Third International Workshop on Distributed Event-Based Systems

DEBS 2004

May 24–25, 2004
Edinburgh, Scotland, UK

Workshop Organization

Program Co-Chairs

Antonio Carzaniga
University of Colorado, USA

Pascal Fenkam
Vienna University of Technology, Austria

Program Committee

Bernhard Aichernig
United Nations University, Macau

Jean Bacon
University of Cambridge, UK

Gianpaolo Cugola
Politecnico di Milano, Italy

Juergen Dingel
Queen's University, Canada

Gabi Dreo-Rodosek
Leibniz Supercomputing Center, Germany

Schahram Dustdar
Caramba Labs, Austria

José Luiz Fiadeiro
University of Leicester, UK

Ludger Fiege
Darmstadt University of Technology, Germany

David Garlan
Carnegie Mellon University, USA

Manfred Hauswirth
Swiss Federal Institute of Technology Lausanne (EPFL), Switzerland

Paola Inverardi
University of L'Aquila, Italy

H.-Arno Jacobsen
University of Toronto, Canada

Mehdi Jazayeri
Vienna University of Technology, Austria

J.-P. Martin-Flatin
CERN, Switzerland

Cecilia Mascolo
University College London, UK

Peter Pietzuch
University of Cambridge, UK

Ivana Podnar
University of Zagreb, Croatia

David Rosenblum
University College London, UK

Rob Strom
IBM T.J. Watson Research, USA

Joe Sventek
University of Glasgow, Scotland

Peter Triantafillou
University of Patras, Greece

Table of Contents

Industrial Challenges in Working with Events

Petre Dini
Cisco Systems, Inc., USA, and Concordia University, Canada

Event-based systems, event-driven management systems, event-based applications, publish/subscribe systems, content-based networking, event-processing, incident handling events, intrusion detection events, application-related events, configuration-change events, and other similar event-based expressions capture so much semantic and complexity that each in turn deserves a special presentation. There is a growing consensus between practitioners and scientists on event taxonomy, event processing operations (filtering, correlating, aggregation, etc.), temporal representation, and the system requirements to receive, subscribe, process, and deliver events. Nonetheless, the hot industrial challenges in management systems to producing, handling and reacting to events are shadowed by very simple problems that are yet unsolved problems across the industry. These problems are in the definition, interoperability, and implementation.

This talk will briefly illustrate several angles of current concerns, through use cases covering device instrumentation, network protocols, domain-oriented application, and management applications. The declared target is to identify challenges and raise appropriate questions, open to a large debate. Some issues may have immediate and simple solutions; others may be simply postponed, due to the lack of feasible and testable solutions.

At the device instrumentation level, we will tackle the loose and strong events forms, and their related notions of simple, complex, and preprocessed events, with examples from Syslog-based and SNMP-based systems. Issues related to the performance of parsing and tagging in normal and high speed networks will be analyzed. We will then focus on some versioning and version conversion aspects concerning SNMP-events' OIDs and naming, the inconsistency in definition/implementation of mandatory/optional recommendations, and the reliability and scalability in event systems at this level.

Generic concerns relate to the temporal issues. In general, multiple timestamps make event processing difficult. On the other hand, some device types may not have clocks at all., and some types of networks may well tolerate delays and clock inconsistencies. A uniform representation of the timestamps, which is still under consideration in standards, will be briefly discussed.

Linking the events to the system behavior raises the problem of a common syntax, if not a common semantic. Alternatively, event conversions and format translations are needed. Domain-oriented events are the most challenging for system interactions and event processing, as we count a growing number of heterogeneous formats covering intrusion detection, accounting, fault, performance, incident, vulnerability, state change, etc. The message intended by this talk is that an educated event design and a consistent implementation could dramatically reduce the set of real concerns, regardless of the systems used to handle (transport, process, and react to) the events, and independent of the technology used for supporting that design (e.g., CORBA, JMS, and TIBCO).

The talk will also cover several basic solutions that are under development at Cisco Systems, Inc., and that are intended to be ported towards standards. These solutions are especially related to reporting complex semantic behavior, embedded preprocessing directives, and smart buffering mechanisms for reliable events.

Several remaining questions will be used as a basis for an open debate. These questions are related to the workshop topics, but they also expand to new industrial paradigms. These paradigms, which include utility-based computing, autonomic computing, diagnosis in-the-box, diagnosis out-of-box, adaptable applications, self-adaptable applications, and reflexive environments require a new approach to formalizing events, architecting event-based systems, and integrating such systems. GRID systems, for example, introduce the concept of intermittent and partial behavior related to resource sharing, which may require a special semantic on SLA/QoS violation events. Events related to traffic patterns and the dynamics of performance and availability changes also require particular metrics and processing. Another hot area, quite poorly covered in terms of event-related requirements, is MPLS OAM and all aspects related to MPLS VPN. The talk will discuss why these areas are perceived as hot topics in industrial research, and where the challenging issues are.

Petre Dini is a senior technical leader and principal architect with Cisco Systems, Inc., being responsible for policy-based strategic architectures and protocols for network management, QoS/SLA, performance, programmable networks and services, provisioning under QoS constraints, and consistent service manageability. He's industrial research interests include mobile systems, performance, scalability, and policy-related issues in

GRID networks. He's also working on particular issues in multimedia systems concerning traffic patterns and security. He worked on various industrial applications including CAD/CAM, nuclear plant monitoring, and real-time embedded software. In early 90's he worked on various Pan-Canadian projects related to object-oriented management applications for distributed systems, and to broadband services in multimedia applications. As a Researcher at the Computer Science Research Institute of Montreal he coordinated many projects on distributed software and management architectures. In this period he was an Adjunct Professor with McGill University, Montreal, Canada, and a Canadian representative in the European projects. Since 1998 he was with AT&T Labs, as a senior technical manager, focusing on distributed QoS, SLA, and Performance in content delivery services.

Petre is the IEEE ComSoc Committee Chair of Dynamic Policy-Based Control in Distributed Systems, and involved in the innovative NGOSS industrial initiative in TeleManagement Forum. Petre is also a Rapporteur in Study Group 4 at ITU-T. He has been an invited speaker to many international conferences, a tutorial lecturer, chaired several international conferences, and published many technical papers.

He is currently an Adjunct Professor at Concordia University, Montreal, Canada, a Senior IEEE member, and an ACM member. He recently founded IARIA, a non-profit organization for promoting new ideas and bridging scientific events across the world.

A Self-Organizing Crash-Resilient Topology Management System for Content-Based Publish/Subscribe

R. Baldoni, R. Beraldi, L. Querzoni and A. Virgillito
Dipartimento di Informatica e Sistemistica
Università di Roma “La Sapienza”
Via Salaria 113, 00198, Roma, Italy
email: {baldoni, beraldi, querzoni, virgi}@dis.uniroma1.it

Abstract

Content-based routing realized through static networks of brokers usually exhibit nice performance when subscribers with similar interest are physically close to each other (e.g. in the same LAN, domain or in the same geographical area) and connected to a broker which is also nearby. If these subscribers are dispersed on the Internet, benefits of such routing strategy significantly decrease. In this paper we present a Topology Management System (TMS), a component of a content-based pub/sub broker. The aim of a TMS is to mask dynamic changes of the brokers' topology to the content-based routing engine. TMS relies on a self-organizing algorithm whose goal is to place close (in terms of TCP hops) brokers that manage subscribers with similar interests keeping acyclic the topology and without compromising network-level performance. TMS is also resilient to broker failures and allows brokers join and leave.

1. Introduction

Content-based publish/subscribe (pub/sub) communication systems are a basic technology for many-to-many event diffusion over large scale networks with a potentially huge number of clients. Scalable solutions are obtainable only deploying a distributed event system made up of a large number of cooperating processes (namely brokers) forming a large scale application level infrastructure.

Current state-of-the-art content-based systems ([3], [4], [6] and [10]) work over networks of brokers with acyclic and static topologies. These systems exploit the content-based routing algorithm (CBR) introduced in SIENA [3] for efficient event diffusion. By making a mapping between the brokers' topology and the subscriptions present at each broker, CBR avoids an event to be diffused in parts of the network where there are no subscribers interested in, thus reducing the number of TCP hops experienced during the event diffusion toward all the intended subscribers. However, such a filtering capability performs at its best only when each broker presents a high physical similarity among its subscriptions (i.e., all subscribers sharing similar interest are physically connected to the same broker). On the contrary, when subscribers sharing similar interests are connected to a dispersed subset of distinct brokers, which are far from each other in terms of hops, the performance gain obtained using the CBR algorithm, with respect to an algorithm that simply floods the network with events, becomes negligible. Therefore such performance gain depends from the distribution of the subscriptions in the brokers' network, which is usually not under control.

In this paper we present a self-organizing topology management system (TMS) for content-based publish/subscribe. TMS encapsulates all the logic associated with the management of a dynamic network of brokers. The main aim of a TMS system is to put a CBR algorithm working at its best condition of efficiency. Therefore, TMS dynamically rearranges TCP connections between pair of brokers in order to put subscribers sharing similar interests as close as possible, even though they are connected to distinct brokers, thus creating the conditions for a reduction of TCP hops per event diffusion. Unfortunately, on TCP/IP networks reducing an application level metric (like TCP hops) can lead to poor performance at the network level. TMS exploits network-awareness capabilities to ensure that a reconfiguration is made permanent only if it does not harm network-level performance.

TMS also includes a module to heal the network after joining/leaving/crashing of brokers in order to maintain an acyclic topology. In general, such a reconfiguration is confined in the neighborhood of the failed node. In this sense, the system provides local self-healing capabilities. Let us finally remark that content-based systems like ([3], [4] and

[6]) require human intervention whenever the brokers' network topology has to be changed in order to add or remove brokers.

In the rest of this paper we give a general overview of the TMS, explaining its features. The paper is structured as follows. Section 2 presents the general architecture of a content-based publish/subscribe system. Section 3 describes the internal architecture of our new Topology Management System. Section 4 describes the details of the algorithms used to handle topology maintenance and self-organization. Section 5 compares our work with the related works in this research area. Finally, Section 6 concludes the paper.

2. Content-Based Pub/Sub Background

This Section defines the key concepts related to a content-based publish/subscribe system and the architecture of a generic broker. Such a system manages subscriptions and events defined over a predetermined event schema, constituted by a set of n *attributes*. An event is a set of n values, one for each attribute, while a subscription is defined as a set of *constraints* over attributes.

A content-based pub/sub system is structured as a set of processes, namely *event brokers*, $\{B_1, .., B_N\}$ interconnected through transport-level links which form an acyclic topology. In particular each broker maintains a set of open connections (*links*) with other brokers (its neighbors). The link connecting broker B_i with broker B_j is denoted as $l_{i,j}$ or $l_{j,i}$. Each broker may have only a bounded number of neighbors (its maximum fan-out F) to ensure that the number of concurrent open connections does not harm process performance.

The internal architecture of a generic broker is depicted in figure 1; each broker is composed by five components:

Subscription table. This table contains all the subscriptions issued by subscribers to the broker. We define as the *zone of interest* of a broker B_i, denoted Z_i, the union of all local subscriptions at B_i.

Pub/Sub manager. A broker can be contacted by clients, that, depending on their role, can be divided in *subscribers* or *publishers*. Publishers produce information by firing events to one of the brokers, while subscribers express their interests for specific events by issuing subscriptions to one of the brokers. The Pub/Sub manager has the responsibility of managing message exchanges with clients (i.e. publishers and subscribers). Subscriptions received by subscribers are inserted in the subscription table, while events received by publishers are passed to the CBR engine. The Pub/Sub manager is also responsible of matching events received by the network (and passed to it by the CBR engine) with local subscriptions contained in the subscription table, in order to notify the corresponding subscribers.

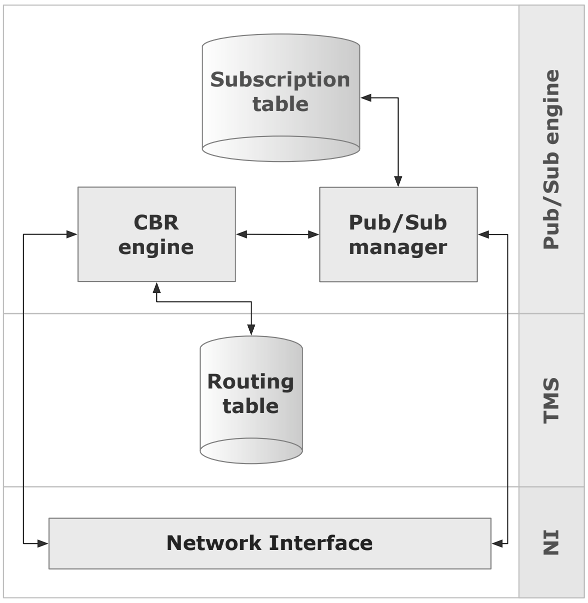


Figure 1. A generic broker architecture

Routing table. This table stores a local view of both the the broker's network topology and the global subscription distribution. Specifically, the number of entries of the routing table at broker B_i is equal to the number of B_i's neighbor brokers. Each entry provides the zone of interest advertised by the corresponding neighbor node. For example let B_j be a neighbor of B_i. The zone of interest stored into the row associated with B_j in B_i's routing table represents the union of all zones of interests behind the link $l_{i,j}$.

CBR engine. This component implements the content-based routing algorithm (CBR) for acyclic peer-to-peer topologies introduced in SIENA [3]. It has two main functions that read and update the routing table: forwarding of events and updating of the zone of interests of each broker.

Each time an event e is received by the broker either from its local publishers (via the Pub/Sub manager) or from a link, this component forwards e *only* through those links which can lead to potential e's subscribers. In this case we say that B_i acts as a *forwarder* for e. Moreover we define *pure forwarder*, a forwarder broker which hosts no subscriptions matching the event e. Forwarding links are determined by the CBR engine from the routing table, by matching the event against the entries contained in it.

When a new subscription S arrives at the pub/sub manager broker of B_i, if S is not a subset of Z_i, it is first added to the Subscription table and then spreaded to B_i's neighbors via the CBR engine. When a CBR component of a broker B_j becomes aware of S through a message received from one of its incoming TCP links, namely l, it updates the row of its routing table by adding S to the l's entry in the routing table. Note that only the zone of interest can be changed by the routing protocol.

Network Interface. This component manages all the low level network-related issues. Different implementations can be used to let the broker work on different types of network.

While the first three components constitute the core of the pub/sub engine, the routing table can be considered the distributed representation of the brokers' network. In this sense we define it as the Topology Management System (TMS) of the broker. For a simple content-based pub/sub broker based on a static and acyclic network topology, like the one depicted in figure 1, the TMS is reduced to a single data structure, i.e. the routing table. Due to the absence of "active components" inside the TMS, this type of systems require human intervention in order to modify the network topology (adding or removing entries inside the routing table).

3. A TMS Architecture

This Section describes the internal architecture of a novel Topology Management System for Content-based Publish/Subscribe. Our TMS is capable of adding automatic network management, failure resiliency and self-organization to a plain CBR-routing algorithm. This is done by a set of components that update transparently the set of neighbors of a broker B_i by modifying the number of rows and/or the content of a row in the routing table.

The architecture of the TMS at broker B_i is depicted in figure 2; we added three main active elements to the simpler TMS shown in the previous figure (a detailed description of each of these components can be found in Section 4.1):

Broker join manager. This component is used by B_i to manage a new broker B_j trying to join the network. The algorithm used by this component (i) tries to obtain a fair distribution of the brokers over the whole network and (ii) updates B_i's routing table as soon as B_i accepts B_j as a new neighbor.

Broker leave manager. This component is used by B_i to manage brokers that want to leave the network and maintain the network healthiness through algorithms that handle broker crash failures. In order to reach this last goal, the component incorporates an eventually perfect failure detector based on heartbeats. The algorithm ensures that the topology remains acyclic without partitioning despite broker crash and/or leave. Each neighbor of the failed broker, say B_j, has to cooperate with the rest of B_j's neighbors to consistently modify its local routing table.

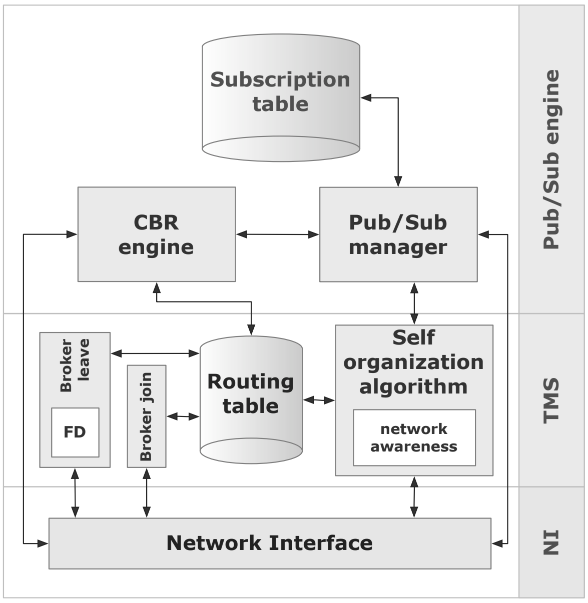


Figure 2. A crash-resilient self-organizing broker architecture

Self-organization engine. This component implements a self-organization algorithm capable of increasing CBR performance through topology reconfiguration of the brokers' network. Each reconfiguration tends to place close to each other, in terms of TCP hops, brokers whose zone of interests are very similar in order to reduce as much as possible the number of pure forwarder. However, a reduction of TCP hops does not always leads to better network level performance (for example establishing a TCP connections between two brokers with a narrow bandwidth can decrease the performance of the overall system even though these brokers share a large zone of interest). Therefore this component embeds a network-awareness module that allows topology reconfigurations only if network level performance are not severely harmed.

It is important to remark that TMS varies the number of entries in the routing table transparently wrt to the CBR engine; a cooperation between TMS and the CBR can however increase the performance of the pub/sub system.

4. A Detailed View of the TMS

4.1. Broker Join

A broker joining the system is added as a new neighbor to one already in the system. The join algorithm ensures a fair distribution of the brokers inside the network (i.e, new brokers are added with higher probability to brokers having a small number of neighbors). Moreover, it ensures that the number of connections handled by each broker is always bounded. When a new broker B_j wants to join the system, it starts contacting a broker randomly chosen among the ones already in the system. A broker receiving a request can choose to accept or forward it to its neighbors (one at a time, with the eventual exclusion of the broker it received the request from). A join request is accepted by a broker B_i with probability $p = \frac{F-|R_i|}{F}$, where $|R_i|$ is the number of rows in B_i's routing table. This ensures that the number of neighbors for a broker does not exceeds the maximum allowed. When a broker B_i accepts a join request, it adds a new entry for B_j to its routing table and sends to B_j the union of its zone of interest Z_i with all the zones covered by its links. This zone is added by B_j to its routing table, associated to the newly created link.

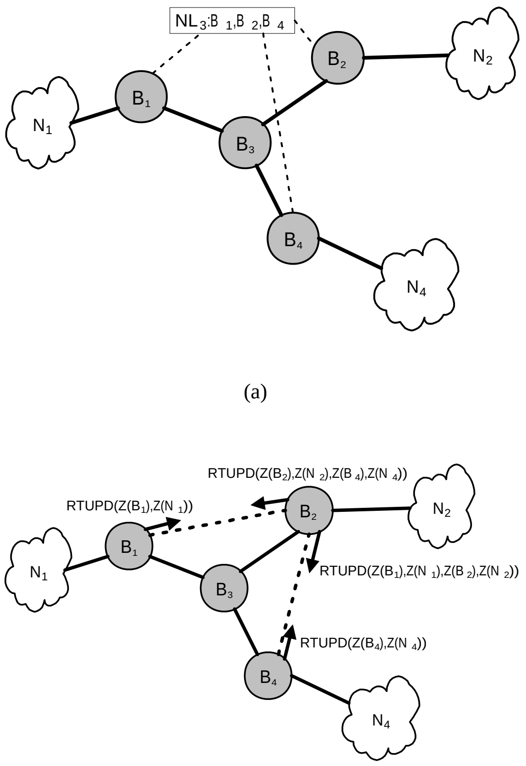
4.2. Broker Leave and Failures

A broker may leave the network voluntarily or if a failure occurs. We assume that brokers can fail by crashing, that they can fail only one at a time, and that the failure of a broker is eventually detected by all its neighbor brokers (using well-know failure detection techniques, like heartbeats). Moreover, a non-faulty broker can be erroneously detected as faulty by its neighbors only for a finite amount of time.

The leave-handling protocol must ensure that after a broker leaves, the global properties of the brokers' network are maintained, i.e. new connections have to be established, such that the network topology remains acyclic. Moreover, the state of the routing tables must be kept consistent removing the subscriptions that were hosted by the leaving broker and adding the newly created paths. We will present the protocol in the following by referring to Figure 3.

The leave-handling protocol requires an additional data structure at each broker. A broker B_i maintains a *neighbors list* NL_j for any of its neighbors B_j. This is an ordered list containing all the brokers that are neighbors for B_j. Figure 3(a) shows the neighbors list NL_3 of Broker B_3, contained in each of its neighbors. Neighbor lists are updated at each new brokers join or leave.

If a broker B_j leaves the network voluntarily, it sends a disconnection message to all its neighbors, while if it is faulty, it simply stops sending heartbeats. In both cases each broker in NL_j will eventually become aware of B_j leave and will request a new connection to the broker following itself in NL_j. Each broker $B_i \in NL_j$ receiving a connection request from the preceding broker in NL_j (with the exception of the first broker in NL_j) will accept the connection and test if B_j has actually left the network; if this is the case B_i will issue a connection request to the next broker in NL_j. When a connection request reaches the last broker in NL_j, a new path connecting all brokers previously connected by B_j has been created. Consider the Figure 3(b). If the broker B_3 leaves the network, eventually the connections shown as dashed lines will be set up. It may happen that a broker is erroneously suspected as faulty by some of its neighbors. A broker B_i suspecting a fault in one of its neighbors B_j, sends it a disconnect message before starting the protocol, forcing the link tear down.



(a)

(b)

Figure 3. Managing Fault of a Broker

At this point, routing tables have to be repaired. This means (i) deleting all the subscriptions that previously belonged to B_j on all B_i's outgoing links, except the one connecting to B_i and (ii) updating the routing tables on all the newly created links.

As far as the latter point is concerned, two route update messages are generated. The first one is generated by the broker that started the reconstruction procedure, say node B', and the other by the one that ended the procedure, say node B''. B' sends its message toward B'' and vice versa. Each message is forwarded by all brokers in NL_j, before reaching the destination. Before forwarding the update message to the destination, each broker adds its local subscriptions and the zones of interest of its other links to it. Figure 3(b) shows the details of the route update messages (RTUPD) sent by each broker. The routing table update involves only the neighbors of the faulty brokers without impacting on other parts of the network. In order to avoid message losses all events and subscriptions in transit while repairing the topology are buffered until the repairing process is completed. It is important to point out that the fan-out constraints could be altered after a broker leave. A neighbor of a leaving node (with the exception of first and the last in the neighbors list) opens two new connections, indeed, while dropping only one, thus possibly reaching a number of open connections which is higher than its maximum fanout.

4.3. Subscription-Aware Self-Organization

Our self-organization component works in synergy with the content-based routing algorithm (i.e., it can read the CBR table and modify it after the self-organization) but its actions are totally transparent to it. In the following we describe the various aspects involved in self-organization.

Cost Metrics. The main target of the self-organization is to do its best to avoid the presence of pure forwarders during event diffusion due to the fact that a pure forwarder adds a useless TCP hop to the event diffusion path. This is done through a reorganization of the brokers' network with the aim of minimizing the TPC-hops experienced by an event to reach all its intended destinations. Lowering the number of TCP hops has a positive impact on the scalability of the system: each broker has to process a lower number of messages, and this leads to lower costs for events matching and forwarding. However, it is important to remark that reducing the TCP hops per notification diffusion through a topology rearrangement not necessarily leads to an equal improvement on network level metrics such as IP hops, latency and bandwidth. In other words, an efficient topology configuration wrt TCP hops could exhibit very poor performance wrt, for example, IP hops. Then, our self-organization component takes into account network-level metrics in order to decide if a new connection can lead to a non-efficient network path.

Measuring Subscription Similarity: the Associativity Metrics. Defining subscription similarity in a content based system can be very difficult due to the very generic language used for subscriptions. We introduce an associativity metric which measures the similarity between the interests of two brokers. In a former version of the self-organization algorithm, we exploited the geometric interpretation of subscriptions to compute associativity, starting from the geometrical intersection of the hyper-rectangles representing brokers' subscriptions. However, the method used to measure these intersections introduces a significant computational overhead and it is not straightforwardly extensible to operators such as prefixes and wildcards.

In this paper we introduce a novel solution for computing associativity, which is based on statistics over events received by each broker. A broker B_i maintains the history H_i of the last n received events. Note that, due to the features of the CBR algorithm, histories of distinct brokers usually are completely different. Any event in H_i that matches a subscription of B_i is included in another list $M(H_i)$. Associativity of a broker B_i with respect to another broker B_j, denoted as $AS(j)_i$ is computed as the percentage of events matched by B_i that are also matched by B_j:

$$AS(j)_i = |M(H_i) \cap M(H_j)|$$

We can also define the total associativity at a broker B ($AS(B)$), as the sum of the associativity among the broker and all its neighbors and, finally, the associativity of the whole system (AS) as the sum of associativity of all the brokers.

Self-Organization Algorithm. The algorithm follows this basic simple heuristic: each broker B tries to rearrange the network in order to obtain an increment of its associativity $AS(B)$ while not decreasing AS. In the following we give a quick sketch of the algorithm; a complete specification, along with implementation details and experimental results, can be found in [1].

A self-organization is triggered by a broker B when it detects (through the reading of the CBR tables) that there can be a broker B', not directly connected to B, that can increase its associativity. The aim of the self-organization is: (i) to connect B to B' and (ii) to tear down a link in the path between B and B' in order to keep the topology of the network acyclic (a requirement of the CBR algorithm). The algorithm does its best to select the link in the path between B and B' such that the two brokers it connects have the lowest associativity. Self-organization takes place only if it leads to an increase of AS and this, as confirmed by experiments, means increasing the probability that two brokers with common interests are placed close in the network. Network-level performance is accounted in the algorithm following this simple principle: self-organization can occur only with those brokers whose network-level distance with the source broker is less than a reference value d. In other words, a broker B starting the reconfiguration will choose as its new neighbor the most similar one having a network distance from it in the range delimited by d. Network distance can be measured indifferently with any metric, either IP hops or latency or bandwidth, depending on the specific requirements of the application.

5. Related Work

Several content-based pub/sub systems have been presented in the literature. Early content-based systems, such as SIENA [3] or Gryphon [2] did not provide any form of self-organization. The creation and the maintenance of the topology is left to the user. Peer-to-peer overlay network infrastructures, such as Pastry [9] and Tapestry [12] exploits self-reconfiguration capability for fault-tolerant routing and for adjusting routing paths with respect to metrics of the underlying network. Overlay network infrastructures represent a general connectivity framework for many classes of peer-to-peer applications, including publish/subscribe. For example, Scribe [8] and Bayeux [11] are two topic-based publish/subscribe systems built on top of two overlay network infrastructures (respectively Pastry and Tapestry). Only recently in ([7] and [10]) the same idea is being applied to two content-based systems.

None of the aforementioned systems include a subscription-driven self-organization mechanism. A reconfiguration algorithm similar to the one included in our TMS is described in [5]. Authors assume that some link may disappear and others appear elsewhere, because of changes in the underlying connectivity (this is the case, for example, of mobile ad-hoc networks). “Reconfiguration” in this case means fixing routing tables entries no longer valid after the topology change. In contrast, in our approach we *induce* a reconfiguration to create more favorable conditions for event routing.

6. Conclusions and Future Work

In this paper we presented a crash-resilient, self-organizing Topology Management System for content-based publish/subscribe. TMS masks topology changes to the content-based routing algorithm in order to keep an acyclic topology. TMS is self-organizing in three senses: (i) it handles autonomically brokers joining/leaving the system, (ii) it can self-heal the application-level network after a broker crash and (iii) it can dynamically reconfigure the connections among brokers in order to create paths that increase the performance of content-based routing.

While the implementation of the TMS is currently under development, we already did a simulation study of all the algorithms sketched in this paper. The simulations show how the self organization algorithm achieves significant improvements wrt the TCP hop metric compared to a simple CBR and without affecting the network latency of notifications (more details can be found in [1]). Moreover the topology maintenance algorithm allows the system to offer self-healing properties in presence of single broker crash.

References

[1] R. Baldoni, R. Beraldi, L. Querzoni, and A. Virgillito. Subscription-driven self-organization in content-based publish/subscribe. Technical report, downloadable from http://www.dis.uniroma1.it/ midlab/docs/bbqv04techrep.pdf, DIS, Mar 2004.

[2] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, and D. Sturman. An Efficient Multicast Protocol for Content-based Publish-Subscribe Systems. In *Proceedings of International Conference on Distributed Computing Systems*, 1999.

[3] A. Carzaniga, D. Rosenblum, and A. Wolf. Design and Evaluation of a Wide-Area Notification Service. *ACM Transactions on Computer Systems*, 3(19):332–383, Aug 2001.

[4] G. Cugola, E. D. Nitto, and A. Fuggetta. The JEDI Event-Based Infrastructure and its Applications to the Development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, 27(9):827–850, September 1998.

[5] A. L. M. G. P. Picco, G. Cugola. Efficient content-based event dispatching in the presence of topological reconfiguration. In *23rd International Conference on Distributed Computing Systems (ICDCS 2003), 19-22 May 2003, Providence, RI, USA*, pages 234–243, 2003.

[6] P. Pietzuch and J. Bacon. Hermes: A Distributed Event-Based Middleware Architecture. In *Proceedings of the 1st International Workshop on Distributed Event-Based Systems (DEBS'02)*, 2002.

[7] P. Pietzuch and J. Bacon. Peer-to-Peer Overlay Broker Networks in an Event-Based Middleware. In *Proceedings of the 2nd International Workshop on Distributed Event-Based Systems (DEBS'03)*, 2003.

[8] A. Rowston, A. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The Design of a Large-Scale Notification Infrastructure. In *3rd International Workshop on Networked Group Communication (NGC2001)*, 2001.

[9] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of International Conference on Distributed Systems Platforms (Middleware)*, 2001.

[10] W. W. Terpstra, S. Behnel, L. Fiege, A. Zeidler, and A. P. Buchmann. A Peer-to-Peer Approach to Content-Based Publish/Subscribe. In *Proceedings of the 2nd International Workshop on Distributed Event-Based Systems (DEBS'03)*, 2003.

[11] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. Katz, and J. Kubiatowicz. Bayeux: An Architecture for Scalable and Fault-tolerant Wide-area Data Dissemination . In *11th Int. Workshop on Network and Operating Systems Support for Digital Audio and Video*, 2001.

[12] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. Katz, and J. Kubiatowicz. Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing. Technical Report UCB/CSD-01-1141, University of California at Berkeley, Computer Science Division, April 2001.

The Event-Based Paradigm in Collaborative Product Data Management: A Case Study

Mauro Caporuscio and Paola Inverardi
Dipartimento di Informatica
Università dell'Aquila
I-67010 L'Aquila, Italy
{caporusc, inverard}@di.univaq.it

Abstract

In this paper we discuss our ongoing experience in integrating a Product Data Management application, THINKTEAM, *and the* SIENA *Publish/Subscribe Middleware. This integration should support* THINKTEAM *users in managing the* lost-update *problem. This problem arises when a client performs a* check-out/modify/check-in *cycle on a document that may be used as reference copy by other clients. In this context* SIENA *should allow for disseminating alerts about the state of the document. The integration between* THINKTEAM, *and* SIENA *poses new requirements on how to manage events and subscriptions. The paper discusses a set of integration problems and their possible solutions.*

1 Introduction

Since the past few years industry has been facing the necessity to decrease its time-to-market. In fact, a design engineer usually spends up to 25-30% of his time for handling information, looking for it, retrieving it, waiting for copies of drawings, archiving new data, etc...

In order to speed up these activities computer systems have been developed to help improving the flow, quality and use of engineering information throughout a company. These systems, known as *Product Data Management* (PDM) [10], support the engineering processes management through the control of engineering data, of engineering activities, of engineering changes and of product configurations. That is, PDM systems maintain the control of data and distribute it to the people who need it when they need it.

The way PDM systems cope with this central role is by holding data (only once) in a secure repository where data integrity can be assured and all changes to data can be monitored, controlled and recorded. Due to the distributed nature of the system, all these actions need coordination and concurrency control. In fact, a usual problem that arises from this kind of systems is the so called "*lost-update*" problem. The lost-update problem (depicted in Figure 2) may occur when a user inserts, deletes, or changes something in the repository, and the effects of his work is nullified by another user.

While an automatic solution of this problem is not possible (as it is critically related to the type, nature and scope of the changes that will be performed on the document) a distributed notification facility would provide the mean to supply the actors with adequate information. In this context content-based Publish/Subscribe systems [6, 5] seem to be a well suited solution for disseminating information about operations made on documents of interest. Moreover, previous work that has been done in exploiting Publish/Subscribe capabilities within Computer Supported Collaborative Work (CSCW) [7, 3], demonstrates how event-based system can be used to build complex, distributed applications for data management.

In this paper we present ongoing work on using the SIENA Publish/Subscribe Middleware [4] within the THINKTEAM PDM [11] application. This work shows the use of an event-notification system in a real business-oriented application, THINKTEAM, and discusses some research issues raised from this experience.

The paper is organized as follows. Section 2 describes the THINKTEAM PDM application and introduces the above mentioned coordination problem. Section 3 outlines the integration of SIENA with THINKTEAM and discusses the problems, and solutions, that are posed by this integration. The last Section concludes and presents future work.

2 ThinkTeam Description

THINKTEAM is a Product Data Management (PDM) application that provides Product Lifecycle Management (PLM) facilities [11]. THINKTEAM provides data (and document) management capabilities to deal with all product documentation: 3D models, 2D drawings, product specifications, analysis results, etc... Moreover, THINKTEAM allows multiple users to access to updated, released and ongoing work of product information.

Structurally, THINKTEAM is a three-tier application, which implements an information management system using an underlying DBMS for persistence and retrieval of the metadata (non-document data). The most typical installation scenario is a network (as shown in Figure 1) where several distributed clients interact with one centralized RDBMS server and one more file server (called *vault*). In this environment, components resident on each client peer supply graphical interface, metadata management and integration services. Persistence services are achieved building on the characteristics of the RDBMS and file servers.

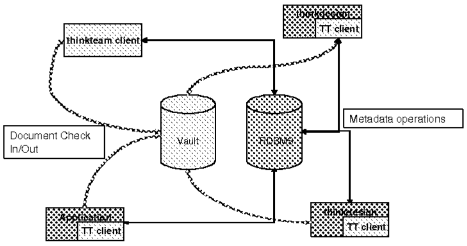


Figure 1. Thinkteam Network

In this context a central role is played by the vault component. The vault is a file system-like repository that controls storage and retrieval of document data in PDM systems. The main functions of the vaulting system are:

1. Providing a single, secure and controlled storage environment where the documents controlled by the PDM system are managed.

2. Preventing inconsistent updates or changes to the document base, while still allowing the maximal access compatible with the business rules.

While the first point involves the low-layer implementation of the vaulting system, the second point concerns the following standard set of operations:

- **get**(*doc*): extracts a read-only copy of the document *doc* from the vault.

- **import**(*doc*): inserts a new document *doc* in the vault.

- **checkout**(*doc*): extracts from the vault a copy of the document *doc* for being modified; on a specified *doc* only one check-out at a time is possible.

- **checkin**(*doc*): replaces a modified document *doc* in the vault. Of course, *doc* must have previously been checked-out.

- **uncheckout**(*doc*): cancels the effects of a previous check-out on *doc*.

Due to the high number of possible clients there is a high degree of concurrency in vaulting operations. In order to maximize concurrency, THINKTEAM does not create exclusive locks for check-out operations. It is therefore possible for clients to perform a **get** operation on already checked out documents. This usually causes the so-called *lost-update* problem. The problem (as depicted in Figure 2) arises when one client performs a *check-out/modify/check-in* cycle on a document that may be used as reference copy by other clients.

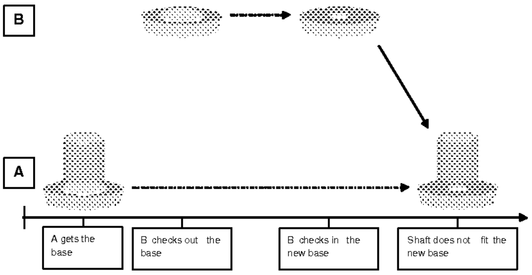


Figure 2. The lost-update problem

Since an automatic solution of this conflict is not possible, an alternative is required. In this context dissemination of information about performed actions would allow for alerting interested clients. Of course, each user will be responsible of his own reactions.

In next Section we show the use of the SIENA Publish/Subscribe Middleware for disseminating information of interest.

3 Event-Notification Within ThinkTeam

As pointed out in the previous Section, an automatic solution of the conflict is not possible, because it is critically related to the type, nature and scope of the changes that will be performed on the document. A distributed dissemination of the information would provide the mean to supply the users with notices by:

1. Advising the user who checks out a document that there exist outstanding reference copies.

2. Notifying the copy holders upon check-out and check-in operation of the document.

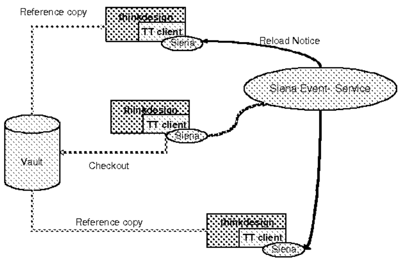


Figure 3. thinkteam and SIENA

Figure 3 shows the new integrated software architecture. In this architecture two additional actors can be recognized: the *Siena Client* and the *event-service*. Each *Siena Client* can be both *publisher* and *subscriber*. Publishers emit events into the event-service which in turn delivers them to the interested subscribers.

Starting from the actions described above and the temporal constraints upon them we can draw the High-level Message Sequence Chart [8] (depicted in Figure 4) and, then derive a set of scenarios. Some of them describe critical use cases that require careful management. For example, Figure 5 shows the scenario involving a **get** and a **checkout** operations: *TTc-proactive* checks out a document doc, subscribes for information about it and, then publishes the associated event in order to inform other clients about its intents. After that, the *TTc-reactive* retrieves the same document from the vault, subscribes for actions involving it and, informs other clients about its action. At this time the event-service should inform *TTc-reactive* that there exists a client that is currently modifying such a document, and the *TTc-proactive* that there exists a client that has a reference-copy of such a document.

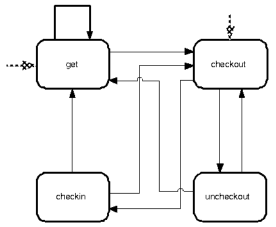


Figure 4. High-level MSC

This scenario represents a case that may not be satisfied. In fact, due to the native semantics of SIENA (and of the event-based systems in general), it is not possible to deliver events to those clients that are not subscribed at the time of publications. This problem (that we refer as "Missed-events Problem"), already pointed out in previous work [1], can be solved in different ways that we discuss in Section 3.1.

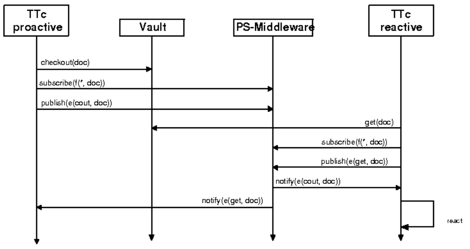


Figure 5. A Scenario concerning get and checkout operations

Moreover, due to the relevance of the events sent, the notification system should be reliable. In fact, if an event of interest gets lost, and then it is not delivered to the clients, the "lost-update" problem occurs again. Currently, SIENA does not solve this problem and then a reliability mechanism must be added. We discuss this aspect in Section 3.2.

3.1 Missed Events

The "Missed Events" problem occurs when a client emits an event *before* another client submitted his subscription matching such event. There are different possible solutions for this problem. (1) To provide an event-persistence mechanism inside the system. (2) To insert an additional component, a *Repeater*, that provides caching services, in the system architecture. (3) To use a pair <subscribe, notify>.

While solution (1) is invasive and requires to change the system, solutions (2) and (3) both operate at *application-level*. That is, they do not change the publish/subscribe architecture, but instead combine its features to reconcile publisher and subscriber actions. We will describe all of them in the following.

3.1.1 Stable Events

A possible solution would be to provide event persistence directly inside the event-service. That is, the event-service itself stores all the events and it will deliver them when somebody will submit a matching filter.

At first we should distinguish two kind of events: *stable-* and *vanish-events*. While *vanish-events* are events that do not need to be stored, *stable-events* represent those events that require persistence. This solution, similar to the one adopted by the Java Message Service (JMS) [9], is particularly invasive, it requires a protocol re-implementation and it introduces many questions that need an answer such as "How long a notification should be stored?" or "May a notification be erased from the repository?" and "Who can delete it?", etc...

3.1.2 Repeater

On the other hand, the *Repeater* acts as a network client and receives and stores all notifications sent through the network (see Figure 6). New users that are looking for a specific event can ask the *Repeater* for old notifications regarding that event. Although easy to develop, this does not look like a good solution. In fact it poses the same questions mentioned in Section 3.1.1 and one more: "Does the presence of a *Repeater* vanish the (asynchronous and loosely-coupled) Publish/Subscribe Architecture philosophy?".

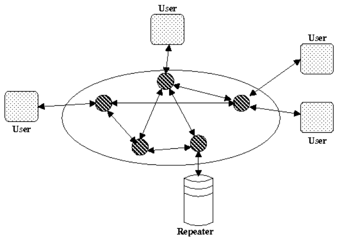


Figure 6. A Subscribe/Notification Architecture with a Repeater.

3.1.3 A subscription-notification combination

The basic idea in this solution is to use a combination of a subscription and a notification. As explained before, the main problem is when a client subscribes for an event after the provider published an announcement for that event.

The following actions explain how a pair <subscribe,notify> works (see also Figure 7):

- provider:

  step p1: subscribe for "I need service *S*"

  step p2: publish "Service *S*"

- client:

  step c1: subscribe for "Service *S*"

  step c2: publish "I need service *S*"

Note that these are not *atomic actions* but there is a little time between the subscription and the notification. Moreover, we assume that the Publish/Subscribe service is unreliable and messages could be delayed through the network. So, different cases are possible. For space reason we show only one of them, the interested reader can refer to [1] where it is shown how the pair <notify,subscribe> works in all cases. Figure 7 depicts how this solution avoids the missed-event problem. We suppose that the client subscribes for an event after the provider sends its notification. Since we are using the pair <subscribe,notify>, the provider has been subscribing for interested parties (action p1). This subscription will catch client's notification (action c2); then the provider knows that a user needs his service. Therefore the provider can publish again his notification.

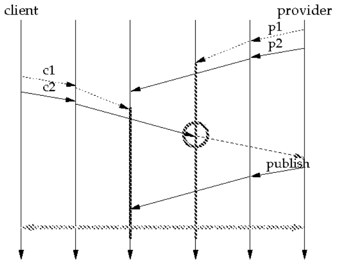


Figure 7. Both users send a pair <subscribe,notify>.

3.2 Reliability Mechanism

Although *reliability* is one of the most important non-functional requirements needed by a distributed application, it is difficult to be guaranteed in a loosely-coupled asynchronous environment such as Publish/Subscribe systems. SIENA does not provide any explicit mechanism for managing reliable communication.

In distributed event-based systems, two different level of reliability can be recognized: *inter-service* and *service-to-client*. While *inter-service* reliability concerns the communication between two nodes in the event-service, *service-to-client* involves a client and its access-point. Since the particular implementation of SIENA does not distinguish between these two kind of communication, it is possible to develop the solution once and apply it in both cases.

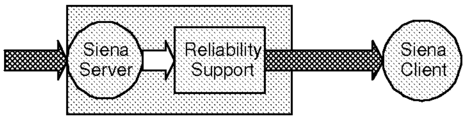


Figure 8. A Support Service for Reliable Communication.

Also in this case we have investigated different approaches: (i) an invasive one that requires to modify the protocol. (ii) a non invasive implementation developed as support-service. Previous work done in designing support-services for the SIENA Middleware [2], suggest how to operate. The idea (see Figure 8) is to build a wrap around the SIENA server that intercepts the outgoing communication, extracts information of interest and implements the new service transparently to the under-layer protocol. This new service should store the notification whenever the communication is not available and keep it until the communication will be restored. Of course some kind of policy is needed in order to avoid infinitely-stored events.

4 Conclusion and Future Work

In this work we have presented and discussed our ongoing experience in integrating the THINKTEAM PDM application and the SIENA Publish/Subscribe Middleware. The use of an event-based system should avoid the so called *lost-update* problem. This problem, usual in such kind of applications, arises when a client performs a *check-out/modify/check-in* cycle on a document that may be used as reference copy by other clients. In this context SIENA should allow for disseminating alerts about the state of the document.

This integration poses new requirements on how to manage events and subscriptions: (i) since alerts must reach every interested client, the system should avoid lost-events. (ii) Moreover, since the native semantics of SIENA does not prevent the *missed-events* problem, a specific solution is required.

This paper has pointed out these aspects and proposed some possible solutions. Currently, ongoing work is in two directions: (i) to validate these solutions with respect to the mentioned problems. They will be implemented as additional services for SIENA and will be tested to measure their impact on the system performances. (ii) Since THINKTEAM is implemented by using MICROSOFT® Technology, such as COM/DCOM, not compatible with SIENA distributed implementations, we have been porting SIENA-client code to this platform.

Acknowledgment

The authors would like to thank Alessandro Forghieri and Maurizio Sebastianis from THINK3 INC. for their contribution to this work.

The authors would also like to acknowledge the Italian national project C.N.R. SP4 that partly supported this work.

References

[1] M. Caporuscio. Co.M.E.T.A. - Mobility support in the Siena Publish/Subscribe Middleware. Master's thesis, Università degli Studi dell'Aquila - Dipartimento di Informatica, L'Aquila - Italy, March 2002.

[2] M. Caporuscio, A. Carzaniga, and A. L. Wolf. Design and evaluation of a support service for mobile, wireless publish/subscribe applications. *IEEE Transactions on Software Engineering*, 29(12):1059–1071, Dec. 2003.

[3] M. Caporuscio and P. Inverardi. Yet Another Framework for Supporting Mobile and Collaborative Work. In *International Workshop on Distributed and Mobile Collaboration*, Linz, Austria, June 2003.

[4] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Achieving Scalability and Expressiveness in an Internet-Scale Event Notification Service. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 219–227, Portland, OR, July 2000.

[5] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3):332–383, Aug. 2001.

[6] A. Carzaniga and A. L. Wolf. Content-based Networking: A New Communication Infrastructure. In *NSF Workshop on an Infrastructure for Mobile and Wireless Systems*, number 2538 in Lecture Notes in Computer Science, pages 59–68, Scottsdale, Arizona, Oct. 2001. Springer-Verlag.

[7] P. Fenkam, E. Kirda, S. Dustdar, H. Gall, and G. Reif. Evaluation of a publish/subscribe system for collaborative and mobile working. In *Proceedings of the Eleventh IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 02)*, 2002.

[8] ITU-T Recommendation Z.120. Message Sequence Charts. ITU Telecommunication Standardisation Sector.

[9] Sun Microsystems Inc., Mountain View, California. Java message service.

[10] The Product Data Management Information Center. Understanding PDM. http://www.pdmic.com.

[11] Think3 Inc. ThinkTeam: a Product Data Management Application. http://www.think3.com.

Efficient Subscription Management in Content-based Networks

Raphaël Chand, Pascal A. Felber
Institut EURECOM
06904 Sophia Antipolis, France
{chand|felber}@eurecom.fr

Abstract

Content-based publish/subscribe systems offer a convenient abstraction for data producer and consumers, as most of the complexity related to addressing and routing is encapsulated within the network infrastructure. A major challenge of content-based networks is their ability to efficiently cope with changes in consumer membership. In our XNET *XML content network, we have addressed this issue by designing novel algorithms to speed up subscription management at the routers, while guaranteeing perfect routing at all times and maintaining compact routing tables thanks to extensive usage of "aggregation." In this paper, we discuss the issue of subscription management in content-based networks, and we specifically describe and evaluate the algorithms that we have developed for* XNET.

1 Introduction

In content-based publish/subscribe systems, messages are routed on the basis of their content and the interests (subscriptions) of the message consumers. This form of communication is well adapted to loosely-coupled distributed systems with large consumer populations, with diverse interests, wide geographical dispersion, and heterogeneous resources (e.g., CPU, bandwidth). Several techniques have been proposed to implement content routing, with various trade offs in terms of algorithmic complexity, runtime overhead, or bandwidth utilization. In particular, support for *perfect* routing (i.e., a message traverses a communication link only if there is some consumer downstream that is interested in that message) introduces significant management complexity at the routers in the presence of dynamic subscription registrations and cancellations. As a consequence, several content-based publish/subscribe networks do not systematically update their routing tables upon consumer departure and let the accuracy of routing degrade over time.

In our XNET XML content network [7], we have addressed this issue by designing novel techniques to speed up the most time-consuming subscription management operation of the routers. Specifically, we propose algorithms that allow routers to quickly determine the "covering" relationships between an incoming subscription and all the entries of their routing table. Covering relationships are at the core of *subscription aggregation* mechanisms, which help limit the size of routing tables and hence improve the efficiency of the filtering engine while ensuring perfect routing. Although the algorithms presented in this paper have been designed for XPath tree-structured subscriptions, they can be readily applied to other subscription language with similar characteristics. Experimental evaluation demonstrates that these algorithms are highly efficient even when the number of subscriptions in the routing table grows very large.

2 Related Work

Several publish/subscribe systems implement some form of distributed content based routing, most notably IBM Gryphon [2], Siena [3], and Jedi [10]. These systems adopt various approaches to subscription management.

IBM Gryphon [2] uses a distributed filtering algorithm based on parallel search trees maintained on each of the brokers to efficiently determine where to route the messages. The system implements perfect routing and supports subscription registrations and cancellations; in fact, registering (canceling) a subscription reduces to inserting (removing) it from the search tree and is thus an efficient operation. However, to maintain and update the parallel search tree, each broker must have a copy of all the subscriptions in the system. As a consequence, this approach may not scale well to large and highly dynamic consumer populations.

Siena [3] also uses a network of event servers for content-based event distribution, and relies upon a routing protocol most similar to ours, but with limited support for subscription cancellation. In a recent paper [4], the authors of Siena introduce a novel routing scheme for content-based networking based on a combination of broadcast and selective routing. Subscription management is simple and efficient but the system does not guarantee perfect routing, in the sense that consumers may receive messages that they are not interested in. The authors have addressed this issue by having routers periodically request for the routing table of other routers.

JEDI [10] proposes several variations for routing events among its networked event servers; in particular, with the *hierarchical* approach, subscriptions are propagated upward a spanning tree and messages are propagated both upward and downward to the children that have matching subscriptions. Subscription management is simple and efficient, but this approach may lead to large routing tables at the root and unnecessary propagation of events upward the tree.

Our subscription containment and matching techniques are related to the widely studied problem of pattern and regular expression matching. There exists several indexing methods to speed up the search of textual data with regular expressions, like the bit-parallel implementation of NFA [1] and suffix trees [14]. In [6], the authors have addressed the reverse indexing problem of retrieving all the regular expressions that match a given string. They propose RE-Tree, an index structure to quickly determine the regular expressions that match a given input string, by focusing the search on only a small fraction of the expressions in the database.

In [12], Tozawa and Hagiya present a containment checking technique for XML schemas, which is based on binary decision diagrams. Little work has been done on the problem of containment checking for tree-structured XPath expressions. In fact, the problem has been shown to be coNP-complete [11]. A sound but non-complete algorithm has been proposed in [5] to determine whether a given tree-structured subscription covers another subscription, but it does not address the problem of covering relationships between large sets of subscriptions.

3 System Overview

This section gives an overview of the XNet content routing network. We also briefly describe its most essential mechanisms, which are relevant for the rest of the paper. More details can be found in [7].

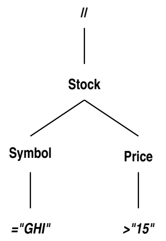
System Model and Definitions. XNet is a distributed publish/subscribe system which consists of a collection of content-based routers (or nodes) organized in an overlay network. Each node routes the messages based on its local knowledge of the consumer subscriptions and the actual content of the messages.

Each data consumer and producer is connected to some node in the network; we call such nodes *consumer* and *producer* nodes. We assume that all routers know their neighbors, as well as the best paths that lead to each producer. We also assume that the number and location of the producer nodes is known. From the point of view of a router, this amounts to knowing which neighbors lead to some producer. The consumer population can be highly dynamic and does not need to be known a priori. The most recent implementation of our routing protocol, XRoute, handles multiple producers [8]; for the sake of simplicity, however, we only consider networks with a single producer in this paper.

Each node has a set of *links*, or *interfaces*, that connects the node to its direct neighbors. We assume that there exists exactly one interface per neighbor, and that communication between two nodes is reliable. Our system also incorporates fault-tolerant mechanisms to handle both transient and permanent failures.

```
<Quotes>
  <Stock>
    <Symbol>DEF</Symbol>
    <Price>34.1</Price>
  </Stock>
  <Stock>
    <Symbol>GHI</Symbol>
    <Price>11.5</Price>
  </Stock>
</Quotes>
```

(a)



(b)

Figure 1: (a) A sample XML document describing two stock quotes. (b) Tree representation of a sample XPath subscription (`//Stock[Symbol="GHI"][Price>15]`) that does not match the XML document.

XNet was designed to deal with XML data, the *de facto* interchange language on the Internet. Producers can thus define custom data types and generate arbitrary semi-structured events, as long as they are well-formed XML documents.

Consumer interests are expressed using a subscription language. Subscriptions allow to specify predicates on the set of valid events for a given consumer. XNet uses a significant subset of the standard XPath language to specify complex, tree-structured subscriptions [13].

An XPath expression contains one or more *location steps*, separated by slashes (`/`). In its most basic form, a location step designates an element name followed by zero or more predicates specified between brackets. Predicates generally specify constraints on the presence of structural elements, or on the values of XML documents using basic comparison operators. XPath also allows the use of wildcard (`*`) and ancestor/descendant (`//`) operators, which respectively match exactly one and an arbitrarily long sequence of element names. We say that an XML document *matches* an XPath expression when the evaluation of the expression on the document yields a non-null object. Figure 1 shows an XML event and an XPath subscription that does not match the event (each branch of the subscription has a matching node in the XML document, but the conjunctive condition at the “Stock” node is not met).

We say that a subscription S_1 *covers* or *contains* another subscription S_2, denoted by $S_1 \supseteq S_2$, iff any event matching S_2 also matches S_1, i.e., $\mathit{matches}(S_2) \Rightarrow \mathit{matches}(S_1)$. The covering relationship defines a partial order on the set of all subscriptions.

The Routing Protocol. XNet implements *perfect routing*, that is, a message traverses a communication link only if there is some consumer downstream that is interested in that message. To do so, each node in the network maintains in its routing table a collection of subscriptions that describe the classes of message that its neighboring nodes are interested in. When receiving a message, a node first determines which subscriptions of its routing table match the event; it then forwards the message to all neighbors that have registered one of these subscriptions. Given accurate routing tables, this process ensures that a message eventually reaches all the consumers, and only those, that are interested in that message.

When a consumer registers or cancels a subscription, the nodes of the overlay update their routing table accordingly by exchanging some pieces of information that represent the registration or cancellation of the consumer. The process starts at the consumer node and terminates at the producer node(s), following the shortest paths. As a consequence, messages published by the producers follow the reverse paths of the subscriptions, along a multicast tree spanning all interested consumers.

The routers in our system reduce the size of their routing tables as much as possible by using elaborate *aggregation* techniques, which are based on the detection and the elimination of subscription redundancies. Subscription aggregation allows us to dramatically improve the routing efficiency of the system both in terms of throughput and latency, because the time necessary to filter a message is proportional to the number of entries in the routing tables. On the other hand, aggregation also adds significant complexity and overhead to the routers, because they need to identify the covering relationships between incoming subscriptions and all the entries of their routing tables. These management operations were the main bottleneck of early XNet implementations and led us to develop the techniques presented hereafter. More details about subscription aggregation and XNet's routing protocol can be found in [7].

4 Subscription Management

Efficient subscription management is critical for the overall performance of the system and to guarantee short registration delays to consumers. As previously mentioned, the cost of subscription management mainly results from the extensive covering checks that have to be performed by the routers when a subscription is registered or canceled.

To determine whether a given tree-structured subscription—also called "tree pattern" henceforth—covers another subscription, we can use the algorithm proposed in [5], which has a time complexity of $O(|S_1||S_2|)$, where $|S_1|$ and $|S_2|$ are the number of nodes of the two subscriptions being compared.[1] Obviously, when an incoming subscription must be tested for covering against all the other subscriptions in the routing table, iterative execution of the algorithm is clearly inefficient. We have therefore designed a novel algorithm, termed XSearch, which efficiently identifies all the possible covering relationships between a given subscription and a possibly large set of subscriptions. This algorithm is described in the rest of this section. Additional details and proofs can be found in a companion technical report [8].

Problem Statement. Consider a tree pattern s and a set R of n tree patterns, $R = \{s_1, \cdots, s_n\}$, which we will refer to as the search set. Our algorithm runs in two different modes according to the relationships that we want to identify. *Covered mode* identifies the set $R_{\supseteq}$ of all the tree patterns in R that are covered by s. *Cover mode* identifies the set $R_{\subseteq}$ of all the tree patterns in R that cover s. We refer to XSearch$_{\supseteq}$ and XSearch$_{\subseteq}$ as the algorithm running in *covered* and *cover* mode, respectively.

Definitions and Notations. Let u be a node of a tree pattern s; we denote by *label*(u) the label of that node and by *child*(u) the set of the child nodes of u in s. Recall that the label of node u can either be a wildcard (*), an ancestor/descendant operator (//), or a tag name. We define a partial ordering $\preceq$ on node labels such that if x and x' are tag names, then (1) $x \preceq * \preceq //$ and (2) $x \preceq x'$ iff $x = x'$.

Algorithm 1 *add*(s, t, u)

```
if $\exists t' \in child(t)$ such that $label(t') = label(u)$ and $s \notin sub(t')$ then
    $sub(t') = sub(t') \cup s$
else
    create $t' \in child(t)$ such that $label(t') = label(u)$ and $sub(t') = \{s\}$
end if
for all $u' \in child(u)$ do
    $add(s, t', u')$
end for
```

Factorization Trees. Our XSearch algorithm does not operate directly on the set of tree patterns R, but on a "factorization tree" built from the set R and defined as follows. The factorization tree of R, denoted $T(R)$, is a tree where each node t has two attributes: a label *label*(t) similar to that of a node of a tree pattern, and a set of tree patterns *sub*(t), which is a subset of R. The root node r_T of $T(R)$ has no label and *sub*$(r_T) = R$. Initially, $T(R)$ consists of only its root node r_T. We incrementally add each tree pattern $s \in R$ to $T(R)$ with the recursive *add*(s, r_T, r_s) function shown in Algorithm 1, where r_s is the root node of tree pattern s. The removal of a tree pattern from $T(R)$ is performed in a similar manner using Algorithm 2. Note that, to keep the presentation simple, we omitted the special

[1] Note that the covering problem has been shown to be coNP-complete [11]. Our algorithm is sound but not complete, i.e., it may fail to detect some covering relationships in rare pathological cases, but all the relationships that it reports are correct. Consequently, a router may fail to aggregate some valid subscriptions, but correctness is never violated.

case of the root node of the factorization tree in the addition and removal algorithms.

Algorithm 2 *remove*(s, t)

```
for all $t' \in child(t)$ such that $s \in sub(t')$ do
    $sub(t') = sub(t') \setminus \{s\}$
    $remove(s, t')$
    if $sub(t') = \{\}$ then
        remove $t'$ from $child(t)$
    end if
end for
```

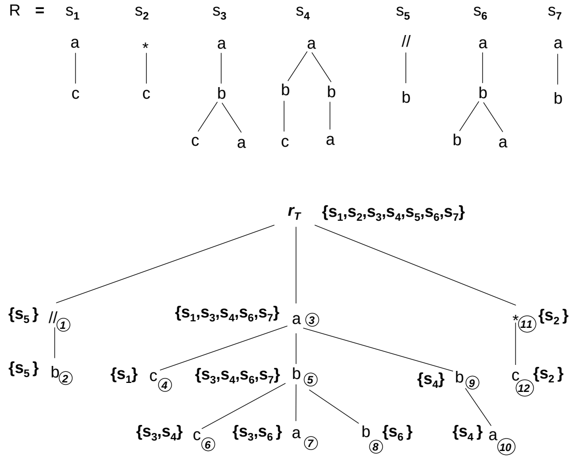


Figure 2: Six tree patterns and a corresponding factorization tree, where a node is represented by its label. Each node is associated with a set of tree patterns, shown between brackets.

Intuitively, a factorization tree enables us to remove the redundancies between the tree patterns in R by "factorizing" identical branches. Thus, $T(R)$ is a compact representation of the tree patterns in R. Figure 2 shows an example with six tree patterns and the corresponding factorization tree. It is important to note that the factorization tree is not unique; depending on the insertion order of the tree patterns, we can have distinct, equivalent trees. This does not affect the correctness of our XSEARCH algorithm, nor its performance.

The Search Algorithm. We first describe the XSEARCH algorithm in covered mode. Consider a subscription set R and a corresponding factorization tree, $T(R)$. Let s be a single tree pattern. The algorithm works recursively on the nodes of s. When executed with the root nodes of $T(R)$ and s, XSEARCH$_{\supseteq}(r_T, r_s)$ returns the set $R_{\supseteq}$ of all tree patterns that are covered by s.

The search process is described in pseudo-code in Algorithm 3. Intuitively, it tries to locate the paths in $T(R)$ that are covered by s; the tree patterns that the union of those paths represent are also covered by s (lines 6 and 8). The process is slightly more complex when encountering an ancestor/descendant operator (//), because we need to try to map it to paths of length 0 (line 11) or ≥ 1 (line 12).

To better illustrate the workings of the algorithm, consider the example runs shown in Figure 3. Two tree patterns, u and v, are matched against the factorization tree $T(R)$ of Figure 2 for clarity. The nodes of u, v, and $T(R)$ are numbered in the figures; we refer to node number i of u, v, and $T(R)$ by u_i, v_i, and t_i, respectively. The different steps of the algorithm are detailed in the two execution traces (the $\hookrightarrow$ symbol represents recursive invocations of the algorithm).

Algorithm 3 XSEARCH$_{\supseteq}(t, u)$

```
1:  if $t$ is a leaf then
2:      XSEARCH$_{\supseteq}(t, u) = \emptyset$
3:  else
4:      if $label(u) \neq$ "$//$" then
5:          if $u$ is a leaf then
6:              XSEARCH$_{\supseteq}(t, u) = \bigcup_{label(t') \preceq label(u)}^{t' \in child(t)} sub(t')$
7:          else
8:              XSEARCH$_{\supseteq}(t, u) = \bigcup_{label(t') \preceq label(u)}^{t' \in child(t)} \bigcap_{u' \in child(u)}$ XSEARCH$_{\supseteq}(t', u')$
9:          end if
10:     else
11:         $S_0 = \bigcap_{u' \in child(u)}$ XSEARCH$(t, u')$
12:         $S_{\geq 1} = \bigcup_{t' \in child(t)}$ XSEARCH$(t', u)$
13:         XSEARCH$_{\supseteq}(t, u) = S_0 \cup S_{\geq 1}$
14:     end if
15: end if
```

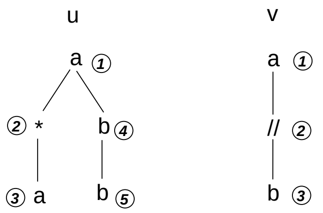
Algorithm 4 XSEARCH$_{\subseteq}(t, u)$

```
1:  if $u$ is a leaf then
2:      XSEARCH$_{\subseteq}(t, u) = sub(t)$
3:  else
4:      if $label(t) \neq$ "$//$" then
5:          if $\nexists u' \in child(u), label(u') \preceq label(t)$ then
6:              XSEARCH$_{\subseteq}(t, u) = sub(t)$
7:          else
8:              if $t$ is a leaf then
9:                  XSEARCH$_{\subseteq}(t, u) = \emptyset$
10:             else
11:                 XSEARCH$_{\subseteq}(t, u) =$
12:                 $\bigcap_{label(u') \preceq label(t)}^{u' \in child(u)} \bigcup_{t' \in child(t)}$ XSEARCH$_{\subseteq}(t', u')$
13:             end if
14:         end if
15:     else
16:         $S_0 = \bigcup_{t' \in child(t)}$ XSEARCH$(t', u)$
17:         $S_{\geq 1} = \bigcap_{u' \in child(u)}$ XSEARCH$(t, u')$
18:         XSEARCH$_{\subseteq}(t, u) = S_0 \cap S_{\geq 1}$
19:     end if
20: end if
```

Algorithm 5 XSEARCH$_{\subseteq}(r_T, r_s)$

```
XSEARCH$_{\subseteq}(r_T, r_s) = sub(t) \setminus \bigcup_{t' \in child(t)}$ XSEARCH$_{\subseteq}(t', r_s)$
```

The second algorithm, XSEARCH$_{\subseteq}$, is described in Algorithms 4 and 5 and works in a very similar manner. The major difference is that the algorithm works recursively on the nodes of $T(R)$, trying to find paths in s that are covered by the tree patterns in $T(R)$. The recursive function in Algorithm 4 returns the subscriptions that do *not* cover s. A subscription t covers s if each branch of s is covered by some branch of t (line 12). Subscriptions that have longer (line 2) or incompatible (line 6) paths cannot cover s, whereas shorter paths (line 9) are acceptable. Finally, when encountering an ancestor/descendant operator (//), we need to try to map it to paths of length 0 (line 16) or



$\textsc{XSearch}(r_T, u_1) = \textsc{XSearch}(t_3, u_2) \cap \textsc{XSearch}(t_3, u_4)$
$\hookrightarrow \textsc{XSearch}(t_3, u_2) = \textsc{XSearch}(t_4, u_3) \cup \textsc{XSearch}(t_5, u_3) \cup \textsc{XSearch}(t_9, u_3)$
$\quad\hookrightarrow \textsc{XSearch}(t_4, u_3) = \emptyset$
$\quad\hookrightarrow \textsc{XSearch}(t_5, u_3) = \{s_3, s_6\}$
$\quad\hookrightarrow \textsc{XSearch}(t_9, u_3) = \{s_4\}$
$\hookrightarrow \textsc{XSearch}(t_3, u_2) = \{s_3, s_6\} \cup \{s_4\} = \{s_3, s_4, s_6\}$
$\hookrightarrow \textsc{XSearch}(t_3, u_4) = \textsc{XSearch}(t_5, u_5) \cup \textsc{XSearch}(t_9, u_5)$
$\quad\hookrightarrow \textsc{XSearch}(t_5, u_5) = \{s_6\}$
$\quad\hookrightarrow \textsc{XSearch}(t_9, u_5) = \emptyset$
$\hookrightarrow \textsc{XSearch}(t_3, u_4) = \{s_6\}$

Finally: $\textsc{XSearch}(r_T, u_1) = \{s_3, s_4, s_6\} \cap \{s_6\} = \{s_6\}$

$\textsc{XSearch}(r_T, v_1) = \textsc{XSearch}(t_3, v_2)$
$\hookrightarrow \textsc{XSearch}(t_3, v_2) = S_0 \cup S_{\geq 1}$
$\hookrightarrow S_0 = \textsc{XSearch}(t_3, v_3)$
$\hookrightarrow S_{\geq 1} = \textsc{XSearch}(t_4, v_2) \cup \textsc{XSearch}(t_5, v_2) \cup \textsc{XSearch}(t_9, v_2)$
$\quad\hookrightarrow \textsc{XSearch}(t_3, v_3) = \textit{sub}(t_5) \cup \textit{sub}(t_9) = \{s_3, s_4, s_6, s_7\}$
$\quad\hookrightarrow \textsc{XSearch}(t_4, v_2) = \emptyset$
$\quad\hookrightarrow \textsc{XSearch}(t_5, v_2) = S'_0 \cup S'_{\geq 1}$
$\qquad\hookrightarrow S'_0 = \textsc{XSearch}(t_5, v_3)$
$\qquad\hookrightarrow S'_{\geq 1} = \textsc{XSearch}(t_6, v_2) \cup \textsc{XSearch}(t_7, v_2) \cup \textsc{XSearch}(t_8, v_2)$
$\qquad\quad\hookrightarrow \textsc{XSearch}(t_5, v_3) = \textit{sub}(t_6) = \{s_6\}$
$\qquad\quad\hookrightarrow \textsc{XSearch}(t_6, v_2) = \emptyset$
$\qquad\quad\hookrightarrow \textsc{XSearch}(t_7, v_2) = \emptyset$
$\qquad\quad\hookrightarrow \textsc{XSearch}(t_8, v_2) = \emptyset$
$\qquad\hookrightarrow S'_0 = \{s_6\}$
$\qquad\hookrightarrow S'_{\geq 1} = \emptyset$
$\quad\hookrightarrow \textsc{XSearch}(t_5, v_2) = \{s_6\}$
$\quad\hookrightarrow \textsc{XSearch}(t_9, v_2) = S''_0 \cup S''_{\geq 1}$
$\qquad\hookrightarrow S''_0 = \textsc{XSearch}(t_9, v_3)$
$\qquad\hookrightarrow S''_{\geq 1} = \textsc{XSearch}(t_{10}, v_2)$
$\qquad\quad\hookrightarrow \textsc{XSearch}(t_9, v_3) = \emptyset$
$\qquad\quad\hookrightarrow \textsc{XSearch}(t_{10}, v_2) = \emptyset$
$\qquad\hookrightarrow S''_0 = \emptyset$
$\qquad\hookrightarrow S''_{\geq 1} = \emptyset$
$\quad\hookrightarrow \textsc{XSearch}(t_9, v_2) = \emptyset$
$\hookrightarrow S_0 = \{s_3, s_4, s_6, s_7\}$
$\hookrightarrow S_{\geq 1} = \{s_6\}$
$\hookrightarrow \textsc{XSearch}(t_3, v_2) = \{s_3, s_4, s_6, s_7\} \cup \{s_6\} = \{s_3, s_4, s_6, s_7\}$

Finally: $\textsc{XSearch}(r_T, v_1) = \{s_3, s_4, s_6, s_7\}$

Figure 3: Two $\textsc{XSearch}_{\supseteq}$ example runs.

≥ 1 (line 17). Note that we implicitely introduce an artificial root node in the tree-structured subscriptions (denoted r_s for subscription s) in order to simplify the description of the algorithm. When called with the roots of the factorization tree and a subscription s, Algorithm 5 recusively searches for subscriptions that do not cover s and return the complement set with respect to R. Because of space limitations, correctness proofs are not given here (they can be found in [8]).

Both Algorithms 3 and 4 perform in $O(|T(R)| \cdot |s|)$ time, where $|T(R)|$ is the number of nodes in the factorization tree and $|s|$ that in the expression being tested. This quadratic time complexity is due to the fact that each node in $T(R)$ and s is checked at most once. As for the space complexity, the size of the factorization tree $T(R)$ grows linearly with the number of tree patterns in the search set R. However, by construction, the factorization tree typically requires much less space than would be needed to maintain the whole search set R, that is, $|T(R)| \ll \sum_{s_i \in R} |s_i|$ when $|R|$ grows to large values.

5 Performance evaluation

Experimental Setup. To test the effectiveness of our subscription management techniques, we have conducted simulations using real-life document types and large numbers of subscriptions. We have generated realistic subscription workloads using a custom XPath generator that takes a Document Type Descriptor (DTD) as input and creates a set of valid XPath expressions based on a set of parameters that control: the maximum height h of the tree patterns; the probabilities p_* and $p_{//}$ of having a wildcard (`*`) and ancestor/descendant (`//`) operators at a node of a tree pattern; the probability p_λ of having more than one child at a given node; and the skew θ of the Zipf distribution used for selecting element tag names. For our experiments, we generated sets of tree patterns of various sizes, with $h = 10$, $p_* = 0.1$, $p_{//} = 0.05$, $p_\lambda = 0.1$, and $\theta = 1$. We used the widely-used NITF (News Industry Text Format) DTD [9] as the input DTD of our XPath generator. All the algorithms were implemented in C++ and compiled using GNU C++ version 2.96. Experiments were conducted on 1.5 GHz Intel Pentium IV machines with 512 MB of main memory running Linux 2.4.18.

XSearch Efficiency. We evaluated the efficiency of the XSearch algorithm for search sets of different sizes. For this experiment, we considered search sets with unique subscriptions, that is, a given subscription does not appear more than once in a set. Indeed, in a given router, XSearch is used to determine the covering relationships between a given subscription and the subscriptions in the routing table, which are all unique.

For each search set, we generated $1,000$ additional subscriptions and, for each of them, we measured the time necessary to determine the subset of the subscriptions that cover, and are covered by, that subscription. For comparison purposes, we have also measured the efficiency of the XSearch algorithm against sequential execution of the containment algorithm of [5], which we call *Linear*.

Figure 4 shows the average search time of the XSearch algorithm. It appears clearly that $XSearch$ is extremely efficient. Even for very large search sets, we can expect an

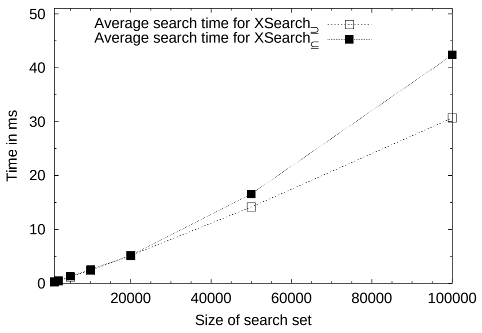


Figure 4: Average search time for the XSEARCH algorithm.

average search time of less than 50 ms. In comparison, the *Linear* algorithm yields to search times that are systematically more than two orders of magnitude higher. This result is not surprising, as the *Linear* algorithm needs to evaluate the entire subscription set R while $XSearch$ only searches through the factorization tree, which is much smaller by construction.

The second variant of the algorithm, $XSearch_\subseteq$, tends to be slightly less efficient than $XSearch_\supseteq$ for large consumer populations. We can explain this observation by the fact that, on average, the $XSearch_\subseteq$ algorithm necessitates deeper traversals of the factorization tree.

Size of search set	1,000	2,000	5,000	10,000
XSEARCH$_\supseteq$	0.23	0.45	1.17	2.41
XSEARCH$_\subseteq$	0.28	0.53	1.30	2.57

Table 1: Average search time of XSEARCH in ms.

One should note that, in practice, the sizes of the routing tables rarely exceed $1,000$ entries, even for very large consumer populations, thanks to subscriptions aggregation. For completeness, we show in Table 1 the absolute average search time of XSEARCH for search sets of small sizes, which are most relevant in the context of content-based routing.

$\|R\|$	1,000	2,000	5,000	10,000	20,000	50,000	100,000
$\sum_{s_i \in R} \|s_i\|$	7.6	15.8	42.1	88.1	183.3	481.8	998.6
$\|T(R)\|$	1.9	3.6	8.2	15.1	28.1	62.1	112.6

Table 2: Space requirements for a given subscription population R and its factorization tree $T(R)$, in thousands of nodes.

Space Efficiency. We have experimentally quantified the space requirements of the factorization tree with subscription sets of various sizes. The results in Table 2 confirm that the number of nodes in the factorization tree is indeed notably smaller than the sum of the nodes of the individual subscriptions.

6 Conclusion

We have described the subscription management techniques that we implemented in our XNET content routing network. These techniques rely on XSEARCH, an algorithm that determines the covering relationships between subscriptions, to efficiently process consumer registrations and cancellations. By capitalizing the performance of this algorithm, our content-based publish/subscribe system can maintain compact routing tables (for improved routing performance) while ensuring perfect routing (for bandwidth efficiency) at all time. Although described in the context of content-based routing and XPath, the XSEARCH algorithm can be readily used with similar subscription languages or to address different data management problems.

References

[1] R. A. Baeza-Yates and G. H. Gonnet. Fast text searching for regular expressions or automaton searching on tries. *Journal of the ACM*, 43(6):915–936, 1996.

[2] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, and D. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *Proceedings of ICDCS*, May 1999.

[3] A. Carzaniga, D. Rosenblum, and A. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.

[4] A. Carzaniga, M. J. Rutherford, and A. L. Wolf. A routing scheme for content-based networking. In *Proceedings of IEEE INFOCOM*, Mar. 2004.

[5] C.-Y. Chan, W. Fan, P. Felber, M. Garofalakis, and R. Rastogi. Tree Pattern Aggregation for Scalable XML Data Dissemination. In *Proceedings of VLDB*, Aug. 2002.

[6] C.-Y. Chan, M. Garofalakis, and R. Rastogi. Re-tree: an efficient index structure for regular expressions. *The VLDB Journal*, 12(2):102–119, 2003.

[7] R. Chand and P. Felber. A scalable protocol for content-based routing in overlay networks. In *Proceedings of NCA*, Cambridge, MA, Apr. 2003.

[8] R. Chand and P. Felber. XNet: An XML Content Routing Network. Technical report, Institut EURECOM, 2004.

[9] I. P. T. Council. News Industry Text Format.

[10] G. Cugola, E. D. Nitto, and A. Fugetta. The JEDI event-based infrastructure and its application to the development of the opss wfms. *IEEE Transactions on Software Engineering*, 27(9):827–850, Sept. 2001.

[11] G. Miklau and D. Suciu. Containment and equivalence for an xpath fragment. In *Proceedings of PODS*, Madison, WI, June 2002.

[12] A. Tozawa and M. Hagiya. XML schema containment checking based on semi-implicit techniques. In *Proceedings of the Conference on Implementation and Application of Automata (CIAA)*, July 2003.

[13] W3C. XML Path Language (XPath) 1.0, Nov. 1999.

[14] S. Wu and U. Manber. Fast text searching: allowing errors. *Communications of the ACM*, 35(10):83–91, 1992.

HOMED: A Peer-to-Peer Overlay Architecture for Large-Scale Content-based Publish/Subscribe Systems

Yongjin Choi, Keuntae Park and Daeyeon Park
Department of Electrical Engineering & Computer Science
Korea Advanced Institute of Science and Technology(KAIST)
Yusung-Gu, Taejon, Korea
{yjchoi, ktpark}@sslab.kaist.ac.kr, daeyeon@ee.kaist.ac.kr

Abstract

Content-based publish/subscrbe systems provide an useful alternative to traditional address-based communication due to their ability to decouple communication between participants. It has remained a challenge to design a scalable overlay supporting the complexity of content-based networks, while satisfying the desirable properties large distributed systems should have. This paper presents a new peer-to-peer overlay called HOMED for distributed publish/subscribe systems. It can construct a flexible and efficient event dissemination tree by organizing participants based on their interest. The delivery depth of an event as well as subscribing/unsubscribing overhead scales logarithmically with the number of participating nodes.

1. Introduction

Publish/subscribe has become increasingly popular in building a large-scale distributed systems. While traditional synchronous request/replay communication needs an explicit IP address of the destination, publish/subscribe systems deliver a message to all, and only those, interested clients based on its content and their subscription set. Thus, publishers do not need to be aware of the set of receivers that varies according to a given message. Example applications that can benefit from this loosely coupled nature of publish/subscribe are news distribution, e-Business (eg. auction), multiplayer online games, system monitoring, and location-based service for mobile devices.

To use a publish/subscribe communication service, receivers register subscriptions that represent or summarize their interests. In turn, they will be notified of an event that matches their interests. Based on the selective power of the subscription language, publish/subscribe can be classified [3] as :

- Topic-based. Topics or subjects are used to bundle peers with methods to classify event content. Participants publish events and subscribe to individual topics selected from a predefined set.
- Content-based. A subscription can specify any predicate (or filter) over the entire content of the publication. In distributed content-based systems, the subscription is flooded to every possible publishers. A published event is forwarded if a neighboring node has a matching predicate. Thus, content-routing is the process of finding an event dissemination tree.

Although a topic-based system can be implemented very efficiently, many of modern applications require a content-based publish/subscribe with high expressiveness. However, as mentioned in [3], scalability and expressiveness are conflicting goals that must be traded-off.

Most content-based systems employ an overlay network of event brokers, which support rich subscription languages (eg. SIENA [3], Gryphon [1]). However, they commonly have the two drawbacks as follows. First, state of the art systems have static overlay networks consisted of reliable brokers under the administrative control, or assume that a spanning tree of entire brokers is known beforehand. Clearly, this is not feasible when a system involves an enormous number of brokers, which join and leave the overlay network dynamically. In extreme case applications, publish/subscribe systems may be organized only by client nodes without any specialized brokers. Second, a broker keeps a large amount of routing state and its control message overhead is huge. This is because every broker can be an intermediate router on the paths of an event dissemination tree. Although the covering relation between subscriptions can reduce this overhead by aggregating them, an unsubscription may have to forward more specific subscriptions covered by it. Hence, the total control traffic effectively has a flooding overhead.

Recently, content-based systems based on peer-to-peer (P2P) networks are proposed [5] [12] [9] [2] to solve these problems. P2P networks (eg. Chord [11], Pastry [10]) address the desirable properties that distributed systems should satisfy. They employ the event broker whose nodeId is the hash of an event type (or a topic name) as the *rendezvous node* (RN), or use P2P routing substrate for the event dissemination tree. However, the features they exploit from P2P systems are limited to self-organization, fault-tolerance, and guaranteed routing depth. Still, their routing state and control message overhead are enormous, since every broker should maintain the predicates of the subscribers whose P2P multicast paths traverse it. In other words, a node may have to participate in routing the events even though it has no interest in them. Due to the same reason, the routing depth for notifications is unnecessarily long. In conclusion, a distributed publish/subscribe needs a structured overlay network, but it must be designed with great care since the underlying peer-to-peer architecture has a significant effect on the performance.

In this paper, we propose a new peer-to-peer overlay, called HOMED, suitable for large-scale publish/subscribes. The design guidelines are (i) a mesh like structure rather than a tree is preferred for a reliable and adaptive event dissemination tree, (ii) a node neighbors with the nodes whose interests are similar to its interest in the overlay network in order that only interested nodes participate in disseminating the event. To ease construction and routing, HOMED organizes the overlay network based on the interest digest of each node rather than the complex selection predicate. HOMED can be used not only for flexible topic or type based systems by nature, but also as a routing substrate for highly selective content-based systems. In HOMED, an event is delivered along the path of a binomial tree. Also, the subscribe/unsubscribe overhead is limited to $O(\log N)$.

The rest of this paper is structured as follows. In what follows, we describe our basic design and the operations of a HOMED in Section 2. Section 3 presents some implementation issues and extensions. We summarize our contributions and discuss future plans in Section 4.

2. Design

HOMED (Hypercube Overlay Mesh for Event Dissemination) is a scalable overlay network for large-scale publish/subscribes. HOMED design centers around a virtual *hypercube*, which is a generalization of a three-dimensional cube into d dimensions. Each node in the HOMED has a d bits identifer based on its subscription predicate. Therefore, a node is mapped onto a vertex in the logical hypercube. An event is propagated to interested receivers along an embedded multicast tree in the hypercube. Due to this awareness of interests, HOMED can find the “best” event dissemination tree with minimum overhead.

First, we describe our HOMED in its basic form. In Section 3, we present several extensions that improve the performance and flexibility.

2.1. ID assignment

Every node joins a HOMED with its predicates specifying the events of interest. A HOMED node can be a broker serving many clients or a client itself in brokerless systems. By some ID generating function, a set of predicates (or a predicate) is transformed to a d bit ID (Interest Digest) .The ID is used as the basis of HOMED topology. Although the ID function has some effect on the performance, ID is just a guide-rule to construct an event dissemination tree and support a content-based service.

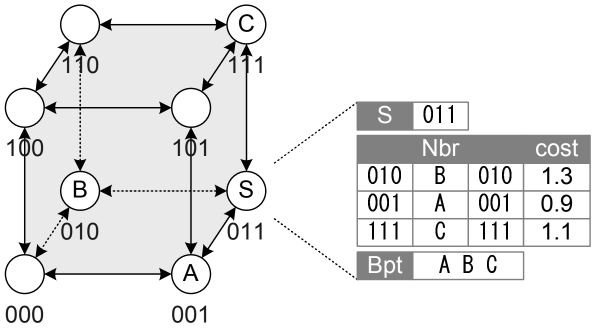
The unique requirement of the ID function is that its resulting IDs preserve a bitwise covering relation. More precisely, if a predicate α is covered by another predicate β, its digest ID$_\beta$ must subsume all 1s of ID$_\alpha$. A simple ID function is that Step 1) split d bits into as many segments as the number of attributes Step 2) divide the range of the attribute values by the number of bits in the segment , and Step 3) set individual bit to 1 if the predicate contains the value the bit represents. For an attribute that specify constrains as a range of values, it is necessary to divide the range into a finite number of intervals and name each.

Bloom-filter based approaches [13] [8] can be interesting alternatives since the bloom-filter also satisfies the above requirement. Subscription partitioning methodology in [14] is also a useful basis of the ID function. In addition, it is often desirable to make an ID based on only a subset of attributes in many applications. However, we will not elaborate on the technical issues here since designing a good ID function depends on the application domain and is beyond the scope of this paper.

2.2. Event Dissemination

To maintain the virtual hypercube topology, every HOMED node has a routing table called *ID cover table* as shown in Figure 1. A HOMED node has an *ID cover* that indicates the set of IDs the node must cover. If all vertices of the hypercube have the corresponding HOMED nodes, the ID cover of each node is the same as its ID. However, as the number of the nodes is much smaller than the number of vertices in the hypercube typically, each node is responsible for larger ID space than its ID.

The ID cover table has d entries, each of them corresponds to an ID whose hamming distance is 1 (i.e. ideal hamming neighbors of the node). Each entry has the physical address and ID cover of the neighbor that covers the corresponding ID. The final column is a scalar metric, such



S	011		
Nbr			cost
010	B	010	1.3
001	A	001	0.9
111	C	111	1.1
Bpt	A B C		

Figure 1. ID Cover Table.

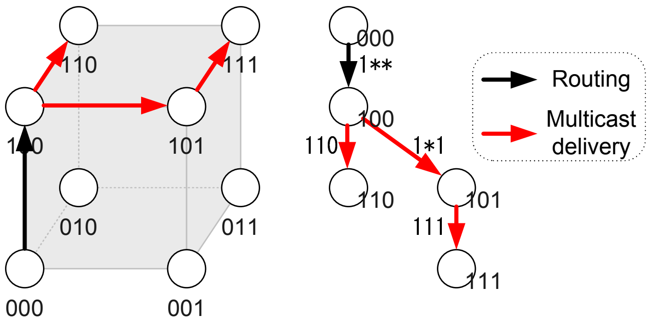


Figure 2. An example of event dissemination.

as the network "distance", which is used for constructing an efficient event dissemination tree. The bottom entry is the list of backpointers that reference the node. When the ID cover of a node changes, the nodes in that list are informed to update or change their neighbors. This will be explained minutely in Section 2.3.

A published event has an eID generated by the ID function. The eID represents the ID space where the event should be delivered. Hence, the dissemination is the process of matching the ID space of the event with nodes' ID. This consists of the two phases: the first phase is *routing* to find some node whose ID matches the eID and the second phase is *multicasting* from that node with an event dissemination tree.

Algorithm 2.1: ROUTE(eID)

```
ID ← publisher;
while ((d = dist(ID, eID)) $\neq$ 0){                    (i)
  //find the node whose hamming distance is shortest among neighbors
  for each (node in ID.Nbr){
    if (dist(node, eID) $<$ d){
      d = dist(node, eID);
      ID = node;
  }}
  forward to ID;
}
```

Algorithm 2.2: DISSEMINATION(ID, eID)

```
for each (node in ID.Nbr){
  if (dist(node, eID) = 0)
    insert node into $clist$;  //an ordered list w.r.t the cost metric
}

if ($clist \neq$ null){
  for each (node in $clist$){
    eID′ = split(ID, node, eID);                    (ii)
    DISSEMINATION(node, eID′);
}}
```

The routing procedure is shown in pseudo code form in Algorithm 2.1. **dist**(A, B) at (i) returns hamming distance between A and B. Wild cards in IDs are ignored in calculating the hamming distance since they mean "don't care". An event is forwarded to the node whose ID is closest in hamming distance among neighbors.

Once reached a matching node, the event is multicast with the tree constructed as shown in Algorithm 2.2. **split**(A, B, eID) at (ii) divides the ID space of eID at the first different bit between A and B, and returns each of resulting split IDspaces to A and B. A node that receives the event forwards it only to the neighbors whose ID is in eID. At the same time, the node splits eID space and assign each of the results to the neighbors. The neighbors are responsible for the delivery of the event to the nodes in their split eID. The event is delivered recursively until all interested nodes receive it. A node engaged in the split process earlier gets more portion of eID since **split()** divides the ID space binomially. In HOMED, a node with better cost metric is given a higher priority in the split process among interested neighbors. Therefore the resulting tree is efficient with respect to the metric in the ID cover table.

Figure 2 shows an example of event dissemination in the 3-dimensional HOMED. An event occurs at the node 000 and the eID of the event is (1**). The event is routed to the first matching node 100 and then multicast. ID spaces on arrows show that eID is split as the event moves toward the leaves of the multicast tree. In this example, the node 101 is assigned the responsibility for delivering the event to 111 since the node 101 has a better cost metric than 110.

It is notable that no node is engaged in the multicast of the event that it is not interested in, which means that the event is disseminated very efficiently with minimum messages. While not explicitly proven here, suppose that IDs of every nodes are different, the expected number of event notification steps is $O(\log N)$ rather than d, where N is the number of HOMED nodes in the network.

2.3. Subscribing

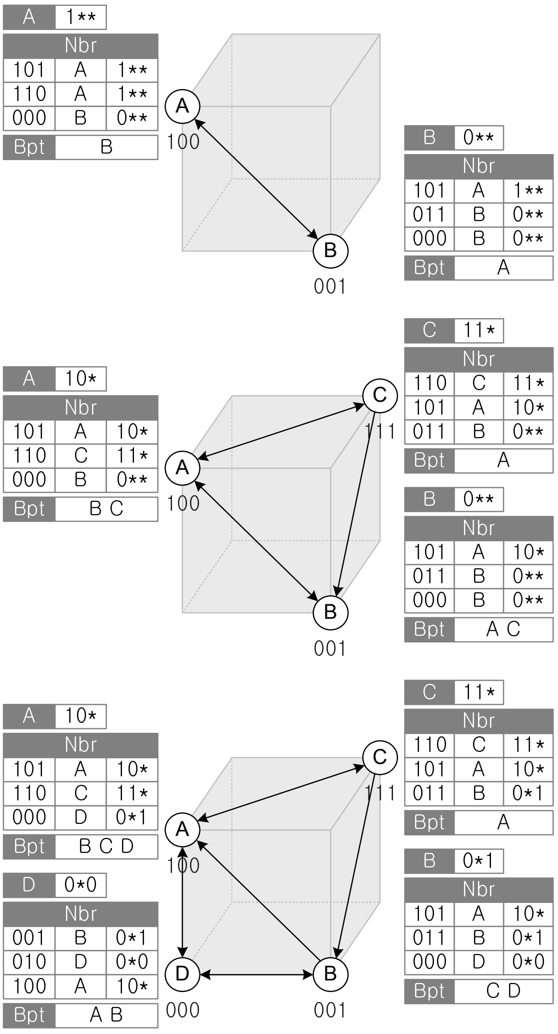
When a new subscriber enters the publish/subscribe network, it contacts any well-known node with subscription information. From that node, it traces ID cover table to find its position in the network. This participation process is called *subscribing*. During the subscribing, the node gets direct neighbor information and its portion of ID cover so that it can complete its ID cover table.

Algorithm 2.3: SUBSCRIBE(ID)

```
ID.Nbr = ID.Bpt = $\emptyset$;   ID.Cvr = $\{$ID$\}$;
$n \leftarrow$ a well-known node;

$//$step1: find the position
$n \leftarrow$ ROUTE(ID);

$//$step2: split ID cover and make the ID table of the joining node
ID.Cvr $\leftarrow$ split($n$, ID, $n$.Cvr);
$L \leftarrow \{$x$|$dist(x, ID) = 1$\}$;   $//$hamming neighbors ($hn$)
for each ($hn$ in $L$)$\{$
  for each (node in $n$.Nbr $\cup$ $n$.Bpt)$\{$
    if ($hn \in$ node.Cvr)$\{$
      ID.Nbr $\leftarrow$ ($hn$, node, node.Cvr);
      node.Bpt $=$ node.Bpt $\cup$ ID;
  $\}\}$
  if ($hn \in$ ID.Cvr)
    ID.Nbr $\leftarrow$ ($hn$, ID, ID.Cvr);
$\}$

$//$step3: update neighbor's table
$L \leftarrow \{n\} \cup n$.Bpt;
for each ($bpn$ in $L$)$\{$
  for each ($hn$ in $bpn$.Nbr)$\{$
    if ($hn \in$ ID.Cvr)$\{$
      replace the $hn$ entry with ($hn$, ID, ID.Cvr);
      move $bpn$ from $n$.Bpt to ID.Bpt;
    $\}$ else if($hn$ = $n$)
      $hn$.Cvr $\leftarrow$ $n$.Cvr;
$\}\}$
```

Algorithm 2.3 shows the sequence of subscribing process with pseudo code. It is organized as three steps. At the first step, a subscriber looks for the node that covers the its ID. This is accomplished simply by call **ROUTE** with its ID. The second step is to determine the ID cover of the subscriber. The subscriber gets its ID cover portion by **split**() function. And, the subscriber makes the neighbor list from neighbors and backpointers of the node that previously covered its ID. The split of ID cover incurs the split of backpointer list. As a final step, nodes that point the ID cover which is split from the original node change the neighbor list to point new owner of the ID cover. New and old owners also update their backpointer lists to reflect those nodes.

For the comprehension, we provide a step-by-step example in Figure 3, where the ID size is 3 bits.



A	1**	
Nbr		
101	A	1**
110	A	1**
000	B	0**
Bpt	B	

B	0**	
Nbr		
101	A	1**
011	B	0**
000	B	0**
Bpt	A	

A	10*	
Nbr		
101	A	10*
110	C	11*
000	B	0**
Bpt	B C	

C	11*	
Nbr		
110	C	11*
101	A	10*
011	B	0**
Bpt	A	

B	0**	
Nbr		
101	A	10*
011	B	0**
000	B	0**
Bpt	A C	

A	10*	
Nbr		
101	A	10*
110	C	11*
000	D	0*1
Bpt	B C D	

C	11*	
Nbr		
110	C	11*
101	A	10*
011	B	0*1
Bpt	A	

D	0*0	
Nbr		
001	B	0*1
010	D	0*0
100	A	10*
Bpt	A B	

B	0*1	
Nbr		
101	A	10*
011	B	0*1
000	D	0*0
Bpt	C D	

Figure 3. An example of subscribing.

In *subscribing*, number of messages to set up ID cover table is $\log N + 2d$ where N is the total number of subscribers and d is the number of bits in ID. A new subscriber sends just one message to find the node that covers its ID and the message moves $\log N$ hops on average. It also contacts all the neighbor nodes and backpointing nodes of the original owner of the ID cover in order to fill the neighbor list. The number of messages are $2d$ as total number of neighbor nodes and backpointing nodes in the network are the same and the average number per node is d.

2.4. Unsubscribing

Subscribers can leave the subscribe network at anytime, which is called *unsubscribing*. To maintain the HOMED network, other subscribers must cover the ID cover of the leaving node and update their neighbor information. Subscribers that want to leave the network send notification to any one neighbor then it inherits the ID cover of the leaving node.

Each node knows the ID cover table of neighbor nodes that is piggybacked in the periodic heart beat message. The node that succeeds to the ID cover of the leaving node sends notification of ID cover change to all the backpointing nodes of itself and the unsubscribing node. It also informs the neighbors of the leaving node that backpointers to the node are no longer needed. Unsubscribing operation is expressed by pseudo code in Algorithm 2.4.

Algorithm 2.4: UNSUBSCRIBE(leaveID)

```
ID.Cvr = ID.Cvr $\cup$ leaveID.Cvr;
//step1: redirect references from leaveID to ID
for each ($bpn$ in leaveID.Bpt){
  for each ($hn$ in $bpn$.Nbr){
    if ($hn \in$ ID.Cvr){
      replace the $hn$ entry with ($hn$, ID, ID.Cvr);
}}}

//step2: eliminate obsolete backpointers
for each (node in leaveID.Nbr)
  node.Bpt = node.Bpt $-$ {leaveID};

//step3: notify ID cover change
for each (node in ID.Bpt){
  for each (nb in node.Nbr){
    if (nb = ID)
      nb.Cvr $\leftarrow$ ID.Cvr;
}}
ID.Bpt = ID.Bpt $\cup$ leaveID.Bpt;
```

During *unsubscribing*, $3d$ messages are delivered by one hop transmission as the node knows where to send ID cover change notifications and obsolete backpointer notifications by hearing neighbor's ID cover table in heart beat message.

2.5. Node Departure

Node departure means a leaving of a node without notification. Reaction mechanism is basically the same as unsubscribing except that neighbors detect the node departure by periodic heart beat messages. The first node that perceives the node departure inherits the ID cover of the leaving node and executes unsubscribing process explained above.

HOMED's mesh like structure gives the benefit of localized recovery of the node departure; only neighbors of the leaving node are involved in the process. While previous approaches need to contact almost all nodes to eliminate obsolete subscription through the reverse path of the tree, HOMED can handle a the node departure with much smaller message and delay overhead.

3. Discussion

In this section, we discuss some design issues and possible improvements.

3.1. Handling duplicate IDs

In HOMED, an ID is not guaranteed to be unique in the network since IDs are determined by interests of nodes. So, several nodes can have the same ID. Although it is desirable to avoid this ID conflict as much as possible with the "good" ID function discriminating interests, HOMED must be able to handle duplicate IDs. One possible solution is that the nodes with the same ID construct an overlay multicast tree [6] [7] rooted at the first joining node or the most powerful one. Another solution is that an ID is composed of interest and identifier of a node. An event also has the corresponding part in its ID, which is normally set to (content ID, ***). In this case, publishers can select receivers among interested ones.

3.2. k-ary Hypercube

In the previous section, an ID is represented by a binary bit vector, and thus the resulting HOMED topology is built on a logical binary hypercube. However, each bit in an ID can be substituted with b bits for the HOMED network that requires a large ID space. [1] The routing table of HOMED is organized with base 2^b in much the same way as that of Pastry [10]. We have the two choices of how many entries a node must have among k $(= 2^b)$ possible values.

1. For each row, a node has two neighbors whose ID segment is numerically closest among smaller and bigger ones. The delivery depth is increased to $O\left(2^b \log_{2^b} N\right)$.
2. A node's routing table has $2^b - 1$ entries for each row as in Pastry. While the routing table size becomes $O\left(2^b \log_{2^b} N\right)$, the deliver depth is $O\left(\log_{2^b} N\right)$.

Therefore, there is a trade-off between the size of routing tables and the depth of event delivery trees.

3.3. Supporting Content-Based Systems

HOMED may suffer from the same problem that topic-based systems have because it is based on the simple ID. Let us assume that a node has an ID segment 0110, which represents an integer attribute $0 < x < 100$, but the actual interest of the node is $30 < x < 80$. Then, it will receive

[1] This is similar to that binary hypercubes is generalized to k-ary n-cubes.

unnecessary messages when they are out of the range of its interest. To reduce those false positives, the exact predicate must be known to all other nodes as existing content-based systems do. This will result in unacceptable overhead. However, HOMED can localize the propagation of exact predicates. For the previous example, the node with ID (0110...) sends its predicate only to the nodes that have the same ID segment 0110, since one of them will deliver matching events.

Content-based networking in HOMED has two advantages. First, a predicate is propagated only to a small number of nodes by a subset of subscriber's neighbors, and thus the number of hops in propagation is limited. Second, predicates do not have to be known in time since they are used only for reducing false positives. In HOMED, predicates are piggybacked on heartbeat messages. Based on the aggregated predicates, each node in the event dissemination tree decides matching descendants using algorithm of [4].

4. Conclusion

In this paper, we have presented a new peer-to-peer overlay called HOMED suitable for large-scale publish/subscribe systems. HOMED can construct a flexible and efficient event dissemination tree with minimum participation of the nodes that have no interest in the event. At the same time, it has the desirable properties such as limited delivery depth and overhead.

Evaluation of publish/subscribe systems is a hard work since they are affected by a lot of components including the probabilistic model of publishers and subscribers. Measuring the performance of HOMED is now in progress.

It is another challenge to exploit physical locality in content-based publish/subscribe. While HOMED already has the scheme to assign different responsibilities to the descendants of a tree, a network-wide optimization is possible using proximity estimation techniques such as IDMaps and GNP. To couple HOMED with them is under study.

References

[1] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, and D. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. *The 19th IEEE International Conference on Distributed Computing Systems (ICDCS '99)*, pages 262–272, May 1999.

[2] A. R. Bharambe, S. Rao, and S. Seshan. Mercury: A scalable publish-subscribe system for internet games. In *Proceedings of the 1st Workshop on Network and System Support for Games (Netgames)*, Braunschweig, Germany, Apr. 2002.

[3] A. Carzaniga and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, Aug. 2001.

[4] A. Carzaniga and A. L. Wolf. Forwarding in a content-based network. In *Proceedings of the 2003 ACM SIGCOMM Conference*, Karlsruhe, Germany, Aug 2003.

[5] M. Castro, P. Druschel, A. M. Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralized publish-subscribe infrastructure. In *Proceedings of the 3rd International Workshop on Networked Group Communication (NGC'01)*, volume 2233, pages 30–43, Nov. 2001.

[6] Y.-H. Chu, S. G. Rao, and H. Zhang. A case for end system multicast. In *Proceedings of ACM SIGMETRICS*, Jun. 2000.

[7] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole. Overcast: Reliable multicasting with an overlay network. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, Oct. 2000.

[8] G. Koloniari and E. Pitoura. Bloom-based filters for hierarchical data. In *the 5th Workshop on Distributed Data and Structures (WDAS)*, Jun. 2003.

[9] P. R. Pietzuch and J. Bacon. Peer-to-peer overlay broker networks in an event-based middleware. *The 2nd International Workshop on Distributed Event-Based Systems (DEBS'03)*, Jun. 2003.

[10] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Hiedelberg, Germany, Nov. 2001.

[11] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, San Diego, California, USA, 2001.

[12] W. W. Terpstra, S. Behnel, L. Fiege, A. Zeidler, and A. P. Buchmann. A peer-to-peer approach to content-based publish/subscribe. *The 2nd International Workshop on Distributed Event-Based Systems (DEBS'03)*, Jun. 2003.

[13] P. Triantafillou and A. Economides. Subscription summaries for scalability and efficiency in publish/subscribe systems. *The 2nd International Workshop on Distributed Event-Based Systems (DEBS'03)*, Jun. 2003.

[14] Y.-M. Wang, L. Qiu, D. Achlioptas, G. Das, P. Larson, and H. J. Wang. Subscription partitioning and routing in content-based publish/subscribe systems. In *16th International Symposium on DIStributed Computing*, 2002.

Dealing with Heterogeneous Data in Pub/Sub Systems: The Concept-Based Approach

M. Cilia[*], M. Antollini[†], C. Bornhövd[‡], A. Buchmann

Databases and Distributed Systems, Dept. of Computer Science,
Technische Universität Darmstadt, Darmstadt, Germany
E-mail: $< lastname >$@informatik.tu-darmstadt.de

Abstract

The event-based approach is well suited for integrating autonomous components or applications into complex systems by means of exchanging events. Event-based systems need an event dissemination mechanism to deliver relevant events to interested consumers. The publish/subscribe interaction paradigm has been gaining relevance in this context. One of the main characteristics of publish/subscribe systems is that they decouple producers and consumers so that they can remain unknown to each other. Therefore, the consideration of data heterogeneity issues is fundamental. However, most pub/sub notification services do not support the interaction among heterogeneous event producers and consumers.

In this paper we describe the concept-based approach *as a high-level dissemination mechanism for distributed and heterogeneous event-based applications. It enhances the notification service by enabling it to pass semantic information across component, institutional or cultural boundaries. It provides a high level abstraction for a meaningful exchange of data within the pub/sub interaction paradigm.*

1. Introduction

The event-based approach is well suited for integrating autonomous components or applications into complex systems by means of exchanging events. Because event-based systems do not require a-priori knowledge about the consumers of events they evolve and scale easily.

The exchanged events encapsulate data about a given happening of interest, which can only be properly interpreted and used when sufficient context information is known. In traditional systems, this context information is typically known by users and developers and left implicit. It is normally lost when data and events are exchanged across component or institutional boundaries. To process events in a semantically meaningful way, explicit information about the semantics of events and data is required.

Event-based systems need an event dissemination mechanism (or notification service) to deliver relevant events to interested consumers. The publish/subscribe interaction paradigm has been gaining relevance for this purpose. It basically consists of a set of clients that asynchronously exchange events, decoupled by a notification service that is interposed between them. Clients can be characterized as producers or consumers. Producers publish notifications[1], and consumers subscribe to notifications by issuing subscriptions, which essentially are stateless message filters. Consumers can have multiple active subscriptions, and after a client has issued a subscription the notification service delivers all future matching notifications that are published by any producer until the client cancels the corresponding subscription.

Since publish/subscribe mechanisms decouple producers and consumers, they should share a common understanding in order to express their mutual interests. In other words, events must be understandable beyond the closed confines of a single component or application. That includes applications that interact across traditional borders regardless of economic, cultural or linguistic differences (e.g. in its simplest form system of units, currency or date/time format). Because the source of an event cannot anticipate who is interested in a given event and where and when it must be delivered, a higher-level publish/subscribe infrastructure is needed.

To the best of our knowledge (see Section 2 for details), existing publish/subscribe mechanisms only expose the *data structure* of events but not the explicit semantics. This reflects a low level support for event consumers that

[*] Also Faculty of Sciences, UNICEN, Tandil, Argentina

[†] Faculty of Sciences, UNICEN, Tandil, Argentina

[‡] SAP Corporate Research, Palo Alto. Christof.Bornhoevd@sap.com

[1] In the context of this paper the terms notification, message and event are used interchangeable to mean the same thing.

based on this scarce information must express their interest without an explicit description of the intended meaning of events, i.e., the implicit assumptions made by event/data producers. Without this kind of information event producers and consumers are expected to comply with implicit assumptions made by participating software components or applications. Even in the case of a very small set of applications within an enterprise this approach is questionable.

For publish/subscribe mechanisms to be effective in open environments they must

- provide for the usage of a common vocabulary for defining interests, and
- support the correct interpretation of information independently of its origin and place of consumption.

We proposed in [2] the use of the *concept-based approach* to support meta-auctions. We have generalized concept-based pub/sub into a high level dissemination mechanism for distributed and heterogeneous event-based applications. Its goal is to provide a higher level of abstraction to describe the interests of event producers and consumers. This is achieved by supporting from the ground up ontologies which provide the base for correct data and event interpretation. Rather than requiring every producer or consumer application to use the same homogeneous namespace (as is common in other pub/sub systems) we provide metadata and conversion functions to map from one context to another. This last feature allows event consumers to simply specify the context to which events need to be converted before they are delivered for client processing.

The rest of this paper is organized as follows. In section 2 related work is presented. Section 3 analyzes and characterizes the support of pub/sub systems for heterogeneous data exchange. The concept-based approach is described in Section 4 including the main characteristics of the data model used and the role adapters play in this context. Finally, conclusions and ongoing work are presented in Section 5.

2. Related Work

In recent years, academia and industry have concentrated on publish/subscribe mechanisms because they allow loosely coupled exchange of asynchronous notifications, facilitating extensibility and flexibility. The channel model has evolved to a more flexible subscription mechanism, known as subject-based addressing, where a subject is attached to each notification [15]. Subject-based addressing features a set of rules that define a uniform and static name space for messages and their destinations. This approach is inflexible if changes to the subject organization are required, implying fixes in all participating applications.

To improve expressiveness of the subscription model the content-based approach was proposed where predicates on the content of a notification can be used for subscriptions. This approach is more flexible but requires a more complex infrastructure [4, 14]. Many projects in this category concentrate on scalability issues in wide-area networks and on efficient algorithms and techniques for matching and routing notifications to reduce network traffic [16, 12, 9, 18]. Most of these approaches use simple boolean expressions as subscription patterns since more powerful expressions cannot be efficiently treated.

In [17] some of the problems mentioned in the introduction are pointed out. The proposed solution consists of the application of two steps: 1) relationship “resolution” (e.g. synonyms and generalization/specialization) and 2) application of mapping functions (if necessary). The first step is aided by a domain-specific ontology so that subscriptions issued by subscribers are then re-written by the centralized notification service according to the default vocabulary. Notice that subscribers are not aware of synonym resolution. The second step involves functions in the sense of the resolution of relationships which otherwise cannot be specified using the relationship solver (e.g. derivation of data).

This approach could work well when human beings are the consumers of notifications but not if subscriber applications act as consumers. This problem arises because when matching notifications are delivered to subscribers they would not expect some fields (specifically those renamed by the synonym solver or resolved via data derivation) in the message content.

As mentioned before, only the data structure of events is exposed to all participants. This puts in evidence the low level support regarding data exchange which directly impacts the broad usability of such an important piece of the communication infrastructure. The resolution of synonyms is a relevant issue but alone does not solve the problem of applications that publish and consume events messages in heterogeneous contexts. Even when a common vocabulary is used the data interpretation problem is still present.

3. Heterogeneous Interaction Panorama

In a pub/sub environment, where new participants can join and leave dynamically, where producers and consumers are unknown to each other and thus fomenting loosely-coupling interactions, the consideration of heterogeneity issues is fundamental. However, most pub/sub notification services do not support the interaction among heterogeneous event producers and consumers.

When trying to tackle the problem of data/event heterogeneity, two main issues need to be considered. The first one concentrates on the vocabulary shared by all participants. The second one relates to the contextual information of applications that produce and consume events. In the following subsections these issues are treated in more detail.

3.1. Sharing a Common Vocabulary

Algorithms used to match published messages and subscriptions are (mostly) based on a string comparison. Therefore, these algorithms require the notifications coming from the publisher side as well as subscriptions to be in the same vocabulary. Because many applications can participate, a richer vocabulary is needed that includes all specific (sub)vocabularies used. Ontologies [10], for instance, are good candidates for this purpose because they support various relationships among terms (e.g. synonyms, specialization/generalization). But notice that within the ontology a set of terms must be identified as the terms used for matching purposes. From now on these terms are referred here as "matching terms" or "matching vocabulary".

Possible differences among participants at the vocabulary level are solved by a vocabulary agreement module. This is responsible for applying resolution strategies based on the navigation of ontology/vocabulary relationships to find the appropriate matching terms.

This module is always located between a publisher or a subscriber application and the pub/sub matching algorithm. From an abstract perspective, it is the delimitation border that distinguishes between the vocabularies used by participating applications and the matching vocabulary used within the pub/sub notification service. But remember that subscriber applications do only know their own vocabulary so that the vocabulary agreement module needs to maintain the applied resolution strategies for each subscription. This is required since matching notifications need to be transformed back from the matching vocabulary to the one known by the consumer application.

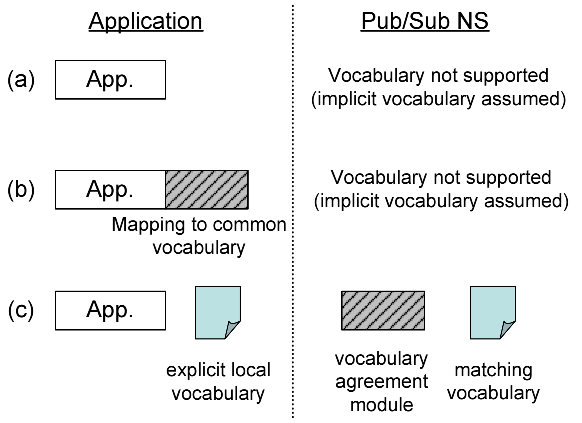

Application
Pub/Sub NS
(a)
App.
Vocabulary not supported
(implicit vocabulary assumed)
(b)
App.
Mapping to common vocabulary
Vocabulary not supported
(implicit vocabulary assumed)
(c)
App.
explicit local vocabulary
vocabulary agreement module
matching vocabulary

Figure 1. Common vocabulary treatment

Figure 1 shows three approaches how applications can deal with the vocabulary problem when interacting by means of a pub/sub notification service. Under the first approach (a) in the figure, applications implicitly assume per default a common vocabulary so that neither the pub/sub infrastructure nor applications explicitly care about this problem. Under the second approach (b), applications themselves internally resolve vocabulary issues while the pub/sub mechanism is unaware of them. The third approach (c) requires that applications can explicitly specify their vocabularies so that the pub/sub infrastructure uses this definition to feed the vocabulary agreement module. Notice that the notification service possesses also an explicit definition of the matching vocabulary that is used within the pub/sub boundaries for mapping purposes.

3.2. Considering the Context of Applications

Sharing a common vocabulary in pub/sub systems is a prerequisite but not the whole solution. The problem of dealing with heterogeneous data is still present even when using a common vocabulary. Relevant contextual information (i.e. the set of assumptions about data) is essential to allow the correct interpretation o the data in question. For instance, consider the common vocabulary terms DateOfIssue, FinalPrice, NetWeight, and Distance. Data associated to these terms require contextual information, like the date format, the currency and the units of measure, respectively. Human beings can often distinguish these cases based on the situation context while computers cannot.

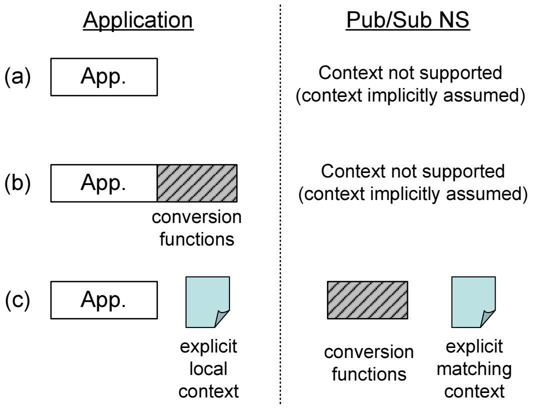


Figure 2. Consideration of application context

According to the handling of context, applications can be organized in three groups as depicted in Figure 2 (which by the way has a striking similarity to Figure 1). In the first group (a), participating applications do not consider contextual information of exchanged data and neither does the pub/sub notification service. Thus, an implicit context is assumed among all participants including the pub/sub infrastructure itself. The range of misinterpretation in this category is wide. In the second group (b), applications are context-aware and they explicitly convert data to and from a common implicit context (assumed by the pub/sub system) when publishing and subscribing respectively. In this case, conversion functions are scattered in many applications. If at least one data producer incorporates terms implying not yet known conversion functions, then all consumers need to be modified to incorporate them. Finally, in the last group (c), the context of applications and the context assumed by the pub/sub system (henceforth *matching context*) are explicitly specified. With this, transformations from and to the common context can be automatically performed. These transformations can now be applied under the responsibility of the pub/sub infrastructure.

4. The Concept-based Approach

Originally introduced in [2] for a specific application and with an initial implementation [5] on top of a commercial pub/sub product, the concept-based approach was proposed to tackle the problem of event-based interaction among heterogeneous applications. On one hand it concentrates on resolving data interpretation problems. On the other hand it provides a pub/sub abstraction that can run on top of (various) notification services independent of the addressing model used underneath.

Towards these objectives it allows subscribers to express their interest considering their local context thus delivering to them as a result *ready-to-process* data which does not require further conversions. It also supports the enrichment of messages produced by publishers by adding their corresponding contextual information.

Figure 3 sketches this approach. The underlying notification service is used as a data delivery mechanism where the concept-based layer is responsible for providing a higher-level interaction among heterogeneous applications. A *vocabulary/ontology manager* is associated with this layer with the purpose of handling domain-specific vocabularies used by the participant applications. Additionally, this layer is in charge of mapping concept-based data into the data structures of the delivery mechanism underneath. *Adapters* are the intermediaries between pub/sub clients and the concept-based layer. At the publisher side they are responsible for resolving vocabulary issues and for enriching message content. At the subscriber side they are in charge of transforming/converting message content according to the subscriber preferences.

In order to incorporate a solution to the problem of data heterogeneity (or data integration) into a pub/sub system several crucial building blocks are needed: i) a shared vocabulary which is the consolidation of vocabularies used by all participating applications, ii) a matching vocabulary which is a subset of the shared one used internally by the pub/sub system for matching purposes, iii) a vocabulary agreement module that maps from the shared to the matching vocabulary, iv) explicit definition of contextual information of all participants including the pub/sub system, and v) conversion functions to map from one context to another.

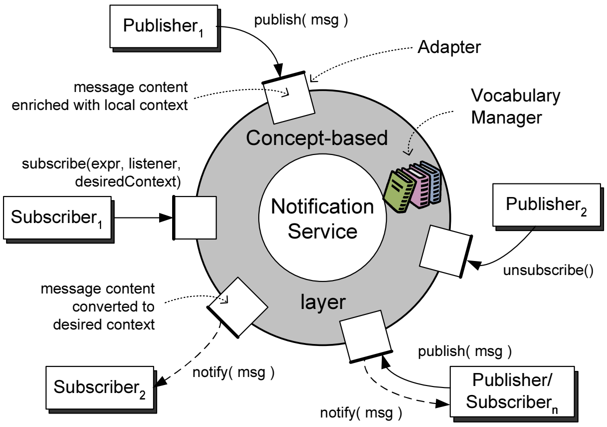


Figure 3. High-level view of the concept-based approach

From the previous section can be seen that context- and vocabulary-aware applications follow a similar pattern with respect to the infrastructure they need to support. The concept-based approach combines both into a model where vocabulary issues, the representation of context information and the conversion functions are under a uniform umbrella. Popular web standards such as RDF, DAML+OIL, and OWL provide adequate support for representing relationships among terms but do not support conversion function mechanisms as an integral part.

4.1. A Model for Data Exchange

To exchange data among loosely coupled systems, it must be taken into account that the structure and semantics of individual data items may vary, even if they describe objects of the same class of real world phenomena. Therefore, *context information* concerning the organization and meaning of data has to be given on an extensional level, i.e., on the level of data values. For this reason, we need description models that allow a flexible association of metadata with the available data items. In the following we describe MIX, the model used to represent event content.

Metadata based Integration model for data X-change [3, 1], or MIX for short, can be understood as a self-describing data model. This is because information about the structure and semantics of the data is not provided as a separately specified data schema, but is given as part of the available data itself. Thus, MIX allows a flexible association of context information in the form of metadata.

This model is based on the concept of a `SemanticOb-`

ject. It represents a data item together with its underlying SemanticContext which consists of a flexible set of meta-attributes that explicitly describe the implicit assumptions about the meaning of the data item. However, because we cannot explicitly describe all modelling assumptions the semantic context always has to be understood as a partial representation.

In addition, each semantic object has a concept *label* associated with it that specifies the relationship between the object and the real world aspects it describes. These labels have to be taken from a commonly known vocabulary, or Ontology. In the MIX model, an ontology is a finite set of concepts and their relationships.

It must be noticed that the SemanticContext is also specified by referring to the ontology/vocabulary. The association of context information with a given data value serves as an explicit specification of the implicit meaning of the data. This allows the determination of semantically equivalent semantic objects even if they are represented differently, i.e., relative to different contexts. For example

$<$*NetWeight, 1234,* { $<$*Unit, "kilogram(s)"* $>$} $>$ and

$<$*NetWeight, 2720.50,* { $<$*Unit,"pound(s)"* $>$} $>$

are semantically equivalent, because they represent the same information and we can specify a *conversion function* by which one representation can be transformed into the other. Such conversion functions are a prerequisite for the integration of semantic objects coming from different sources, by converting these objects, as far as possible, to a common context. In detail, two kinds of conversion functions can be distinguished. The first are those attached to the concept definition. The second are those defined in a function conversion manager (because, for instance, they may change over time, e.g. currency conversion).

This approach provides the central management of conversion functions and the flexibility to extend them if necessary. This avoids scattering conversion functions that in other cases need to be embedded in participating applications.

4.2. Adapters

Adapters are the contact/mediator of applications with the notification service. They maintain the different views of client applications by enriching the ontology with: i) the corresponding contextual information, and ii) the mapping from their local vocabularies to the matching one.

From the view point of client applications, the first task is the vocabulary agreement. Here, synonyms within the ontology are resolved. For consumer applications in particular, subscriptions are rewritten considering other relationships, like generalization/specialization. The applied resolution strategies need to be preserved since they are fundamental to map back the content of incoming notifications. Remember that consumer applications can only process messages expressed in their local vocabulary.

The vocabulary agreement module can also be statically defined if the kind of message content applications publish and/or subscribe to is stable. Adapters are not necessarily located at the client side. In this sense, configuration information is passed (at bootstrap- or connection-time) to the notification service for setting up vocabulary agreements for a particular client application and as a consequence the ontology is enhanced with this information.

The second task relates to the context information. In the case of data producers, adapters are responsible for enriching message content with meta information. This is done based on the explicit specification of clients about their contextual information. In this case, the ontology is enriched with it for the use of this particular application. Thus, this information is used when publishing events (by simply looking up in the ontology the required contextual information that needs to be added to this particular attribute of the message). When incoming notifications/messages arrive they are converted according to the context specified for each term as defined in the enriched ontology.

Adapters can also be configured to minimize the context information attached to each message. For this purpose, an addressable context can be defined so that participant applications explicitly state their contexts. Messages can now simply refer to such a context instead of putting all relevant context information as part of the payload of messages. This is referred to as *context sharing*. Conversion functions are aware of this feature and can resolve this kind of references.

4.3. Building on Top of Delivery Mechanisms

The concept-based approach provides an abstraction where participating applications do not care about details of the underlying delivery mechanism. In this sense, applications involved in such interaction can replace this mechanism by relying on the concept-based approach causing minimal changes on applications if any. This allows, for instance, the migration from a topic-based dissemination (à la JMS) to a content-based without major effort.

From the publisher's point of view, the message content (represented with MIX) is mapped into the corresponding data structures of the underlying dissemination mechanism. The concept-based software layer is responsible for dealing with the "features" that the underlying delivery mechanism does not support. For details about the mapping into different addressing models see [6] for subject-based addressing and [7] for content-based addressing.

In many cases during the mapping to the underlying data structures data is converted according to the matching context defined by the notification service. The Rebeca framework [13] was specialized in order to delay the conversion as much as possible by enabling message routers to automatically apply conversion functions on demand to appropriately evaluate expressions.

5. Conclusions and Ongoing Work

The concept-based approach enhances the scope of use of notification services by enabling it to cross component, institutional or cultural boundaries. It provides a higher-level of abstraction for a meaningful exchange of data within the pub/sub interaction paradigm. The concept-based software layer is responsible for resolving data heterogeneity problems by minimizing misinterpretation of exchanged data. This is achieved by integrating the use of ontologies, meta-data information and conversion functions provided by MIX. A notification service based on the concept-based approach delivers *ready-to-process* notification to subscriber applications so that no further data conversions are needed. This layer runs on top of various (commercial) notification services (such as, Sun's JMS reference implementation, JMS/JBoss, WebSphere MQ, OpenJMS, Rebeca, TIB/Rendezvous).

This approach was successfully used in research projects like Internet-enabled vehicles [8] and the meta-auction approach [2].

The MIX implementation used in this prototype is based on a pure java development while an OWL-based implementation is almost ready-to-use [11].

A centralized administration user interface for the concept-based layer is under development. Through this interface a common vocabulary can be defined, the ontology can be edited, new conversion functions can be defined, the matching context can be specified, and the mapping strategies into the underlying addressing model can be configured.

Currently, a framework for composite events is under development with the purpose of supporting event correlation in a variety of environments. Additionally and in order to encompass the OWL implementation of MIX we are experimenting with different XML routing strategies within the Rebeca framework.

References

[1] C. Bornhövd and A. Buchmann. Semantically Meaningful Data Exchange in Loosely Coupled Environments. In *Proc. Intl Conf on Information Systems Analysis and Synthesis (ISAS'00)*, July 2000.

[2] C. Bornhövd, M. Cilia, C. Liebig, and A. Buchmann. An Infrastructure for Meta-Auctions. In *Proceedings of WECWIS*, pages 21–30. IEEE Computer Society, June 2000.

[3] C. Bornhövd. *Semantic Metadata for the Integration of Heterogeneous Internet Data (in German)*. Ph.D. Thesis, Department of Computer Science, Darmstadt University of Technology, Germany, July 2000.

[4] A. Carzaniga, D. R. Rosenblum, and A. L. Wolf. Challenges for Distributed Event Services: Scalability vs. Expressiveness. In *Engineering Distributed Objects (EDO'99)*, Los Angeles, CA, May 1999.

[5] M. Cilia, C. Bornhövd, and A. Buchmann. Moving Active functionality from Centralized to Open Distributed Heterogeneous Environments. In *Proceedings of CoopIS*, volume 2172 of *LNCS*, pages 195–210, Trento, Italy, Sept. 2001.

[6] M. Cilia, C. Bornhövd, and A. Buchmann. CREAM: An Infrastructure for Distributed, Heterogeneous Event-based Applications. In *Proceedings of CoopIS*, volume 2172 of *LNCS*, Italy, Nov. 2003. Springer.

[7] M. Cilia, C. Bornhövd, and A. Buchmann. Event Handling for the Universal Enterprise (to appear). *Information Technology and Management (ITM), Special Issue on Universal Global Integration*, Jan. 2005.

[8] M. Cilia, P. Hasselmeyer, and A. Buchmann. Profiling and Internet Connectivity in Automotive Environments. (demo paper). In *Proceedings of VLDB*, pages 1071–1074, 2002.

[9] F. Fabret, F. Llirbat, J. Pereira, A. Jacobsen, K. Ross, and D. Shasha. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe. In *Proceedings of ACM SIGMOD*, pages 115–126, 2001.

[10] T. R. Gruber. Towards Principles for the Design of Ontologies Used for Knowledge Sharing. *Int. Journal of Human-Computer Studies (IJHCS)*, 43(5/6):907–928, 1995.

[11] P. Kabus. Implementing semantic data integration for the internet (in german). MSc Thesis, Department of Computer Science, Darmstadt University of Technology, Germany, Oct. 2003.

[12] G. Mühl, L. Fiege, and A. Buchmann. Filter Similarities in Content-Based Publish/Subscribe Systems. In *International Conference on Architecture of Computing Systems (ARCS)*, volume 2299 of *LNCS*, pages 224–238. Springer, 2002.

[13] G. Mühl and L. Fiege. The REBECA Notification Service, 2001. `http://www.gkec.informatik.tu-darmstadt.de/rebeca/`.

[14] G. Mühl. *Large-Scale Content-Based Publish/Subscribe Systems*. PhD thesis, Darmstadt University of Technology, Germany, Sept. 2002.

[15] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The Information Bus – An Architecture for Extensible Distributed Systems. In *Proceedings of SIGOPS*, pages 58–68, 1993.

[16] L. Opyrchal, M. Astley, J. Auerbach, G. Banavar, R. Strom, and D. Sturman. Exploiting IP Multicast in Content-based Publish-Subscribe Systems. In *Proceedings of Middleware*, volume 1795 of *LNCS*, pages 185–207. Springer, 2000.

[17] M. Petrovic, I. Burcea, and H.-A. Jacobsen. S-ToPSS: Semantic Toronto Publish/Subscribe System. (demo paper). In *Proceedings of VLDB*, pages 1101–1104, Sept. 2003.

[18] P. Pietzuch and J. Bacon. Hermes: A distributed event-based middleware architecture. In *In Proceedings of the 1st International Workshop on Distributed Event-Based Systems (DEBS'02)*, Vienna, Austria, July 2002.

Extending Message-Oriented Middleware using Interception

Edward Curry, Desmond Chambers, and Gerard Lyons
Department of Information Technology,
National University of Ireland, Galway, Ireland.
{edward.curry, des.chambers, gerard.lyons}@nuigalway.ie

Abstract

Varieties of Message-Oriented Middleware (MOM) platforms are available each with their own propriety functionality to solve specific messaging challenges. At present, it is not possible to mix and match these propriety features into a customized MOM solution.

A number of patterns have been identified that allow a software systems implementation to be more flexible and extensible. This work investigates the use of one such pattern, the POSA Interceptor pattern, in the construction of a MOM framework that is easily customised and extended in a structured way.

This framework, Chameleon, is designed to support the use of interceptors (message handlers) with a MOM platform to facilitate dynamic changes to the behaviour of the deployed platform. The framework also allows for interceptors to be used on both the client-side and server-side, permitting advance functionality to be deployed to the client, and for co-operation between client-side and server-side interceptors.

1. Introduction

Message-Oriented Middleware (MOM) platforms are available in wide range of implementations such as WebSphere MQ (formerly MQSeries) [1], TIBCO [2], Herald [3], Hermes [4], SIENA [5], Gryphon [6], JEDI [7] and REBECCA [8] each of these providers have been designed with specific goals and employs unique functionality to achieve these. A number of these providers are designed as centralized servers or server clusters, others as server/broker networks, some as federated services. Providers have also been designed for a specific task such as enterprise integration, mobile clients, internet-level scalability, wide-area networks, sensor networks, and ubiquitous environments. In order to achieve these design goals, each provider has its own unique services for their target audience such as message translations, filtering algorithms, mobile profiles, broker routing algorithms, distributed state persistence technologies, and so on.

At present, the vast majority of these features are not compatible with one another, nor are they transferable between MOM platforms. The goal of this paper is to illustrate a potential approach for packaging these services into reusable chunks that may be mixed and matched to create a customised messaging solution.

The remainder of this paper briefly introduces the Interceptor design pattern and its current usage, gives an overview of our work on providing message handler chains (interceptors within the messaging domain) within the Chameleon framework. The frameworks architectural design and implementation is explained along with possible uses of the framework and future development plans.

2. Interception design pattern

The ever-increasing demands placed on software systems require them to often perform beyond the scope of their original requirements. Such behaviour may not always be anticipated during their initial development phase, thus making it important to design systems that can be easily extended during their life cycle.

Systems designed for large target audiences with diverse interests will often include functionality that is only utilized by a small percentage of users. While such functionality may be vital to some users, incorporating it into the core system makes it unnecessarily bloated and increases overhead for the majority of remaining users.

The POSA Interceptor design pattern [9] is a variant of the Chain of Responsibility pattern from the Gang of Four (GoF) [10] . This pattern enhances a system by increasing flexibility and extensibility. The pattern also enables functionality to be easily added to the system in order to dynamically change its behaviour. This seamless integration of functionality can be performed without the need to stop and recompile the system, allowing its introduction at runtime.

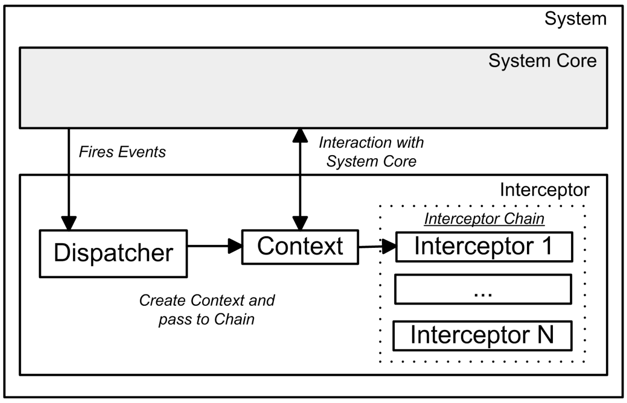


Figure 1. POSA Interceptor Pattern

The basic interceptor pattern has four main elements:

- System Core
- Dispatcher
- Context
- Interceptors

The Interceptor pattern, illustrated in Figure 1, follows a straightforward sequence of events.

Interceptors are registered with the system dispatcher; the system core may perform registration, the interceptors may self-register, or they may be registered from an external source (parent server/master server). Once the interceptors are registered, the system core notifies the dispatcher of any events that have occurred. Upon receiving an event, the dispatcher examines the event to determine which interceptors need to be notified. The dispatcher then packages the event and any relevant information into a context, the context may be provided by the system core. The dispatcher then notifies the relevant interceptors or interceptor chains (a ordered/unordered collection of interceptors) by passing them the context containing the event. When triggered, the interceptor examines the context and executes its related functionality.

An optional addition to the pattern is to allow interceptors access to the internals of the core system state and to provide a mechanism to control the system by altering its state.

2.1. Uses of the interception pattern

Interceptors are utilised in a broad range of domains to increase flexibility and extensibility; such systems include CORBA ORBs (TAO, Orbix) for infrastructure and support services, web browsers (Microsoft Internet Explorer) for plug-in integration, and web servers (Apache 2.0) to allow modules to register handlers (interceptors) with the core server. The JBoss J2EE [11] application server also uses the interceptor design pattern to provide customized functionality in the areas such as transactions, security, remoting and life cycle support.

Currently, no message service has exploited interception as a mechanism for extending its core-messaging functionality.

2.2. Evaluation

The Interceptor pattern has a number of advantages and disadvantages; benefits of the pattern include the decoupling of communications between a sender and receiver of an interceptor request, this permits any interceptor to fulfil the request and allows interceptors to change system functionality, even at run-time.

The pattern also has a number of drawbacks that if left unresolved may lead to a number of issues in the system design. One of the main drawbacks is increased complexity in design, the more interceptors can hook into the system the more bloated its interface. The inherent openness of the pattern also introduces potential vulnerabilities into systems. With such an open design, malicious interceptors or simply erroneous ones may be introduced resulting in system corruption or errors.

Another important issue to consider is the possibility of incompatibilities between interceptors and potential infinite interceptor loops whereby an event produced by an interceptor triggers another interceptor that in turn generates an event that triggers the original interceptor. Such errors will only occur at runtime and may be difficult to locate.

When used within the messaging domain the abstract generic interceptor pattern is implemented using a customised context or *'Message Context'* and '*Message Handlers'* as interceptors. The remainder of this paper uses such terminology.

3. Chameleon

The goal of this research is to implement the Chameleon message framework to allow message handler chains (Interceptors) to augment functionality onto a base MOM platform. In order to achieve this objective the Java Message Service (JMS) Application Protocol Interface (API) [12] is used as an interface to the underlying core messaging platform, the Chameleon framework sits on top of this Java Message Service System Core (JMSSC) platform. MOM services/features can be packaged as handlers and deployed to add functionality on top of the base service, enhancing its functionality.

The remainder of this section examines the architecture of Chameleon and discusses its ability to allow handlers to be deployed on both the client-side and server-side of a MOM platform.

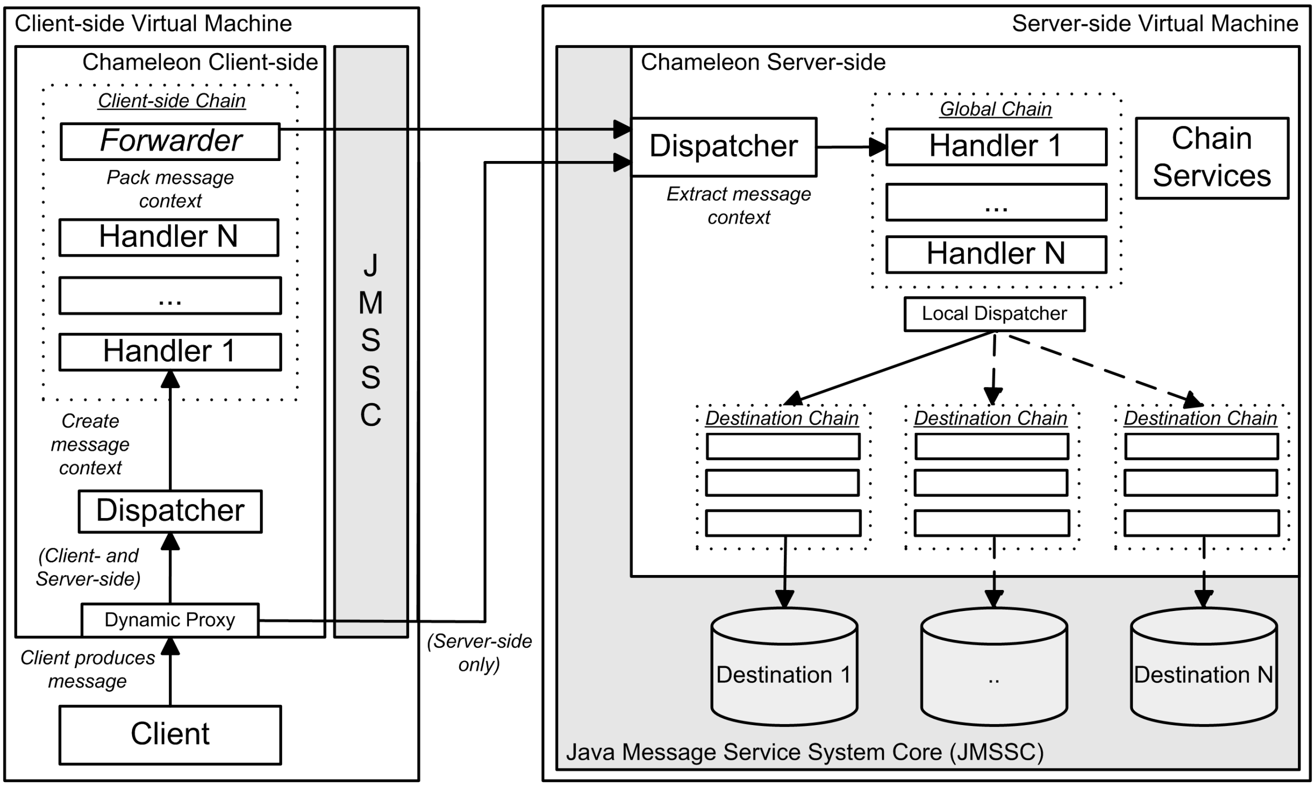


Figure 2. Chameleon Framework Overview

3.1. Server-side

The core of the Chameleon framework exists in its server-side deployment. When the framework initialises, it first registers any handler chains present in its start-up configuration. Server-side chains are constructed from local configuration files or under the guidance of a master or peer server within a broker network. This configuration may also be updated at runtime to adapt the chains for operating conditions.

When a message is passed from the JMSSC, chameleon first checks for the existence of a client-side message context, if one exists it uses it as the basis to create a server-side context. Alternatively, it creates a new context for the inbound message. Once the context is ready, it is passed to any relevant global server-side chains. After the global dispatcher has completed evaluating the message it is passed on to the local-chain dispatcher. This local-dispatcher is responsible for triggering any local-chains associated with specific destinations; chain scopes are discussed in more detail in section 3.5.

Server-side handlers have the ability to interact with a client-side handler (if one exists); this topic is covered in more details in the section 3.3 of this paper.

3.2. Message context

The message context is a handler's main point of interaction with the framework and JMSSC. It is central to the invocation of handlers and is used to communicate the message/event that has triggered it; the handler then processes the message. Message contexts can be used as a medium to store data, communicate with other handlers, or interact with the JMSSC and framework. Handlers are able to store and retrieve information within the context using the setProperty() and getProperty() methods. These methods can save any serializable object and the basic Java primitive types within the context.

Contexts can also be used to communicate information between client-side and server-side handlers, this allows for the behaviour on the server-side or client-side to be dynamically altered based on events and information from either side.

The context is also a mechanism to enable a handler to alter the behaviour of the JMSSC. The context achieves this by exposing an API to control core system operations. Handlers can be granted permissions to access parts of this API. Such an approach provides a safe method to manipulate the system core in a secure and controlled manner.

3.3. Client-side

Client-side handlers offer the ability to extended and enhance the functionality of the JMSSC on its client-side. Client-side chains allow the message service to dynamically alter the behaviour of its client at run-time.

Client-side handlers can be constructed in two manners, the first approach is to build the chain from local handlers which reside on the clients machine, this approach requires the machine to have the relevant handlers installed or to use a distributed classloader. The second approach involves the construction of chains on the server-side and transmission to the client; this removes the need to have application-specific stubs that would need to be pre-installed on the client machines. Once the chain is constructed it now needs to be configured, similar to server-side chains two approaches can be used, configuration can be obtained from a local file or from the server or peer node that the client is connect too. Client-side chain configuration may be updated at runtime to adapt the chains for operating conditions.

Client-side handlers introduce a number of advantages to MOM/DEBS by permitting computational tasks and behaviour to be easily distributed to client machines. With this support framework in place, advanced features may be developed with co-operation and co-ordination between both the client and server-side of the platform. Such capabilities could be used to increase the scalability of centralized servers by distributing tasks to the clients, such as message transformation, filtering, etc.

The dynamic retrieval and configuration of client-side handlers has a number benefits; services can now be deployed to the client without any special arrangements on the client-side. This streamlined distribution of services reduces the amount of administration needed to alter a deployed system, making frequent changes to its behaviour and configuration more feasible. A service can adapt itself into a more optimal state based on its current operating conditions. Clients may now connect to multiple servers and retrieve their specific client-side chain without the need for extra configuration or intervention by a system administration.

The prospect of deploying functionality to the client side is an interesting proposition, however due diligence must be taken when considering the use of this “mobile code”. If a client-side chain can be configured from a remote location or is downloaded, it presents a number of security issues and potential vulnerabilities to the client system; such issues are covered in more details in [13].

Once client-side chain initialization has complete, the outbound/inbound message is placed into a message context and passed to the client-side dispatcher. As soon as the message has passed through the chain, the message context is packaged into the JMS messages and sent to the server-side. Upon receipt of a new message, the server checks for the existence of a client-side context and uses this in the construction of the server-side context. This process allows client-side handlers to communicate with the server-side, allowing data exchange between them, such as the results of a distributed task or the results of any computations or pre-processing carried out on the message by the client-side.

The inverse of this process is also possible, whereby server-side handlers wish to pass message specific information details to the client-side; achieved by updating the message context of an outbound message.

3.4. Chain services

Message handlers within the Chameleon framework can provide a wide variety of functionality to the underlying JMSSC. To successfully carry out many of these tasks, handlers will require access to a number of infrastructure services. The Chameleon framework offers a numbers of support services for handlers; such services include logging facilities, persistence frameworks (hibernate), usage statistics, and core system states. Other potential services also include transactions, security, hardware usage statistics (CPU, memory, hard disk) and fault-tolerant system/event recovery logs.

Interactions with the core message service can use two methods; exploitation of the message context to control the core service has already been covered. The second approach involves using a chain service to access the system core. Such a service can expose models of the core message services internal structures and state. Chameleon provides one such service that exposes the current destinations that exist within the message service; the model contains destination configuration, usage statistics, and subscriber details. This service also provides destination administration capabilities (add, remove, move, etc). Such services are designed to allow handlers access to the internal state of the system in order to increase the functionality that they provide and enhance their ability to examine and adapt the core system at runtime.

3.5. Chain scope

Handler chains provide a powerful method of augmenting a message service with dynamic functionality; this ability can be extend by using multiple chains and attaching them at multiple interception points within the service core. An interception point is a place of execution within a system in which handler chains can be triggered to inject functionality. In order to achieve this, the systems behaviour must be modelled to allow chains to be attached to specific points within the model. A good quality system model will help to identify appropriate locations for interception points, this can also help to determine what handlers should be grouped together. Within the Chameleon framework, three chain interception points or interception scopes are available:

- Global (system-wide – all destinations)
- Local (per destination)
- Hierarchy (per branch)

Global-scoped chains are designed to operate on all incoming messages into the service and are triggered before any other scoped chains; global chains are designed for implementing system-wide services such as auditing, logging, or usage monitoring.

Local-scoped interception points work on a per-destination basis, allowing for chains to be attached to a single or group of destinations, once a message arrives for a given destination the chains attached to that destination (if any) will be triggered before the message is delivered to the destination. These interception points allow functionality to be added/extended at the destination level; this scope of chain allows highly specific behaviour to be attached to a single destination.

Hierarchy interception points allow chains to be associated with a channel within a hierarchy. Similar to local-destination interception points, hierarchy interception points work on the principle that each leaf (channel) of the hierarchy can have a local chain associated with it. The absolute chain for a branch consists of this local chain and the absolute chain of its parent, this recursive approach to interception continues up the tree until it reaches the root channel. This results in a very powerful mechanism for processing messages submitted to a channel hierarchy structure. This mechanism is similar to inheritance within object-oriented programming, where an object (channel) inherits the functionality of its ancestors (parents chains) and can augment this functionality with its local implementation (local chain).

4. Framework benefits

When compared to alternative techniques for system extension, such as method-call interception, reflection, or aspect-oriented programming, Chameleon provides a non-invasive message-centric method for augmenting any JMS compatible message service without needing access to its source code for mass-refactoring and recompilation.

Besides offering a flexible method of integrating MOM technologies, the Chameleon framework has a number of benefits for the development and deployments of MOM services. It provides a unique ability to easily deploy functionality to the client-side of a MOM without the need for any application-specific stubs to be present. MOM systems can be easily extended and their behaviour can be altered at run-time, this allows for a number of unique MOM features to be developed.

Chameleon can facilitate the development of reflective and adaptive MOM services, reflection has been advocated for advanced adaptive behaviour, the reflective middleware model is a principled and efficient way of dealing with highly dynamic environments yet supports development of flexible and adaptive systems and applications [14]. Reflective flexibility diminishes the importance of many initial design decisions by offering late and runtime-binding options to accommodate actual operating environments at the time of deployment, instead of anticipating the operating environments at design time [15].

Chameleon also facilitates the development of services that can co-ordinate their behaviour with the client-side of the message producer or consumer. This allows the development of a service that can easily distribute tasks to client machines; such capabilities open the possibility of developing new dynamic services for MOM. Such services could request clients to transforms their message payloads into the desired format for a destination, this would substantially reduce the workload of message brokers, for example each destination within the service may provide a XML schema and relevant XSLT stylesheets to transform incoming XML messages with on the client-side.

Chameleon also allows behaviour to be easily packaged as message handlers and dynamically added to the core service. Once packaged, behaviours can be mixed and matched to create customised messaging solutions. This process reduces the effort required to dynamically add and remove behaviour such as auditing, accounting, security, transactions, filters, message transformations (XSLT), and load balancing from a MOM service.

5. Future plans

The next stage of our research is to complete the development of the framework and to develop a number of services that exploit the virtues of the framework. The first of these services involves the utilisation of adaptive and reflective techniques with channel hierarchies [16].

Channel Hierarchies are structures that allow channels to be defined in a hierarchical fashion, so that they may be nested under other channels. Each sub-channel offers a more granular selection of the messages contained in its parent. Clients of the hierarchy *subscribe* to the most appropriate level of channel in order to receive the most relevant messages. In large-scale systems, the grouping of messages into related types (i.e. into channels) helps to manage large volumes of different messages [17].

Current MOM platforms do not define the structure of channel hierarchies. Application developers must therefore manually define the structure of the hierarchy at design-time. This process can be tedious and error-prone. To solve this problem the Chameleon messaging architecture implements reflective channel hierarchies [18] with the ability to autonomously self-adapt to their deployment environment. The Chameleon architecture exposes a causally-connected meta-model to express the configuration and structure of the channel hierarchy, this enables the run-time inspection and adoption of the hierarchy.

In order for handlers to be interoperable between services, a number of standards will need to be defined to regulate interaction with the core message service. Models are needed to represent the internal state of the service, its destinations, message producers and consumers, subscriptions, filters, usages statistics, and so on. If MOM behaviour is to be truly portable between implementations, such standards will need to be defined in co-operation with the entire MOM community.

Plans are also in place to port propriety functionality from a third-party message service to act as a proof of concept for the approach. We also plan to investigate further services, tools, and techniques to offer streamlined integration of disparate MOM services.

6. Conclusions

This work forms a necessary step in integrating Message-Oriented Middleware (MOM) technology from disperse implementations. While not claiming to be a sliver bullet, it provides a useful non-invasive mechanism to add/extend the functionality of an underlying MOM implementation. The process of augmenting functionality upon this core message service is achieved though the use of the POSA Interceptor pattern.

The Chameleon framework is designed to support the use of message handlers (interceptors) with a JMS compatible MOM platform. Chameleon allows MOM behaviours to be easily packaged as message handlers and dynamically added to the core service. Once packaged, behaviours can be mixed and matched to create customised messaging solutions.

The framework permits handlers to be used on both the client and server-sides, allowing for advanced functionality to be deployed to client systems, and for co-operation between client-side and server-side handlers. Handlers can also be added and removed at run-time; this enables the application of reflective and adaptive techniques within a MOM service.

7. Acknowledgement

The support of the Informatics Research Initiative of Enterprise Ireland is gratefully acknowledged.

8. References

[1] L. Gilman and R. Schreiber, *Distributed Computing with IBM MQSeries*. New York: John Wiley, 1996.

[2] D. Skeen, "An Information Bus Architecture for Large-Scale, Decision-Support Environments," presented at USENIX Winter Conference, 1992.

[3] L. F. Cabrera, M. B. Jones, and M. Theimer, "Herald: Achieving a Global Event Notification Service," presented at 8th Workshop on Hot Topics in OS, 2001.

[4] P. R. Pietzuch, "Event-Based Middleware: A New Paradigm for Wide-Area Distributed Systems?," 2002.

[5] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Achieving Expressiveness and Scalability in an Internet-Scale Event Notification Service," presented at Nineteenth ACM Symposium on Principles of Distributed Computing (PODC2000), Portland OR, 2000.

[6] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward, "Gryphon: An Information Flow Based Approach to Message Brokering," presented at International Symposium on Software Reliability Engineering, Paderborn, Germany, 1998.

[7] G. Cugola, E. D. Nitto, and A. Fuggetta, "The JEDI Event-Based Infrastructure and Its Application to the Development of the OPSS WFMS," *IEEE Transactions on Software Engineering*, vol. 27, pp. 827--850, 2001.

[8] L. Fiege and G. Mühl, "Rebeca", http://gkpc14.rbg.informatik.tu-darmstadt.de/rebeca/

[9] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, vol. 2: Wiley & Sons, 2000.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*: Addison-Wesley, 1995.

[11] M. Fleury and F. Reverbel, "The JBoss Extensible Server," presented at Middleware 2003, Rio de Janeiro, Brazil, 2003.

[12] M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout, "Java Message Service Specification v1.1," Sun Microsystems, Inc., 2002.

[13] D. M. Chess, "Security issues in mobile code systems," in *Mobile Agents and Security*, vol. 1419, *Lecture Notes in Computer Science*: Springer-Verlag, 1998.

[14] F. Kon, F. Costa, G. Blair, and R. H. Campbell, "The Case for Reflective Middleware," *Communications of the ACM*, vol. 45, 2002.

[15] R. E. Schantz and D. C. Schmidt, "Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications," in *Encyclopedia of Software Engineering*: Wiley & Sons, 2001.

[16] E. Curry, D. Chambers, and G. Lyons, "Could Message Hierarchies Contemplate?," presented at 17th European Conference on Object-Oriented Programming (ECOOP 2003), Darmstadt, Germany, 2003.

[17] P. R. Pietzuch and J. M. Bacon, "Hermes: A Distributed Event-Based Middleware Architecture," 2002.

[18] E. Curry, D. Chambers, and G. Lyons, "Reflective Channel Hierarchies," presented at 2nd Workshop on Reflective and Adaptive Middleware, Middleware 2003, Rio de Janeiro, Brazil, 2003.

On Methodologies for Constructing Correct Event-Based Applications [*]

Pascal Fenkam, Mehdi Jazayeri, Gerald Reif
Technical University of Vienna, Distributed Systems Group
A-1040 Vienna, Argentinierstrasse 8/184-1
{p.fenkam, m.jazayeri, g.reif}@infosys.tuwien.ac.at

Abstract

Various application domains exist where the advantages of the event-based paradigm make it a key technology. In general, this architectural style allows better control of the structural and behavioral complexity of applications: components can be developed independently and loosely integrated. The computational behavior of this paradigm, however, remains poorly understood. This position paper argues on the necessity of a new methodology for constructing event-based applications as well as a new logic that clarifies the computational behavior of such applications. For this, the paper presents some factors that make the event-based style so troublesome and discusses the (non-) adequacy of existing formal techniques for the construction of event-based applications.

1 Introduction

The different facets of the event-based (EB) architectural style (also called *publish/subscribe* or *implicit invocation*) are currently widely investigated. In this paradigm, some components (called *subscribers*) specify their interest in some kind of data (called *event notification* or simply *event*) that other components publish (or announce). The publishing component is called the *publisher*. The specification stating what kind of events a subscriber is interested in is called a *subscription*. The communication between the publishers and the subscribers is done through the *event-based infrastructure*. Events are sent to this infrastructure which then matches them against existing subscriptions and invokes interested subscribers. The dispatching of events to subscribers is completely transparent to the publishers, leading to loose coupling of components. This makes the event-based paradigm a key technology for many application domains.

Despite the wide acceptance of the EB concept, the development of applications based on it remains an ad hoc and informal process. Consequently, as EB systems proliferate, including in safety-critical domains, it remains difficult [20, 21] to reason about the reliability of such systems. This is especially unfortunate in the case of the EB paradigm because the loose component coupling it supports naturally leads to a component based approach to the construction of distributed applications and therefore a consequent expectation of increased reliability.

The aim of this paper is to shed some light on the strange behaviors of the EB paradigm. We present three behavioral aspects of this style that we have distilled and that make us strongly believe that at least a new logic is needed that captures the quintessential properties of this style. First, we show that the traditional meaning of the sequential composition of programs does not hold anymore. This is related to the second issue, where we claim that the associativity of the sequential composition is not present in the context of EB applications. The third strange behavior is called the “assertion fleeing effect”: the result of a subscriber is always postponed to the “end” of the post-conditions of programs. Finally, we discuss the applicability of existing formal techniques to the construction of EB applications. We claim that although a new formal specification language may not be required, a logic must be developed that clarifies the behavior of the EB architectural style and renders it unambiguously understood.

The remainder of the paper is organized as follows. We condition methodologies for the development of correct EB applications with a set of requirements that we present in the next section. Section 3 summarizes existing approaches for constructing EB applications and discusses their shortcomings. In section 4, we briefly present an abstract semantics of the concept of EB system. This abstract semantics is the basis for the elicitation of event-based systems’s behaviors in Section 5. We pursue the discussion of the need of a new methodology for constructing EB applications in Section 6. Section 7 concludes the paper.

[*] This work is supported by the Austrian Research Foundation (FWF) project RAY (Number P16970-No4).

2 Requirements for a Methodology

We present some requirements for a methodology for developing correct EB applications. For the elicitation of these requirements, we consider three application areas: component based software engineering (CBSE), mobile computing, and coordination languages. Our requirements are clearly not exhaustive as the EB paradigm is applied in many other domains.

2.1 Generic Requirements

Before going to the specific requirements of the above areas, there are generic requirements that a software development methodology must strive for, namely the scalability to large systems and the ability to be used to consistently produce maintainable and high-quality software at low cost and with a small turnaround time [25]. Four issues are identified in this requirement: scalability, quality, cost and schedule, and maintenance issues. Since the degree to which a methodology supports these issues is in general hard to measure, it may be argued that any development method supports them more or less. The goal is therefore, to find methods that better respond to these requirements than those currently available.

2.2 CBSE related Requirements

CBSE is an emerging methodology that promises to adequately address many of the above issues. And, indeed, many researchers are working on making it a reality.

One of the issues that need to be solved is the composition and integration of components; the EB paradigm is currently intensively deployed for this purpose. Any software development methodology that supports the development of EB applications must, therefore, also explicitly support CBSE. That is, such an approach must be compositional, both at the abstract level (conceptual level) as well as at the implementation level. At the abstract level, it must be possible to compose component specifications as well as proofs about properties of these specifications. In particular, the methodology must allow building models in a clearly defined and intuitive way such that one can understand the whole by understanding the parts and recombining them in predictable ways [13].

2.3 Mobility related Requirements

Mobile computing is a paradigm in which users are able to communicate and use their applications while they are moving [9]. The difficulty in such computational models is that one can not rely on the permanent availability of the network. The resulting connectivity intermittence has led to the argument that traditional client/server middleware are not suitable for mobile computing [3, 9, 10, 23]. The EB paradigm is viewed as a valuable replacement to client/server; it does not require subscribers to be present at the time of announcement. This asynchrony in the communication is mapped one-to-one to mobile computing. A subscriber represents a mobile device that comes to and leaves the network at will.

From this short analysis, we derive that an approach for the construction of EB applications must support components that leave and come intermittently. In addition, to support easy maintenance of systems, or to be applicable to pervasive computing, such an approach must be able to foresee the integration of new components in a running system. In the EB terminology, this implies that the reasoning should not be based on a pre-defined static set of subscribers.

2.4 Coordination related Requirements

Coordination includes the specification, analysis, and control of the cooperation between components of a system. A significant amount of work is currently undergoing on this topic (see [4, 5, 29]). Many of the proposed solutions are based on the EB architectural style. The requirement that are placed upon methodologies for constructing EB applications by coordination languages are in essence similar to those of component based engineering and mobile computing. In particular, it is argued that compositionality, evolvability, and easy handling of volatile business requirements must be supported [4].

Due to the similarities in the goals of the coordination paradigm and the many uses of the EB architectural style, it is important to notice that neither subsumes the other; the EB paradigm is used for other purposes than coordination while there are coordination techniques that are not based on the EB technique (e.g. [5, 7]).

3 Related Work

Despite the wide acceptance of the EB paradigm, not much work has been presented on building correct applications using it.

3.1 Formalizing Architectural Styles

Several researchers have attempted to provide formal techniques that can support the treatment of EB applications. Although the ultimate goal of such works is reliability, the developed approaches have not been successful. Examples of such approaches are that of Garlan and Notkin [22], as well as that of Abowd, Allen, and Garlan [1, 2] who propose frameworks for formalizing architectural styles in general and implicit invocation in particular. These approaches primarily focused on taxonomic issues, and do not provide an explicit computational model that permits compositional reasoning about the behavior of EB applications [12, 11]. This claim is justified by observing that the semantics of a method (also called operation or function) is not defined. It is, therefore, not possible to reason about the behavior of these methods (at least not without extending the framework). Moreover, it is not said how a method must be specified. For instance, a key question in formally specifying EB applications is how does a designer specify that a method m announces an event e whenever the state satisfies the condition Q? The next issue in such approaches is that there is no indication on whether the specifications are realizable or not. Given such a specification, what is the next step in building an application? How does a designer show that a given application satisfies such a specification?

3.2 Verification of EB Applications

Dingel et al.[12] propose an approach for verifying the correctness of EB applications. Since this is an a-posteriori approach, the components of the completed program are verified in isolation and then put together where general properties are attempted to be proved. The scalability to large applications can not be achieved, since erroneous design decisions taken in early steps are propagated until the system is implemented and attempted to be proven correct [26, 16]. Further, given that this approach assumes a static set of subscriptions the evolvability and the maintainability of systems are therefore hard to support.

3.3 Model checking EB Applications

Model checking EB applications is interesting in that a significant part of the process is carried out automatically. In [8, 20, 21], attempts to apply model checking to the verification of EB applications are discussed. The authors provide generic frameworks to be reused by modelers in the process of defining the abstract structure related to their systems. Indeed, the authors succeeded in factoring the work such that, for instance, the event delivery policy is now a pluggable element with various packaged policies (prepared by the authors) that can be used off-the-shelf.

In general, in addition to being an a-posteriori approach, model-checking EB applications suffers from the fact that there is no known adequate way for specifying the announcement of events in presence of interference. Ideally, it should be possible to specify components, verify their properties, implement them, and integrate them in a predictable way.

The discussion in this paper results from our early attempts to construct a methodology for the stepwise development of event-based applications [16, 15, 17, 14].

4 Abstract Semantics of the EB Paradigm

To make clear the differences in the behavior of the EB paradigm, we need to clarify the programming language and the semantics that we assume. We will attempt as far as possible to remain informal in this paper. Let us assume the following while-parallel language that also allows announcement of events.

$$\begin{aligned} P ::= x := e \mid P_1; P_2 \mid \textbf{if}\ \ b\ \textbf{then}\ \ P_1\ \textbf{else}\ P_2\ \textbf{fi} \mid \textbf{skip} \\ \mid \{P_1 \| P_2\} \mid \textbf{while}\ \ b\ \textbf{do}\ \ P\ \textbf{od}\ \mid \textbf{announce}(\mathrm{e}) \end{aligned}$$

The semantics of the constructs in this language seem to have the traditional meaning. Indeed, the semantics of the announcement construct is simple: the set of subscribers to an event are executed in parallel with the remainder of the announcing program. That is, if we assume that there is no event announcement in the program z_1 and that the program z_e is subscribed to the event e, then $z_1; \mathbf{announce}(e); z_2$ is a program that behaves as z_1 and follows as $\{z_e \| z_2\}$ if z_1 terminates. In this semantics, we abstract from the EB infrastructure and simply model its behavior: the invocation of subscribers. For simplicity, we have removed the await construct that we propose [16, 15, 17] for synchronization and mutual exclusion in EB systems. The await construct gives more power to the language and allows it to simulate more types of EB systems. The aspects related to this construct will not be investigated in this paper. Further, we have ignored passing the event to the subscribers; this is discussed in [14].

5 Challenges of the EB Paradigm

We take a close look at the above semantics and present some aspects that make reasoning about EB applications difficult.

5.1 The meaning of sequential composition

In traditional systems (e.g. Hoare Logic [30], VDM [27]), a program $z_1; z_2$ is a program that behaves as z_1 and follows as z_2 if z_1 terminates. If the result of z_1 (respectively z_2) is captured by means of its post-condition E_1 (respectively E_2), then the result of $z_1; z_2$ is E_2 if the pre- condition of z_2 follows from E_1. This is expressed in Hoare's style as:

$$\frac{\begin{array}{c}\{E_0\}\ z_1\ \{E_1\} \\ \{E_1\}\ z_2\ \{E_2\}\end{array}}{\{E_0\}\ z_1; z_2\ \{E_2\}} \tag{1}$$

Let us now consider the case with an event-announcement. If the subscribers of e, say z_e, is such that $\{E_1\}\ z_e\ \{E_e\}$ holds, then the following rule follows from a naive double application of the above rule:

$$\frac{\begin{array}{c}\{E_0\}\ z_1\ \{E_1\}\\ \{E_1\}\ z_e\ \{E_e\}\\ \{E_e\}\ z_2\ \{E_2\}\end{array}}{\{E_0\}\ z_1;\mathbf{announce}(e);z_2\ \{E_2\}} \qquad (2)$$

This rule is, however, unsound because the computational model of the EB paradigm requires to interpret the announcement of the event e as the parallel execution of z_e and z_2. If we assume that z_e and z_2 coexist, that is, there is no interference of one with the other, then the following rule must hold.

$$\frac{\begin{array}{c}\{E_0\}\ z_1\ \{E_1\}\\ \{E_1\}\ z_e\ \{E_e\}\\ \{E_1\}\ z_2\ \{E_2\}\end{array}}{\{E_0\}\ z_1;\mathbf{announce}(e);z_2\ \{E_2\ \wedge\ E_e\}} \qquad (3)$$

Hence, sequential composition of programs when the EB paradigm is involved does not always have the traditional meaning. As a natural consequence, the iteration rule looses its traditional properties.

5.2 Non-associativity of sequential composition

A natural consequence of the above rules is that the associativity of the sequential composition of programs may no longer be assumed. Without event announcement, $z_1;\{z_2;z_3\}$ has the same results as $\{z_1;z_2\};z_3$. This does not hold when z_2 announces an event. Let us replace z_2 with the announcement of the event e. The naive way of composing the three programs z_1, $\mathbf{announce}(e)$, and z_3 is to start by composing z_1 with $\mathbf{announce}(e)$ which results in the program $z_1;z_e$ whose post-condition is E_e (if the pre-condition of z_e follows from E_1). Next, we compose $z_1;\mathbf{announce}(e)$ with z_3 which results into a program satisfying E_2 if the pre-condition of z_3 follows from E_e.

This result is, however, unsound. The semantics of the announce construct requires that the program z_e be executed in parallel with the remainder of the announcing program, i.e. z_3. The sequential composition operator is, therefore, right associative in the presence of event announcement.

5.3 Fleeing Assertions

As shown in the previous section, in the presence of event announcement, sequentially composing a program z_1 with the program z_2 does not result in a program that behaves as z_1 and follows as z_2. In absence of interference, the result of the subscribers to e (an event announced by z_1) will always hold in the final state of the program. Let us explain this in the light of the following figures.

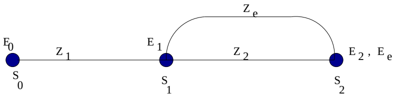


Figure 1. Behavior of $z_1;\mathbf{announce}(e);z_2$

Figure 1 captures the behaviors of $z_1;\mathbf{announce}(e);z_2$ as expressed by Rule 3. The first program z_1 is executed in a state satisfying E_0. Next an event announcement is executed in the state S_1 satisfying E_1, resulting in the parallel execution of z_e and z_2. The programs z_1 and z_2 both terminate in the state S_2 which satisfies E_2 and E_e.

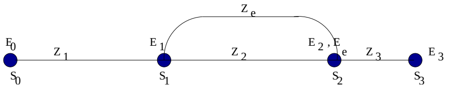


Figure 2. Naive Interpretation of $z_1;\mathbf{announce}(e);z_2;z_3$

We now perform a sequential composition of the above program with z_3. Figure 2 illustrates the case of a naive sequential composition: z_3 starts after z_2 and z_e are terminated. This is the wrong interpretation discussed in the previous section.

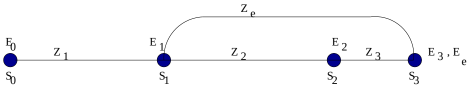


Figure 3. Fleeing Assertion

Figure 3 shows the sound behavior of $z_1;\mathbf{announce}(e);z_2;z_3$. The subscriber z_e is executed in parallel with $z_2;z_3$. The post-condition of z_2 which held in the state S_2 in Figure 1 now instead holds in the state S_3. In general, the result of a subscriber can not be " caught" by a sequentially composed program. We say that the post-condition of such a subscriber is a fleeing assertion; it keeps "fleeing" to the final state of any computation (provided there is no interference).

5.4 Similarity with the π-Calculus

The semantics that we have given to the announce construct in this paper (and also in [16, 15, 17, 14]) has some resemblance to asynchronous communication. In this communication mechanism, in fact, a process/agent sends a message to the other process and both continue their execution running in parallel with each other. The behavior advocated can, therefore, also be observed in such systems.

One has, however, to be careful when analyzing this behavior in the context of formal frameworks for asynchronous communication. Considering for instance the π-calculus [28], the agent $(u(a).Q_1\|\overline{u}(b).Q_2).P$ is an agent such that $\overline{u}(b).Q_2$ sends the message b to $u(a).Q_1$ and evolves as Q_2 while the recipient evolves as $Q_1[b/a]$ and run in parallel with Q_2. And, once the agent $(u(a).Q_1\|\overline{u}(b).Q_2)$ is terminated, the agent P can be executed. This is completely different from event-based applications where it is not possible to constrain a program to start after the termination of a subscriber. If we replace for instance $\overline{u}(b).Q_2$ with $\mathbf{announce}(b).Q_2$ and let $u(a).Q_1$ be the subscriber to the event b, then, the above agent evolves into $(Q_2.P)\|Q_1[b/a]$, which is completely different from the result obtained when there is no event announcement.

5.5 Shared Variable Interference

The EB style is intended for the loose coupling of components. This inspires that shared variables are not a primary issue in constructing methods for developing such applications. Yet, we argue that the issue of shared variables can not be ignored.

Let us illustrate a simple case in which loose coupling and shared variable are present. Two Java classes *Stack* and *Counter* are assumed where *Stack* contains the methods *push* and *pop* while *Counter* contains *increment* and *decrement*. The example is from [12, 11]. The meaning of these methods is indeed what the reader may expect. We require that *push* (resp. *pop*) announces an event of type *EventA* (resp. *EventB*) after performing its supposed task. Next, we construct an instance *somestack* of *Stack* and an instance *somecounter* of *Counter*. Using an EB middleware, we subscribe *somecounter.increment* (resp. *somecounter.decrement*) to events of type *EventA* (resp. *EventB*). The reader should agree that this is indeed a loosely coupled system: *Stack* knows nothing about *Counter* and vice versa.

To be convinced of the presence of shared variables interference, one observes that the result of *somestack.push* depends on whether it is executed alone or in parallel with *somestack.pop*, in which case there is indeed interference between *somestack.push* and *somestack.pop*. This kind of interference is indeed very difficult to master in the case of EB systems since the two methods *somestack.push* and *somestack.pop* may be invoked in a non-deterministic way by the environment.

6 Do we need "Yet another formalism"?

We have presented the requirements that we expect from a methodology that supports the EB architectural style. We have also presented some behavioral properties that contribute to making reasoning about the correctness of EB applications a non-trivial task. We are now armed to address the question of whether we need a new formalism for constructing such applications.

In general, we plead that there is the need for a new methodology that allows the development of reliable event-based applications. While a new formalism may not be required, we claim that an adequate logic is required that renders the computational behavior of the event-based paradigm unequivocally understood. We expect such a logic to illuminate the derivation and the inference of programs that conform to the EB computational model. Such a logic does not need to be orthogonal to existing logics, but should ideally be based on them such that existing tools can be leveraged.

Although such logics are often not widely used in practice, we believe that they give rise to sound and well founded less formal frameworks such as model checking and formal testing that are indeed more used in practice. Further, as acknowledged by Dingel et. [12, 11], the intuition gained in the development of formal techniques are of particular significance to software engineers.

7 Conclusions

The increasing use of the EB paradigm motivates the need for methodologies to support not only the construction of systems but also reasoning about their correctness. While some argue that reasoning about the EB paradigm is hard [20] other instead claim that no new methodology is required for constructing EB applications [19, 18]. The aim of this paper was to bring light on this starting "dispute."

Based on the requirements put on the EB style by its application domains, we deduce that there is need for a suitable methodology for EB applications. On the other hand, based on some identified behavioral aspects of this style, we plead that this methodology must be supported by a logic that allows a clear understanding of the EB paradigm.

Given that there are many factors that contribute to making it difficult to construct a methodology for the developing correct EB applications, following Michael Jackson [24], we further suggest that the EB architectural style be refined into architectural types [6] which are obtained from architectural styles by fixing some of their parameters.

This conclusion is guiding our work in the area of event-based applications [16, 15, 17, 14].

References

[1] G. D. Abowd, R. Allen, and D. Garlan. Using style to understand descriptions of software architectures. *ACM Software Engineering Notes*, 18(5):9–20, 1993.

[2] G. D. Abowd, R. Allen, and D. Garlan. Formalizing styles to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4):319–364, 1995.

[3] E. Anceaume, A. K. Datta, M. Gradinariu, and G. Simon. Publish/subscribe scheme for mobile networks. In *Proceedings of the second ACM international workshop on Principles of mobile computing*, pages 74–81. ACM Press, 2002.

[4] L. Andrade and J. Fiadeiro. Coordination – the evolutionary dimension. In *Proceedings of TOOLS Europe 2001*, pages 136–147. IEEE Computer Society Press, 2001.

[5] F. Arbab, M. Bonsangue, and F. de Boer. A coordination language for mobile components. In *Proceedings of the 2000 ACM Symposium on Applied Computing (SAC 2000)*, pages 166–173. ACM press, 2000.

[6] M. Bernardo, P. Ciancarini, and L. Donatiello. On the formalization of architectural types with process algebras. In *Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 140–148. ACM Press, 2000.

[7] M. Bonsangue, J. Kok, and G. Zavattaro. Comparing coordination models based on shared distributed replicated data. In *Proceedings of the 1999 ACM Symposium on Applied Computing (SAC '99), San Antonio, Texas, USA*, pages 146 – 155. ACM press, February 1999.

[8] J. S. Bradbury and J. Dingel. Evaluating and improving the automatic analysis of implicit invocation systems. In *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 78–87. ACM Press, 2003.

[9] G. Cugola and E. D. Nitto. Using a Publish/Subscribe Middleware to Support Mobile Computing. In *Proceedings of the Workshop on Middleware for Mobile Computing, in association with IFIP/ACM Middleware 2001 Conference, Heidelberg, Germany*, November 2001.

[10] G. Cugola, E. D. Nitto, and G. P. Picco. Content-based dispatching in a mobile environment. In *Proceeding of WSDAAL 2000, Ischia (Italy)*, Septenber 2000.

[11] J. Dingel, D. Garlan, S. Jha, and D. Notkin. Reasonning about Implicit Invocation. In *Proceedings of the 6th International Symposium on the Foundations of Software Engineering, FSE-6, Lake Buena Vista, FL*, pages 209–221. ACM, November 1998.

[12] J. Dingel, D. Garlan, S. Jha, and D. Notkin. Towards a formal treatment of implicit invocation using rely/guarantee reasoning. *Formal Aspects of Computing*, 10:193–213, 1998.

[13] D. F. D'Souza and A. C. Wills. *Objects, Components, and Frameworks with UML, The Catalysis Approach*. Addison Wesley Longman, Inc., 1998.

[14] P. Fenkam. *A Systematic Approach to the Development of Event-Based Applications*. PhD thesis, DSG, Technical University of Vienna, October 2003.

[15] P. Fenkam, H. Gall, and M. Jazayeri. A Systematic Approach to the Development of Event-Based Applications. In *Proceedings of the 22nd IEEE Symposium on Reliable Distributed Systems (SRDS 2003), Florence, Italy*. IEEE Computer Press, October 2003.

[16] P. Fenkam, H. Gall, and M. Jazayeri. Composing Specifications of Event Based Applications. In *Proceedings of FASE 2003 (Fundamental Approaches to Software Engineering 2003), Warsaw, Poland*, LNCS, pages 67–86. Springer Verlag, April 2003.

[17] P. Fenkam, H. Gall, and M. Jazayeri. Constructing Deadlock Free Event-Based Applications: A Rely/Guarantee Approach. In *Proceedings of FM 2003: the 12th International FME Symposium, Pisa, Italy*, LNCS, pages 632–657. Springer Verlag, September 2003.

[18] L. Fiege, M. Mezini, G. Muhl, and A. P. Buchmann. Engineering event-based systems with scopes. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP 2002)*, volume 2374, pages 309–333. LNCS, 2002.

[19] L. Fiege, G. Muhl, and F. Gartner. A modular approach to building structured event-based systems. In *Proceedings of the 2002 ACM Symposium on Applied Computing (SAC'02)*, pages 385–392, Madrid, Spain, 2002. ACM Press.

[20] D. Garlan and S. Khersonsky. Model checking implicit-invocation systems. In *Proceedings of the 10th International Workshop on Software Specification and Design, San Diego, CA*, pages 23–30, November 2000.

[21] D. Garlan, S. Khersonsky, and J. S. Kim. Model checking publish-subscribe systems. In *Proceedings of the 10th International SPIN Workshop on Model Checking of Software (SPIN 03), Portland, Oregon*, pages 166–180, May 2003.

[22] D. Garlan and D. Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *Proceedings of Fourth International Symposium of VDM Europe: Formal Software Development Methods*, Noordwijkerhout, Netherlands, October 1991. LNCS 551.

[23] Y. Huang and H. Garcia-Molina. Publish/subscribe in a mobile enviroment. In *Second ACM international workshop on Data engineering for wireless and mobile access*, pages 27–34. ACM Press, 2001.

[24] M. Jackson. *Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices*. Addison-Wesley, 1995.

[25] P. Jalote. *An Integrated Approach to Software Engineering*. Springer Verlag, 1997.

[26] C. Jones. Tentative steps towards a development method for interfering programs. *Transactions on Programming Languages and Systems*, 5(4), October 1983.

[27] C. B. Jones. *Systematic software development using VDM*. Prentice-Hall International, 1990. 2nd edition.

[28] R. Milner. *Communicating and Mobile Systems: the pi-Calculus*. Cambridge University Press, May 1999.

[29] G. A. Papadopoulos and F. Arbab. Coordination models and languages. *Advances in Computers*, 46, 2001.

[30] D. von Oheimb. Hoare logic for mutual recursion and local variables. In V. R. C. Pandu Rangan and R. Ramanujam, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1738 of *LNCS*, pages 168–180. Springer, 1999.

Security Aspects in Publish/Subscribe Systems

L. Fiege A. Zeidler A. Buchmann
TU Darmstadt
{fiege,az,buchmann}@dvs1.informatik.tu-darmstadt.de

R. Kilian-Kehr
SAP Corporate Research
roger.kilian-kehr@sap.com

G. Mühl
TU Berlin
gmuehl@acm.org

Abstract

Publish/subscribe is emerging as a very flexible communication paradigm that is applicable to environments demanding scalable and evolvable architectures. Although considered for workflow, electronic commerce, mobile systems, and others, security issues have long been neglected in publish/subscribe systems. Recent advances address this issue, but only on a low, technical level. In this paper, we analyze the trust relationships between producers, consumers, and the notification infrastructure. We devise groups of trust to model and implement security constraints both on the application and the system level. The concept of scopes helps to localize and implement security policies as an aspect of structured publish/subscribe systems.

1 Introduction

The publish/subscribe paradigm is an interest-oriented communication model [3]. Event notifications are published by producers, and consumers receive those that match one of the subscriptions they have specified. The paradigm is successfully applied in many areas of distributed computing, and the loose coupling of producers and consumers leverages reconfigurability and evolution. Recent research mainly focused on functional aspects of the intermediary pub/sub service that conveys the notifications. It is considered a black box optimized for notification routing and scalability in distributed settings. Today, an increasingly important emerging aspect of publish/subscribe systems is *security* and *trust*. This includes *access control* to the pub/sub infrastructure (and the data it transports), as well as the need to establish mutual *trust* between producers and consumers of data, i.e., granting the *authenticity* and *validity* of data in the system.

This imposes the question of *how* the mutual trust between publisher and consumer can be established despite the decoupling facilitated by the pub/sub paradigm. The obvious approach is to delegate some of the aspects of trustworthy interaction to the pub/sub service for enforcement. For instance, access control and secured delivery can be added to the pub/sub infrastructure [1]. Unfortunately, this often implies that the infrastructure as a whole is trusted, a frequently found assumption.

At Internet scale, however, the pub/sub infrastructure itself has to be considered as a security issue. A distributed network of event brokers likely spans a larger number of service providers and many administrative domains. Consequently, the security considerations of producer-consumer interaction must include the infrastructure, and a black box view on it is no longer applicable.

Initial work is available on security issues in publish/ subscribe. A general description of requirements is given by Wang et al. [12]. An apparent problem is access control to the pub/sub service and certain classes of notifications [8, 1]. Miklós [8] uses the Siena covering relations to constrain allowed subscriptions and advertisements; a trusted broker network is assumed. Perhaps the most advanced result is Belokosztolszki et al. [1], who combine role-based access control with a distributed notification service. The privilege to publish or subscribe to a specific *type* of event is granted by a designated owner of this type. A relaxation of the trusted network assumption is sketched that finds connected broker subgraphs that use encrypted communication links. However, globally valid type hierarchies are problematic to establish and limited in their modeling capabilities [7].

In this paper, we want to weaken the assumption of a trusted network to a large degree. The settings we consider are systems where the notification service (a) can be part of a larger network consisting of different transport networks of unknown trustworthiness; and (b) the notification delivery itself may span several separate notification services or administrative domains. Obviously, mechanisms must be in place to bridge potentially malicious networks or brokers, as well as to establish mutual trust between different administrative domains on behalf of a client. We discuss these issues in greater detail in Section 2. Then, in the remainder of this paper, we show our approach of applying *scopes* to the problems aforementioned. Scopes originally were designed to model *visibility* of event dissemination within the distributed pub/sub notification service REBECA (cf. Section 3). In Section 4 we exploit scopes to enforce and maintain security aspects in Internet-scale pub/sub systems. Section 5 sketches an implementation using aspect-oriented programming techniques, before Section 6 concludes this paper.

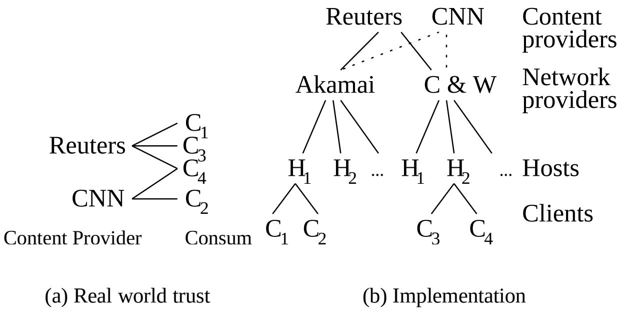
2 Trust in Pub/Sub Systems

2.1 System Model

A minimal pub/sub system consists of producers, consumers, and the intermediary pub/sub service to convey the published notifications. The pub/sub service offers three simple-to-use primitives: `subscribe`, `advertise`, and `publish` to register consumer interests, to announce potential future notifications, and to publish a notification. The notification service itself acts as a black box and is conceptually centralized, which we refine later. We assume a distributed implementation in a network of event brokers; the brokers to which clients are connected are called border brokers. Each broker maintains a routing table that keeps track of the network links and the subscriptions that were received on them. Notifications are forwarded on those links for which a matching subscription is stored.

2.2 Trust

Trust in the sense we use it in this paper has two different aspects: a "real-world" aspect of *trusting someone or something* on the basis of some contract (Fig. 1(a)) and, second, the aspect of implementing trust through some security measures in a more technical sense (Fig. 1(b)). In traditional systems, security is mostly based on knowing the identity of involved parties, which is not possible in publish/subscribe. Indeed, at first sight one might argue that security contradicts its open and decoupled nature.



(a) Real world trust (b) Implementation

Figure 1. Trust

Figure 1 depicts a common example from the domain of e-commerce applications: a customer subscribes to a "premium stock market ticker" provided by *Reuters*, for instance. As the service comes with a monthly fee, contracts are concluded between customers and the service provider (Fig. 1(a)) describing the terms of their trust relationships.

Obviously, the provider of such a service has an interest in *access control*. Only *authenticated* and *authorized* customers should be able to receive the stock market quotes. The most basic requirement for a security implementation is to allow access to the service for the group of valid customers and to deny access to anybody else. On the other hand, a customer of such a premium service wants to be sure that information received from the service is *authentic*, i.e., originates from the premium service and is not manipulated. Therefore the customer has to trust the *authenticity* and *validity* of the received information. Taken together, provider and customers share a common group of trust in which they interact.

Obviously, the presence of the pub/sub infrastructure as an intermediary introduces an additional level of trust concerns. Not only that the infrastructure must be trustworthy itself, it also must be leveraged to implement the trust relationship between the producer and consumer (cf. Fig. 1(b)). From the point of view of a *group of trust*, as described above, the infrastructure must be part of the application-specific group of trust that customers and provider share.

Consequently, an implementation of real-world trust must secure groups of application components *and* the underlying groups of event brokers necessary to connect the components. On both levels, measures must be taken to separate communication within the group from outsiders and to base group admission on credentials sufficient to establish mutual trust.

2.3 Current Deficiencies

Contemporary design of pub/sub services focuses on functional aspects of the pub/sub paradigm, i.e., efficiency of message routing, scalability, expressiveness of filter languages, or event composition, to name only a few.

However, trust and security is not part of the pub/sub paradigm, and the trust relationship is not directly enforcable in producers and consumers. Security is a separate aspect of publish/subscribe, *outside* of the pure *ability* to convey messages. Trust is injected into a system based on external contracts on the level of applications. The goal must be to map trust agreements to the underlying notification service for implementation and enforcement. The current model of pub/sub assumes that the black box model of a conceptually centralized pub/sub service is applicable at all times. But implementing a group of trust requires additional control on *how* messages are delivered in the infrastructure.

What is needed to implement trust and security on top of the pub/sub paradigm, is fine-grained control over every part of the infrastructure used to transport messages from a producer to a consumer. Each part on this way has to have the same trustworthiness *as if* producer and consumer would communicate directly. To inject this extensive level of control we exploit the concept of scoping introduced in the next section.

3 Scoping

Scopes in publish/subscribe systems [5, 6] delimit groups of producers and consumers on the application level and control the dissemination of notifications within the infrastructure. Hence, they offer a technical basis to realize groups of trust. This section describes their basic functionality and security extensions are shown in Section 4.

3.1 Model

The fundamental idea of the scoping concept is to control the visibility of notifications outside of application components and orthogonal to their subscriptions. A scope *bundles* a set of simple application components, i.e., producers and consumers, which are typically not aware of this bundling. Additionally, it may contain other scopes as well. The resulting structure of the system is given by a directed acyclic graph of simple and complex components .

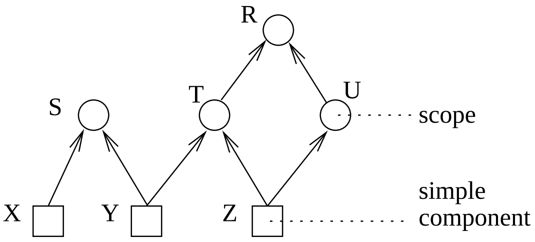


Figure 2. An exemplary scope graph

The visibility of notifications is initially limited to the scope they are published in. The transition of notifications between scopes is governed by *scope interfaces*, i.e., a scope issues subscriptions and advertisements in order to act as regular producer and consumer in its superscope(s). The scope's interface selects the eligible notifications that are forwarded to their superscopes and the external notifications that are relayed towards the scope's (sub-)components. In Figure 2, a notification published by Z is delivered to Y and to any other consumers in T and U if their subscription matches. Also, it is visible in R if it matches the output interface of T or U, but it is not visible in S.

3.2 Using Scopes

Four new functions are needed for maintaining a scope graph: *creating* new scopes (`cscope`), *destroying* scopes (`dscope`), *joining* an existing scope (`jscope`), and *leaving* a scope (`lscope`). Two approaches to scope administration exist. First, the functions may be directly accessed by the clients of the pub/sub service, i.e., the producers and consumers, i.e., the components of applications. In this case the functions are provided as extensions of the publish/subscribe API.

However, in accordance with the loose coupling of the event-based paradigm, scope management should be done outside of the application components. We identified the role of an administrator who is responsible for orchestrating existing components into new scopes, which in turn are available for higher level composition. At deployment time, *descriptors* assign newly created components to certain scopes. At runtime, we leverage management interfaces to remotely administrate scope membership of existing components. For the implementation of *trust* we can exploit the same mechanisms for assigning components to certain application-dependent scopes, representing a group of trusted components.

3.3 Scope Architectures

We sketch a distributed implementation of scopes as an extension of the REBECA distributed notification service [10]. This approach opens the black box and determines groups of event brokers that implement a specific scope, thus correlating groups on the application and the system level.

Integrated routing reconciles distributed notification routing with the visibility constrains defined by the scope graph. The original routing table is broken into multiple tables, one for each locally available scope. Thus, for each scope a connected subset of event brokers constitute an overlay within the broker network that conveys scope-internal traffic. Another routing table, the scope routing table, records scope-link pairs signifying in which directions brokers of the respective scope can be found.

Upon scope creation, an initially empty routing table is created at some broker, together with any management information regarding this scope, such as interface definitions. The creation is announced with a notification that is distributed in the network to update the scope routing tables. The overlay can either be extended manually by administrative commands to preset a certain extent of the overlay, or it is extended dynamically when other components are to join the scope. Both ways, a scope *join request* is always issued at a broker currently not part of the overlay. A request is traveling in the direction stored in the scope's routing table, leaving a temporary trail of references to the request source. The first broker encountered that is part of the requested scope, processes the join request and sends a reply back along the trail. If affirmative, the reply contains management information needed to set up routing tables in the involved brokers; they become part of the scope's overlay.

The transition of notifications between two scopes requires the two scope overlays to share at least one broker. Consider scopes T and R of Figure 2. T is a component of R and has joined R. For each subscription of T, a respective entry is added to the routing table of R that points to the table of T. For each advertisement an entry is added in

T's table that points to R. Mechanisms are in place to prevent multiple transitions at different brokers, but they are omitted here.

With this implementation, scopes not only group clients of the pub/sub service on the application level. They are also an important tool to group event brokers, extending their structuring capabilities to the infrastructure. They determine which subset of brokers belong to the same grouping and even allow for different routing algorithms in separate overlays as long as the transition between the scopes adhere to the constraints of the scope graph.

4 Security in Scopes

The preceding discussion introduced scopes as a means to group application and infrastructure components. They are therefore an apparent place to implement groups of trust. A scope isolates intra-scope traffic from the rest of the system, if the infrastructure is trusted. In Section 4.1 we address access control of clients, i.e., at the application level. Section 4.2 enhances scope overlay management to extend application-depend trust groups to the infrastructure.

4.1 Client Access Control

In many scenarios, like e-Commerce applications or mobile applications, access to the pub/sub infrastructure must be controlled on the level of subscriptions, advertisements, and publications, i.e., client access control. It must be ensured that only authorized clients have access to the network of brokers to publish and subscribe to notifications they are privileged to. In general, access control is implemented at the border brokers of a system, assuming a trusted infrastructure (cf. Section 4.2).

The presented solution uses rather simple policies because the main focus lies on how security is integrated—more sophisticated policies would be available if role-based access control schemes are bound to scopes, cf. [1]. *Attribute certificates* (AC) as specified in RFC 3281 [4] are utilized to encode privileges. An AC is a credential with a digitally signed (or certified) identity and a set of attributes. It carries here a reference to a public key certificate to authenticate the client and authorized filter expressions the client is allowed to advertise or subscribe for. ACs are issued by the provider of the broker network itself or by some other trusted *attribute authority* (AA). A legitimate content provider has got an AC from the network provider that authorizes its advertising. On the other hand, access to premium content may require an AC of the content provider, which is checked by the network.

Consider a service requesting the pub/sub system to propagate an advertisement A. To do so, it calls `advertise` of the pub/sub interface together with an AC showing its privilege to do so. The border broker verifies the AC by checking the contained signature of the network provider and, depending on the result of the check, grants access or, e.g., simply discards the advertisement. Included in a valid advertisement is another certificate carrying the public key of the content provider for this advertisement. Later, this certificate is used to authorize subscribers to the content published after the advertisement (e.g., the group of "premium content subscribers"). Advertisements are flooded through the overlay network of the scope they are published in. Thereby, access control information for subscriptions matching the advertisement is made available at all border brokers—overlay extensions are handled transparently as the network is trusted, so far.

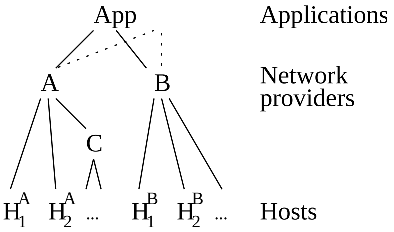


Figure 3. Trust relationships

When a client subscribes to some information at a border broker, it also gives its credentials in form of an AC. The border broker checks the signature of the certificate based on the network provider key or the keys contained in its list of received advertisements. The AC the client provides must match the attributes specified by the signing AA contained in the advertisement. If and only if they match, the subscription is processed further like in the standard pub/sub case without additional security.

4.2 Infrastructure Security

So far, we considered access control on the level of applications, i.e., in the graph of scopes. As discussed in Section 2, the trust relationship manifested in the application must also be secured within the infrastructure. Routing and the decision (how) to use specific parts of the broker network is subjected to application-specific security considerations. Scoping is exploited to correlate groups of application components with groups of trusted event brokers, making the scope overlay accessible to system engineers.

The previous assumption of having a homogeneous trusted network is relaxed and we first investigate when to extend a scope overlay. One can suppose large broker networks to be hosted by different service providers, like it is the case for the Internet. City carriers are likely to provide event brokers on wireless access points while global players link cities, countries, and continents. We assume a trust relationship as sketched in Fig. 3, e.g., negotiated in business contracts. Certificates authenticate hosts, their relation to providers, and the provider-application relation.

Connected Overlays. Assume that a scope overlay in a trusted network of brokers H_1^A, H_2^A, and H_3^A exists (Figure 4) and that a scope join request is received from a neighbor broker H_1^B, which is not yet part of the overlay. The decision whether the requesting, directly connected broker is trusted is application- and scope-specific. If positively ascertained, the implementation described in Section 3.3 is used for extending the scope overlay. A likely pre-installed policy of the network provider is to trust all the brokers within its own administrative domain.

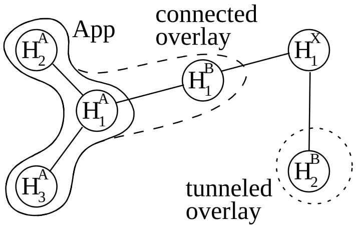


Figure 4. Extending the scope overlay

If a broker from a different administrative domain, say H_1^B, is asked to join the scope, it forwards a scope join request according to its scope routing table towards H_1^A. It appends to the request a list of chained attribute certificates of the path in the trust graph of Figure 3 from its node to the node of the respective scope. Upon receiving such a request, H_1^A tests the included certificates. If a shared ancestor in the trust relationships is found, extending the overlay may proceed as described. At this place, various security policies could be applied that are assigned to the scope *App* to govern its extension in the broker network. For instance, a scope might mandate link layer encryption with Transport Layer Security (TLS).

Tunneling. If a trustworthy node is about to join that is only reachable via an untrusted broker, the previous approach is not applicable. Consider a join request from a host H_2^B that is routed through an untrusted broker H_1^X. The latter is assumed to have a routing entry for the scope *App* in its scope routing table. H_2^B digitally signs the request and includes its own public key. If H_1^A accepts the request, the scope overlay would include an untrusted intermediary if the above implementation is used. The solution applied here is to *tunnel* the traffic through H_1^X. The clear text part of the reply contains an indication of whether to tunnel the scope and, if so, triggers an update of the scope routing table to include an entry pointing to H_2^B—provided that H_1^X cooperates. Notifications are encrypted and tagged with the scope's name so that they can be forwarded by H_1^X, although they are not part of its content-based routing. Eavesdropping and modifications are prevented, while malevolent omissions are detectable by application level heartbeats.

The tunnel can span more than one broker and it may even be used to connect clients via untrusted border brokers. The problem is, however, that multiple join requests lead to multiple tunnels. A second broker requesting to join the scope via H_1^X, or multiple clients connected via point-to-point tunnels at untrusted border brokers, will result in duplicated messages individually encrypted for the various destinations. At least in the former case of multiple trusted brokers behind an untrusted one, scope-level broadcast with a shared session key can attenuate bandwidth consumption. The same session key is forwarded to any newly attached broker so that the overlay connected via H_1^A-H_1^X is reached with only one message. Of course, this trades computing resources with bandwidth, for the new brokers have to filter out notifications consumed at other brokers.

The described tunneling is similar to secure (application-level) multicast, giving raise to the known problems of multicast key management [9]. Shared session keys must be changed if some brokers leave the overlay. However, if session keys are only used between brokers, it is plausible to assume that fluctuation is rather low and the frequency of key changes is limited.

5 Implementation

Clients access the REBECA notification service via *local event brokers*, which offer the plain pub/sub API as a library collocated to the client code. Local brokers maintain connections to the event broker network. There, event brokers are implemented as separate processes, which maintain TCP connections to other brokers and clients and at least one routing table for unscoped traffic. Brokers are customizable *software containers* and thus the implementation of the routing engine, connection pooling, transmission protocols and message handlers is specified at deployment of the broker. REBECA messages transmitted between brokers may contain (a) control messages, like subscriptions and scope admin messages, or (b) notifications, which consist of a management header and notification data. Appropriate message handlers process these messages according to their type (a) or (b).

Scope configuration is accessed through a remote management interface to the event broker functionality, using the Java Management Extensions (JMX) (cf. Figure 5). On creation of a scope, the desired scope parameters can either be specified directly, or via a *scope type*, which refers to a predefined configuration.

Although flexible, the current REBECA architecture does not allow for an easy inclusion of security policies. A number of core classes would have to be re-implemented for the integration of each specific kind of security handling. Furthermore, the proposed implementation of security is only partially tied to the sketched scope implementation – integrated routing in this case – and is designed to be applicable to other forms, as well.

```
interface ScopeEBIf :
    public RemoteEBInterface {
  createScope (ScopeName, IOInterface)
  createScope (ScopeName, IOInterface,
               SuperScopeName)
  createScope (ScopeName, Type)

  joinScope    (ComponentName, ScopeName)

  subscribe (ScopeName, Filter)
  ...
}
```

Figure 5. Remote event broker interface

To achieve greater flexibility, we employ aspect-oriented programming (AOP) techniques and AspectJ [2] to implement security aspects of scopes. We briefly sketch three main security extensions. First, access control on the API level is required for the authorization of the invocation of management functions. Certificates are stored with the callee and are transparently sent with each call to the remote management interface. These are verified before access to management functionality is granted by the broker. Second, some features, like admission tests of join requests, are also applicable to implementations other than integrated routing. Thus, a new function was introduced that calls a list of *interceptors* before starting to update the routing tables. Third, specific to integrated routing, extending the scope overlay must be governed by an authorization test of the original requesting broker and the next-hop broker. This test checks chains of certificates according to Figure 3 and is evaluated prior to calling the handler that processes the extension. Depending on the result a new *session key* may be generated. It must be added to all affected routing table entries by the extension handler and is used for secure message exchange between brokers over untrusted parts of the broker network. The encrypted fan-out to consumers uses point-to-point connections; in case of performance problems, caching schemes like [11] may be employed to reduce the number of encryptions needed.

6 Conclusion

Trust in publish/subscribe systems cannot be associated with specific producers and consumers without impairing their loose coupling. Instead, we have associated trust with the interaction within a group of components. This facilitates the design of loosely coupled applications and their security policies. Security is considered on the level of both applications and its implementation in the infrastructure, allowing for enforcement of security measures even across different administrative domains.

We introduced scopes as a suitable means to model and implement the above issues. Originally designed as general concept to control visibility, they make component interaction explicit. By opening up the black box of the pub/sub service, they provide for the appropriate locations to *weave* security aspects into a distributed pub/sub notification service. The separation of intra-scope traffic makes it possible to implement different security implementations that are bound to different parts of an application's structure, depending on the actual needs for security and trust. To avoid re-implementing larger parts of the pub/sub service every time the system evolves, we employed AOP technology to add the implementation to the REBECA notification service.

References

[1] A. Belokosztolszki, D. M. Eyers, P. R. Pietzuch, J. Bacon, and K. Moody. Role-based access control for publish/subscribe middleware architectures. In *Proc. of the 2nd International Workshop on Distributed Event-Based Systems (DEBS'03)*, June 2003. ACM Press.

[2] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Communications of the ACM*, 44(10):29–32, 2001. Special Issue on Aspect-Oriented Programming.

[3] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[4] S. Farrell and R. Housley. An internet attribute certificate profile for authorization. Request For Comment 3281 (RFC 3281), April 2002.

[5] L. Fiege, M. Mezini, G. Mühl, and A. P. Buchmann. Engineering event-based systems with scopes. In *Proc. of the ECOOP 2002*, *LNCS* 2374, Malaga, Spain, June 2002. Springer-Verlag.

[6] L. Fiege, G. Mühl, and F. C. Gärtner. Modular event-based systems. *The Knowledge Engineering Review*, 17(4):359–388, 2003.

[7] W. Harrison and H. Ossher. Subject-oriented programming (A critique of pure objects). In *Proc. of OOPSLA 1993*, 411–428, 1993.

[8] Z. Miklós. Towards an access control mechanism for wide-area publish/subscribe systems. In *Proc. of the 1st International Workshop on Distributed Event-Based Systems (DEBS'02)*, Vienna, Austria, 2002. IEEE Press.

[9] R. Molva and A. Pannetrat. Network security in the multicast framework. In *NETWORKING 2002 Tutorials*, *LNCS* 2497, pages 59–82, Pisa, Italy, 2002. Springer-Verlag.

[10] G. Mühl. *Large-Scale Content-Based Publish/Subscribe Systems*. PhD thesis, Darmstadt University of Technology, 2002. http://elib.tu-darmstadt.de/diss/000274/.

[11] L. Opyrchal and A. Prakash. Secure distribution of events in content-based publish subscribe systems. In *10th USENIX Security Symposium*, Aug. 2001.

[12] C. Wang, A. Carzaniga, D. Evans, and A. L. Wolf. Security issues and requirements for Internet-scale publish-subscribe systems. In *Proc. of the HICSS-35*, Big Island, Hawaii, Jan. 2002.

Secure “Selecticast” for Collaborative Intrusion Detection Systems

Philip Gross, Janak Parekh and Gail Kaiser
Columbia University
Programming Systems Lab
[phil|janak|kaiser]@cs.columbia.edu

Abstract

The problem domain of Collaborative Intrusion Detection Systems (CIDS) introduces distinctive data routing challenges, which we show are solvable through a sufficiently flexible publish-subscribe system. CIDS share intrusion detection data among organizations, usually to predict impending attacks earlier and more accurately, e.g., from Internet worms that tend to attack many sites at once. CIDS participants collect lists of suspect IP addresses, and want to be notified if others are suspicious of the same addresses. The matching must be done efficiently and anonymously, as most organizations are reluctant to share potentially revealing information about their networks. Alerts regarding external probes should only be visible to other CIDS participants experiencing probes from the same source(s). We term this type of simultaneous publish/subscribe “selecticast.” We present a potential solution using the secure Bloom filter data structure propagated over the MEET publish-subscribe framework.

1. Introduction

Increasingly, malicious attackers attempt to avoid tripping an Intrusion Detection System (IDS) by spacing out their probes over long periods of time, and interleaving probes among multiple sites over these extended periods [1]. Most current IDS’s have a limited memory window, or employ thresholding before signaling an alert in order to avoid generating a flood of false alerts. In contrast, Collaborative Intrusion Detection Systems (CIDS) attempt to correlate low-level alerts from multiple enclaves in the same organization or across organizations over a long period of time in order to detect these stealthy scanners prior to an actual attack. Participating sites submit lists of suspicious addresses, including those that fall below their own thresholds for escalating alerts (e.g., to human administrators). Each site compares others’ lists against its own, looking for matches, which are then escalated to a higher level of suspicion and monitoring.

One approach to implementing CIDS is a centralized repository that receives all the “watchlists” and sends back augmented watchlists to each participant. However, this creates a central point of failure and a tempting target for denial-of-service attacks. It also requires revealing all the data to the central site, but the organizations may only want to share with others that are targets of the same prospective attacker(s). The nature of the latter communication suggests a content-based messaging system, but in a context where participants (particularly when from independent institutions) may demand that their data be anonymized. That is, if participant A has no low-level alerts in common with participant B, it will learn nothing about participant B’s network structure or traffic from the data set. Only if there is an alert in common will the information of that particular alert, and only that alert, be revealed.

Such a peer-to-peer system should ensure that the data streams remain private, using some form of encryption. However, content-based messaging systems are inherently difficult to use for encrypted message streams. Content-based routing (CBR) implies that routers inspect the contents of every packet at each routing point, which is unacceptable if routers aren’t trusted. Encryption is not a problem in channel-based routing, since routers along the path don’t need to inspect content, and only privileged subscribers can decrypt the events. However, channels are not well suited for this problem domain, as each IP probe source would need its own channel, which is potentially a huge number. Encryption is also less of an issue if the content-based routers can be trusted, as each message could safely be decrypted and inspected within the router (although this is computationally expensive). Additionally, trusted routers could enforce access control policies (e.g., [2] or [3]) to guarantee that events are forwarded only to subscribers whose security credentials match those acceptable to the publishers.

However, ensuring that all intermediate routers are trustworthy will not generally be possible. We present a new approach that supports a restricted form of content-based event routing with only minimal trust required of routers. In particular, we trust the routers to forward messages to subscribers without forging, altering, or discarding them, and to implement our specialized algorithms correctly. Rogue routers that do attempt to interpret the events themselves for malicious purposes, that forward false matches, or forward to additional entities pose little or no threat, as explained below. The main restriction on our approach is that content filtering is limited to equality and inequality, as opposed to other comparisons on event content such as less than, greater than, substring of, etc.

Our main insight is to employ *Bloom filters* for representing hashed alert sets. A Bloom filter [4], described further below, is essentially a compact representation of a set of hashes. A router receives sets of hashed values from participants (publishers), which are then checked against others' (subscribers') Bloom filters. Matched values are sent back to the submitter and to the matched participants; we call this symmetric form of publish-subscribe "selecticast". The router then converts the submitted values to a much smaller Bloom filter, and discards the originals. There is a small possibility of false matches, which can be decreased by increasing the size of the Bloom filters and/or the number of hashes per input value [5]. For the CIDS application domain, false positives are not a major issue, assuming the rate is sufficiently low, as it merely implies that a small percentage of addresses will be temporarily flagged for closer monitoring despite being innocent.

We present one such system, including its methods for minimizing false matches. We first describe the collaborative intrusion detection system motivating this work. We then explain how Bloom filters operate and criteria for selecting the hashes. We compare several alternative approaches to CBR based on secure hashes. We briefly discuss the system's integration into a modular event system and its extension to a distributed implementation. Finally, we survey related work, and conclude with the current status and real-world evaluation plans.

2. Motivation

The current emphasis on security has led to the development and deployment of sophisticated traffic analysis tools, honeypots, and intrusion detection systems. A fundamental limitation of such systems is the single-point perspective on suspicious activity they offer: such activity can only be examined from the point of view of the sensor, which is attached to only one network. Patient attackers can slowly scan several targets in parallel, without creating enough traffic against any one of them to warrant an alert. Such low-frequency events can be easily lost in the sea of alerts generated by IDS's.

A *collaborative* intrusion detection system, such as described in [6], shares IDS alert information among sites within a large organization or across different organizations, thereby enriching the information available to each, and revealing far more detail about the behavior of attackers than would otherwise be possible.

Assume we already have a collection of sites, each hosting an IDS performing surveillance detection, i.e., tracking connections and failed or incomplete connection attempts, and mapping these activities to source IP addresses (or ranges of addresses) as much as memory permits. The Antura Recon detector is a commercial example of such a surveillance-detection-enabled IDS [7]. By identifying such sources, sites progress from detecting attacks as they occur to predicting impending attacks.

In CIDS, we correlate these alerts from IDS's across multiple sites. With sufficient participation in the collaboration, it now becomes possible to detect stealthy scanners who relatively rarely probe any given site but are slowly testing multiple target sites. Collaborative "watchlists" across multiple sites aggregate the activities of source IPs (or ranges of IPs) that would likely fall under the radar at any individual site. A critical concern, however, is that sites be able to participate without revealing confidential or sensitive information about their networks and traffic. By hashing the alerts before transmission and correlating and distributing them with selecticast, we can solve the problem simply and efficiently.

3. Secure Hashes and Bloom Filters

A Bloom filter [4] is a "one-way" data structure, consisting of a bit-string that represents hash hits. It is one-way in the sense that one can test to determine whether a given filter has seen a particular datum before, and the filter will answer with no false negatives and rare false positives. Thus, the Bloom filter does not reveal its contents; it can only confirm whether a specific value is stored.

More precisely, Bloom filters are used to probabilistically and compactly represent subsets of some universe U. A Bloom filter is implemented as an array of m bits, uses k hash functions mapping elements in U to $[0..m)$, and supports two basic operations: **add** and **query**. Initially, all bits in the Bloom filter are set to 0. To **add** $u \in U$ to a Bloom filter, the hash functions are used to generate k indices into the array and the corresponding bits are set to 1. A location can be set to 1 multiple times, but only the first has an effect. A **query** is positive if and only if all k referenced bits are 1. A negative query clearly indicates that the element is not in the Bloom filter, but a positive query may be due to a false positive: the case in which the queried element was not added to the Bloom filter, but all k queried bits are 1 due to other additions. We use k independent indices, instead of just a single index, to reduce the probability of a such a false positive.

The probability of false positives is an important metric because minimizing it is the key to making effective use of Bloom filters. The analysis proceeds as follows. If p is the probability that a random bit of the Bloom filter is 1, then the probability of a false positive is p^k, the probability that all k hash functions map to a 1. If we let i be the number of elements that have been added to the Bloom filter, then p = 1-(1-1/m)^{ik}, as ik bits were randomly selected, with probability 1/m, in the process of **add**ing i elements. In [8] and [5], it is shown that the probability of false positives is minimized when k is approximately $m/i \ln 2$.

The reason the above analysis is with respect to k, the number of hash functions, is that we do not control i, the number of additions, in our application – it's dictated by network traffic. What we can control is m, the amount of memory used and k, the number of hash functions. The collaborative security participants need to choose m and k so that the probability of false positives is acceptably minimized. Small values of k lead to large values of m, whereas small values of m lead to large values of k. The smaller m, the more compact our Bloom filters, but the smaller k, the faster the implementation – at routers as well as subscribers. As discussed in the next section, hash computation is the dominant cost and it depends on k.

4. CBR With Bloom Filters and Hashes

Bloom filters efficiently represent a set of hash values in a small space. Good results are typically obtainable with filters with only eight times as many bits as there are items being stored [5]. One can merge two or more Bloom filters by simply binary-ORing them together (at the cost of a higher false-positive rate).

Bloom filters also have some disadvantages, most significantly the impossibility of deletion, although alternate schemes allowing deletion (at the cost of less space efficiency) have been proposed [9]. Additionally, as with standard hash tables, the hash values have usually been adjusted modulo the size of the table, so increasing the size requires rehashing the original values.

We will assume that it is undesirable (insecure) for the routers to see raw, unhashed values, which will typically be alerts containing sensitive IP address and port information. There are several different ways to leverage Bloom filters to represent the hash-value sets participants are publishing. We could have clients submit Bloom filter bit-strings representing the hashed addresses of interest, and have routers use Bloom filters internally. We could have clients submit the sets of hash values and have routers organize them into Bloom filters. Or we could have clients submit lists of hash values and have the routers use standard hash tables.

There are a number of costs to consider, and the optimal solution will depend on the specific attributes of our network and the needs of our CIDS "selecticast." Among the important cost metrics are the size of the "selecticast" submissions and notifications in transit, the size of the subscription representations in router memory, and the speed with which the router can compute intersections. Another important variable is the specificity of participant notification: do participants merely need to know that others have seen a particular alert, the number who have seen the alert, or the actual identities of all who have seen an alert?

Plain Hash Tables

If clients submit lists of the hash values, the most straightforward approach for the router is to simply maintain a hash table of all submissions. Each entry in the hash table links to a list of submitters. When a new set is submitted, the router adds each entry to the master hash table. If the submitter list for that entry was non-empty, a notification containing the new submitter list and the hash value is sent to all entries in the list.

This implementation has the advantages that there will be no false positives except for rare hash collisions. It also allows deletion, enabling the submission of updates, as opposed to complete lists. If a hashed alert is flagged for deletion, the submitter will be silently deleted from the submitter list for that entry in the master hash table.

Its main disadvantage compared with Bloom filters is size. The exact size depends on the specific type of hash table constructed and the underlying architecture of the system (e.g., pointer size). Assuming both hash values and pointer size are 32 bits (reasonable for our expected alert set cardinality of 10^5-10^6), an optimal open hash table (load factor 0.50) storing n items will use around 64n bits, and an optimal chained hash table (load factor around 0.75) will use around 85n bits (64 bit entries, 32 bits for value and 32 for pointer, times 4/3 for optimal load). For either type, if each entry also has a linked list of submitters, add an extra 64n.

Pure Bloom Filters

At the opposite extreme, we could deal purely with Bloom filters. In this case, participants would submit a Bloom filter representation of their dataset. In order to look for matches, we would directly compare the Bloom filters, counting the number of bits in common. Filters with any matching elements must have at least k bits in common, where k is the number of Bloom filter bits we set for each input value.

Unfortunately, this approach will usually not be practical. The number of bits that match due to random chance will be huge for any typical pair of Bloom filters.

Suppose we have two Bloom filters of size *m* bits, each storing *n* distinct values (i.e., no values in common) using *k* bits per item. A bit in the filter will be set with probability *p*, roughly $1-(1-1/m)^{kn}$. For optimal Bloom filters, *p* should be near 0.5, although we can make it much sparser at the cost of increased memory usage.

For our candidate Bloom filter, we can view the case that one of our bits coincidently matches a bit in the other filter as a Bernoulli trial with chance of "success" *p*. The number of matching bits will then follow a binomial distribution, with the expected number of successes in *kn* trials equal to *knp*. Computer simulation confirms the accuracy of this analysis. *knp* will be vastly greater than *k*, and thus the number of false positives will be enormous. Even if we try to lower *knp* by making *p* extremely small (and making the filter extremely sparse), large values of *n* will rapidly make the situation unworkable.

For instance, if *k* is 6, and thus we want our expected number of collisions to be less than 6, we must put fewer than 1200 items into an 8 million bit (1MB) Bloom filter (k=6, n=1183, $m=2^{23}$, $p\approx 0.0008$, $knp\approx 6.0$). Note that this cost is only incurred in memory. We can compress the filter during network transmission by a large factor using standard compression tools. Nonetheless, and even given the speed of simply ANDing the two large Bloom filters together, 7000+ bits per item stored is a highly unattractive ratio.

Hybrid Bloom Filter

We can successfully leverage the size advantage of Bloom filters by combining them with the set-of-hash-values approach. Participants submit the list of hash values of interest, representing noted instances of suspicious network activity. The router uses the actual hash values to check against the Bloom filters of the other participants to find matches with reasonable accuracy. If matches are found, the router sends the matching values as a notification to all matching participants. Additionally, the router converts the submitted set of hash values into a Bloom filter of size $8\hat{n}$ bits, where $\hat{n}$ is the estimated total number of values per participant (making all filters the same size, and giving a chance of false positive around 2%). This filter is then stored and associated with the submitter. After all of the submitted hash values have been checked against everyone else's Bloom filters, the router can then discard the submitted hash value list, leaving only the subscriber's (much smaller) Bloom filter.

Thus publishers submit large sets of hash values, which are used to find matches, and then leave behind much smaller Bloom filter "residues" that act as subscriptions. Matching hash value sets (optionally tagged with the identity of their submitter) are sent out as the actual notifications. For this domain, we assume that the number of notifications is very small in relation to the number of values submitted (experiments show correlation rates of 0.01% or lower). If the number of matches is expected to be large, the matching sets could themselves be converted to Bloom filters before being sent as notifications, with the attendant space savings.

Note that the hash values used for Bloom filter generation are much larger than the hash values used by a plain hashtable, even though the resulting filter structure is smaller than the corresponding hashtable. For each item entered, Bloom filters need *k* indices into an *m*-bit table, and thus a total of $k\ln m$ bits of hash per item. If $m=8n$, then $k(3+\ln n)$ bits. For sets of 2,000 to 128,000 items and k=6, this works out to 84-120 bits per item, or a factor of 3-4 increase over the size of the hash values needed for plain hashtables. This would potentially be a problem for the submission of large sets of hashed values in the hybrid case.

However, we can avoid this problem by hashing our alerts to 32 bits for transmission, and then rehashing each to 120+ bits after it arrives at the server (and then splitting up those bits into the k indices of size ln*m* that we need), thus making the transmission cost no more expensive than for plain hashtables. Since the original alerts will typically contain less than 32 bits of entropy, no information should be lost with this two-stage hashing process.

Optimization with Two-Stage Compare

In the hybrid case described above, we assumed that the router maintains a separate Bloom filter for each of the *C* collaborating parties, representing the specific set of alerts seen by that party. When a new set of values is published, it must be compared against each of the *C-1* other sets. We can speed processing by entering all of the submitted value sets into a single large "master" Bloom filter in the Router and checking this first.

If we find a match in the master Bloom filter, we must then check each individual filter to discover the specific participants who matched. Despite this, we will show that this approach can offer substantial space efficiencies over the hash table approach, and the speed disadvantage can be reduced.

We can speed our Bloom filter lookups by taking advantage of arithmetic modulo 2^m on binary numbers. Just as a base 10 number modulo 10^m is the least significant *m* digits of the number, a binary number modulo 2^m is just the bottom *m* bits of the number, which can be extracted by ANDing the number with an appropriate bit mask (`(1<<m)-1` using the C-language bit operators).

Let *n'* be the power of 2 closest to *n*. We create one master Bloom filter of size *Cn'* and a filter of size *n'* for each of the *C* participants. Sizing these at 8 bits per item, we have a total space of $16Cn'$. The single hash table approach, as described above, will use $128Cn$-$150Cn$ bits to encode the same information. Even if we choose $n'=2n$, the total size of our Bloom filters will be a quarter or less of the size of the hash table solution. To do a lookup, we take our k hash values, compute indices modulo Cn', and do our lookups into the master Bloom filter. If all k indices match, we simply take the bottom $\log_2 n'$ bits of each master table index value, and use these as our search indices into the size n' subtables. Thus we only need to compute a nontrivial modulus once. If C is also a power of 2, both hash resizings become single AND instructions.

Aging

The master Bloom filter still has one major weakness vis-à-vis the master hash table solution. Participants will be publishing new alert lists to the network on a regular basis. While we can easily add new values to the master Bloom filter (just keep setting the appropriate bits), we have no way to delete out-of-date entries, and our master filter will gradually fill up with junk bits, until the probability of a positive response for any input approaches 1.

One solution is to maintain a "shadow" copy of the "primary" master Bloom filter (at a cost of an extra $8Cn'$ bits), and periodically swap the two. At startup, after the primary master Bloom filter is initialized from all the participants' data, the shadow copy is cleared. When participants subsequently publish a new set of values, their individual Bloom filter is replaced, and both the primary and shadow filters are updated with the new values. After all participants have submitted new data (or a preset time interval is elapsed), the shadow table becomes the new primary table, and the old primary table is cleared and becomes the new shadow table.

During the period where many new sets have been added to the current primary table, the number of false positives it returns will increase. However, as the individual participant subtables are always up-to-date, this should not result in a much higher rate of actual false positive messages transmitted back to subscribers, as all of the secondary checks for specific matches will fail. The only effect will be a reduction of the primary table's efficiency in filtering. If value set updates are largely similar to the previous set, the performance degradation will be even smaller.

MEET

The Multiply Extensible Event Transport (MEET) is a modular publish-subscribe system currently under development that allows users to define their own data types and predicates on those types to be used as filters. MEET allows enhancements to the classic publish-subscribe paradigm through the addition of new modules.

In the above discussion, we have assumed a single router node. We can use this extensible system implement a fully distributed solution. We wish to distribute the task of matching values among multiple routers. We can achieve even distribution of the computation by assigning particular hashes to particular routers with a mechanism based on Distributed Hash Table routing. For instance, if we have 16 routers, the first handles all hashed values ending with 0000, the second all hashes ending with 0001, etc.

MEET enables DHT routing and selecticast through the addition of data type, filter, and routing modules.Further discussion of MEET is outside the scope of this paper.

5. Related Work

A number of sophisticated publish-subscribe systems have been developed, including Siena [10], Gryphon [11], JEDI [12], ECho [13], CORBA Events [14], and Elvin [15]. Siena, Gryphon, JEDI, and Elvin are all content-based, where intermediate routers analyze the contents of each packet to determine appropriate forwarding destination(s). Wang et al. [16] examined security issues for CBR, but focused on the (as yet unsolved) problems of evaluating complex filters (i.e., more complex than simple equality testing, e.g., ad-hoc range checking) on encrypted data.

Bloom filters have been studied for a number of applications, including wide-area service discovery [17], IP packet traceback [18], and distributed caching services[8], in addition to being a primary tool for relational database joins [19]. The most germane work is probably by Triantafillou and Economides [20, 21], who use Bloom filters to create "subscription summaries," allowing for radical speedups to standard content-based routing. To our knowledge, no one has proposed using it for management of large numbers of opaque subscriptions in publish/subscribe systems. We also know of no other "selecticast" systems where publications are also subscriptions.

Others have investigated CIDS, e.g., [22] [23] [24] [25], but none have proposed an event infrastructure for data distribution.

6. Status and Conclusions

We believe that our proposed two-level system of Bloom filters will allow efficient and secure correlation of data as required by collaborative intrusion detection systems. The launch of a second generation CIDS, using MEET with Bloom filters extended as discussed here, is planned as a joint project between Columbia, Georgia Tech, Florida Institute of Technology, Syracuse

University, MIT, USC/ISI and the Brookings Institute. We expect to be able to compare our performance data against first generation CIDS trials involving Columbia, George Tech, and the University of Pennsylvania, where the raw data was collected and correlated at a centralized site.

Our extensions to Bloom filters may also prove useful for other secure content-based routing applications where equality/inequality testing of values is sufficient. Publications and subscriptions do not necessarily have to be symmetric, as in our "selecticast", but instead subscriptions could be provided directly as Bloom filters.

7. Acknowledgements

We would like to thank Sal Stolfo and the other members of his Intrusion Detection Systems Lab for developing the collaborative security framework motivating this event system research; Angelos Keromytis, Vishal Misra, Jason Nieh, Dan Rubenstein and Henning Schulzrinne for helpful suggestions; and the other members of our Programming Systems Lab. PSL is funded in part by National Science Foundation grants CCR-0203876, EIA-0202063 and EIA-0071954, and by Microsoft Research.

8. References

[1] CERT Coordination Center. *Module 4 - Types of Intruder Attacks*, *CERT/CC Overview Incident and Vulnerability Trends*. 2003.

[2] Blaze, M., J. Feigenbaum, and A.D. Keromytis. KeyNote: Trust Management for Public-Key Infrastructures. *Security Protocols International Workshop*. 1998. Springer LNCS.

[3] Keromytis, A.D., et al. The STRONGMAN Architecture. *3rd DARPA Information Survivability Conference and Exposition*. 2003.

[4] Bloom, B.H., Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 1970. 13(7): p. 422-426.

[5] Mitzenmacher, M., Compressed bloom filters. *IEEE/ACM Transactions on Networking*, 2002. 10(5): p. 604-612.

[6] Locasto, M., et al. *Secure and Efficient Privacy–Preserving Multi–Organization Intrusion Detection*, *Technical Report CUCS-012-04*. 2004.

[7] Robertson, S., et al. Surveillance Detection in High Bandwidth Environments. *3rd DARPA Information Survivability Conference and Exposition*. 2003.

[8] Fan, L., et al. Summary cache: a scalable wide-area Web cache sharing protocol. *SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. 1998.

[9] Cohen, S. and Y. Matias. Spectral bloom filters. *ACM SIGMOD International Conference on Management of Data*. 2003.

[10] Carzaniga, A., D.S. Rosenblum, and A.L. Wolf, Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 2001. 19(3): p. 332-383.

[11] Aguilera, M.K., et al. Matching events in a content-based subscription system. *18th Annual ACM Symposium on Principles of Distributed Computing*. 1999.

[12] Cugola, G., E. Di Nitto, and A. Fuggetta, The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, 2001. 27(9): p. 827-850.

[13] Eisenhauer, G., F.E. Bustamante, and K. Schwan. Event services for high performance computing. *9th International Symposium on High-Performance Distributed Computing*. 2000.

[14] Harrison, T.H., D.L. Levine, and D.C. Schmidt. The design and performance of a real-time CORBA event service. *ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages and Applications*. 1997.

[15] Segall, B., et al. Content based routing with Elvin. *Australian UNIX and Open Systems User Group*. 2000.

[16] Wang, C., et al. Security Issues and Requirements for Internet-scale Publish-Subscribe Systems. *35th Annual Hawaii International Conference on System Sciences*. 2002.

[17] Czerwinski, S.E., et al. An Architecture for a Secure Service Discovery Service. *Mobile Computing and Networking*. 1999.

[18] Snoeren, A.C. Hash-based IP traceback. *Conference on Applications, technologies, architectures, and protocols for computer communications*. 2001.

[19] Mackert, L.F. and G.M. Lohman. R* optimizer validation and performance evaluation for local queries. *ACM SIGMOD International Conference on Management of Data*. 1986.

[20] Triantafillou, P. and A. Economides. Subscription summaries for scalability and efficiency in publish/subscribe systems. *22nd International Conference on Distributed Computing Systems Workshops*. 2002.

[21] Triantafillou, P. and A. Economides. Subscription summarization: A new paradigm for efficient publish/subscribe systems. *24th International Conference on Distributed Computing Systems*. 2004.

[22] Yegneswaran, V., P. Barford, and S. Jha. Global Intrusion Detection in the DOMINO Overlay System. *11th Annual Network and Distributed System Security Symposium*. 2004.

[23] Markatos, E. *A European Network of Affined Honeypots*, Private communication, 2004.

[24] Balasubramaniyan, J.S., et al. An Architecture for Intrusion Detection Using Autonomous Agents. *Annual Computer Security Applications Conference*. 1998.

[25] Cuppens, F. and A. Miege. Alert correlation in a cooperative intrusion detection framework. *IEEE Symposium on Security and Privacy*. 2002.

Service-Oriented Event Correlation - Workflow and Information Modeling Approached

Andreas Hanemann, David Schmitz
Munich Network Management Team
Leibniz Supercomputing Center
Barer Str. 21, 80333 Munich, Germany
{hanemann, schmitz}@lrz.de

Abstract

The paradigm shift from device-oriented to service-oriented management has also implications to the area of event correlation. Today's event correlation mainly addresses the correlation of events as reported from management tools. However, a correlation of user trouble reports concerning services should also be performed. This is necessary to improve the resolution time and to reduce the effort for keeping the service agreements. We refer to such a type of correlation as service-oriented event correlation.

For introducing service-oriented event correlation for an IT service provider, an appropriate modeling of the workflow and of the information is necessary. Therefore, we examine the process management frameworks ITIL and eTOM for their contribution to the workflow modeling in this area. The MNM Service Model, which is a generic model for IT service management proposed by the MNM Team, is used to derive an appropriate information modeling. The different kinds of dependencies that we find in our general scenario are used to develop a workflow for the service-oriented event correlation.

1. Introduction

In huge networks a single fault can cause a burst of failure events. To handle the flood of events and to identify the root cause of a fault, event correlation approaches like rule-based reasoning, case-based reasoning or the codebook approach have been developed. The main idea of correlation is to condense and structure events to retrieve meaningful information. Until now, these approaches address primarily the correlation of events as reported from management tools or devices. We call these approaches resource-oriented (for an overview of the state-of-the-art see [1]).

As in today's IT environments the offering of services with an agreed service quality becomes more and more important, this change also affects the event correlation. To avoid service level agreement (SLA) violations, it is especially important for service providers to identify the root cause of a fault in a very short time, when trouble reports are received from customers or the provider's own service surveillance. We call the kind of event correlation for such a scenario *service-oriented* as it uses knowledge about services, service provisioning and SLAs. As we showed in [1] the following reasons for service-oriented event correlation can be identified:

Resolution time minimization: The time interval between the first symptom (recognized either by provider, network management tools, or customers) that a service does not perform properly and the verified fault repair needs to be minimized. This is especially needed with respect to SLAs.

Effort reduction: If several trouble reports are symptoms of the same fault, the fault processing should be performed only once and not several times. If e.g. the fault has been repaired, all affected customers should be informed about that automatically.

Impact analysis: In case of a fault in a resource, its influence on associated services and affected customers can be determined. This analysis can be performed for short term (when there is currently a resource failure) or long term (e.g. network optimization) considerations.

To receive the benefits of the service-oriented event correlation, it is necessary to have an appropriate information modeling, e.g. with respect to the dependencies from services to subservices and resources. The workflow also needs to be modeled to show which steps are necessary during the event correlation process.

The rest of the paper is organized as follows. In Section 2 we present the management models ITIL and eTOM and examine their contribution to the area of fault management and especially to event correlation and show that the MNM Service Model is useful as basis for the information modeling of service-oriented event correlation. Our workflow design for this kind of correlation as well as the derived information modeling for the service events are presented in Section 3. The last section concludes the paper and presents future work.

2 Usability of Existing Models for Service-Oriented Event Correlation

In the following we examine the established IT process management frameworks ITIL and eTOM. The aim is find out where event correlation can be found in the process structure and how detailed the frameworks currently are. This is helpful to model the workflow for the service-oriented event correlation.

2.1 ITIL

The British Office of Government Commerce (OGC) and the IT Service Management Forum (itSMF) [2] provide a collection of best practices for IT processes in the area of IT service management. The collection is called “IT Infrastructure Library (ITIL) [3]”. The service management is described by 11 modules which are grouped into Service Support Set (provider internal processes) and Service Delivery Set (processes at the customer-provider interface). Each module describes processes, functions, roles and responsibilities as well as necessary databases and interfaces. In general, ITIL describes contents, processes, and aims at a high abstraction level and contains no information about management architectures and tools.

The fault management is divided into Incident Management process and Problem Management process.

Incident Management: The Incident Management contains a service desk as interface to the customer (e.g. receives reports about service problems). In case of severe errors structured queries are transferred to the Problem Management.

Problem Management: The Problem Management’s tasks are to solve problems, take care of keeping priorities, minimize the reoccurrence of problems, and to provide management information. After receiving requests from the Incident Management the problem has to be identified and information about necessary countermeasures is transferred to the Change Management.

The ITIL processes describe only what has to be done, but contain no information how this can be actually performed. As a consequence, event correlation is not part of the modeling. The ITIL incidents could be regarded as input for the service-oriented event correlation, while the output could be used as a query to the ITIL Problem Management.

2.2 TOM/eTOM

The TeleManagement Forum (TMF) [4] is an international non-profit organization from service providers and suppliers in the area of telecommunications services. Similar to ITIL a process-oriented framework has been developed at first, but the framework was designed for a narrower focus, i.e. the market of information and communications service providers. A horizontal grouping into processes for customer care, service development & operations, network & systems management, and partner/supplier is performed. The vertical grouping (fulfillment, assurance, billing) reflects the service life cycle.

In the area of fault management three processes have been defined along the horizontal process grouping.

Problem Handling: The purpose of this process is to receive trouble reports from customers and to solve them by using the Service Problem Management. The aim is also to keep the customer informed about the current status of the trouble report processing as well as about the general network status (e.g. planned maintenance). It is also a task of this process to inform the QoS/SLA management about the impact of current errors on the SLAs.

Service Problem Management: In this process reports about customer-affecting service failures are received and transformed. Their root causes are identified and a problem solution or a temporary workaround is established. The task of the “Diagnose Problem” subprocess is to find the root cause of the problem by performing appropriate tests. Nothing is said how this can be done (e.g. no event correlation is mentioned).

Resource Trouble Management: A subprocess of the Resource Trouble Management is responsible for resource failure event analysis, alarm correlation & filtering, and failure event detection & reporting. Another subprocess is used to execute different tests to find a resource failure. There is also another subprocess which keeps track about the status of the trouble report processing. This subprocess is similar to the functionality of a trouble ticket system.

The process description in eTOM is not very detailed. It is useful to have a check list which aspects for these processes have to be taken into account, but there is no detailed modeling of the relationships and no methodology for applying the framework. Event correlation is only mentioned in the resource management, but it is not used in the service level.

2.3 MNM Service Model

The MNM Service Model [5], which was developed by the Munich Network Management Team, is a generic model for service modeling. It distinguishes between *customer side* and *provider side*. The customer side contains the basic roles *customer* and *user*, while the provider side contains the role *provider*. The provider makes the service available to the customer side. The service as a whole is divided into usage which is accessed by the role user and management which is used by the role customer.

The model consists of two main views. The *Service View* (see Fig. 1) shows a common perspective of the service for customer and provider. Everything that is only important for the realization of the service is not contained in this view. For these details another perspective, the *Realization View*, is defined (see Fig. 2).

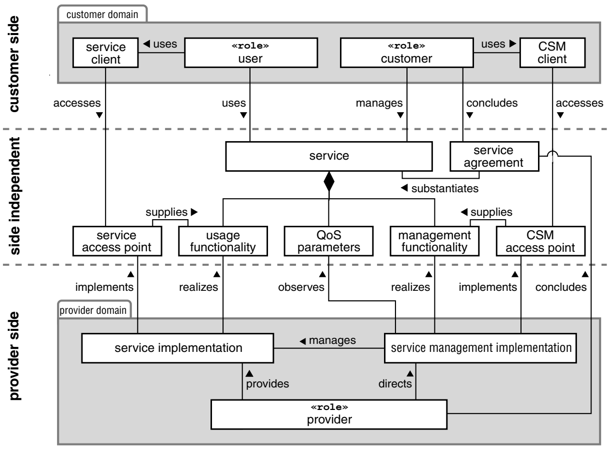


Figure 1. Service View

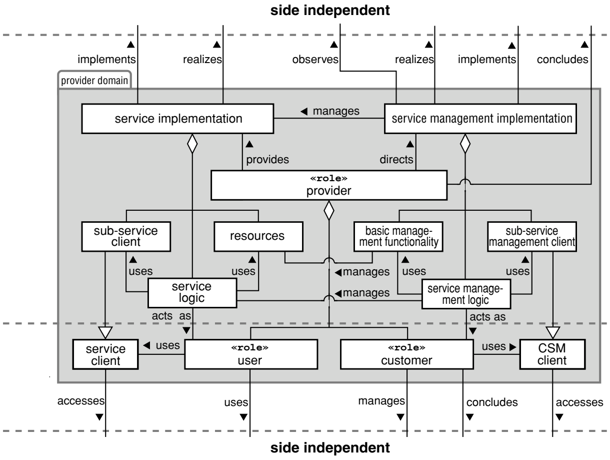


Figure 2. Realization View

The Service View contains the *service* for which the functionality is defined for management as well as for usage. There are two access points (service access point and CSM access point) where user and customer can access the usage and management functionality, respectively. Associated to each service is a list of QoS parameters which have to be met by the service at the service access point. The QoS surveillance is performed by the management.

In the Realization View the service implementation and the service management implementation are described in detail. For both there are provider-internal resources and subservices. For the service implementation a service logic uses internal resources (devices, knowledge, staff) and external subservices to provide the service. Analogous, the service management implementation includes a service management logic using basic management functionalities [6] and external management subservices.

The MNM Service Model can be used for a similar modeling of the used subservices, i.e. the model can be applied recursively.

As the service-oriented event correlation has to use dependencies of a service from subservices and resources the model is used in Subsection 3.4 to derive the needed information for service events.

3 Workflow and Information Modeling for Service-Oriented Event Correlation

Fig. 3 shows a general service scenario which we will use as basis for the workflow modeling for the service-oriented event correlation. The provider offers different services which depend on other services called subservices (service dependency). Another kind of dependency exists between services/subservices and resources. These dependencies are called resource dependencies. These two kinds of dependencies are in most cases not used for the event correlation performed today. This resource-oriented event correlation deals only with relationships on the resource level (e.g. network topology).

As both ITIL and eTOM contain no description how event correlation and especially service-oriented event correlation should actually be performed, we propose the following design for such a workflow (see Fig. 4). The workflow is divided into the three phases fault detection, fault diagnosis, and fault recovery. In general, we have two kinds of events: Resource events, which contain information about failures in resources, and service events, which contain information about service problems.

In the fault detection phase these events can be generated from different sources. The resource events are issued during the use of a resource, e.g. via SNMP traps. The service events are originated from customer trouble reports, which are reported via the Customer Service Management (see below) access point. In addition to these two “passive” ways

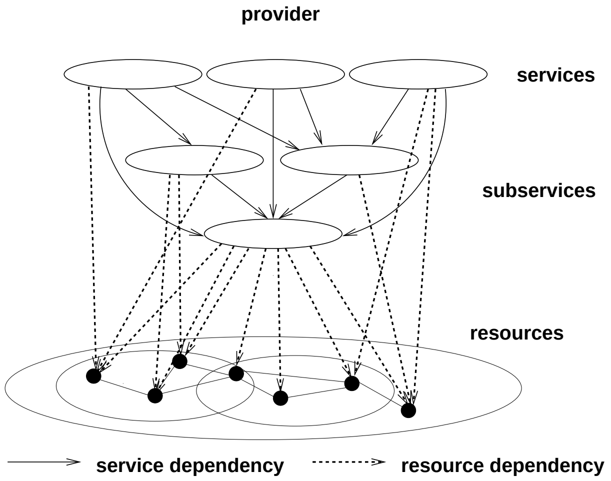


Figure 3. Different kinds of dependencies for the service-oriented event correlation

to get the events, a provider can also perform active tests. These tests can either deal with the resources (resource active probing) or can assume the role of a virtual customer and test a service or one of its subservices by performing interactions at the service access points (service active probing).

An important part of the fault diagnosis phase is the event correlation. The correlation contains the resource event correlator which can be regarded as the event correlator in today's commercial systems. Therefore, it deals only with resource events. The service event correlator does a correlation of the service events, while the aggregate event correlator performs a correlation of both resource and service events. If the correlation result in one of the correlation steps shall be improved, it is possible to go back to the fault detection phase and start the active probing to get additional events. These events can be helpful to confirm a correlation result or to reduce the list of possible root causes.

After the event correlation an ordered list of possible root causes is checked by the resource management. When the root cause is found, the failure repair begins. This last step is performed in the fault recovery phase.

The next subsections present different elements of the event correlation process.

3.1 Customer Service Management and Intelligent Assistant

The MNM Service Model contains a Customer Service Management (CSM) access point as a single interface between customer and provider. Its functionality is to provide information to the customer about his subscribed services, e.g. reports about the fulfillment of agreed SLAs. It can also be used to subscribe services or to allow the customer to manage his services in a restricted way. Reports about problems with a service can be sent to the customer via CSM.

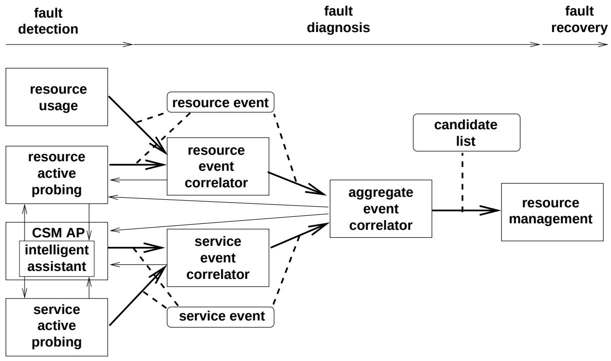


Figure 4. Event correlation workflow

To reduce the effort for the provider's first level support, an Intelligent Assistant can be added to the CSM. The Intelligent Assistant structures the customer's information about a service trouble. The information which is needed for a preclassification of the problem is gathered from a list of questions to the customer. The list is not static as the current question depends on the answers to prior questions or from the result of specific tests. A decision tree is used to structure the questions and tests. The tests allow the customer to gain a controlled access to the provider's management. At the LRZ a customer of the E-Mail Service can e.g. use the Intelligent Assistant to start a “ping” request to the mail server. But also more complex requests could be possible, e.g. requests of a combination of SNMP variables.

3.2 Active Probing

Active probing (see Fig. 5) is useful for the provider to check his offered services. The aim is to identify and react to problems before a customer notices them. The probing can be done from a customer point of view or by testing the resources which are part of the services. It can also be useful to perform tests of subservices (own subservices or subservices offered by suppliers).

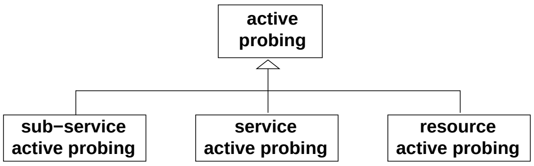


Figure 5. Active Probing

Different schedules are possible to perform the active probing. The provider could select to test important services and resources in regular time intervals. Other tests could be initiated by a user who traverses the decision tree of the Intelligent Assistant including active tests. Another possibility for the use of active probing is a request from the event correlator, if the current correlation result needs to be improved. The results of active probing are reported via service or resource events to the event correlator (or if the test was demanded by the Intelligent Assistant the result is reported to it, too). While the events that are received from management tools and customers denote negative events (something does not work), the events from active probing should also contain positive events for a better discrimination.

3.3 Event Correlator

Because we have to deal with two types of events (resource events and service events) in the service-oriented scenario, the event correlation should be performed in different steps. The reason for this are the different characteristics of the dependencies (see Fig. 3).

On the resource level there are only relationships between resources, e.g. caused by the network topology. An example for this could be a switch linking separate LANs. If the switch is down, events are reported that other network components which are behind the switch are also not reachable. When correlating these events it can be figured out that the switch is the likely error cause. At this stage, the integration of service events does not seem to be helpful. The result of this step is a list of resources which could be the problem's root cause. The resource event correlator is used to perform this step.

In the service-oriented scenario there are also service and resource dependencies. As next step in the event correlation process the service events should be correlated with each other using the service dependencies. The result of this step which is performed by the service event correlator is a list of services/subservices which could contain a failure in a resource. If e.g. there are service events from customers that the video conference service and e-mail service do not work and both services depend on a common subservice (in this case e.g. the DNS), it seems more likely that the resource failure can be found inside the subservice.

In the last step the aggregate event correlator matches the lists from resource event correlator and service event correlator to find the problems possible root cause. This is done by using the resource dependencies.

Fig. 6 shows the different event correlators.

3.4 Resource Events and Service Events

Today's event correlation deals mainly with events which are originated from resources. Beside a resource identifier these events contain information about the resource status, e.g. SNMP variables. To perform a service-oriented event correlation it is necessary to define events which are related to services. These events can be generated from the provider's own service surveillance or from customer reports at the CSM interface. They contain information about the problems with the agreed QoS. In our information modeling we define an event superclass which contains common attributes e.g. time stamp. Resource event and service event inherit from this superclass (see Fig. 7).

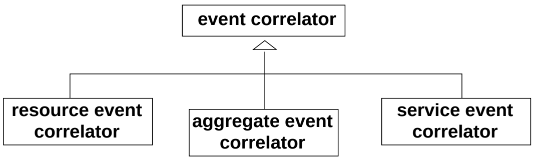


Figure 6. Event Correlators

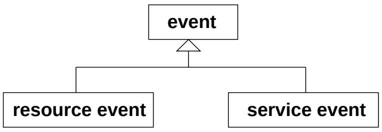


Figure 7. Events

Derived from the MNM Service Model we can define the information which is necessary for a service event.

Event description: This field has to contain a description of the problem. Depending on the interactions at the service access point (Service View) a classification of the problem into different categories should be defined. It should be possible to add an informal description of the problem.

Issuer's identification: This field can either contain an identification of the customer who reported the problem, an identification of a service provider's employee (in case the failure has been detected by the provider's own service active probing) or a link to a parent service event (see below). The identification is needed, if there are ambiguities in the service event or the issuer should be informed (e.g. that the service is available again). The possible issuers refer to the basic roles (customer, provider) in the Service Model.

Dates: This field contains key dates in the processing of the service event such as initial date, problem identification date, resolution date. These dates are important to keep track how quick the problems have been solved.

Status: This field represents the service events actual status (e.g. active, suspended, solved).

Priority: The priority shows which importance the service event has from the provider's perspective. The importance is derived from the service agreement, especially the agreed QoS parameters (Service View).

Assignee: To keep track of the processing the name and address of the provider's employee who is solving or solved the problem is also noted. This is a specialization of the provider role in the Service Model.

Service: As a service event shall represent the problems of a single service, a unique identification of the affected service is contained here.

QoS parameters: For each service QoS parameters (Service View) are defined between the provider and the customer. This field represents a list of these QoS parameters and agreed service levels. The list can help the provider to set the priority of a problem with respect to the service levels agreed.

Resource list: This list contains the resources (Realization View) which are needed to provide the service. This list is used by the provider to check if one of these resources causes the problem.

Subservice service event identification: In the service hierarchy (Realization View) the service for which this service event has been issued may depend on subservices. If there is a suspicion that one of these subservices causes the problem, child service events are issued from this service event for the subservices. In such a case this field contains links to the corresponding events.

Other event identifications: In the event correlation process the service event can be correlated with other service events or with resource events. This field then contains links to other events. This is useful to, e.g., send a common message to all affected customers when their subscribed services are available again.

The fields date, status, and other events are not derived directly from the Service Model, but are necessary for the service event correlation process.

4 Conclusion and Future Work

In our paper we showed the need for a service-oriented event correlation. For an IT service provider this new kind of event correlation makes it possible to automatically map problems with the current service quality onto resource failures. This helps to find the failure's root cause earlier and to reduce costs for SLA violations. In addition, customer reports can be linked together and therefore the processing effort can be reduced.

To receive these benefits we presented our approach for performing the service-oriented event correlation as well as a modeling of the necessary correlation information. In the future we are going to apply our workflow and information modeling for different services offered by the Leibniz Supercomputing Center.

Several issues have not been treated in detail so far, e.g. the consequences for the service-oriented event correlation if a subservice is offered by another provider. If a service of the provider does not work, it has to be determined whether this is caused by the provider himself or by the subservice. Another issue is the use of active probing in the event correlation process which can improve the result, but can also lead to delay in the correlation.

Acknowledgments

The authors wish to thank the members of the Munich Network Management (MNM) Team for helpful discussions and valuable comments on previous versions of the paper. The MNM Team, directed by Prof. Dr. Heinz-Gerd Hegering, is a group of researchers of the Munich Universities and the Leibniz Supercomputing Center of the Bavarian Academy of Sciences. Its Web server is located at `wwwmnmteam.informatik.uni-muenchen.de.`

References

[1] Hanemann, A. and Schmitz, D.: Why is Service-Orientation Necessary for Event Correlation? Proceedings of the DAIS/FMOODS PhD Student Workshop, Paris, France, November 2003.

[2] Websites: www.itil.co.uk and www.itsmf.com

[3] ITIL Service Delivery, ISBN 0113300174
ITIL Service Support, ISBN 0113300158
ITIL Planning to Implement Service Management, ISBN 011330014X

[4] Website of the TeleManagement Forum: www.tmforum.org

[5] Garschhammer, M., Hauck, R., Hegering, H.-G., Kempter, B., Langer, M., Nerb, M., Radisic, I., Rölle, H., and Schmidt, H.: Towards generic Service Management Concepts - A Service Model Based Approach. In: Pavlou, G., Anerousis, N., and Liotta, A. (eds.): Integrated Network Management, VII, pages 719-732, IEEE/IFIP, May 2001.

[6] Hegering, H.-G., Abeck, S., and Neumair, B.: Integrated Management of Networked Systems - Concepts, Architectures and their Operational Application. Morgan Kaufmann Publishers, ISBN 1-55860-571-1, 1999.

Towards an Architecture for Reasoning about Complex Event-Based Dynamic Situations

Gabriel Jakobson, John Buford
Altusys Corp, USA
jakobson,buford@altusys.com

Lundy Lewis
Southern New Hampshire University, USA
l.lewis@snhu.edu

Abstract

In this paper we are concerned with event-based situation analysis. Application areas include the understanding and awareness of complex unfolding scenarios such as homeland security threats and future battlespace engagements. The paper (i) discusses the differences between the environments/requirements for event-based management and situation management, (ii) presents an argument for an integration of an event correlation system and a situation awareness system, and (iii) proposes an integrated architecture that combines rule-based spatio-temporal event correlation and case-based reasoning for understanding and managing situations. In addition, the paper discusses the hard problem of the mutual influence between event management and situation awareness.

1. Introduction

Consider that one is driving an automobile down a country road. What does one attend to mentally when driving on a country road? One might be enjoying the scenery, listening to music, or simply cruising. However, if a deer runs across the road in front of the car then the situation changes. For a brief second, a new kind of situation presents itself – an unsettling event occurs that must be dealt with immediately. After the event is dealt with, where hopefully the deer and car don’t collide, the situation returns to normal.

Consider now that the country road leads into a four-lane street with the hustle and bustle of shops, restaurants, malls, stoplights, and heavy traffic. The driving situation changes once again. The driver attends to events such as changes in stoplights, brake lights, pedestrians, 491 Amherst St., the cheapest gas prices, and the like. Further, if one hears a siren nearby, one is likely to pull over – that is, the situation changes again.

The salient features of our driving story are (i) observations of events, (ii) dynamic situation awareness, and importantly (iii) the mutual influences between event observations and situation awareness. In the driving story, for example, it is easy to see that certain observations of events invoke certain situation templates, while at the same time certain situation templates lead to the expectation of certain event observations.

The focus of this paper is on analysis of dynamic situations. In particular, we are concerned with situations such as those encountered in the management of a battlespace, complex technological systems and processes, and real-time emergency situations in health care, homeland security, and other applications. These applications involve a large number of dynamic objects that change state in time and engage each other into fairly complex spatio-temporal relations. From the management viewpoint it is important to understand the situations in which these objects participate, to recognize emerging trends and potential threats, and to undertake protective actions that lead to predefined safe situations.

For example, in a tactical land/air battlespace, the operational units (troop formations, vehicles, weapon systems) maybe be under attack by multiple enemy sources, and thus the commanders need to know the changes in the battlespace situation, the direction, and the strength and severity of potential enemy actions [1]. Understanding dynamic battlefield situations is not a trivial task: it requires complex cognitive modeling, i.e. modeling the semantics of event perception, situation comprehension, and action projection [2]. These tasks should be undertaken in a dynamic environment, where events, actions, and situations follow the time constraints of the domain and the logic of temporal reasoning.

Modeling dynamic situations has been the research focus of several scientific disciplines, including operations research, ergonomics, psychology, and artificial intelligence. Most notably, John McCarthy introduced the notion of Situation Calculus in 1963, and several years later it was formalized by McCarthy and Patrick Hayes [3]. Informally, situations were considered as snapshots of the world at some time instant, while a strict formal theory was based on non-monotonic reasoning. Reiter proposed a formal situation calculus designed for action planning purposes, where situations were defined as a sequence of actions [4]. Reiter's situation calculus has been applied in areas such as agent programming and robotics [5,6]. Reiter's situation calculus has been extended with a fluent-based (i.e., situation dependent actions) approach to model long-term, autonomous services in ubiquitous telecommunication applications [7]. Several other approaches to dynamic situation modeling have been studied, including approaches based on Petri nets, finite state machines, and Bayesian networks [8].

Our approach to modeling and reasoning about dynamic situations is inspired by two AI disciplines in which the authors have been involved for several years: Real-Time Event Correlation (EC) [9] and Case-Based Reasoning (CBR) [10]. EC is a widely recognized approach for telecommunication network root cause fault analysis and CBR is an effective paradigm for reasoning and decision support in applications such as health care, diagnostics, and legal reasoning. In this approach we propose the paradigm of Dynamic Case Based Reasoning, where the notion of classical cases used in CBR is extended by dynamic capabilities based on EC technology. The dynamic cases are used as the fundamental units of constructing complex situations, where the dynamics of situations is driven by the events and event correlation functions, as shown in Figure 1.

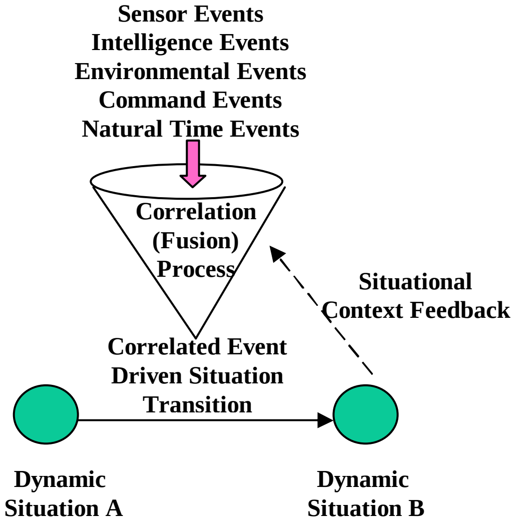


Figure 1. Correlated events and dynamic situations

Section 2 discusses the different features of dynamic event-based situation awareness and the traditional tasks of event management in communications networks. Section 3 discusses a conceptual framework for reasoning about situations. Section 4 discusses a design based on the integration of event correlation and case-based reasoning, and section 5 provides a conclusion and direction for future research.

2. Modeling complex event-based situations

Reasoning about complex event-based dynamic situations differs from the traditional task of analyzing a stream of events caused by faults in communications networks. For example, traditional event management systems are characterized as follows:

- The operational environment is formed from a large number, 100s to 1000s, of interconnected elements.
- The topology of the operational environment is defined by two main structural relations: connectivity and containment.
- The class relations are formed by studying information that is given or derivable from vendor documentation.
- The operational environment is largely static or with minor changes during the event analysis process.
- Network events are well-defined structured text messages with limited syntactic variations; heterogeneity of the events results from syntactic sugar-coding of internal vendor policies.
- There are strong causal relations between the events due to the propagation of faults through inter-connected network components.

The objectives of network event management include root-cause analysis, discovery of network performance degradations, and re-routing network traffic. Those tasks often require an execution of a relatively long chain of simple rules. Each step involves limited semantic processing and could be implemented by a rule-based system. A certain amount of temporal reasoning is required, but this is not a dominant feature. As a rule the events are fast flowing and require very fast processing -- during critical events this might reach processing several hundred to a thousand events per second.

In contrast, dynamic situation management such as battlespace management can be characterized thusly:

- The operational environment is formed by a medium number (hundreds, rarely thousands) of inter-connected elements.
- The operational environment is defined by complex temporal, spatial, and domain specific relations.
- The class relations are formed by multiple complex ontologies, which are usually ill defined.
- The operational environment is highly dynamic, and often unpredictable.
- Operational events are very diverse in nature and modality, involving signal, textual, visual and other types of information; a high level of heterogeneity is the norm; and the fusion of data, information, and knowledge is a required component of the operational space situation analysis.

- Causal relations are typically weaker (compared with network events); however there are very strong temporal and spatial relations.

The overall objectives in situation awareness are to understand the complexity of dynamic situations, discover potential developments, analyze potential threats and catastrophic situations, and undertake actions which avoid undesired situations. The realization of those tasks requires an execution of a relatively small number of complex reasoning steps involving the recognition of semantically rich event patterns.

The task of situation representation is a constructive task based on the principles of object-orientation and domain ontology [11]. Before defining the situations, let's introduce an object state as a set of object parameter values. Following the ideas of McCarthy, we consider situations as states, which have an assigned time value, either a point or an interval time. Time is the critical distinguishing factor between states and situations, e.g. two identical states with different time values represent two different situations. Considering dynamic situations, we are interested not only in the state value at some particular time, but also in the nature of situation changes, their speed, and directions. The dynamics of situations is reflected by state transitions, i.e. movement of an object from a state to state. Theoretically, it is possible to use the model of Finite State Machines (FSM) to describe these transitions; however, the simplicity of state specifications and augmentation of transitions with simple input/output variables make the FSM approach ineffective. Using cases for describing situations and using correlated events for determining situation transitions provides a more powerful tool for defining the dynamics of the situation changes.

The fundamental unit for representing a situation is a dynamic case: the smallest conceptual unit, which has a closed and semantically independent meaning in a domain. A tank at a specific location performing a specific task at a specific time is a situation.

Each situation is represented by its object specification, including:

- Identification
- Class specification
- Slots (parameters)
- Slot attributes
- Predicates and relations over the slot values
- Actions
- Time

In the following discussion we consider situations and cases as synonyms.

Abstract situation classes are organized into hierarchies according to the domain ontology. As usual, the specific situations are constructed by parametric instantiations of abstract situation classes. It will be unrealistic to represent complex situations with a single case. The representation of complex situations relies heavily on combination of situations using relations, e.g.:

- Spatial relations: s1 LOCATED_AT s2,
  s1 ABOVE s2, s1 CO-LOCATED s2,
  s1 NEAR s2
- Administrative relations: s1
  SUBORDINATE_TO s2, s1 DIRECTS s2
- Structural relations: s1 PART_OF s2,
  s1 CONNECTED_TO s2
- Other domain specific relations

While modeling dynamic situations, certain situations are identified as the start, target, undesirable, and transitional situations. One of the tasks, e.g. in dynamic battle-space situation modeling, is the identification of enemy threats and actions to avoid catastrophic or reach winning situations. The threats are considered as potential enemy hostile actions. In real situations the threats could be forces of nature, or unintentional actions.

An important component of transition of dynamic situations is the definition of the conditions of transitions. As discussed earlier, we will use event correlation technology to define these transitions.

3. A conceptual framework for reasoning about situations

Figure 2 shows the conceptual framework for reasoning about situations. The Dynamic Situation Model (DSM) represents the top-level model when one reasons about events and situations. To stress our point, consider the task of battle-space operations modeling. We distinguish between three levels of modeling of events and situations: the Operational Space Level, the Network Communication Level and the System Service Level. On the Operations Space Level the DSM describes the battle-space objects, relations, events, actions, and situations as described in Section 2. The dynamics of changes of situations is defined by event correlation functions over the multiple sources of sensor, intelligence, security and other information sources, as well operations management events such as commands and actions.

The actual information flow between different battle-space operational entities, such as humans, human groups, vehicles, weapon systems, infrastructure elements and other entities is conducted over the battlespace data communication networks. The Network Communication Level deals with Network Event Models that describe the traditional tasks of network fault, performance, and traffic event management. While modeling the battlefield situations, reasoning about potential threats, and planning battlespace management commands and

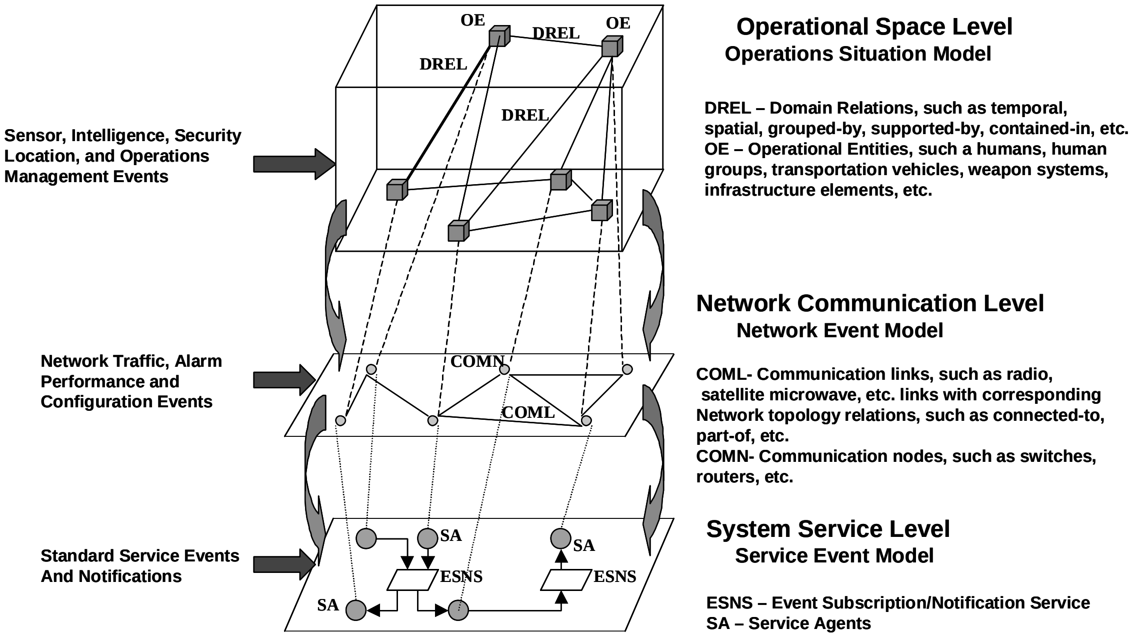

OE
DREL
OE
DREL
DREL
Operational Space Level
Operations Situation Model
DREL – Domain Relations, such as temporal, spatial, grouped-by, supported-by, contained-in, etc.
OE – Operational Entities, such a humans, human groups, transportation vehicles, weapon systems, infrastructure elements, etc.
Sensor, Intelligence, Security Location, and Operations Management Events
Network Communication Level
Network Event Model
Network Traffic, Alarm Performance and Configuration Events
COMN
COML
COML- Communication links, such as radio, satellite microwave, etc. links with corresponding Network topology relations, such as connected-to, part-of, etc.
COMN- Communication nodes, such as switches, routers, etc.
System Service Level
Service Event Model
Standard Service Events And Notifications
SA
SA
ESNS
ESNS
SA
ESNS – Event Subscription/Notification Service
SA – Service Agents

Figure 2. Modeling of dynamic event-based systems

actions, an important task is to map the events associated with the dynamic situation model to the events occurring in the corresponding network communications infrastructure. The overall task of such mapping is out of scope of this paper; however, we note that this mapping task is similar to the task of mapping events from Virtual Private Networks (VPNs) to the actual physical infrastructure networks.

The lowest level, the System Service Level, deals with software infrastructure objects, services, and events, which comprise the implementation architecture of the battlespace management platform. An example of such architecture has been described in [12]. It is built from standard CORBA services using structured CORBA events and the CORBA Event Notification Service. The CORBA Event Notification service was used as an event subscription mechanism and as a pipe to construct complex event passing channels.

4. Dynamic event-based situation awareness

4.1. Temporal Event Correlation

Event Correlation as a branch of Computer Science and Information Technology has been the focus of extensive research and practical applications over the last 15 years. The impetus for Event Correlation research was motivated by the fact that very often complex systems, being under operational stress, with malfunctioning of internal components, or being targets of malicious attacks, produce a large number alarms, which analyzed independently without recognizing the synergy between multiple information sources, do not reveal the actual internal situation of the system. In addition, large numbers of generated alarms might form chains of causally dependent events and mask the true root cause of the system failure. Due to the high speed of incoming events, the alarms may pass unnoticed or be noticed too late. Failure to capture the sequences of time-dependent events makes it hard to see the trends in the system of internal processes and predict the future behaviors of the system.

We will follow the event correlation model described in [9], where event correlation is broadly defined as a conceptual interpretation procedure of assigning a new meaning to a set of events that happen within a predefined time interval. This conceptual interpretation procedure stretches from a trivial task of event compression to a complex dynamic pattern-matching task involving Boolean functions, temporal relations and testing connectivity, containment and other structural relations between the objects that generate events.

In broad terms, an event is an act of internal transition of a system from a state to state, or in our area of interest – from a situation to a situation. The external manifestation of such an event is informational and as such it is an artifact created for human interpretation and practical utility. In more practical connotation, we consider an informational event as a time-stamped dynamic piece of information, which represents a change in the state of an object or manifests an action. Relative to the correlation process, we make a distinction between the raw (i.e. base) events and the derived (i.e. correlated) events. The raw events are external events originating outside the correlation process, while the derived events are results of a correlation process. The overall event correlation process is run by the Correlation Engine, which uses the structural, spatial and other constraints defined by the objects in the situation model, and the temporal relations existing between the events.

Temporal relations between events, such as x AFTER y, x ENDS_BEFORE y, x SIMULANEOUS_TO y, play a critical role in event correlation applied to dynamic situation analysis. Some classes of temporal event correlation in an interval time have been discussed in [9].

Each event correlation process has an assigned correlation time window -- a maximum time interval during which the component events should happen. The correlation process is started upon the arrival of the first component event and stopped as the last component event arrives. As any other event, correlation has its time of origination, time of termination, and lifespan. By definition, the time of origination of the correlation is equal to the time of origination of the last component event. Event correlation is a dynamic process, so the arrival of any event instantiates a new correlation time window for some correlation. Time is a critical factor in the correlation process. First, very often the correlation processes should follow "event floods", which might reach hundreds if not thousands events per second. Second, event correlation patterns should take into account temporal orders and relations between events. In addition, the physical latencies in the communication lines could distort the actual order of incoming events causing incorrect pattern matching.

4.2. A CBR interpretation of situations

What is needed is a way to model dynamic situations. Further, the model should allow (i) learning from experience and mistakes and (ii) adapting more-or-less classic standard situations to accommodate the nuances of current situations.

Our approach to these tasks is to use a model of cognition, case-based reasoning (CBR), where a case is a template for some generic situation [10]. A set of events is posed to the CBR system, whereupon four processes are carried out. First, the set of events is compared to a library of situation templates, and a set of maximally similar cases is retrieved by a Retrieve Module. In the CBR literature, a number of retrieval algorithms have been proposed. The simplest and weakest algorithm is key-term matching; the most complex but strongest algorithm is analogy-based matching [13, 14].

The case library can be thought of as a set of former experiences with situations that are potentially similar to the situation at hand. Typically a former situation has to be tweaked in some way to render it applicable to the nuances of a current situation. This is the task of an Adapt Module. In the CBR literature, a number of adaptation algorithms likewise have been proposed. Null adaptation, for example, covers those episodes wherein a past situation is exactly like a current situation; adaptation by substitution covers those episodes in which an object that occurs as a descriptor in the current situation should be substituted throughout for an object that occurs as a descriptor in the retrieved case [13].

An Execute Module is straightforward. The user may choose to execute a command or action recommended by the retrieved/adapted case. The execution may be conducted manually or may be carried out automatically by the decision-maker, either in supervised or unsupervised mode. The execution of an action or plan may involve cooperation with other individuals.

Importantly, the results of the execution are recorded in the case and the case is entered back into the case library. In most CBR systems, the case library is structured as a sequential list, much like a stack of paper forms. Of course, decision-makers do not structure their problem-solving knowledge in this way. There have been several proposals for more complex memory structures in the literature. An interesting proposal is the concept of a master case [15]. A master case is one in which all the problem-solving experiences with a particular, well-defined situation are subsumed in one case. This is in contrast with the sequential memory in which each problem-solving experience is confined to a unique case.

4.3. An integrated EC/CBR architecture

Figure 3 shows an integrated architecture with an event correlation engine at the front end and a CBR engine on the back end. Based on the preceding sections, the design of the conceptual architecture is straightforward. The new feature that Figure 3 emphasizes is the flow of information in both directions between the correlation engine and the CBR engine. The events issuing from the correlation engine are used to invoke cases, or situation templates, that serve as interpretations of multiple events. Figure 3 suggests two cases that might be hypotheses about a current situation. In reverse direction, a case might suggest further information which, if it were available, would strengthen a hypothesis. The CBR engine could send a message to the correlation engine whereupon the engine attempts to prove the truth or falsity of a proposition or try to find the value of a parameter. If that fails, the CBR engine could alert the command center of an opportunity for seeking out information that would strengthen a hypothesis.

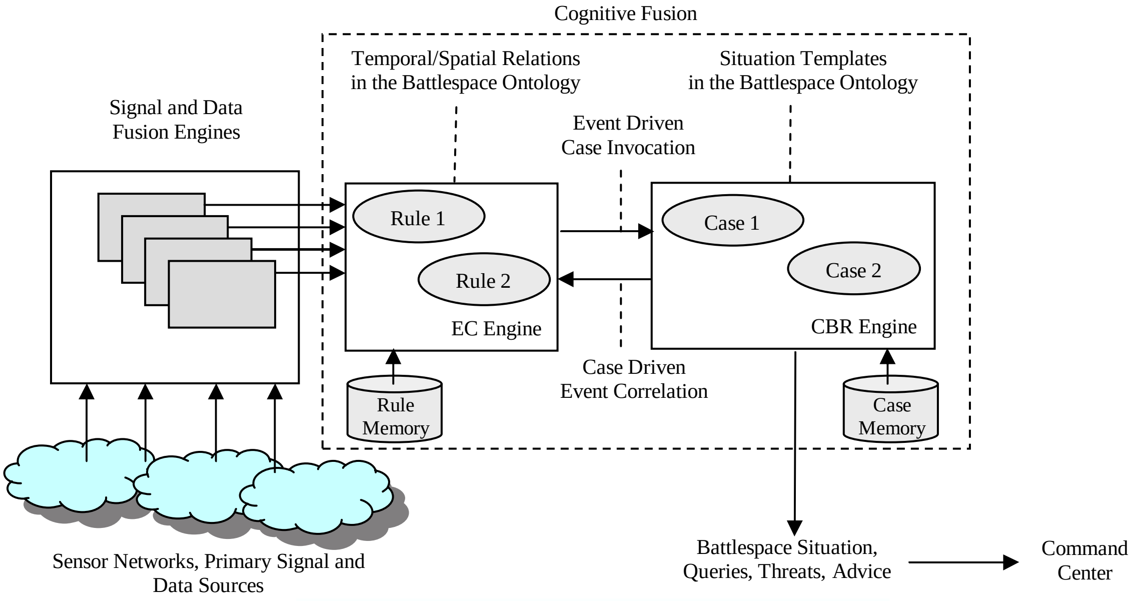

Cognitive Fusion
Temporal/Spatial Relations in the Battlespace Ontology
Situation Templates in the Battlespace Ontology
Signal and Data Fusion Engines
Event Driven Case Invocation
Rule 1
Rule 2
EC Engine
Case 1
Case 2
CBR Engine
Rule Memory
Case Driven Event Correlation
Case Memory
Sensor Networks, Primary Signal and Data Sources
Battlespace Situation, Queries, Threats, Advice
Command Center

Figure 3. An Integrated EC/CBR Architecture

5. Conclusions and further work

In this paper we discussed an architecture for dynamic event-based situation awareness. The contributions of the paper are (i) introducing dynamic cases, (ii) proposing the EC/CBR integration model, (iii) analyzing the specifics of dynamic situation awareness in comparison with traditional event management in communications networks, and (iv) introducing the multi-level model of dynamic event-based systems.

Further research includes a formal representation of the apparatus suitable for prototype building and testing. Some hard problems are (i) modeling unpredictable situations, (ii) developing algorithms for learning dynamic cases, (iii) developing a situation ontology, and (iv) choosing an appropriate implementation medium. The foundation for the implementation will be a distributed services architecture. The use of standard services and components with well-defined functionality and standard inter-component communication protocols allows the building of open, scalable, and customizable systems. Various technologies could be used for building the infrastructure of the EC/CBR system, e.g. CORBA, RMI, and Jini.

6. References

[1] A. Steinberg, C. Bowman, and F. White. Revisions to the JDL Data Fusion Model. *NATO IRIS Conference Proceedings*, Quebec, Canada, Oct. 1998.

[2] D. Rumelhart. The Architecture of Mind: a Connectionist Approach, in *Mind Readings*, MIT Press, 1998.

[3] J. McCarthy and P. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence, in *Machine Intelligence 4*, American Elsevier, 1969.

[4] F. Pierry and R. Reiter. Some Contributions to the Situation Calculus. *Journal of ACM*, 46(3), 325-364, 1999.

[5] Y. Lesperance, H. J. Levesque, et al. A Situation Calculus Approach to Modeling and Programming Agents, in *Foundations and Theories of Rational Agency*, Kluwer, New York, 1997.

[6] H. J. Levesque, R. Reiter, et al. GOLOG: A Logic Programming Language for Dynamic Domains. *Journal of Programming*, 31, 59-84, 1997.

[7] D, Wen-Yu, X. Ke, L. Meng-Xiang. A Situation Calculus-based Approach To Model Ubiquitous Applications, Cornell University e-print Archive, 2003.

[8] S. Mahoney and K. Laskey. Constructing Situation Specific Networks, in *Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence*, 1998

[9] G. Jakobson and M. Weissman. Real-Time Telecommunication Network Management: Extending Event Correlation with Temporal Constraints. *Integrated Network Management IV*, IEEE Press, 1995.

[10] L.Lewis. *Managing Computer Networks: A Case-Based Reasoning Approach*. Artech House, 1995.

[11] *D. McGuinness.* Ontologies and Online Commerce. *IEEE Intelligent Systems*, Vol.16, No 1, 2001.

[12] G. Jakobson, M. Weissman, L. Brenner, C. Lafond, and C. Matheus. GRACE: Building Next Generation Event Correlation Services, *2000 IEEE Network Operations and Management Symposium Proceedings*, April 2000.

[13] J. Kolodner. *Case-Based Reasoning*. M. Kaufman, 1993.

[14] D. Gentner. Structure-Mapping: A Theoretical Framework for Analogy. *Cognitive Science 7*, 1983.

[15] G. Dreo and R. Valta. Using Master Tickets as a Storage for Problem-Solving Expertise, in *Integrated Network Management IV*, Elsevier Publishers, 1995.

An Extended Publish/Subscribe Protocol for Transparent Subscriptions to Distributed Abstract State in Sensor-Driven Systems using Abstract Events

Eleftheria Katsiri[1], Jean Bacon[2] and Alan Mycroft[2]
[1] Laboratory for Communication Engineering, [2] Computer Laboratory,
University of Cambridge,
William Gates Building,
15 JJ Thomson Avenue, Cambridge CB3 0FD, UK
ek236@eng.cam.ac.uk, {Jean.Bacon,Alan.Mycroft}@cl.cam.ac.uk

Abstract

Event-based middleware is emerging as the major paradigm for large-scale and widely distributed systems, especially sensor-rich environments. Here, many primitive events are low-level and effort has been directed towards defining more meaningful composite events, which are typically recognised by finite-state machines. We argue that finite-state-machines (FSMs) are insufficient for meeting the requirements of users in Sentient Computing environments; user intuition may be concerned with notions such as state and negation which FSMs cannot support with reasonable efficiency. We aim to support querying and subscribing transparently to distributed state, which necessitates an alternative model for Sentient Computing. We propose a state-based, temporal first order logic (TFOL) model whose implementation is based on a deductive knowledge-base.

Furthermore, we propose a generalised notion of an event, an abstract event, which we define as a notification of transparent changes in distributed state. An extension to the publish/subscribe protocol is discussed, in which a higher-order service (Abstract Event Detection Service) publishes its interface; this service takes a TFOL abstract event definition as an argument and in return publishes an interface to a further service (an abstract event detector) which notifies transitions between the values true and false of the formula, thus providing a more natural and efficient interface to applications.

1. Introduction

Our research focuses on Sentient Computing [10] which is a computing paradigm for *context-awareness* in sensor-driven systems. Sentient Computing systems monitor the stimuli provided by environmental sensors in order to *notify* Sentient applications, *in real-time*, of changes that are relevant to the rapidly changing context of the user. An example of a Sentient application is one where the user can define his requirements in terms of *abstract knowledge predicates*. Such predicates represent states of entities that share abstract spatial and temporal properties, which are of interest to the user. The application then receives notifications from the Sentient Computing system when the defined knowledge has been acquired, and as a result, it delivers relevant reminders. E.g., a user may ask to be reminded to return John's book, whenever he is in the same room as John.

Current trends in programming paradigms in the area of context-awareness, such as *ubiquitous* and *pervasive computing*, advocate a separation of concerns between the way the user perceives knowledge about the physical world and the way knowledge is produced and maintained within Sentient Computing systems. The user's view is *abstract and state-based*, i.e., events are perceived as changes of state. The user's view should be *transparent*, i.e., the required application functionality should be available irrespectively of the heterogeneity of the underlying distributed modelling (in terms of the physical entities it contains) and the heterogeneity should be concealed, even when the user is mobile. E.g., locating the *closest, empty, meeting room* should be feasible, both when the user is walking in the Computer Laboratory and within Engineering. Furthermore, to accomodate user requirements, Sentient Computing environments need to be *dynamically extensible in real-time*. New user requirements need to be satisfiable without taking the system offline, or recompiling existing applications, even when entities are added or removed from the underlying model.

This paper argues that in context-aware, sensor-driven systems such as Sentient Computing systems, events in their traditional sense are an unsustainable metaphor. We introduce a generalised concept of an event, an *abstract event*, as a notification of changes of *abstract state* of model entities. Although we abide by the pub/sub protocol, we propose an extension to it, a higher-order service that allows *real-time dynamic extensibility*. This service allows for *abstract event detectors* to be created dynamically based on an abstract event definition in TFOL (Section 3.1). The abstract event detectors are implemented in accordance with our state model, by a deductive knowledge-base, which maintains the state of the entities of that component in an internal memory and *deduces* higher-level changes of abstract state, from changes in concrete state. This allows richer expressiveness in user requirements, as demonstrated in Section 7.

2. Deficiencies of Current Event Models

In event-based modelling, the state of an entity is represented by means of an *event history* entity, which is in turn implemented by finite state machines (FSMs). However, events in sensor-driven systems are primitive, they are not always *significant*, and they only convey *positive, concrete* knowledge. Although each received event may affect the truth property of high-level, abstract knowledge, unless all changes are explicitly *deduced*, the semantic mapping between concrete and abstract knowledge is incomplete. On the other hand, transparent reasoning with state requires both reasoning with negative (missing) abstract knowledge as well as concealing the details of the universe of discourse from the reasoning tool. It is well known that FSMs are limited in both aspects. This is discussed in detail next.

Negation in Transparent Reasoning with State. *There does not exist a finite-state machine which can accept the expression $\neg P$, when there is no instance of the predicate $\neg P$ available in the system as a symbol.* This means, that a symbol that corresponds to either P or $\neg P$ must be explicitly generated for the FSM to be able to process it. This problem is interlinked with reasoning transparently with *parametric* state. We demonstrate that known methodologies for parametric FSM-based reasoning [1, 8] are not directly applicable here. In those methods, a parametric expression is modelled with an FSM with free variables and for each free variable in the initial parametric FSM, an identical, non-parametric FSM is spawned, whenever a symbol that instantiates the free variable occurs at a given state. In the spawned FSM, all instances of the parameter that corresponds to the symbol which has been encountered, are substituted with the actual symbol. However, in constrast to an event that occurs in a deterministic way, a state can be activated and de-activated, in responce to any received event. Therefore, each state S of the parametric FSM that models P needs to have a transition to a state S' that models $\neg P$ and which cancels the execution. Unless $\neg P$ can be directly extracted from a primitive event, $\neg P$ does not exist as a symbol in the system. The *Closed World Assumption* can be used to determine a set of states $\{Q_i, i = 1, 2, \cdots\}$ where $\neg P$ can be assumed to hold. This is possible only if $\{Q_i, i = 1, 2, \cdots\}$ is a set of states that represent concrete or deduced knowledge. Even so, creating the transitions from S to $\{Q_i, i = 1, 2, \cdots\}$ requires knowledge of the underlying universe of discourse, which makes the reasoning *non transparent*. Fig. 1 illustrates this with an example. In Fig. 1(a), if an event occurs that corresponds to user a_1 being inside region r_1, then the automaton of Fig. 1(b) will be spawned from the one in (a). In this automaton, the transitions from s_1 to s_3 must represent all events that report user a_1 exiting from region r_1. This means that there must exist one transition for each region in the universe which is different from r_1. If instead user a_1 had moved into r_2, the automaton, which would need to be constructed as a result, would be that of Fig. 1(c).

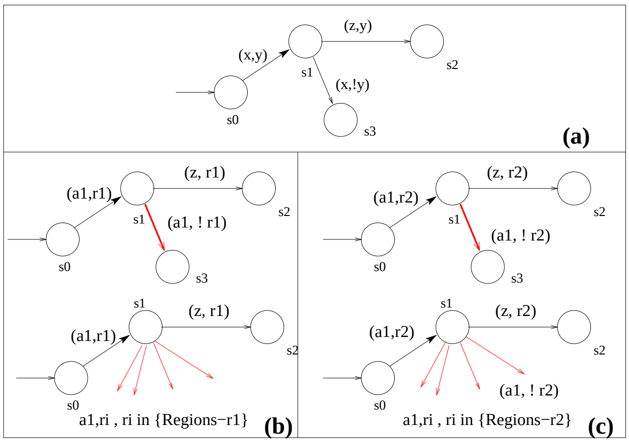


Figure 1. "Any two users are co-located".

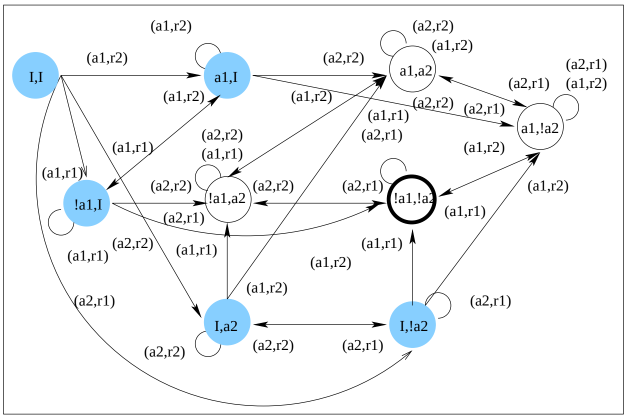


Figure 2. "Everybody is in r_2" with 2 users.

Uncertainty in Existential and Universal Quantification. The FSM of Fig. 2, models the expression "Everybody is in

r_2" in a world with two users a_1, a_2 and two regions r_1, r_2. Until an event for each user a_1 and a_2 has occurred, the system has only *partial knowledge*. States in grey are *uncertain*. Uncertainty is inherent in reasoning with expressions that signify existential and universal quantification (see Section 7).

Dynamic Extensibility. The Closed World Assumption often leads to state explosion. E.g., assuming that user a_3 is added to the previous closed world, the FSM of Fig. 2 would need to be re-compiled to that of Fig. 3. That renders dynamic extensibility unsustainable.

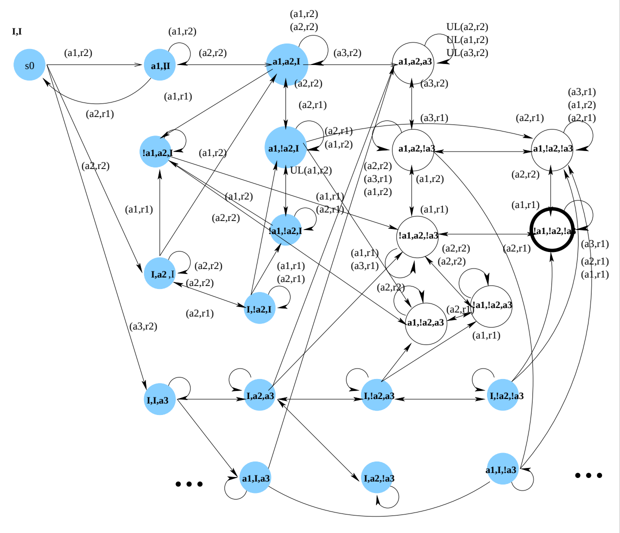


Figure 3. "Everybody is in r_2" with 3 users.

3. Abstract Events

Using the definitions and nomenclature proposed in [12] and extending the temporal notions of [11], we define a *state-based* model for distributed sensor-driven systems. We assume that a system consists of several physical *domains* such as an office domain. Each domain contains a set of physical objects such as mobile users and equipment. Modelled physical locations include *regions*, ranged over *rid*, and *positions (x,y,z)*. Each user *(u)* is associated with a *role* and each region is associated with a *regional attribute (rattr)* that describes relevant contextual information such as its functionality or ownership, e.g., kitchen, Alan's office etc. This allows the expression of *semantic abstractions* such as *the closest system administrator who is not busy.* Objects, locations, roles and regional attributes have *states* which are either concrete or abstract. Concrete state predicates represent state that is directly derived from the sensors and in this paper are always prefixed with *L_*(Low-level). Example of concrete state predicates are those that represent a user's position in terms of his co-ordinates, rooms in a building and nested locations such as floors. These are modelled [12] with the first-order logic predicates:[1]

$$L\_UserAtLocation(u, role, (x, y, z), timestamp)$$
$$L\_AtomicLocation(rid, rattr, polygon)$$
$$L\_NestedLocation(rid, rattr, rid\text{-}list)$$

E.g.,

$$L\_UserAtLocation(John, Phd, (13.45, 5.76, 1.75), 13{:}56)$$
$$L\_AtomicLocation(Room7, Meeting\text{-}room, Polygon37)$$
$$L\_NestedLocation(Floor4, [Room2, Room9])$$

A set of *function* predicates, as in Prolog, represent functions where the last variable represents the value of the function. The distance between a user's position and the centre of the polygon that defines a region is represented through the predicate $Distance(u, role, rid, rattr, v)$.

Abstract state predicates represent high-level state that is derived from concrete state by means of TFOL on properties of interest. A user's high-level location in terms of the region that contains him, a user's presence or absence and the fact that one or more people are co-located are examples of such predicates. Initially, when the system is started up, only concrete state predicates exist. Abstract state predicates in this paper are always prefixed with *H_*(High-level), $H\_UserAtLocation(u, role, rid, rattr, t)$.

A *timestamp* denotes the moment when a *current* abstract predicate instance becomes active. Previous instances are stored as *historical* predicates and are associated with an additional timestamp that denotes the moment when the predicate instance became inactive. Timestamps allow for *temporal abstractions* such as *now, today* and *temporal operations* such as sequence, iteration, equality and inequality with temporal intervals, over current and historical data.

Definition 1 *An abstract event is detected when an instance of an abstract state predicate which previously evaluates to true next evaluates to false and vice versa.*

3.1. Abstract Event Specification and Filtering

The *Abstract Event Specification Language (AESL)*[2] for creating abstract event definitions is a subset of TFOL that corresponds to Horn Clause Logic. *An abstract event definition* (AESL def) consists of one or more *implications* (Horn clauses). In case of only one implication, the RHS is the abstract predicate of interest. In case of more than one implication, the RHS of the last rule is the abstract predicate of interest.[3]

[1] As a convention, predicate attributes are in lower case and their values have the initial character capitalised, e.g., *polygon* is a variable whereas *Polygon27* denotes the value of the attribute *polygon* in a given co-ordinate system.

[2] AESL is a typed language, however types are ignored in this paper.

[3] As usual with these clauses all free variables are presumed to be implicitly universally quantified.

Example 1 *Locate the closest location to each user that has been empty for at least 5 min.*

Writing for brevity UL for $H\_UserAtLocation$, AL for $L\_AtomicLocation$, EL for $H\_EmptyLocation$, CL for $H\_ClosestLocation$, CEL for $H\_ClosestEmptyLocation$ and D for $Distance$:

$$\begin{aligned}
&(\nexists u\ UL(u, rid, role, rattr) \wedge AL(rid, rattr)\\
&\quad \Rightarrow EL(rid, rattr))\\
&D(u, role, rid_2, rattr_2, v_1) > D(u, role, rid_1, rattr_1, v_2)\\
&\quad \Rightarrow CL(u, role, rid_1, rattr_1)\\
&CL(u, role, rid, rattr) \wedge EL(rid, rattr)\\
&\wedge |EL(rid, rattr), CL(u, role, rid, rattr)|_{t>300}\\
&\quad \Rightarrow CEL(u, role, rid, rattr) \qquad (1)
\end{aligned}$$

Eq. $|EL(rid, rattr), CL(u, role, rid, rattr)|_{t>300}$ follows the design of [16] and extends the sequence operator in [11]. It is used to select a pair of EL and CL predicates whose temporal distance is greater than 300 sec.

Filtering. Horn Clause logic is used for defining filters during client subscription. Filtering is equivalent to selecting a subset of instances of a specific predicate by specifying a set of constraints on its attributes. A filter that selects only the instances in (1) which are meeting-rooms, in respect to a system administrator, is shown in (2). Note that we encourage AESL definitions to use variables as arguments for predicates rather than constants. This is done for implementation optimisation and it ensures that the deduction of the abstract predicate, which is computationally expensive is performed once, and multiple instances of the predicate are selected later on, by filters. Section 3.2 discusses the implementation of AESL defs and filters in more detail.

$$(rattr = \textit{Meeting-room}) \wedge (role = \textit{Sysadm}) \qquad (2)$$

3.2. Abstract Event Detectors

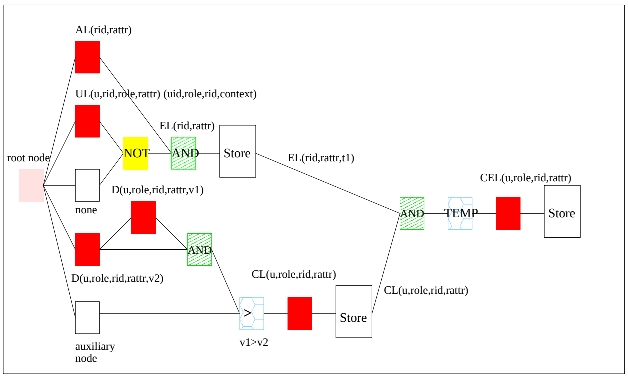


Figure 4. An abstract event detector for (1).

Each AESL definition is compiled into one or more abstract event (AE) detectors, which are structured as a deductive knowledge-base and which can perform semantic operations on instances of knowledge predicates that correspond to TFOL. They are implemented as Rete networks [7] and they consist of nodes and arcs. Every time a sensor creates a new instance of a concrete state predicate, corresponding tokens are created and propagated through the arcs to the nodes. Each node checks whether the received tokens correspond to a particular condition, e.g., if they are of class $H\_UserAtLocation$. It then forwards the tokens that satisfy the check on to the the child nodes. *Two-input* nodes concatenate tokens where shared variables are bound correctly. *Store* nodes, act as buffers for the current and historical instances of a predicate type and forward all stored instances on to the child nodes. This allows for temporal reasoning. When a token is forwarded to the final node, an instance of the abstract predicate that is being defined is created or deleted accordingly and an “activation” or “de-activation” abstract event is triggered, respectively. An abstract event detector for (1) is portrayed in Fig. 4.

Abstract event detection can be distributed so that each implication in an AESL def. is implemented by a separate detector, forming an hierarchical topology similar to SIENA [6] and HERMES [14]. The optimal placement of the detectors in the system can be determined as appropriate [16].

A filter is implemented as an AE detector with a linear complexity. Filters can be combined whenever there is a shared condition. E.g., the filter of (2) can be combined with that where $(rattr = \textit{M}\text{eeting-room}) \wedge (role = \textit{Ceo})$ as shown in Fig. 5.

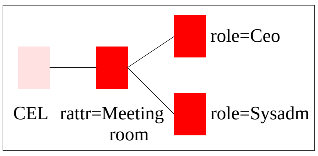


Figure 5. Filter combination.

4. An AED Service

We propose an extension to the pub/sub protocol, a *high-order service* which we call *Abstract Event Detection (AED) Service*, where subscribers do not just subscribe to event notifications as in the traditional form of this protocol, but to the establishment and configuration of an abstract event detector for a new abstract event of interest. The AED Service, acts as a mediator between the subscriber and the publisher and is responsible for detecting abstract events from primitive events. It interacts with publishers and subscribers using the pub/sub primitives *(Subscribe(), Notify())* according to the following extension to traditional pub/sub:

the AED Service publishes its interface using an event service such as the CORBA Notification Service, to all the subscribers and publishers of primitive events. Subscribers register their interest in subscribing to abstract event types of interest by subscribing at a dedicated *"AbstractEventSubscriptionListener"* (*AESListener()*) interface, at the AED Service. Each subscription carries the subscriber id, an AESL definition and a filter.

$$\textit{AEDListener}.\textbf{subscribe}(subscriberId, \textit{AESL}\ def, \textit{filter})$$

The AED Service uses the AEDListener (Fig. 6) to listen for subscriptions of the above type. For each received subscription, it checks in the *abstract event repository* whether an event type with the same name or AESL definition exists and if so, adds the subscriber to the list of subscribers for this event type. If it does not exist already, it registers the new event type with the underlying event service, so that the abstract event is available for filtering. The AESL definition is made available at the abstract event repository along with the new abstract event type. Next, the AESL definition and filter are extracted in order to construct an abstract event detector, that detects an abstract event of the requested type, as explained in Section 3.2. Using again the AESL definition, the primitive events of interest are selected and the underlying notification service is used for subscribing to the primitive events which are then translated appropriately and forwarded as inputs to the abstract event detector.

Each time an abstract event is detected, *Notify()* is invoked in order to publish the abstract event. *Notify()* publishes both the abstract event and the AESL def. This protects the service from malevolent event subscription in case of duplicate subscriptions to the same abstract event type with incorrect AESL defs. The *AEUListener()* listens to *Unsubscribe()* requests and removes the client that corresponds to the unsubscribe event from the notification list for that abstract event type.

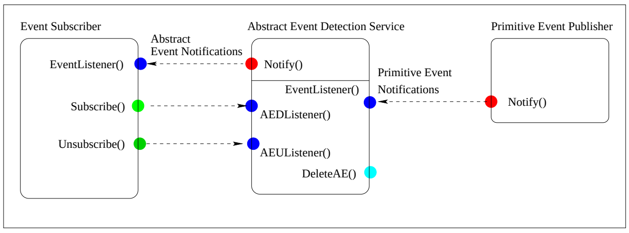


Figure 6. The AED Service.

The following call allows a subscriber to register interest in the events of type $H\_ClosestEmptyLocation$. The AESL definition in (1) and the filter in (2) are applied during subscription.

$$\textit{AEDListener}.\textbf{subscribe}(a, \Phi_1, \Phi_2)$$

Φ_1 and Φ_2 correspond to (1) and (2) respectively, where a is the subscriber's Interoperable Object Reference (IOR).

4.1. AED Service Properties

The AED Service can be distributed to implement distributed event detection (Section 3.2). It supports a garbage collection process for unused abstract predicates, when no subscribers are interested in a particular event type. This is implemented through the *DeleteAE()* interface. It also supports a satisfiability checking mechanism that ensures the correctness of the abstract event specifications e.g., conflicting requests are detected, such as "Whenever the system administrator is at a different room from the health and security officer, notify me", where the system administrator is the same person as the health and security officer.

5. Implementation and Scalability Analysis

A prototype scalability analysis for the example of the $H\_ClosestEmptyLocation$ in (1) compares the traditional pub/sub, where *instances* of the concrete state predicates $L\_AtomicLocation$, $H\_UserAtLocation$ and $Distance$ are available directly as primitive events, with the extended pub/sub, which uses the AED Service. We assume that the AED Service subscribes to the primitive events from the respective publishers and uses the AESL defs to create the abstract event detector of Fig. 4. Assuming an architecture that consists of k domains, each containing on average n users and m rooms and assuming that the user moves at a rate of $1\ event/sec$, then as long as an AED component is placed close enough to the primitive event sources, the overall event bandwidth generated from the above scenario can be reduced to $O(n) notifications/sec\ (per\ domain)$ vs. $O(n^2) notifications/sec\ (per\ domain)$ for the case of the traditional pub/sub. The overall, worst-case computational cost is $O(n^2)\ tests/sec$, using the dual-layer knowledge-base architecture of [12].

6. Related Work

There seems to be little attention to applying distributed systems theory to the special behaviour of ubiquitous sensor-driven networks. SIENA [6] is a distributed event architecture consisting of *event brokers*. It aims to optimise message traffic by deciding on the optimal places where message processing should take place and it takes advantage of overlapping subscription filters in order to reduce computational costs. HERMES [14] is similarly distributed and aims to improve the efficiency of the delivery of event notifications by using peer-to-peer routing techniques for creating overlay broker networks [15]. OASIS [4, 5] extends HERMES with *role-based access control*. Significant work has also been done in the area of *event composition* [2, 3, 13, 16, 17] i.e. the combination of primitive events into composite events by applying a set of *composition operators*. There, composite events are regarded as regular expressions with extensions for sequencing, concurrency and time.

The seminal notion of abstract events in sensor-driven systems appears to have originated in [3] and it is mentioned again later on in [2]. However, this early intuition is not directly applicable to distributed sensor-driven systems as it simulates the process of abstracting a region from a given position by means of a location service such as SPIRIT [9] and does not investigate further logical abstractions, such as TFOL reasoning. Our work allows the dynamic creation of abstract events in an asynchronous manner.

7. Conclusions and Future Work

Current event models have limitations in reasoning transparently with distributed state, which is due to the incomplete mapping between the application requirements in abstract knowledge and the concrete knowledge of the implementation domain, as well as the insufficiency of finite automata to deal with negation and quantification. In order to address these limitations we introduce the concept of abstract events as changes of abstract state and we describe a higher-order service that takes as an argument an abstract event definition and in return publishes an interface to a further service, an AE detector, which notifies transitions between the values *true* and *false* of the defined predicate. Detectors can be generated automatically. Because abstract events are high-order, the notification bandwidth is reduced. Furthermore, greater expressiveness in the subscription and querying language is achieved. Expressions that are implemented transparently by the proposed model include all expressions that negate knowledge from a source of context such as *not exists, nobody, nowhere, not seen, absent, empty, idle* as well as those that quantify abstract and concrete context and model entities: *somebody, somewhere, anybody, everybody, everywhere, exists, for all* ⟨*context*⟩, *there is* ⟨*context*⟩. We have created a prototype implementation of the proposed system. Our system can be applied to stored events as well. Future work involves addressing the lack of global time appropriately e.g., interval timestamps have been used in [16]. Future work will also consider sensor failure which is a common source of uncertainty in sensor-driven systems, as well as extending the reasoning beyond boolean logic.

References

[1] J. Bacon, J. Bates, R. Hayton, and K. Moody. Using Events to Build Distributed Application. In *2nd International Workshop on Services in Distributed and Network Environments, Whistler, British Columbia, Canada,* June 1995.

[2] J. Bacon and K. Moody. Toward Open, Secure, Widely Distributed Services. *Communications of the ACM*, 45(6), June 2002.

[3] J. Bacon, K. Moody, J. Bates, R. Hayton, C. Ma, A. M. Neil, O. Seidel, and M. Spiteri. Generic Support for Distributed Applications. *IEEE Computer*, 33(3):68–76, Mar. 2000.

[4] J. Bacon, K. Moody, and W. Yao. A Model of OASIS Role-Based Access Control and its Support for Active Security. *ACM Transactions on Information and System Security (TISEC)*, 5(4):492–540, Nov. 2002.

[5] A. Belokosztolszki, D. Eyers, P. Pietzuch, J. Bacon, and K. Moody. Role-Based Access Control for Publish/Subscribe Middleware Architectures. In *International Workshop on Distributed Event-Based Systems (DEBS03), San Diego, CA*, June 2003.

[6] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, Aug. 2001.

[7] C. L. Forgy. RETE: A fast Algorithm for the Many Pattern/ Many Object Pattern Match Problem. *Artificial Intelligence*, 19:17–37, 1982.

[8] N. Gehani, H. V. Jagadish, and O. Shmueli. Composite Event Specification in Active Databases: Model & Implementation. In *18th VLDB Conference, Vancouver, British Columbia, Canada*, Aug. 1992.

[9] A. Harter, A. Hopper, P. Steggles, A. Ward, and P. Webster. The Anatomy of a Context-Aware Application. In *Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking, Seattle, WA*, pages 15–19, Aug. 1999.

[10] A. Hopper. The Royal Society Clifford Paterson Lecture: Sentient Computing, 1999.

[11] D. Ipina and E. Katsiri. A Rule-Matching Service for Simpler Develpment of Reactive Applications. In *IEEE Distributed Systems Online*, Nov. 2001.

[12] E. Katsiri and A. Mycroft. Knowledge-Representation and Abstract Reasoning for Sentient Computing. In *Workshop on Challenges and Novel Applications of Automated Reasoning, CADE-19, Miami Beach, FL*, July 2003.

[13] M. Mansouri-Samani and M. Sloman. GEM: A Generalised Event Monitoring Language for Distributed Systems. *Distributed Systems Engineering Journal*, 4(2), June 1997.

[14] P. Pietzuch and J. Bacon. HERMES: A Distributed Event-Based Middleware Architecture. In *the First International Workshop on Distributed Event-Based Systems (DEBS02)*, pages 611–618, July 2002.

[15] P. Pietzuch and J. Bacon. Peer-to-Peer Overlay Broker Networks in an Event-Based Middleware. In *International Workshop on Distributed Event-Based Systems (DEBS03), San Diego, CA*, June 2003.

[16] P. Pietzuch, B. Shand, and J. Bacon. Composite Event Detection as a Generic Middleware Extension. *IEEE Network Magazine, Special Issue on Middleware Technologies for Future Communication Networks*, Jan. 2004.

[17] M. Spiteri. *An architecture for the Notification Storage and Retrieval of Events*. PhD thesis, University of Cambridge, 2000.

Providing content-based services in a peer-to-peer environment

Ginger Perng, Chenxi Wang, Michael K. Reiter
Carnegie Mellon University
Pittsburgh, PA, USA
gperng@ece.cmu.edu, chenxi@cmu.edu, reiter@cmu.edu

Abstract

Information dissemination in wide area networks has recently garnered much attention. Two differing models, publish/subscribe and rendezvous-based multicast atop overlay networks, have emerged as the two leading approaches for this goal. Event-based publish/subscribe supports content-based services with powerful filtering capabilities, while peer-to-peer rendezvous-based services allow for efficient communication in a dynamic network infrastructure. We describe Reach*, a system that integrates these two approaches to provide efficient and scalable content-based services in a dynamic network setting.*

1 Introduction

Event-based publish/subscribe (pub/sub) services have been widely studied as a basis for information dissemination in large networks (e.g., [3, 4, 6]). In this communication model, a *subscriber* registers a long-standing *subscription* to the pub/sub service and receives published messages that match that subscription. Pub/sub services are typically supported by a fixed infrastructure of routers that disseminate subscriptions and forward messages based on their *content* rather than the addresses of the subscribers. Current pub/sub systems are capable of supporting rich subscription languages and provide powerful content-based services.

A different model for information dissemination is rendezvous-based communication services layered atop structured peer-to-peer overlays (e.g., as implemented using Distributed Hash Tables (DHTs))). Motivated by a separate set of goals, the DHT-based information services typically do not support content-based routing but instead focus on providing efficient communication in a highly dynamic network where servers can join and leave at will. In this model, overlay nodes serve as logical meeting points for senders and receivers to exchange information. The overlay layer is responsible for efficiently locating the corresponding rendezvous node for a particular message and forwarding the message to interested subscribers (e.g., [5, 9, 13, 15]).

The pub/sub and rendezvous models present dramatically different approaches to information dissemination, each with its advantages. The pub/sub models exploit information content for routing, which supports highly expressive subscription and filtering capabilities. In contrast, peer-to-peer rendezvous-based services—notably DHTs—route in a way that ignores content and thus provides limited information services. However, the rendezvous-based services are built on top of peer-to-peer networks which have the ability to adapt dynamically via members joining and leaving the service. This permits a degree of flexibility and resource sharing that conventional pub/sub systems do not exploit.

Despite these contrasts, we argue that peer-to-peer rendezvous-based services and pub/sub communication can be integrated in such a way that gains the advantages of both. Our goal in this paper is to examine using rendezvous-based communication as a primitive for implementing pub/sub services. Specifically, we describe the design of a system we call *Reach*, which supports an important class of content-based services—namely content-based multicast—atop a peer-to-peer service, using the rendezvous abstraction. At a high level, the rendezvous service is the means by which subscriptions are stored in the network (like *triggers* in [12]), and by which published messages are directed to "find" the subscriptions they match. This rendezvous node is then an entry point into a "subset tree" of nodes hosting other, weaker (more general) subscriptions, and thus to which this message should also be routed. This tree is implemented in such a way that it offers join-and-leave flexibility and maximum efficiency as the nodes in the tree are nearby neighbors in the overlay.

We describe the basics of our design in Section 2 and 3, discuss research issues in Section 4, present related work in Section 5, and conclude in Section 6.

2 System Model and Design

In Reach, we assume that the universe of information content is characterized according to n enumerated attributes. Each distinct event (publication) bears an n-bit identifier, where a "1" in the i-th bit indicates that the event pertains to the i-th attribute; we denote the identifier for event e by id_e. Subscriptions in Reach are expressed in the same manner—each subscription contains an n-bit identifier in which each bit set indicates an interest in the corresponding attribute. A subscription indicates a conjunction of the indicated interests, i.e., an event matches a subscription if for every set bit in the subscription, the bit in the event identifier is set, as well.

This encoding scheme defines an identifier hierarchy — id_1 is said to be a parent of id_2 if and only if $id_1 \supset id_2$. For example, as shown in Figure 1, identifier 0011 is a parent for both 0001 and 0010 . More intuitively, a parent identifier contains at least all the attributes of a child identifier. This hierarchy is a fundamental concept in Reach and is the basis for content-based multicasting. The precise benefits of this hierarchy will become clear in Section 3.

Reach consists of a network of overlay nodes. Each node serves as a rendezvous point for some subscriptions and messages. They are also responsible for forwarding messages to interested clients. We say that a node i is the rendezvous node for message e if i hosts id_e (the same is true for subscriptions). For illustration purposes, we first discuss how identifiers are mapped onto physical nodes when the network size is a power of two. The other cases are discussed in Section 3.3. Recall that n is the size of the attribute space. Assume that we have a network with 2^m nodes, $m \leq n$, where each node can be identified by an m-bit string (we call this the node ID). An identifier, id, is mapped to node i if the lower-order m-bits of id coincide with the node ID of i. For example, node 01 in a network of four nodes would host identifiers 0001, 0101, 1001, and 1101 from an 4-bit attribute space. For convenience, we denote that node 01 hosts **01.

Messages and subscriptions entering Reach are routed to their designated rendezvous node. A subscription is stored at its rendezvous node where messages are matched to the subscriptions. Once a match occurs, the message is forwarded to the interested client. A message e is said to match a subscription f if and only if $id_e \supseteq id_f$. Note that not all subscriptions f's satisfying $id_e \supseteq id_f$ are located on the same rendezvous node. Therefore, the challenge is to efficiently locate all nodes that host f's such that $id_e \supseteq id_f$. The routing and forwarding algorithms described in Section 3 are designed specifically with this goal in mind.

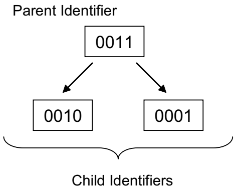


Figure 1. Parent/child relationship.

3 Routing and Content-Based Multicasting

Reach is a rendezvous-based network that supports content-based multicast. To achieve this, Reach uses a unique peer-to-peer lookup service that maintains high-level semantic relations within the overlay. We begin with a description of Reach's basic peer-to-peer routing infrastructure. We then describe content-based multicasting on top of this infrastructure. In Section 3.3 we discuss how Reach handles dynamically changing network sizes. We start our discussions assuming a power of two network size; this restriction is later relaxed.

3.1 Point-to-Point Routing

In Reach, we use a Hamming-distance based routing scheme. More specifically, each Reach node maintains a routing table that contains the addresses of all the nodes whose identifiers are one Hamming distance away from its own. For instance, node 1011 would have a neighbor set containing 0011, 1001, 1010, and 1111. In a network of 2^m nodes, each Reach node has a routing table with m entries. We note that the neighboring relationship in Reach is in the overlay layer and therefore is entirely logical.

The Reach routing algorithm is straightforward: a message is incrementally forwarded to its destination in such a fashion that each hop puts the message one Hamming distance closer to its destination. Consequently, the number of overlay hops between a pair of nodes is exactly the Hamming distance between their respective node IDs.

For a network of 2^m nodes, the average point-to-point path length in Reach is $O(m)$. This is similar to other peer-to-peer lookup services such as [13, 15, 10]. However, we stress that our contribution is not in Reach's ability to act as a routing infrastructure, but rather in its ability to support content-based multicasting.

3.2 Content-based Multicasting

To support content-based multicasting, Reach must locate all nodes who host subscriptions that are subsets of the event identifier of a particular message. For example, a message with identifier 1001 needs to reach the node hosting 1001, as well as nodes hosting 1000 and 0001. To achieve this, our high-level strategy is simple: each event message that enters Reach is first routed to its rendezvous node (1001 in our example). From there the event is progressively forwarded to nodes that host subsets of the event identifier (1000 and 0001). Each traversed node forwards the event to its subscribers if matching subscriptions are found. Routing toward the rendezvous point is simply a point-to-point overlay communication that follows the algorithm described in Section 3.1. Discussions in this section focus on the post-rendezvous dissemination operations. We call this part *subset routing* (see Figure 2).

Recall the definition of hierarchical event types in Section 2. This hierarchy defines a superset-subset relationship that is fundamental to achieving subset routing. Informally, subset routing follows a dissemination tree rooted at the rendezvous node. The ID of every child node in the tree is one Hamming distance away from its parent and has Hamming weight less by one. The shaded nodes in Figure 2 illustrate a subset routing tree. As shown, node 0111 has three immediate children, 0011, 0101, and 0110. When an event reaches its rendezvous node (in Figure 2, event 1100111 reaches node 0111), the event is recursively forwarded to the nodes one level down the tree until it reaches all nodes in the tree. In the example in Figure 2, event 1100111 is sent to 0110, 0101, 0011, and from there to 0010, 0100 and 0001. The ways in which routing tables are constructed in Reach guarantee that children nodes are always in the parent's routing table (and vice versa). In a subset routing tree, every non-root node hosts identifiers that are subsets of the event identifier. Thus, an event eventually reaches all subset subscriptions by starting from the rendezvous node and traversing down the tree.

One might observe that the subset routing tree is not unique (e.g., in Figure 2, 0110 can also be parent to 0100). We use a deterministic algorithm to build a tree and at the same time, eliminate potential duplicate messages. The details of the subset routing algorithm are as follows: When a message e reaches its rendezvous node i, i finds the children of id_i and forwards the event to each such child. When a child node j, such that $id_j \subset id_i$, receives this message, it generates the children of id_j and forwards the message to them, and so forth. In the meantime, each node uses a deterministic algorithm to eliminate duplicate messages to children in common with other nodes. A possible algorithm would be to use a public hash of the message identifier as an indexing mechanism to determine who is responsible for the common children.

At a more intuitive level, the content-based multicasting mechanism in Reach ensures that a message is forwarded to each subscription that requests all or a subset of the attributes present in the message content. The multicast algorithm generates $2^b + l$ messages where b ($b \leq m$) is the number of marked attributes in the m-bit suffix of the message identifier, and l is the number of overlay hops to reach the rendezvous node (i.e. the root of the tree).

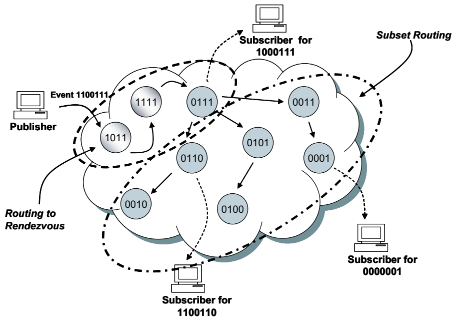


Figure 2. Dissemination of event 11000111 in a network of size 2^4 (m = 4, n = 8).

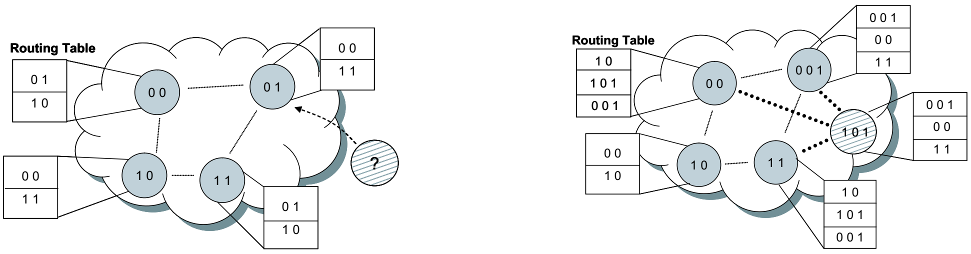
In contrast to other DHT-based overlay networks, Reach identifiers encode meaningful application-level information, i.e., attributes. As a result, the neighboring relation in Reach reflects a semantic relation that is not preserved in standard peer-to-peer overlay networks. This semantic relationship in the overlay layer is the key to an efficient implementation of content-based multicast. Without it, a separate overlay message would be required to reach each potentially distant subset ID in the overlay layer.

3.3 Dynamic Networks

We have thus far described Reach when the network size is a power of two. In this section we relax this assumption and discuss arbitrary network sizes. Assuming Reach bootstraps from a network of 2^m nodes, we discuss node joins and leaves in turn.

Node joining: We assume that a new node joins Reach by contacting an existing Reach node. The Reach node, upon receiving a *join* request, divides the set of identifiers that it hosts between the new node and itself. Consider the scenario of a four-node network. Suppose node 01 receives a *join* request, it first creates two new node IDs, 001 and 101, by adding a leading bit to its original 2-bit node identifier. It then updates its own ID to 001 and labels the new node 101. Node 001 will continue to host identifiers whose 3-bit suffix match 001, and give the rest to the new node, 101. As a last step, 001 adds 101 to its routing table, informs its original neighbors, 11 and 00, of its new ID, and introduces 101 as a new neighbor to them. Node 00 and 11 update their routing table accordingly. Figure 3 depicts an example node join scenario.

It should be noted that a Reach node processes a *join* request only when its own ID has equal or fewer number of bits than all of its neighbors. If the node's ID has more bits than any of its neighbors, it simply forwards the *join* request to a neighbor whose ID has fewer bits than its own. Note



(a) Reach network before a node joins.

(b) Reach network after the node joins.

Figure 3. Example of a node joining a Reach network with 2^2 existing Reach nodes.

that in some cases a node will need to drop neighbors from its routing table after a *join* operation. Consider the example of a split at node 001, whose ID is being updated to 0001. A neighbor 1011 before the split is now two Hamming distances away from 0001. As a result, 0001 simply drops 1011 from its neighbor set.

Assuming k is the network size, on average $O(\log k)$ nodes are affected for each *join* operation (i.e., having to update their routing tables). At the end of a *join* operation, each node hosts a set of identifiers that has the node ID in its suffix.

Node leaving: Our algorithm for node leaving is similar to node joining. If node i with an $m+1$-bit identifier is to leave the network, it contacts node j who shares the m-bit suffix with i.[1] j will then host all of i's virtual identifiers and update its node ID to the m-bit suffix. i will inform all of its neighbors to drop i from their routing tables and to add j if it's not present. j will update its neighbors on its new node identifier. Again, $O(\log k)$ nodes will be affected by a single leave operation.

We note that a network with varying length node IDs has a size that is not a power of two. It should be clear that the routing and event dissemination mechanisms in Reach can carry on exactly in the same manner even with varying length node IDs—Hamming-distance based routing with different length IDs would simply route on the largest suffix. The average path length in such a network is still $O(\log k)$ when the network size is between k and $2k$. The joining and leaving operations can be repeated as many times as the attribute space allows—for an n-bit attribute space, the network can grow to at most 2^n nodes where each node hosts a single virtual identifier.

Concurrent Joins/Leaves: The join/leave algorithm described thus far assumes that nodes join and leave the network in a serialized manner. In a scenario with concurrent joins and leaves, nodes in the network may operate on out-of-date neighbor tables. We define the *global gap* as the largest difference in the ID length of any pair of nodes in the network. As shown in [1], networks with large global gaps have longer average path lengths than those that do not. Handling concurrent joins and leaves with out-of-date routing tables can result in a large global gap. A possible remedy to reduce the global gap in Reach is by imposing a limit on the number of join/leave requests a node can process between subsequent heartbeat messages from neighbors. Assuming the heartbeat messages carry up-to-date IDs from the neighbors, a node can update its routing table before processing subsequent requests. A detailed discussion and analysis of concurrent joins/leaves is out of the scope of this paper.

4 Discussion

4.1 Naming

In Reach, we assume a globally-static attribute space and use a bit vector for naming. This naming scheme preserves high-level application semantics, which gives rise to a content-based abstraction that is fundamentally more powerful than standard DHTs. An interesting question is whether information content can be effectively represented by such a bit-vector representation. For example, if the application in question consists of querying the content of newspaper articles, using a separate bit to represent every newspaper ever published is clearly not an option. A more rational approach would call for some kind of hierarchical attribute space for which high-level filtering and routing are followed by filtering on more fine-grained attributes.

We emphasize that the attribute representation problem is independent of the Reach architecture and routing scheme. Use of a hierarchical attribute space, for instance, may better facilitate certain applications, but it will not result in fundamental changes to the Reach infrastructure design.

[1] There should only be one such node in the network and it is in i's routing table.

4.2 Finer-Grain Filtering

Currently, Reach only allows clients to express interest in particular attributes. A more powerful subscription language would allow filtering based on the values of attributes. Augmenting Reach's functionality to achieve this is simple: the routing scheme (including point-to-point and subset routing) in Reach need not change, but subscriptions can be accompanied by arbitrarily complex filters on the values of the attributes. When a message is being matched to the subscriptions, the finer-grain filters on attributes values will be invoked. This way, an event matches a subscription if and only if the event identifier matches the subscription identifier *and* the value-based filter covers the event attribute values.

4.3 Fault Tolerance

Our current design does not specifically deal with fault tolerance. We note that link failures at the physical network layer can be masked with similar mechanisms as those in [16, 11]. However, our routing scheme will break down if a physical intermediate node fails. Since every Reach node serves as a rendezvous point for some particular content, straightforward replication of all the rendezvous nodes is not a viable option. We note that Reach falls in the general vein of hypercube models, and fault tolerance can potentially be dealt with by adopting a hypercube fault-tolerance mechanism (e.g.,[8]). Further investigation is needed to determine if these mechanisms can adapt to dynamic networks such as Reach. We are also investigating other more fault tolerant methods for traversing the subset tree.

4.4 Load Balancing

In a random subscription and event workload, Reach's mapping of event IDs to rendezvous node is evenly distributed between all nodes in the system. However, in reality, it is possible that certain rendezvous nodes will become overloaded due to popular event identifiers. Reach currently does not have specific mechanisms to deal with load balancing. A potential load-balancing scheme could replicate subscriptions up the dissemination tree. A parent node can then probablistically decide whether to fulfill the children's subscriptions. Another scheme could allow an overloaded node, A, to push an identifier, id, to another less overloaded node, B. A would send all subscriptions that match id to B. When an event labeled with id is published, A would then forward the event to B to let B fulfill the subscriptions for id. This is similar to load balancing mechanisms as found in [12].

4.5 Quenching

Another potential optimization for Reach is incorporating a quenching mechanism to reduce unnecessary communication due to lack of subscriber interest. A quenching mechanism would be as follows: when a subscription reaches its rendezvous node, the node sends an *interest* bit to all its neighbors whose IDs are supersets to his ID. Quenching is performed recursively; that is, an interest bit from a child node will cause an interest bit to propagate upward to the parents and superset nodes only if the *interest* bit has not been communicated previously. Therefore, an event will only be delivered to children nodes that have indicated interest prior to the publication of the event. This is similar to quenching performed by Siena[4].

5 Related Work

As Reach is inspired by supporting content-based services with dynamic server infrastructures, it is important to contrast our design with others that have similar goals.

5.1 Publish/Subscribe Networks

Current pub/sub networks such as [2, 4] are capable of supporting powerful content-based services. Compared to Reach, they provide more expressive subscription languages and finer grain content-based services.

However, many pub/sub implementations require broadcasting subscriptions to the entire network. As network size increases, subscription registration becomes prohibitively expensive. There exist pub/sub designs that explore covering relationships between subscriptions to reduce broadcast traffic [4], but such designs only benefit if the workload lends itself well to such optimizations. In contrast, subscription registration in Reach does not require broadcast and traverses only $log(n)$ of the network nodes.

In general, pub/sub services are built on top of a static network infrastructure, and indeed many routing and forwarding algorithms rely on this static nature. We note that many applications may benefit from more general network settings including dynamically changing server populations. Reach, as other peer-to-peer systems, supports a dynamic network infrastructure and as a result allows the network to scale gracefully as the server population changes.

A pub/sub system that allows for dynamic server populations is presented in [14]. However, as with other pub/sub systems, broadcasts to the network are necessary to establish the content-based routing paths.

5.2 Application Level Multicast

Information services such as Bayeux [16], Scribe [11] and *i*3 [12] are built on top of structured overlay networks.

These systems provide application-level multicast services using the rendezvous-based communication model. In these systems, information is associated with an identifier that is a hash of the data content. Each data identifier is mapped to the rendezvous node whose ID is the closest match to the identifier. These systems only allow exact (or near exact) matching of subscriptions and data IDs and are incapable of supporting more sophisticated content-based operations.

Furthermore, application-level multicast services require the explicit creation of multicast groups and setup of rendezvous nodes. As [4] points out, there does not appear to be a universally optimal mapping from multicast groups to recipient interests. In Reach, multicast groups are implicitly defined by the superset/subset relationship, and thus do not need explicit management. As a result, subscribers only receive messages matching their interests, and publishers need not duplicate messages to reach all relevant multicast groups.

A system that shares similar goals with Reach is Hermes [7]. Hermes implements a "type- and attribute-based" routing scheme that extends the expressiveness of subscriptions and supports event hierarchies. Hermes incorporates event hierarchies in the system by explicitly notifying "ancestor" identifers of the existence of new "descendant" identifiers. Thus, a published event with the ancestor identifier will also be forwarded to the rendezvous node of the descendant identifier. This scheme adds support for event hierarchies. However, nodes in a Hermes event hierarchy are more likely to be scattered throughout the network, thereby resulting in a large overhead for propagating an event to all of its descendants. In contrast, Reach attains efficiency by constructing a network wherein the super/subset identifiers are immediate neighbors in the overlay infrastructure.

6 Conclusion

In this paper, we presented a design for content-based multicasting atop a peer-to-peer rendezvous communication abstraction. Our approach offers benefits from both pub/sub and rendezvous models. From pub/sub, we implement an important class of information services, namely content-based multicasting. We adopt an expressive form of subset matching, and route publications efficiently on "subset trees" by utilizing subset relationships in the routing protocol. From the peer-to-peer model, we accomodate dynamic joins and leaves, and thereby gain benefits of resource sharing that have not been previously exploited in pub/sub systems. We showed that by exploiting semantic relationships in the overlay structure, we are able to achieve an efficient implementation of content-based multicast.

References

[1] I. Abraham, B. Awerbuch, Y. Azar, Y. Bartal, and D. Malkhi. A Generic Scheme for Building Overlay Networks in Adversarial Scenarios. *International Parallel and Distributed Processing Symposium* (Nice, France), April 2003.

[2] M. Aguilera, R. Strom, D. Sturman, M. Astley, and T. Chandra. Matching events in a content-based subscription system. *Symposium on Principles of Distributed Computing*, pages 53-61, 1999.

[3] G. Banavar, M. Kaplan, K. Shaw, R. E. Strom, D. C. Sturman, and W. Tao. Information flow based event distribution middleware. *International Conference on Distributed Computing Systems*, pages 114–121. IEEE Computer Society, 1999.

[4] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, **19**(3):332–383. ACM Press, August 2001.

[5] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. *Fourth USENIX Symposium on Internet Technologies and Systems* (Seattle, WA), March 2003.

[6] T. S. Inc. TIBCO Rendezvous.
http://www.tibco.com/solutions/products/active_enterprise/rv/.

[7] P. R. Pietzuch and J. M. Bacon. Hermes: A Distributed Event-Based Middleware Architecture. *International Workshop on Distributed Event-Based Systems*, 2002.

[8] P. Ramanathan and K. G. Shin. Reliable Broadcast in Hypercube Multicomputers. *IEEE Transactions on Computers*, **37**(12):1654–1657, December 1988.

[9] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. *ACM SIGCOMM Conference*, pages 161–172. ACM, 2001.

[10] A. Rowstron and P. Druschel. Pastry: scalable, decentralized object location and routing for large-scale peer-to-peer systems. *IFIP/ACM International Conference on Distributed Systems Platforms*, pages 329–350, 2001.

[11] A. Rowstron, A.-M. Kermarrec, P. Druschel, and M. Castro. SCRIBE: The design of a large-scale event notification infrastructure. *3rd International Workshop on Networked Group Communications*.

[12] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet Indirection Infrastucture. *ACM SIGCOMM Conference* (Pittsburgh, PA, August 2002), 2002.

[13] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Conference*. Published as *Computer Communication Review*, **31**(4):149–160, 2001.

[14] W. W. Terpstra, S. Behnel, L. Fiege, A. Zeidler, and A. P. Buchmann. A Peer-to-Peer Approach to Content-Based Publish/Subscribe. *International Workshop on Distributed Event-Based Systems*, 2003.

[15] B. Y. Zhao, J. Kubiatowicz, and A. D. Joseph. *Tapestry: an infrastructure for fault-tolerant wide-area location and routing*. UCB Technical Report UCB/CSD–01–1141. Computer Science Division (EECS) University of California, Berkeley, April 2001.

[16] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. Kubiatowicz. Bayeux: An Architecture for Scalable and Fault-tolerant Wide Area Data Dissemination. *International Workshop on Networking And Operating System Support for Digital Audio and Video*, June 2001.

Supporting Mobility with Persistent Notifications in Publish/Subscribe Systems

Ivana Podnar and Ignac Lovrek
Department of Telecommunications
Faculty of Electrical Engineering and Computing, University of Zagreb
Unska 3, HR-10000 Zagreb, Croatia
ivana.podnar@fer.hr, ignac.lovrek@fer.hr

Abstract

The paper proposes a novel approach to mobility in publish/subscribe (pub/sub) systems that relies on notification persistency. The existing solutions apply proxy subscribers that take the role of disconnected subscribers, store notifications published during disconnections, and deliver them to subscribers after reconnection to the pub/sub system. The system perceives a constant number of subscribers which is inefficient if it serves a large number of subscribers that are often disconnected. We argue that the system should not be burdened by proxy subscribers during subscriber disconnections, but rather store persistent notifications, and deliver valid notifications to subscribers when they reconnect to the system. Preliminary experiments show that the proposed solution can improve the observed scalability weakness of the existing solutions.

1. Introduction

The inherent characteristics of pub/sub middleware, such as loose coupling, asynchronous and persistent communication, and system extendibility, are found suitable for the design of mobile applications that need to adapt to highly dynamic and volatile conditions in mobile environments [3]. The existing pub/sub systems are optimized for stationary environments [2, 4, 8, 6]. The mobility-related operation is managed by the application layer through a sequence of *subscribe-unsubscribe-subscribe* requests: A subscriber defines subscriptions when connecting to a pub/sub system, unsubscribes prior to disconnection, and re-subscribes after reconnecting to the system. However, the subscriber will not receive notifications that are published during the time of disconnection. The *queuing approach* is commonly applied to solve the problem of lost notifications [1, 5, 10]. A system broker, or a special proxy acts as a proxy subscriber during subscriber disconnections. The proxy stores notifications published during disconnections in a special subscriber's queue, and delivers them to the subscriber after its reconnection. If the subscriber reconnects through a new system broker, a handover procedure is performed which transfers the stored notifications to the subscriber and updates its subscriptions in the broker network.

We propose a novel approach to mobility in pub/sub systems: Rather than queuing notifications for disconnected subscribers, the network of brokers stores persistent notifications until their validity period expires. We argue that notification publishers need to define the validity period of published notifications, the system must assure notification storage, and deliver valid notifications to subscribers when they reconnect to the pub/sub system. If a subscriber connects to the system after notification expiry, the notification will not be delivered to the subscriber because it no longer holds valuable information. Notification persistency is by no means a new characteristics in the existing pub/sub systems: The Java Message Service (JMS) supports mobility using durable subscriptions and persistent notifications. However, the available JMS implementations are mainly centralized and offer no routing optimizations for distributed pub/sub architectures.

We have implemented a prototype system to show the adequacy of the proposed solution. The prototype can be distinguished from other pub/sub implementations by the inherent support for publisher and subscriber mobility. Furthermore, we have used the prototype implementation to evaluate the proposed solution, and to compare it with the approach based on queues.

The paper is structured as follows. Section 2 presents the proposed solution: We outline the extensions of the routing algorithms to support mobility using persistent notifications, and discuss the characteristics of the approach. Section 3 describes the prototype implementation, defines the metrics for performance evaluation of pub/sub systems, and presents initial evaluation results that compare the proposed approach with the queuing approach. In Section 4, we discuss mobility solutions in the existing systems, and Section 5 concludes the paper.

2. Mobility Support with Persistent Notifications

A pub/sub system comprises a set of publishers and a set of subscribers that interact by performing actions. The occurrence of an action is an event that changes a system state. The events occurring in the system are *connect*, *disconnect*, *publish*, *subscribe*, *unsubscribe*, and *notify*. Publishers and subscribers connect and disconnect from the system, and change the system state by modifying the set of connected publishers and subscribers. Publishers publish notifications using the event *publish*, and change the set of published persistent notifications. Publishers define a validity timestamp which specifies the validity period of a published notification. A notification is removed from the set of persistent notifications when its validity period expires. Subscribers activate subscriptions by generating an event *subscribe*, and invalidate them using the event *unsubscribe*. The system is responsible for matching a published notification to the set of active subscriptions, and for notifying subscribers with a matching subscription about the existence of the notification using the event *notify*.

A pub/sub system may have a distributed architecture to solve the scalability problem which is particularly important in mobile environments where clients roam in different network domains and can connect to “the closest” broker. A distributed pub/sub system comprises a set of interconnected brokers that exchange messages carrying events to maintain a consistent view of active subscriptions and persistent notifications in the system. Brokers exchange subscription messages to create *delivery paths* connecting a publisher to a set of potential subscribers. The routing algorithms used for creating delivery paths are mainly based on *reverse path forwarding* [2, 6], and use subscription equality or subscription covering to reduce the number of messages exchanged between system brokers, and the number of entries in broker routing tables. A recently proposed solution applies an algorithm based on core trees [8].

The existing solutions for client mobility in distributed pub/sub systems use the queuing approach (Q-app) for storing notifications in special queues during subscriber disconnections [1, 5]. We propose an approach based on persistent notifications (PN-app) that requires the delivery of valid notifications after the activation of a new subscription in the system. When a subscriber reconnects to the system, it reactivates its subscriptions and therefore receives valid stored notifications. The PN-app is an extension of the subscribe-unsubscribe-subscribe procedure that does not modify the applied routing algorithm, e.g., the reverse path forwarding, or the core-based tree algorithm. Here we shortly describe the extensions of the basic algorithm:

- When a broker receives a *publish* message, it first stores the notification in the persistent notification container, and subsequently delivers it to interested subscribers and neighboring brokers. The broker removes the notification from the container when its validity period expires, and maintains a list of valid notification ids that have been sent to subscribers and brokers.
- When a broker receives a *subscribe* message, either from a subscriber, or a neighboring broker, it checks the list of persistent notifications and delivers a matching valid, and previously undelivered notification to the subscriber, or the broker.
- When a broker receives a *connect* message from a subscriber, it activates the subscriber’s subscriptions. The subscriber must provide a list of active subscriptions and a list of received valid notification ids to avoid duplicate notifications.
- When a broker receives a *disconnect* message from a subscriber, it deactivates the existing subscriber subscriptions, and terminates delivery paths in the broker network.

The mechanism that compares the list of valid received notification ids to those that should be sent to a subscriber, or a neighboring broker prevents the delivery of duplicate notifications. We assume that notification ids are unique within the system, and that they are removed from the list when notifications expire. A notification can be undelivered because a broker network may introduce the delay that can lead to notification expiry prior to its delivery to a subscriber.

The differences between the Q-app and the PN-app are in the following:

- **Notification storage.** In case of the Q-app, system brokers store queues per each disconnected subscriber. For the PN-app, brokers maintain persistent notifications, and the list of valid notifications sent to subscribers and neighboring brokers.
- **Subscriber’s reconnection to the system.** When applying the Q-app, a reconnecting subscriber first reactivates its subscriptions at the new broker to update the delivery path in the broker network, and next, the new broker retrieves the notifications from the subscriber’s queue maintained by the old broker and delivers them to the subscriber. In case of the PN-app, the subscription reactivation will initiate the delivery of valid notifications along the new delivery path to the subscriber. Prior to notification delivery to the client, the edge broker checks whether the subscriber has already received a valid notification by comparing it to the list of received notification ids.
- **Subscriber’s data.** A subscriber in the system applying the Q-app needs to know the identifier of the old broker together with the list of active subscriptions to reconnect to the system. In case of the PN-app, a list of received and valid notification ids, and the list of active

subscriptions is needed.

- **Perceived number of system subscribers.** Subscriber queues act as proxy subscribers for disconnected clients which gives the impression that subscribers are constantly active in a system that uses the Q-app. The PN-app maintains no active subscriptions for disconnected subscribers.

A persistent notification is stored by a subset of network brokers until its validity period expires. It is maintained by a single broker if, at the time of its publishing, there were no remote subscribers for the notification. The notification will eventually reach a reconnecting subscriber following a newly-created delivery path, and be stored on each broker it traverses. The subscriber might have already received the notification if it has previously resided on the broker through which the notification has been published: The notification will be routed to the new broker, however, it will not be delivered to the subscriber since its id is in the list of subscriber's received notifications. This in the known overhead of the approach that causes superfluous traffic in the broker network, and increases the usage of broker memory and processing time. At the other extreme is the situation in which all brokers have a notification copy. A reconnecting subscriber will receive a notification copy from the access broker without causing extra traffic in the broker network.

Potential advantages of the proposed approach when compared with the Q-app are the following: avoidance of the handover procedure that transfers notifications from the old to the new broker, reduced size of broker routing tables due to decreased number of perceived subscribers in the system, and memory consumption related to the storage of notifications in the system. The expected disadvantage is related to control traffic: The PN-app generates an increased number of subscriptions and unsubscriptions for terminating the old and creating the new delivery paths. The Q-app suffers from the same problem if the probability that a subscriber reconnects to the same broker is low. If subscribers frequently connect to the same broker, the number of control messages is reduced because there is no need to update an existing delivery path. The maintenance of the list of received and valid notification ids is an additional broker overhead in case of the PN-app.

3. Evaluation

We use the implementation of the prototype system MOPS (**Mo**bile **P**ublish **S**ubscribe) to investigate the adequacy of the proposed approach, to evaluate the system performance in mobile environments, and to compare it with the system performance when applying the Q-app. Both approaches are used with the routing algorithm based on subscription covering.

3.1. The Prototype System MOPS

The MOPS infrastructure comprises a set of interconnected brokers that form an acyclic communication graph. There is a single spanning tree for notification delivery connecting a publisher to a group of subscribers, and each broker is a single point of system failure. The broker network is built incrementally by connecting a new broker to an active broker, and can be extended during system operation. Clients, i.e., publishers and subscribers, are mobile entities that can connect to different brokers. The communication is realized in the form of messages transported using TCP to assure reliable communication. The communication between a client and a broker is truly push-based without open connections. A client registers as connected with a broker, and provides a time-to-live parameter (TTL) specifying the period it expects to be served by the broker. The broker maintains a list of connected clients, and removes a subscriber from the list in case a notification cannot be delivered to the subscriber prior to TTL expiry.

The MOPS system supports typed notifications that carry a list of attributes. It offers type-based and attribute-based subscriptions, and implements type-based routing with support for subscription covering in the broker network. Attribute-based notification filtering is performed by the edge broker prior to notification delivery to subscribers. What distinguishes it from other pub/sub prototype implementations is the inherent support for publisher and subscriber mobility that has been integrated into the system design, rather than added as an extension to an existing system supporting stationary clients. The system can be configured to use either queues for storing notifications on behalf of disconnected subscribers, or persistent notifications maintained by the brokers.

3.2. Metrics

The performance of a pub/sub system in a dynamic environment with mobile clients is largely influenced by its efficiency. We propose the usage of the following metrics for mobile pub/sub system evaluation:

- **Broker processing load.** The processing load experienced by a broker can be measured by the rate of processed messages. Messages carrying notifications transport the actual information, while subscription and unsubscription messages represent control load that creates and updates delivery paths. We differentiate between received and sent messages, and classify them according to the type of events they transport.
- **Bandwidth consumption.** A desirable property of a distributed pub/sub system is to consume minimal bandwidth. The rate of processed messages, as in the case of broker processing load, gives a good estimate

of the physical bandwidth consumption.

- **Notification delay.** Efficient notification delivery requires minimal delay, i.e., the period between notification publication and receipt. In case of mobile subscribers, the delay is increased due to subscriber disconnections from the system, and it depends on the duration of disconnection periods and notification validity periods.

3.3. Preliminary Experimental Results

We show the experimental results that enable preliminary assessment of system performance when applying the PN-app, and the Q-app. The results are obtained using a working prototype that simulates the real working environment, instead of a model simulation. There are some implementation differences that cause an increased processing load for system brokers in case of the PN-app due to a garbage collector that purges expired notifications from persistent notification containers. Therefore, we have decided to use the metrics that are not largely influenced by the processing latency to enable a just comparison of the two approaches. The experiment investigates the broker's processing load, and bandwidth consumption in terms of the rate of processed messages, and the number of stored notifications. The experiment does not investigate the notification delay because it is largely influenced by the applied routing policy, which is the same for both approaches, and by the actual system implementation which would favor the Q-app.

We ran the experiment under the same initial conditions for the Q-app and the PN-app. After forming a network of brokers, we initiated a set of mobile subscribers and stationary publishers. Each experiment run lasted 20 minutes, and we conducted 5 runs with the same initial setting.

Input parameters. We are using a network of seven brokers forming a tree structure. The number of publishers is constant, $p = 7$, and each publisher is stationary and connected to one of the brokers. Publishers publish notifications at a constant rate of $pubRate = 0.5$ notifications/s. We use a complex type hierarchy consisting of 20 types, and publishers generate notifications of a randomly chosen type with a uniform probability. Each notification carries a payload of 100 bytes. The validity period for notifications in case of the PN-app is set to 5000 ms to avoid undelivered notifications.

We varied the number of subscribers in the system, $s = 1, 5, \ldots, 35$. Subscribers are mobile and can connect to all system brokers except to B_1 because it is the root node of the broker network, and therefore the system bottleneck. We use the random mobility model in the experiment: A subscriber chooses the next broker randomly from the set of available brokers. Each subscriber connects to a new broker with a constant connection rate in the range from 0.2 to 0.6 connections/s, and the connection duration is 50% of the connection period.

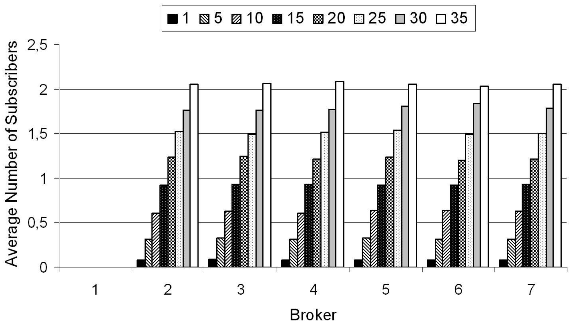


Figure 1: Connected subscribers per broker

Figure 1 shows the measured average number of subscribers connected to a broker as the total number of subscribers in the system changes. It is visible that subscribers do not connect to B_1, and that other brokers evenly share the subscriber load. In case of the total of 15 subscribers in the system, there is on average one subscriber connected to each broker. All subscribers subscribe to the top notification type, and should receive all published notifications. We use the simple subscription scenario to test the implementation and check that notifications are received without duplicates.

Experimental results. Figure 2 shows the average number of stored notifications on a broker as the number of subscribers in the system increases. As expected, the number of stored notifications increases linearly with the number of subscribers for the Q-app, since notifications are stored per each disconnected subscriber in a separate queue. The number of stored notifications for the PN-app reaches its maximum value as soon as each published notification is stored once on each broker and remains constant regardless of the number of subscribers in the system.

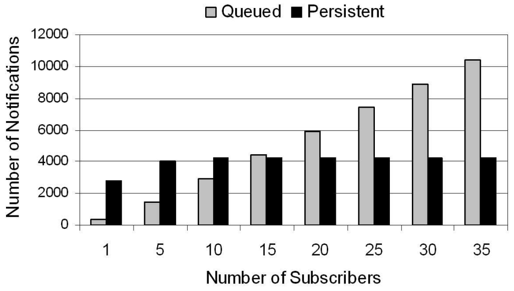


Figure 2: Queued and persistent notifications

Figure 3 shows the average rate of *received and sent notification messages* per each broker. The rate of received notification messages increases for both approaches until

$s = 15$ when there is on average 1 subscriber connected to each broker, and reaches the maximum value determined by the number of system publishers, and their publishing rates. Clearly, the rate of sent notification messages increases as the number of subscribers increases, because the number of notification destinations increases accordingly. The Q-app generates a larger number of notification messages than the PN-app when $s \geq 7$, because in case of the Q-app, notifications are sent to subscriber queues during the disconnection period, while in case of the PN-app, notifications are cached on brokers they traverse, and from there delivered to reconnecting subscribers. The notification rate in case of the Q-app is further increased by notification exchange during the handover procedure.

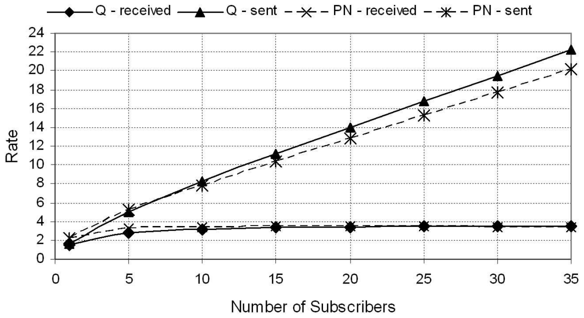


Figure 3: Rate of received and sent notifications

Figure 4 shows the average rate of *received and sent unsubscription messages* per broker. As expected, the Q-app generates less unsubscription messages than the PN-app because the Q-app does not generate unsubscription messages in case the old and the new broker are the same, which is the case for the PN-app. The rate of sent unsubscription messages decreases as the number of subscribers increases, because there is no need to propagate unsubscriptions since there are other subscribers with an active matching subscription that are connected to brokers.

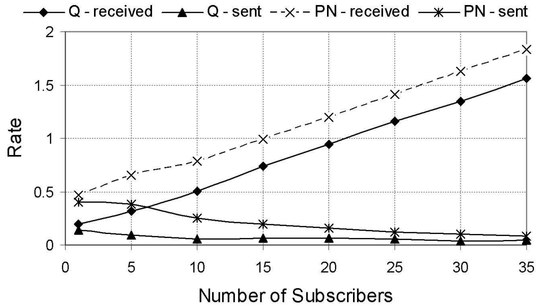


Figure 4: Rate of received and sent unsubscriptions

Finally, Figure 5 depicts the average rate of all received and sent messages per broker and can be used to asses the broker processing load. It is visible that the PN-app poses less load on a broker as the number of subscribers in the system increases. The gain is 10% in this particular experiment when $s = 35$, and would become significant in case of a large number of subscribers.

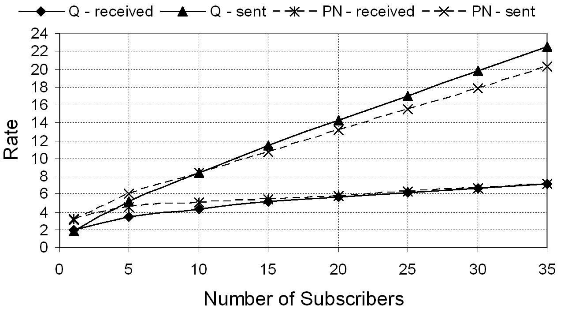


Figure 5: Rate of received and sent messages

To conclude, the PN-app is superior when compared to the Q-app with respect to the consumption of broker memory for storing undelivered notifications if we assume that the number of subscribers in the system is significantly larger than the number of brokers. Furthermore, the PN-approach introduces less processing load on brokers for the conducted experiment as the number of subscribers in the system increases, and therefore consumes less bandwidth on links connecting the brokers. The preliminary results show that the load introduced by control messages for the PN-app is acceptable when compared to the Q-app.

4. Related Work

A number of authors agree that the mobility support in pub/sub systems should be offered by the pub/sub middleware itself, and not delegated to the application layer [9, 12]. The existing solutions extent the established pub/sub systems that are optimized for stationary environments by adding special proxy subscribers taking the subscriber's role for disconnected subscribers [1, 5].

The mobility service developed within the project SIENA [1] uses *client proxies* and a special *client library* to manage subscriptions on behalf of a subscriber. A client proxy runs as a special component at an access point, and stores messages for a disconnected subscriber in a special queue. The client library mediates a move-out procedure from the old broker, and a move-in procedure when a subscriber reconnects to the new broker. The old and the new proxy perform a handover procedure that transfers messages from the old proxy to the new one, and subsequently to the subscriber. Clearly, the overhead of the proposed solution is the existence of a new component in system, the client proxy. The reported experimental results prove the applicability of the implementation, investigate the number of duplicate and lost messages, but define no other metrics to evaluate system performance. Best to the author's knowledge, the presented experimental results are the first attempt to evaluate pub/sub system performance in a mobile setting.

The solution presented in this paper is compliant with the algorithm for physical mobility developed within the project REBECA [5, 12] that uses the Q-app where the last serving broker takes the role of a proxy subscriber. When a subscriber connects to a new broker, it re-issues its subscriptions, and the broker network finds the old broker by locating a broker at the junction of delivery paths to the new and the old broker. It is straightforward to find a junction broker if the broker network applies simple routing, because each broker keeps entries about all active subscriptions. The notifications stored by the old broker are routed through the junction to reach the new broker, and subsequently the subscriber. The authors do not justify why subscribers cannot maintain the information about the last visited broker which significantly complicates the algorithm that needs to locate the junction broker. The algorithm needs further extensions in case routing based on covering is applied since simple routing generates large routing tables. There are currently no results that evaluate the performance of the approach.

Mobility support in Elvin [10] is one of the first implementations offering mobility to subscribers in a pub/sub system. The solution puts a proxy server between the original server and a mobile device for storing messages for disconnected subscribers. A subscriber must always connect to the central proxy server which can become a performance bottleneck and induce significant network traffic due to potential triangular routing.

Recently, some of the systems that implement the JMS specification support mobility [7, 11] by offering a lightweight JMS-compliant API for Java-enabled mobile terminals that can be used to implement JMS-based publishers and subscribers. However, the available implementations offer no routing optimizations for distributed architectures.

5. Conclusion

The paper presents an approach to mobility in pub/sub systems that uses a sequence of subscribe-unsubscribe-subscribe requests, and relies on notification persistency to assure the delivery of notifications published during subscriber disconnections. We have investigated the adequacy of the approach using a prototype system, presented initial results that investigate its performance, and compared it to the solution based on queues. The preliminary experimental results show that the proposed approach is superior to the queuing approach with respect to the broker processing load and memory consumption as the number of subscribers in the system increases, assuming that subscribers move following the random mobility model. Future work should examine the notification delay introduced by the approach, and investigate system performance for different subscriber mobility models and subscription scenarios.

Acknowledgements. The authors thank Branimir Turk and Matija Mićin for their contributions to the implementation of the MoPS system.

References

[1] M. Caporuscio, A. Carzaniga, and A. L. Wolf. Design and evaluation of a support service for mobile, wireless publish/subscribe applications. *IEEE Trans. on Software Engineering*, 29(12):1059–1071, Dec. 2003.

[2] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. on Computer Systems*, 19(3):332–383, August 2001.

[3] G. Cugola and H.-A. Jacobsen. Using publish/subscribe middleware for mobile systems. *ACM SIGMOBILE Mobile Computing and Communications Review*, 6(4):25–33, 2002.

[4] G. Cugola, E. D. Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Trans. on Software Engineering*, 27(9):827–50, September 2001.

[5] L. Fiege, F. C. Gärtner, O. Kasten, and A. Zeidler. Supporting mobility in content-based publish/subscribe middleware. In *Proc. of the Middleware 2003*, volume 2672 of *LNCS*, pages 103–122. Springer-Verlag, June 2003.

[6] G. Mühl, L.Fiege, F. C. Gärtner, and A. Buchmann. Evaluating advanced routing algorithms for content-based publish/subscribe systems. In *Proc. of the 10th IEEE Int. Symp. on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'02)*, pages 167–176. IEEE Computer Society, Oct. 2002.

[7] ObjectWeb Open Source Middleware. JORAM (release 3.6.0), August 2003. http://www.objectweb.org/joram/.

[8] P. Pietzuch and J. Bacon. Peer-to-peer overlay broker networks in an event-based middleware. In *Proc. of the 2nd Int. Workshop on Distributed Event-Based Systems (DEBS'03)*. ACM Press, June 2003.

[9] I. Podnar, M. Hauswirth, and M. Jazayeri. Mobile Push: Delivering content to mobile users. In *Proc. of the 1st Int. Workshop on Distributed Event-Based Systems (DEBS'02)*, pages 563–568. IEEE Computer Society, July 2002.

[10] P. Sutton, R. Arkins, and B. Segall. Supporting disconnectedness – Transparent information delivery for mobile and invisible computing. In *Proc. of the IEEE Int. Symp. on Cluster Computing and the Grid*, pages 277–285. IEEE Computer Society, May 2001.

[11] E. Yoneki and J. Bacon. Pronto: MobileGateway with publish-subscribe paradigm over wireless network. Technical Report UCAM-CL-TR-559, Computer Laboratory, University of Cambridge, 2003.

[12] A. Zeidler and L. Fiege. Mobility support with REBECA. In *Proc. of the 2nd Int. Workshop on Distributed Event-Based Systems (DEBS'03)*, pages 354–360, May 2003.

Federating Heterogeneous Event Services

Conor Ryan, René Meier, and Vinny Cahill
Distributed Systems Group, Department of Computer Science, Trinity College Dublin, Ireland
cahryan@eircom.net, rene.meier@cs.tcd.ie, vinny.cahill@cs.tcd.ie

Abstract

As event-based middleware is currently being applied for application component integration in a range of application areas, a variety of event services have been proposed to address different application requirements. The emergence of ubiquitous computing systems has given rise to application integration across multiple areas and as a result, has led to systems comprising several, independently operating event services. Even though event services are based on the same communication pattern, application component integration across heterogeneous services is typically prevented by the constraints imposed by their respective event models.

This paper presents the design and implementation of the Federated Event Service (FES). The FES enables heterogeneous event services to cooperate and to operate as a single logical service. It therefore facilitates building event-based systems in which the application requirements cannot be met by a single event service.

1. Introduction

Event services provide asynchronous, decoupled, anonymous message-based communication. This facilitates scalable distributed systems composed of autonomous concurrently-executing entities. There are many event services in existence addressing wide ranging issues such as Internet scale (Siena [1]), quality of service (CORBA Notification Service (CNS) [2]), and mobility and location awareness (STEAM [3]). When integrating systems that use distinct event services it may be necessary to inter-work their event services to facilitate communication between the systems.

There is currently no standard solution available for heterogeneous event service inter-working. In the absence of a standard solution, system developers are forced to roll their own solutions. This is problematic as such solutions can cost time, money and effort. These solutions may be sub-optimal since developers, unless they are experts in event systems and event system inter-working, may not have considered or understood all of the issues involved.

A federated service is a collection of autonomous concurrent services that may be linked together to provide a single logical service. This paper presents the design and implementation of the Federated Event Service (FES), a standard mechanism for federating heterogeneous event services. We believe that such a mechanism is a valuable solution for addressing the event service inter-working problem described above. We also believe that this mechanism is a viable alternative to bespoke solutions for building or extending event-based systems, when requirements cannot be met by a single event service.

The remainder of this paper is structured as follows: Section 2 surveys related work. Section 3 discusses event service federation and related issues. Section 4 describes the design of the FES. Section 5 and section 6 discuss a test implementation and usage of the FES. Section 7 concludes this paper by summarizing our work.

2. Related Work

Based on our survey of related work there seems to be little research into the area of federating or inter-working heterogeneous event services. Most of the interest, for commercial reasons, lies in the inter-working of the CORBA Notification Service (CNS) and the Java Message Service [4]. This inter-working requirement is a less difficult problem than generic event service inter-working as it is a bilateral inter-working requirement supporting similar event models, feature sets and event structures.

In [5] events are used to federate components of any granularity including event services. Similar to the FES, the approach in [6] uses a common event model and gateways to provide interoperability between event services. However, these approaches do not emphasize the issues involved with and the opportunities to be gained from heterogeneous event service federation. Our work is specifically concerned with addressing the issues involved with inter-working and federation of heterogeneous event services and aims to provide this mechanism as transparently as possible to existing systems.

3. Federating Event Services

A federated event service provides a single logical event service to clients; however, it consists of a number of autonomous event services. The federation mechanism is transparent to the participating event services and event services are unaware of other event services in the federation. Federation improves reliability – if an event service fails, the rest of the event services are still available. Services in a federation share the processing load and this can improve performance and scalability. Event services can still be administered individually. This facilitates easier management of a large service.

Many issues must be considered in the design of a system for the federation of heterogeneous event services. Some of these issues face any system inter-working effort while others are particular to event service inter-working. Important issues that we are aware of are briefly summarized here.

Event model heterogeneity: An event model consists of a set of rules describing a communication model that is based on events. Event models are discussed and classified in [7]. For a federated service to be valuable it must cater for a wide variety of event models. Important features including event propagation support, event type support, event filtering support, and event service specific features such as mobility and QoS must be considered.

Communication: Requests issued across multiple event services may be subject to failure if an individual event service fails. Allowances have to be made for the fact that mobile event services such as STEAM may be involved in the communication path. Adequate routing protocols must be considered

Naming: Event services use varying mechanisms to identify event types and instances, e.g. event channels (CORBA Event Service (CES) [8], CNS), subscription filters (Siena, CNS) and subject identifiers (STEAM, COSMIC [9]). The design must also consider identifier uniqueness across the federation, case sensitive names and varying maximum name lengths. Event service federation introduces the requirement for individual event service identification, to allow requests to be directed at a subset of the federation and events to be forwarded to the correct event services. Routing events and requests to all event services is not a scalable option. The event service federation may also require a unique identifier to allow event services to participate in more than one federation.

Scalability: How can the sizing limits of an individual event service or system be maintained when the event service participates in a federation? What performance overheads are introduced?

Security: Security in federated event services unveils many issues such as system wide administration, authentication and authorization issues. For a good overview of security issues as they pertain to event services, see [10].

Transparency: As previously explained it is important that the federation remains as transparent as possible to event services and systems. In addition event services may have multiple client applications and it may not be realistic or even possible to update them to provide support for federation due to cost, time or unavailability of source code.

Integrating semantically misaligned events: Events generated in different systems may contain identical contents but in different forms. As discussed in [6], such events may be propagated between systems assuming mapping information is available in the systems involved.

4. FES Architecture

As shown in Figure 1, a FES system consists of two or more event services and one or more *gateways* that bridge them. A gateway interfaces to each event service by means of an *adapter*. The gateways in a FES system form a completely distributed system. Each gateway is an equal peer in the system. There are no centralized points of control or failure and gateways do not maintain any global state.

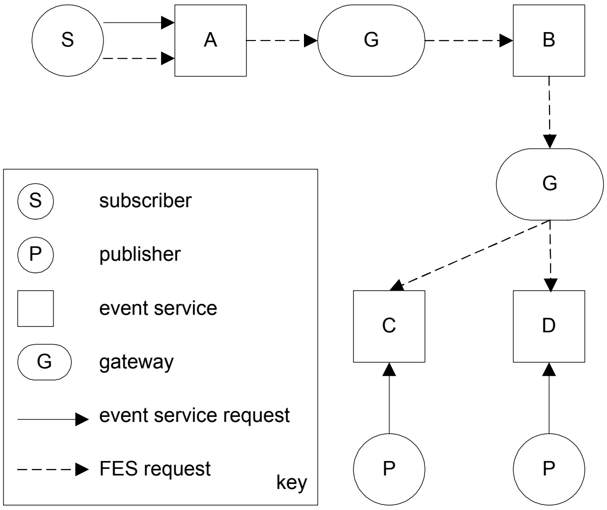


Figure 1. An example FES system.

An event service that a gateway, or any event service client, is directly connected to is known as a *direct event service*. An event service that a gateway, or any event service client, is not directly connected to is known as an *indirect event service.* An event service that is used to route a request is known as an *intermediate* event service.

The FES supports event announcement, subscription and publication requests. These requests may be issued at any event service or gateway. The gateways propagate requests to the relevant event services. A distinction must be made between normal event service requests as issued by a client to its direct event service (*event service request*) and requests issued by a client to event service(s) in the federation (*FES request*).

For example in figure 1, subscriber *s* issues an event service request to event service *A*. In addition *s* also issues the same subscription request as a FES request to the federation. The request is propagated to event services *B*, *C*, and *D* by the FES gateways. Any event published in the federation that matches the subscription request will be propagated back to event service *A* by the gateways and finally to subscriber *s*.

4.1. The FES Event Model

The FES event model acts as a common language between event services. To define the event model for proof of concept purposes, three basic types are required: `string`, `double` and `long`. These basic types are based on CORBA basic IDL types [11].

A FES `Event` is a structured event that is composed of a subject, a set of parameters and a set of attributes. Identifiers are case sensitive. The subject (type `string`) identifies the application event type, e.g. "DeviceOffline". The identifier (or the subject) must be unique within a FES system. Parameters contain application specific data. Attributes represent the non-functional properties of an event such as the delivery priority of an event. Parameter and attributes may be accessed by index and by identifier. An event may contain 0 or more parameters. A parameter consists of an event unique parameter index of type `long`; an event unique parameter identifier of type `string`; a type identifier of type `long` that specifies the type of the parameter data and the parameter data. Attributes have the same structure as parameters. The FES does not place any limit on event size although this implementation of the FES does not support event fragmentation.

The FES supports any event service attribute that may be applied on an event-service-by-event-service basis (hop-by-hop) by adapters. The FES defines an open ended set of attributes and associated semantics, such as event delivery priority and event validity proximity. Adapters may ignore attributes that their event services do not support. Default semantics are also specified for each attribute. These defaults are applied by adapters when attributes must be supplied at an event service but are not specified in an incoming event, for example when mapping a location ignorant CNS event to a location-aware STEAM event.

The structured event type was chosen as this type is commonly supported in event models. It allows flexible filtering. It is relatively easy to map an un-typed event to a structured event. The CNS specification defines how CES un-typed events should be mapped to CNS structured events. The CES/CNS typed events are rarely used, as they are difficult to understand and implement [2, p.212]. Parameter/attribute access by identifier and index and case sensitive identifiers help to facilitate the mapping of event models to the FES event model.

The FES supports event filtering via the FES filtering language. At a minimum the FES filtering language must support subject based filtering. The FES approach to filtering does not depend on the extent of its filtering prowess. The FES makes use of two filters whenever a consumer makes a subscription request to an indirect event service. (1) The subscription as made by the consumer at the direct event service in the direct event service filtering language (*direct filter*). (2) The subscription as made by a gateway on behalf of the consumer at an indirect event service in the indirect event service filtering language (*indirect filter*). The filter that is applied at the indirect event service must always define the same set of events or a superset of the events that was defined by the filter that was specified by the subscriber at the direct event service. In the case where a superset of events is specified at the indirect event service, unwanted events may cross a FES system to the direct event service. However, these events will not reach the consumer, as the direct event service filter will filter them out. The original filter in the FES filtering language must be preserved at all times so that it may be applied consistently at all indirect event services.

FES requests define the functions that are supported by the FES. A request specifies the event service where it originated from (the *source event service*), the event service(s) at which the request should be applied (*destination event service(s)*), and the request parameters. For example, a subscription request specifies the filter that should be applied. A publication request specifies the event that should be published.

The FES may distribute requests to one or more destination event services. Requests are one-way functions that may be applied at most once to each destination event service.

An *announcement* request specifies a particular event type that may be published by an event service producer. Event services may propagate this information to consumers. This facility allows event services and consumers to prepare for future event arrival. An *unannouncement* request specifies an event type that will no longer be published by an event service producer in the future. This facility allows event services and consumers to tear down resources that are will no longer be required to handle events of a certain type. A *subscription* request defines the events that a consumer of an event service is interested in. The consumer supplies a filter to specify this. An *unsubscription* request defines the events that a consumer of an event service is no longer interested in. The consumer supplies a filter to specify this. A *publication* request defines an event that a producer of an event service has published to an event service. These requests are different from the other kinds of requests as they are generated automatically by the FES whenever it receives an event from an event service.

A client of a FES system may specify the event services that a request is sent to via a request *distribution list*. A distribution list provides functionality that is not part of normal event services, i.e. clients may target specific sets of subscribers and publishers. Clients need not be bound to certain event service instances. Instead they may specify the type of event service, or a range of event services to send a request to assuming that a suitable event service naming convention was employed. Distribution lists improve the scalability of the system.

A FES request is encapsulated in an `Event` derived `ControlEvent`. Control events are the only means by which requests may be communicated to gateways and by which gateways communicate. A gateway acts as a producer and a consumer of control events for each of the event services that it is connected to. Therefore a request may be forwarded to a gateway by publishing the relevant control event to an event service to which the gateway is connected. A request may be propagated over many gateways and event services in this fashion to reach a particular event service. In addition control events may be passed to a gateway by other means such as user input or via command line parameters.

4.2. FES Gateways and Adapters

The FES is realized by a set of event services that are connected by gateways. Gateways subscribe to their direct event services via adapters for control events.

When a gateway receives a control event it examines the event's distribution list to determine whether the request contained within should be applied at a direct event service and/or whether the event should be forwarded to other gateway(s) for application at indirect event service(s).

If the request should be applied at a direct event service then the gateway unwraps the request details and carries out the necessary request. For example, if the request is a subscription request, then the control event contains a filter. The subscription request is then made via the event service's adapter. If for example the request is a publication request then the control event contains an event. This event is extracted and published to the event service via the adapter.

If the request should be applied at an indirect event service(s) then the gateway must make a routing decision to decide which of its directly connected event services it should publish the control event to in order to route the request to the correct gateway(s).

The gateway must manage some local state information regarding the requests that it has made to its direct event services. For example, this includes information pertaining to subscriptions that have been made at a direct event service. When a publish request is then received from an adapter, the gateway can determine the distribution list for the event.

The adapter pattern is used to encapsulate heterogeneity among event services in the FES. This includes encapsulating event service requests and the mapping of FES requests to event service specific requests and vice-versa. FES model mapping is an implementation detail of an adapter. Generally, there are three kinds of event mapping that an adapter may perform: user-defined (via configuration information and/or plug-in code), automatic, and combined user-defined/automatic event mapping. The integrity of a control event must be maintained at all times so requests may be applied consistently at event services. The size of control events can vary dynamically since they may contain serialized FES events. Therefore, depending on the maximum event size in a FES system, event services with limited event size may not be suitable as intermediate event services.

The FES adapter interface defines five main methods corresponding to the FES requests described above. The adapter implementation must map these methods and events to event service specific functions and events. If an event service does not support announcements and/or subscriptions then null implementation can be provided for these methods. On start-up an adapter implementation must subscribe to its event service for control events. If an event service does not support filtering then the adapter must do its own filtering to single out control events. Received control events must be passed to the gateway for processing. All other events received by an adapter must be converted to publication control events before passing them to the gateway.

4.3 Using the FES

The following steps outline how the FES may be used to federate heterogeneous event services.

1) Identify the events to be propagated between event services.
2) Select and/or implement appropriate event service adapters.
3) Configure gateways with event mapping information if necessary.
4) Place gateways between appropriate event services to allow inter-event service communication to occur.
5) Event propagation between event services is initiated by forwarding a relevant subscription request to the appropriate gateway(s). This can be generally

achieved by publishing the corresponding control event to any event service in the federation.

4.4. Assessment

The FES architecture addresses some of the issues outlined in section 3. A common, flexible FES event model and the adapter pattern are used to address event model heterogeneity and naming issues. The FES is transparent to event services. Existing event service clients require modification to support dynamic FES requests. Modifications are not required to propagate a static set of event types between event services. Distribution lists can aid scalability. The FES cannot provide end to end request/event context support (e.g. QoS attributes), unless all event services in the request path provide the necessary support. Other issues are left open for future work.

5. Implementing FES Gateways and Adapters

Two approaches were considered for implementation of the FES.

In the *compiled* approach, a configuration file that describes event mappings, event services and gateways is input into a tool that generates FES systems. This tool produces the necessary FES system code including gateways and adapters. Support for different types of event services can be plugged into the tool. This approach produces efficient run-time translation and mapping code, as there is no need to look up and interpret this information at run time. This approach can also produce closer mappings to event service APIs and interface languages. However, a change in the configuration will require a re-build of the system or parts of the system and a reinstallation.

The *interpreted* approach requires the development a generic gateway component and an adapter for each event service. Gateways and adapters read event mapping and configuration information on startup and apply this information when translating and mapping data. This approach produces slower run-time code than the compiled approach as configuration information is accessed and interpreted at run time for each event and request. However, a change in the configuration will only require a restart of the relevant FES components.

It was decided to initially implement the interpreted approach as this approach is easier to develop, test and debug. The implementation supports subject based filtering only. To test the FES design the STEAM, Siena and CNS event services were chosen as FES participants. These event services have sufficiently different event models, event services, feature sets and implementations to test the FES design. All adapters implement automatic event mapping, automatically mapping between event service structured events and the FES structured event at run time. The development platform was Visual C++ 6.0 on Windows 2000 Professional. The Win32 TAO CNS implementation was used [12, 13].

Implementing the STEAM adapter was relatively straightforward. STEAM proximity information is mapped to a FES “Proximity” attribute. The subject based filtering of STEAM easily maps to the FES filtering requirement.

The CNS adapter maps the CNS priority attribute to a FES “Priority” attribute. The CNS allows filtering on any part of a CNS structured event. The CNS adapter maps the FES event subject to the `event_name` field of a CNS structured event header.

The Siena C++ API does not support the event push propagation model. Therefore the Siena adapter manages a separate thread to pull events from Siena and push these events to the gateway. The Siena structured event maps well to the FES event. However, Siena has no concept of an event subject. Therefore for automatic event mapping the Siena parameter “FES_Subject” is used by the adapter implementation to specify the subject of the event Siena filtering supports filtering on any parameter in the event.

6. Using FES Gateways and Adapters

The following use case describes a traffic monitoring system that is composed of three heterogeneous event services that are federated via the FES. This system monitors traffic speeds at various locations in a city and logs the license number and speed of vehicles that exceed speed limits. In this system, vehicles broadcast various events over an ad hoc wireless network using the STEAM event service that include the current speed of the vehicle. The current speed of the vehicle is published every second via a “Speed” event on the car’s onboard real time network via a real time event service (RTES). This event contains the car’s current speed and its license number. A FES gateway is used to inter-work the RTES and STEAM event services. Fixed roadside traffic monitors located at or near speed limit signs subscribe for these speed events and publish them on a wide area fixed Siena event service. Each monitor contains a FES gateway inter-working the STEAM and Siena event services. The subscription filter employed at the STEAM event service in each monitor depends on the speed limit in the area. In the city traffic control office there exists a traffic control application. This application allows the operator to set the speed limits for various areas in the city. The roadside signs dynamically display the current speed limit. In addition, setting a speed limit changes the corresponding subscription to the STEAM event service at the roadside monitor.

Figure 2 outlines the configuration of the test application that we developed to simulate this use case. Here CNS acts as the RTES. *G1* is a CNS/STEAM gateway. It is passed simulated GPS locations to 'move' it between traffic monitors. *CNS PUB* is a CNS publisher in the 'vehicle' that publishes varying "Speed" events every second to the CNS event service. *G2* and *G3* act as the roadside STEAM/Siena gateways. *SIENA SUB* is a Siena subscriber, subscribing for specific "Speed" events. The STEAM event service is collocated with the relevant gateways. On start-up a 'hardwired' subscription request is issued to G1 to specify a filter of "Speed", with a distribution list of "Cns" and with the source specified as "Siena".

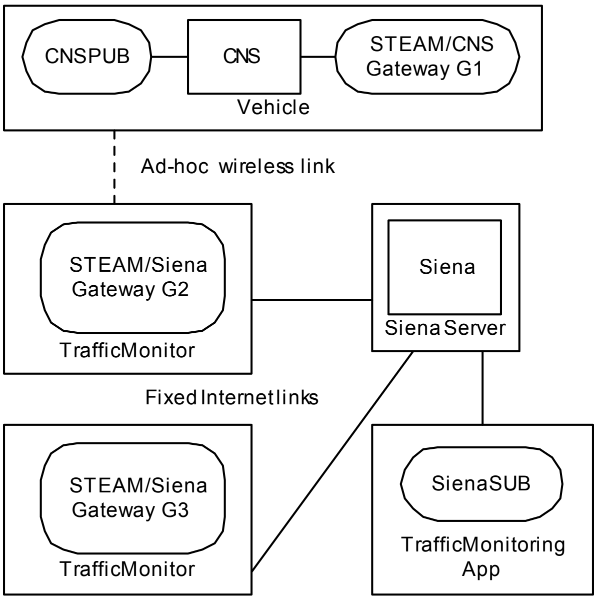


Figure 2. Test FES Application - Traffic Monitoring System.

7. Conclusions

This paper presented the design of the Federated Event Service (FES) – a system for inter-working and federating heterogeneous event services. A proof of concept implementation and test application were presented in order to show that distinct event services can be federated with the FES.

Several issues pertaining to heterogeneous event service federation were outlined and discussed. Some of the raised issues as well as issues related to request tunneling, automatic configuration, and federation monitoring remain open for future research.

References

[1] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Design and Evaluation of a Wide-Area Event Notification Service," *ACM Transactions on Computer Systems*, vol. 19, pp. 283 - 331, 2001.

[2] Object Management Group, *CORBAservices: Common Object Services Specification - Notification Service Specification, Version 1.0.1*: Object Management Group, 2002.

[3] R. Meier and V. Cahill, "Exploiting Proximity in Event-Based Middleware for Collaborative Mobile Applications," in *Proceedings of the 4th IFIP Int. Conference on Distributed Applications and Interoperable Systems (DAIS'03)*, *LNCS 2893*. Paris, France: Springer-Verlag Heidelberg, Germany, 2003, pp. 285-296.

[4] M. Aleksy, M. Schader, and A. Schnell, "Implementation of a Bridge Between CORBA's Notification Service and the Java Message Service," in *Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS)*. Big Island, Hawaii, USA, 2003.

[5] J. Bates, J. Bacon, K. Moody, and M. Spiteri, "Using Events for the Scalable Federation of Heterogeneous Components," in *Proceedings of the 8th ACM SIGOPS European Workshop: Support for Composing Distributed Applications*. Sintra, Portugal, 1998, pp. 58-65.

[6] J. Bacon, A. Hombrecher, C. Ma, K. Moody, and W. Yao, "Event Storage and Federation Using ODMG," in *Proceedings of the 9th International Workshop on Persistent Object Systems (POS 2000)*, vol. LNCS 2135. Lillehammer, Norway, 2000, pp. 265-281.

[7] R. Meier and V. Cahill, "Taxonomy of Distributed Event-Based Programming Systems," in *Proceedings of the Int. Workshop on Distributed Event-Based Systems (ICDCS/DEBS'02)*. Vienna, Austria, 2002, pp. 585-588.

[8] Object Management Group, *CORBAservices: Common Object Services Specification - Event Service Specification*: Object Management Group, 1995.

[9] J. Kaiser, C. Brudna, C. Mitidieri, and C. Pereira, "COSMIC: A Middleware for Event-Based Interaction on CAN," in *Proceedings of the 9th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA2003)*. Lisbon, Portugal, 2003.

[10] C. Wang, A. Carzaniga, D. Evans, and A. L. Wolf, "Security Issues and Requirements for Internet-scale Publish-Subscribe Systems," in *Proceedings of the 35th Hawaii International Conference on System Sciences (HICSS)*. Big Island, Hawaii, USA, 2002.

[11] Object Management Group, *The Common Object Request Broker: Architecture and Specification, Revision 3.0; Chapter 3: OMG IDL Syntax and Semantics*: Object Management Group, 2002.

[12] D. C. Schmidt, "Real-Time CORBA with TAO (The ACE ORB)," www.cs.wustl.edu/~schmidt/TAO.html, 2004.

[13] T. Harrison, D. Levine, and D. Schmidt, "The Design and Performance of a Real-Time CORBA Event Service," in *Proceedings of the 1997 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97)*. Atlanta, Georgia, USA: ACM Press, 1997, pp. 184-200.

Self-stabilizing Routing in Publish-Subscribe Systems

Zhenhui Shen and Srikanta Tirthapura
Department of Electrical and Computer Engineering
Iowa State University
Ames, IA 50011
{zhshen,snt}@iastate.edu

Abstract

Publish-subscribe systems route events to interested subscribers through a distributed network of routing tables. We present a self-stabilizing algorithm that maintains these routing tables in a consistent distributed state, and recovers from faults in the network. Neighboring message routers periodically exchange their routing table state, and take corrective actions if (and only when) necessary. We formally prove that the resulting algorithm brings the system back to a legal global state if it starts out in a faulty state.

Further, we show how to reduce the size of the periodic message exchanges by exchanging "sketches" of the routing tables which are much smaller than the routing tables themselves. We present a message size/accuracy tradeoff of using these sketches, which are based on Bloom filters. We have simulated our algorithm, and present our results of studying the important special case of a transient edge failure in greater detail.

Due to space constraints, we will give an overview of our results in this paper, and we refer to our technical report for further details.

1 Introduction

Publish-subscribe systems provide a loosely coupled middleware to distributed applications. In such systems, messages from senders (or publishers) are routed to receivers (or subscribers) based on their content, rather than on a fixed destination address. Receivers in turn express their interest in receiving a certain class of messages by submitting *subscriptions*, which are predicates on the message content. If the publish-subscribe systems are to be scalable to large networks, then the event routing must be performed in a in a distributed fashion. Many systems such as [CRW01, ASS^{+}99, Muh02] show how to construct a distributed network of routing tables to accomplish this task.

In this work, we are concerned with the fault tolerance of the routing tables. Faults are inevitable in such a distributed system built over a wide-area network. For example, a loss of a subscribe/unsubscribe message could lead the system to a state where the routing tables are inconsistent with each other. A more serious fault might be the corruption of a routing table, or even a router crash. Any fault detection/recovery mechanism must be *decentralized*, and should have a low overhead of detection/correction.

We propose a fault detection and recovery mechanism for distributed publish-subscribe networks based on *self-stabilization*. Self-stabilization is a general fault-tolerance technique, introduced by Dijkstra[Dij74]. Informally, a system is self-stabilizing if, starting from an arbitrary initial global state (perhaps faulty), it quickly reaches a "legal" global state. The advantage of self-stabilization is that it addresses all faults through an uniform mechanism. Rather than enumerate all possible faults that could occur in the network and take corrective actions for each of them, we present one set of rules and actions which handles all possible faults in the router state. When these actions are consistently followed, the system will be quickly restored to a "legal" state after router faults.

The correctness of the publish-subscribe system is a global system property. No single node in the system will be able to say whether or not the system is in a correct state. However, we show that the predicate specifying that the global system state is correct can be written as the conjunction of many local predicates, each of which can be checked in a decentralized way using local actions. This property will help us in designing a local algorithm for the fault-tolerance. We prove that our algorithm leads the system to a legal state, and the time taken is proportional to the diameter of the graph.

If implemented in a straightforward way, self-stabilization presents a large message overhead. At first glance, it seems necessary for the nodes to pass their complete routing tables to their neighbors. The routing tables are typically large (a few thousands of subscriptions), and since this exchange has to take place periodically, this could lead to significant message traffic. To reduce this overhead of checking, we propose to communicate small space "sketches" of the routing tables, instead of the whole routing tables themselves. Our sketches are based on Bloom Filters[Blo70]. These can be used to detect inconsistencies between routing tables very accurately, and yield space improvements of two orders of magnitude.

We formally analyze the space/accuracy tradeoff of checking using our sketches. Our analysis of the false positive probability of using Bloom filters to check set equality is novel. All previous analyses of Bloom filters focused on the false positive probability for checking the set membership.

In summary, our contributions are as follows:
(1)We present a local, self-stabilizing algorithm for adding fault-tolerance and recovery to publish-subscribe systems. We prove that irrespective of what global state the system begins in, it will reach a legal global state, and quantify the time (number of parallel steps) taken to do so.
(2)We present a way to reduce the overhead of fault-tolerance by communicating "sketches" of the routing tables as opposed to the routing tables themselves. We analyze the space-accuracy tradeoff provided by these sketches.
(3)We have simulated the self-stabilization algorithm, and these simulations reinforce our theoretical analysis. We found that though the algorithm is correct in all cases, it needs further tuning, and may not lead to the most efficient recovery in every case. We have analyzed an important special case, that of a transient edge failure, and this demonstrates how the choice of a timeout for initializing the fault recovery can be very important for the overall performance.

Related Work: Existing work study the fault tolerance problem from the perspective of topological changes, such as tree partitioning, grafting/pruning of branches. Siena[CRW01] briefly suggests using system primitives, like subscribe/unsubscribe, to allow subtrees to be merged or to be split. In [PCL03], the authors argue that subscribing and unsubscribing should be treated asymmetrically and propose an optimization over the Siena approach. In another paper [CFLP04], they optimize the special case of a link failure and a link formation occurring in parallel and push the reconfiguration overhead for this special case to a minimum.

On the other hand, some systems like Scribe[RKCD01] and Hermes [PB02] are built on top of a distributed hash table, which take advantage of the peer-to-peer routing substrate to achieve fault-tolerance.

All current work handles a limited range of faults such as message losses and link formation/failure; they do not provide a comprehensive mechanism like we do.

2 Model

We deal with a publish-subscribe network whose nodes are organized into a single spanning tree. We assume that all communication links are FIFO. Each node holds a routing table. Many data structures have been proposed for fast matching and forwarding of events[CW03, ASS$^+$99]; we will not be concerned with the exact form of this data structure. For our purposes, the routing table is a set of tuples of the form (sub, R) where sub is a subscription, and R is a set of neighboring nodes from which the subscription was received. If any event arrives that matches sub, then it is forwarded to all nodes in R, except for the node from which the event arrived.

Self-stabilizing algorithms can be built in a modular fashion[GM91]. Our algorithm stabilizes the state of the routing tables on a tree-based publish-subscribe system. This can be layered on top of another algorithm which stabilizes the spanning tree itself.

3 Self-stabilization Algorithm

The self-stabilization algorithm is concerned with the consistency between the routing tables held by the nodes. Neighboring nodes periodically exchange the states of their routing tables. A node takes corrective actions if its routing information is inconsistent with some neighbor. Each node makes local corrections independently and asynchronously. Through a sequence of local corrections, we restore the consistency among the distributed routing tables.

3.1 Local Legality Implies Global Legality

The definitions of legal local/global states lie at the core of our algorithm. Before defining them, we first introduce some notations and concepts.

An undirected edge connecting nodes a and b is denoted by $\{a, b\}$. It is composed of two directed edges, denoted by (a, b) and (b, a). For node v, let $N(v)$ denote the neighbor set of v. If we remove edge $\{a, b\}$ from the tree T, the whole tree is divided into two parts. The subtree rooted at a is denoted by T_a^b and the subtree rooted at b is denoted by T_b^a.

For directed edge (a, b), the *filter* $F_{a\to b}$ is the union of all subscriptions registered at node a, which sends matching events to node b; in other words, it is the set of all subscriptions that a has so far received from b. The set S^a_b is the union of all subscriptions that are generated by nodes in the subtree T^a_b.

Definition 1 *A system is* quiescent *if there are no subscribe/unsubscribe messages in transit.*

Suppose the neighbor set of b, $N(b) = \{n_1, n_2, \ldots n_k, a\}$. Let $X_{b\to a}$ denote the subscriptions in transit from b to a and $Y_{b\to a}$ denote the unsubscriptions in transit from b to a. Let L_b denote the local subscriptions issued by node b. We first define what it means for an edge to be locally legal.

Definition 2 *The* directed edge (a, b) *is* locally legal, *iff* $F_{a\to b} \cup X_{b\to a} - Y_{b\to a} = \cup^k_{i=1} F_{b\to n_i} \cup L_b$.

Definition 3 *The* undirected edge $\{a, b\}$ *is* locally legal, *iff both* (a, b) *and* (b, a) *are locally legal.*

We now define what it means for an edge to be globally legal.

Definition 4 *If the system is quiescent, then edge* $\{a, b\}$ *is globally legal, iff* $F_{a\to b} = S^a_b$ *and* $F_{b\to a} = S^b_a$.

We assume that every received subscription will be forwarded unless it is a duplicate. Definition 4 holds under this assumption. Regarding any other optimization in subscription forwarding, the computation of S and F can be adapted to keep Definition 4 valid.

Definition 5 *A publish-subscribe system is in a legal state if one of the two conditions holds:*
(1)it is quiescent and all edges are globally legal or
(2)it can be reached from a legal quiescent state by a finite sequence of transitions.

The global legality of edges is hard to check directly, since it is a predicate that involves the state of the whole system. However, the local legality of an edge can be (more) easily checked. We now state a theorem which shows that the predicate defining the global legality of the system can be written as the conjunction of many local predicates, one for each edge.

Theorem 1 *The publish-subscribe system is in a legal state iff every edge is locally legal.*

This proof and the following proofs are all omitted due to space constraints. We refer interested readers to our technical report[ST04].

3.2 The Edge Stabilization Algorithm

Given the above theorem, we only need to stabilize each directed edge into a legal state, and the system will reach a globally legal state. It is easy to set a (faulty) directed edge to a legal local state using appropriate subscriptions/unsubscriptions. However, stabilizing a faulty edge might "disturb" a neighboring edge, and cause it to move from a legal to an illegal state, so that the global state is still illegal. Informally speaking, such "disturbances" can flow only along a simple path in the tree, and have to eventually stop at a leaf. Thus, eventually the system will reach a globally legal state.

A timer is assigned to each directed edge in the network, and the (directed) edge stabilization procedure is initiated upon expiry of the timer. The period of the timer controls the frequency of stabilization, and hence the message overhead (more discussion of the timer appears in Section 5). The source node of a directed edge is responsible for the stabilization. A single round of the procedure consists of two phases: an observe phase followed by a correction phase. We describe the algorithm for directed edge (a, b). All the directed edges are being stabilized in parallel in this manner.

Variables:

```
$N(a) = \{n_1, n_2, \ldots, n_k, b\}$
$S(a) = \cup^k_{i=1} F_{a\to n_i} \cup L_a$
$C_1 = S(a) - F_{b\to a}$
$C_2 = F_{b\to a} - S(a)$
```

Actions at Node a
Event: timeout at t_1 (observe phase)

```
compute $S(a)$ at time $t_1$
send an "observer" to $b$
reset the timer for the next round
```

Event: get the reply from b (correction phase)

```
(comment: $b$'s reply is $F_{b\to a}$)
if ($S(a) = F_{b\to a}$)
    return;
else
    compute $C_1, C_2$
    send $C_1, C_2$ to $b$
```

Actions at Node b
Event: receive an "observer" from a (observe phase)

```
compute $F_{b\to a}$
return $F_{b\to a}$ to $a$
```

Event: receive C_1 from a (correction phase)

```
subscribe to each record contained in $C_1$
```

Event: receive C_2 from a (correction phase)

```
unsubscribe to each record contained in $C_2$
```

It is important to note that the correction phase at node b is initiated only if edge (a, b), and hence the whole system was not in a legal state. Thus, if the system is in a legal state, then the self-stabilization will not add any additional subscriptions/unsubscriptions to the system.

Theorem 2 *Starting from any initial state (perhaps faulty), if (1)no further faults occur and (2)every directed edge in the tree executed the above stabilization process, then the system will reach a legal global state.*

4 Reducing the Message Overhead

An important component of the local stabilization algorithm is the checking of the equality between the two tables $S(a)$ and $F_{b\to a}$. One way to do this is to send the entire table $F_{b\to a}$ across from node b to node a, but this would result in a large message overhead for the following reasons:
(1)The objects being sent across and compared are large sets of subscriptions. These routing tables might contain thousands of subscriptions, and if each subscription takes a few tens of bytes, then these messages would be of the order of a few hundred kilobytes or more. In addition, comparing these large sets would be significant computational overhead. (2)Self-stabilization is a periodic system behavior, which further exacerbates the above problem.

Our approach to reducing this overhead is as follows. Instead of sending the entire routing tables across, we send only a sketch of the table to the neighboring node. This sketch takes much smaller space than the table itself. These sketches are compared at the neighboring node, and if they are found to be inconsistent, then there must be a fault, and now a full comparison of the routing tables is initiated to recover from the fault. For the common case when the routing tables are consistent with each other, the whole routing table will not have to be sent across, leading to a very efficient checking process.

However, these sketches are inherently lossy. There is some probability that the routing tables are actually inconsistent, but the sketches do not reveal it. We are able to quantify this probability of a false positive, and we show that a sketch which uses only a few bits per subscription is able to provide a false positive probability of less than 10^{-3}.

We note that this is the first work, to our knowledge which suggests compression of messages in this way in a self-stabilizing algorithm. We summarize the desirable properties of a sketch:

(1) The size of the sketch should be small compared to the original routing table size
(2) It should be able to detect inconsistencies with high probability, and with low computational overhead
(3) The cost of maintenance should be low, i.e. every time a subscription or unsubscription is received, we should be able to update the sketch quickly.
(4) Since we need to compute the union of routing tables while checking, we need to be able to (quickly) compute the sketch of the union of sets given the sketches of the sets.

We considered various techniques for maintaining these sketches, including hashing and checksums. Our final solution, which satisfies all the above properties, is based on *Bloom filters*. Below we analyze the various tradeoffs associated with using a Bloom filter for the purpose of testing equality between sets.

4.1 Bloom Filter for Testing Set Equality

A Bloom filter is a compact representation of a set to support membership queries. It was invented by Burton Bloom in 1970 [Blo70]. In [FCAB00], it is shown how to derive the probability of a false positive for a membership query in using the Bloom filter.

In self-stabilization, we do not use a Bloom filter to test for set membership, but to compare if two sets are equal. More precisely, we want to check if the union of a few sets ($S(a) = \cup_{i=1}^{k} F_{a\to n_i} \cup L_a$) equals another set ($F_{b\to a}$). This calls for a new analysis of the tradeoffs between the false positives and the parameters of the Bloom filter. We now sketch our analysis of the false positive probability for the context of set equality, and graph the resulting tradeoffs obtained.

Let B_S denote the Bloom filter of set S. Let m denote the size (in bits) of the Bloom filter, and k the number of hash functions. We want to compute the probability that B_A and B_B are equal, though A and B are unequal. Let $\alpha = e^{-\frac{k|B|}{m}}$ and $\beta = e^{-\frac{k|A|}{m}}$. Let p denote the false positive probability, i.e. the probability that $B_A = B_B$ though $A \neq B$.

Theorem 3

$$p \leq \min\{p_A, p_B\}$$

where

$$p_A < \alpha \cdot \prod_{i=2}^{k\cdot|A-B|} \left(\frac{i-1}{m} + \left(1 - \frac{i-1}{m}\right)\cdot\alpha\right) \quad (1)$$

and

$$p_B < \beta \cdot \prod_{j=2}^{k\cdot|B-A|} \left(\frac{j-1}{m} + \left(1 - \frac{j-1}{m}\right)\cdot\beta\right) \quad (2)$$

We now sketch the false positive probabilities for various values of $k, m/n$ and $|A-B|$ in Figure 1. Clearly, the probability decreases (leading to a more accurate test for equality) when the difference between the sets is getting large. As k increases, the computation overhead for maintaining the Bloom filter increases. The parameter m/n is the number of bits used per element. As it increases, the space overhead also increases, but the false positive probability decreases. Thus, by using 4 bits per element and 2 hash functions, the false positive rate for sets differing by 3 elements is only about 0.001, and this can be further decreased by increasing k or m/n.

5 Simulations

The self-stabilization procedure is a periodic system behavior, whose period is controlled by a

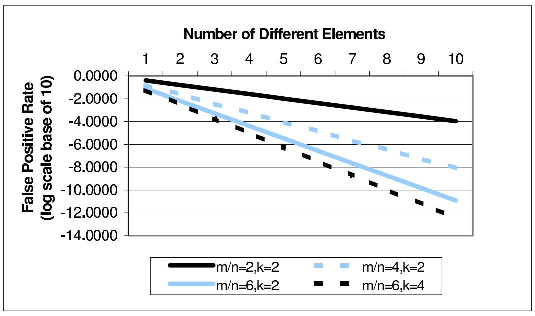


Figure 1. The Probability of a False Positive Under Various m/n and k Combinations.

The y-axis is the error probability, in a log-scale. The x-axis is the size of the difference, $|A - B|$.

timer(Section 3.2). Since this period is fixed, the procedure is executed at regular intervals. However, this regular behavior does not always yield the best performance, though it is always correct. One special case is that of a transient edge failure, which we consider now.

In face of an edge failure, some subscriptions become obsolete, since they were generated by nodes in a partition which is now unreachable. These subscriptions have to be (usually) removed, else the endpoints of the broken edge will continue to receive useless events. But, if the broken edge comes back up quickly, then these obsolete subscriptions become useful again. In such a case, it is better to delay the unsubscribing after the link failure.

Since the setting of the timer is independent from the event of link failure, there is no synchronization between the time of the link failure and the activation time of the next round of stabilization. As a result, the next round of stabilization might kick in soon after the link failure, which may not be ideal for the performance.

A better approach is to reset the timer based on the network condition. This leads to our *adaptive strategy*, which delays the stabilization timeout until the node receives a sufficient number of unwanted events, rather than depending on a fixed timeout.

We conclude by presenting some experimental results which compare various timeout strategies. Our performance metric is the *reconfiguration overhead*, which is defined to be the total number of hops traversed by both unsubscribe/re-subscribe messages and unwanted events under a single link failure. We study the following three strategies:
(1)The "strawman algorithm" (the name borrowed from [PCL03]). Upon a link failure, it resets the timer and initiates the stabilization without delay.
(2)The "static algorithm" with no change to the preset timer.
(3)Our "adaptive algorithm". Upon a link failure, the timer is reset, but the stabilization is activated once a sufficient number of unwanted events are received, or when the link is re-formed.

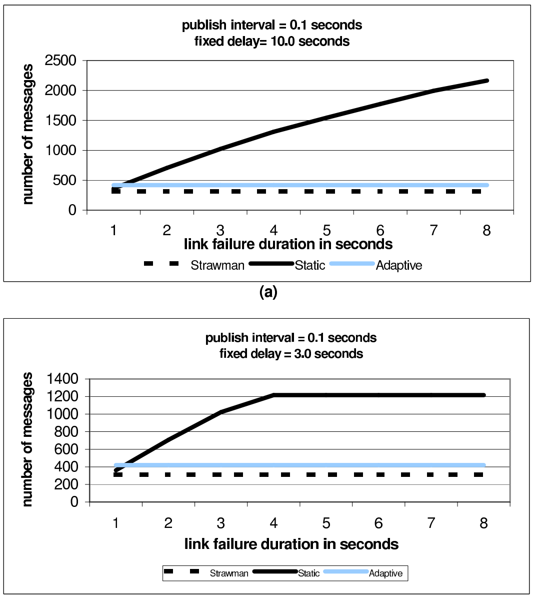
5.1 Simulation Setting

In our simulations, both an event and a subscription are chosen to be 3-character random strings. An event matches a subscription if the two strings are identical. The topology is a single spanning tree, consisting of 100 nodes.Each simulation scenario is uniquely identified by a combination of the following parameters:

Publish Rate: The publish rate regulates the system load. We simulate two scenarios: a *light system load* using a publish interval of 5.0 seconds and a *heavy system load* using a publish interval of 0.1 seconds.

Subscribe Rate: The subscribe rate controls the density of the subscriptions. We set the subscribe interval to be 2.0 seconds. In addition, each router can subscribe to at most 20 event patterns.

Fixed Delay: This is the timeout used by the "static" algorithm. We choose two timeout values: a longer one of 10 seconds and a shorter one to be 3 seconds.

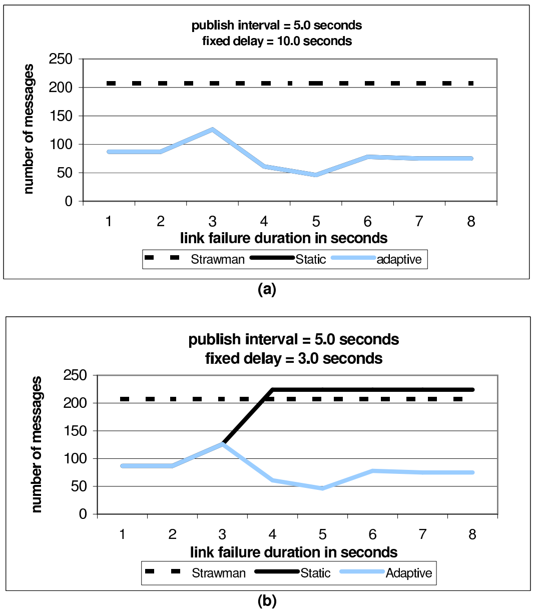


(a)

(b)

Figure 2. Reconfiguration Overhead Under Heavy System Load

Under a heavy load (Figure 2), the endpoints expect to receive more unwanted events. It's ideal to unsubscribe early to limit the increasing cost of unwanted events. In Figure 2, the strawman curve has the lowest



(a)

(b)

Figure 3. Reconfiguration Overhead Under Light System Load

overhead, as it unsubscribes without delay. The adaptive curve has a slight increase, for it delays unsubscribing a little bit. The static algorithm has a poor performance, but a shorter timeout brings down the overhead by 50%.

Under a light load (Figure 3), the cost of unwanted events is negligible due to the rare occurences of unwanted events. It is better to delay unsubscribing. Therefore any static algorithm with large timeout performs well under this condition. Meanwhile the adaptive algorithm also yields the same amount of cost. This time both the strawman and the static strategy with short timeout value generate huge overhead. As a comparison, our adaptive algorithm saves two thirds of the cost of the strawman approach.

In summary, neither the *static* nor the *strawman* do well for both the lightly loaded and heavily loaded cases. However, the *adaptive* algorithm for triggering the reconfiguration shows a good (though not optimal) performance in both cases.

References

[ASS$^+$99] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 1999.

[Blo70] Burton Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 13(7), Jul 1970.

[CFLP04] Gianpaolo Cugola, David Frey, Amy L.Murphy, and Gian Pietro Picco. Minimizing the reconfiguration overhead in content based publish subscribe. In *Proceedings of the 19th ACM Symposium on Applied Computing (SAC04)*, Mar 2004.

[CRW01] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.

[CW03] Antonio Carzaniga and Alexander L. Wolf. Forwarding in a content-based network. In *Proceedings of ACM SIGCOMM 2003*, pages 163–174, 2003.

[Dij74] E. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the ACM*, 17:643–644, 1974.

[FCAB00] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[GM91] M. G. Gouda and N. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40:448–458, 1991.

[Muh02] G. Muhl. *Large-Scale Content-Based Publish/Subscribe Systems*. PhD thesis, Darmstadt University of Technology, 2002.

[PB02] Peter Pietzuch and Jean Bacon. Hermes: A distributed event-based middleware architecture. In *In Proceedings of the 1st International Workshop on Distributed Event-Based Systems (DEBS'02)*, July 2002.

[PCL03] Gian Pietro Picco, Gianpaolo Cugola, and Amy L.Murphy. Efficient content-based event dispatching in the presence of topological reconfiguration. In *Proceedings of the 23rd International Conference on Distributed Computing Systems(ICDCS'03)*, pages 234–244, 2003.

[RKCD01] Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. Scribe: The design of a large-scale event notification infrastructure. In *Proceedings of the Third International COST264 Workshop (NGC 2001)*, nov 2001.

[ST04] Zhenhui Shen and Srikanta Tirthapura. Self-stabilizing routing in publish-subscribe systems. Technical Report TR-2004-04-4, Iowa State University, Electrical and Computer Engineering, 2004. http://archives.ece.iastate.edu/archive/00000033/.

Fault-Tolerance in the SMILE Stateful Publish-Subscribe System

Robert E. Strom
IBM TJ Watson Research Center
strom@watson.ibm.com

Abstract

We present the algorithms for achieving service guarantees in the SMILE distributed relational publish-subscribe system in the presence of lost or reordered messages due to broker and/or link failures.

SMILE extends the content-based publish-subscribe paradigm by allowing subscribers to request continually updated derived views, specified as relational algebraic (SQL-like) expressions over published event histories.

The SMILE system performs compile-time analysis of subscriptions, and generates tailored code for computing incremental state updates, and for detecting and recovering from lost or permuted messages. We exploit: (1) the language's monotonic type system, and (2) a refined service guarantee of eventual correctness.

We first present an abstract protocol capturing the common features of all fault detection and recovery protocols implemented in the SMILE prototype. We then describe the specialized protocols for particular relational operators. We discuss potential optimizations and performance tradeoffs.

1. Introduction

SMILE (Smart MIddleware, Light Ends) is a distributed publish-subscribe system with enhanced computational capabilities. In SMILE, clients subscribe to states rather than to individual events as in conventional pub-sub systems such as SIENA[3]; the states represent continually updated views derived from event histories by relational algebraic expressions that may involve operators such as aggregation, join, and top-k.

SMILE is targeted to large-scale deployments (e.g. thousands of clients, hundreds of distinct views). The process of deploying and running a SMILE system includes the following steps:

- Producers define the schema (type and structure) for the event streams they wish to advertise.
- Consumers specify their subscriptions as views derived from one or more producer event streams (or from other derived views). Specifications are written in a declarative *Middleware Programming Language (MPL)* based on relational algebra.
- Network administrators define a broker network that will host the middleware implementation.
- A compilation service consolidates and compiles the initial subscription set into an optimized *delivery plan* organized as a *transform graph* consisting of *relation* and *transform* objects that incrementally compute and store derived views in response to changes in inputs, and that recover from faults.
- A *placement service* takes the transform graph together with information about the broker capacities, network topology, and location of publishers and subscribers, and determines the best assignment of relations and transforms to nodes, in order to optimize latency and throughput. A *deployment service* uploads the code to the appropriate brokers.
- During system operation, events from producers are logged, and are then propagated through the transform graph within the broker network, causing views to be recomputed and additional state update messages to propagate. Updates to subscribed views are delivered to client applications. The transform objects embed protocols that are resilient to lost, duplicated or reordered messages due to link failures, and to loss of state due to broker crashes and restarts.
- If performance parameters change, or if a broker becomes unavailable, the system reacts by modifying and/or reassigning the objects in the transform graph. If subscriptions change, the system may need to dynamically recompile the changes and decide whether to re-optimize or reassign other objects.

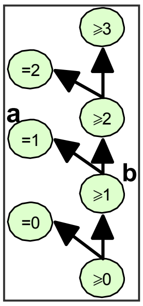
The general architecture of SMILE has been presented in a previous paper [7]. The present paper describes the details of the algorithms that augment the incremental transforms in order to tolerate lost and reordered messages, and broker failure. Section 2 presents the features of the monotonic language model and of the "eventual correctness" guarantees, which our algorithms exploit. Section 3 presents the fault-tolerance algorithm in its most abstract general form. Section 4 presents the concrete instances of this algorithm, tailored to each kind of relational operator, illustrating their application by means of an example that uses most of SMILE's relational operators. Section 5 discusses open problems and related work.

2. SMILE Model Summary

SMILE is based on a relational model. Events enter the system associated with a specific *topic*. The history of events on any topic corresponds to a *base relation.* Base-relations are append-only. All other relations are called *views*. Views cannot be independently modified; their contents are defined by algebraic expressions defining them in terms of either base relations or other views. Expressions are mapped to an intermediate language modeled after Date and Darwen's Tutorial-D[6]: operators include **select**, **project**, **extend**, aggregation (**sum**, **min**, **max**), **join**, **merge**, and **top-k**.

2.1. Relations

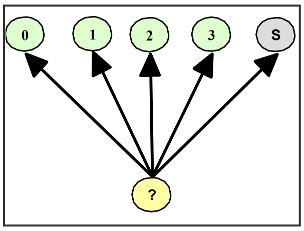
SMILE relations are statically typed. The type of a base relation is given by the client who defines and advertises a new topic. The type of a view is computed by the compiler, based on the algebraic expression defining the view. Relations are modeled as maps from 0 or more *key* domains to 0 or more *non-key* domains. A map is abstractly represented as a table of rows. Each row has a column for each key domain and a column for each non-key domain. There are as many rows as there are possible values of the key domains. Relations may vary over time in the following constrained ways: Values in key columns are fixed. Values in non-key columns may change only by progressing from "lower" to "higher" values in a partially ordered *monotonic domain.* ("Lower" and "higher" do not necessarily correspond to algebraically smaller or larger; "higher" should be viewed as "has more information".) If value *a* is higher than or equal to value *b*, it is said to *imply* value *b* (*a* ⇒ *b*). Each domain has a unique bottom element which is the initial value; there may be multiple "highest" values, called *final* values. The illustration shows a simple domain of integers. In this domain, **a** ⇒ **b** and **a** is final. Implication is extended in a straightforward way to relation values: if R1 and R2 are different possible values of a relation R, then R1 ⇒ R2 iff each value v1 in R1 implies the corresponding (same row key and column name) value v2 in R2.



Certain non-key values are conventionally displayed as "invisible" when viewed by users; any row whose non-key values are all invisible is not shown. What appears to users to be insertion or deletion of rows is represented in our formalism by a change of columns from invisible to visible values or from visible to invisible; the formalism lacks any concept of insertion or deletion. The tables in Figure 1 illustrate conventionally displayed relations; key columns are shown to the left of the heavy line, and non-key columns to the right.

2.2. Event Histories

Base relations represent event histories. Each event has a unique key; by default, this key is either a "time tick" or a "unique id" assigned by the system when the event is first published. The event schema defines the domains of the non-key columns. Each non-key type is extended to a monotonic domain by augmenting its domain with two invisible values: *unknown*, meaning it is not known whether or not an event was published at this time tick; and *silence*, meaning it is known that no event was published at this tick. ***Unknown*** is the bottom value; all other values are final. The illustration shows values in an event history belonging to an integer domain with range [0..3]. "Appending" an event at a particular tick is represented as replacement of an unknown value at that tick with a known value, and replacing the unknown values of prior ticks since the last event with silence values.



If one sorts the rows of an event history by tick order, some prefix of rows is final, each row containing either an event or a silence, and the suffix will be unknown rows.

2.3. Service Guarantee

The safety and liveness guarantees are formalized in our earlier paper [7], and are summarized here:

Safety: Given a snapshot of values of all relations, if C is the value of the derived view computed from the base relations, and D the delivered value, then C⇒D.

Liveness: Given a snapshot, and C defined as above, eventually the system will deliver a value D where D⇒C.

A corollary of these two rules is that if publishers quiesce, eventually D will be precisely equal to C. Because of this corollary, our service guarantee is called *eventual correctness*. Although similar to the *eventual consistency* property used in database replication and in gossip-based systems such as Astrolabe[8], our safety property contains stronger guarantees about the approximate information delivered prior to quiescence. The combination of monotonic domains and the eventual correctness guarantee is essential to the design of our algorithms.

3. Abstract Fault-Tolerance Algorithm

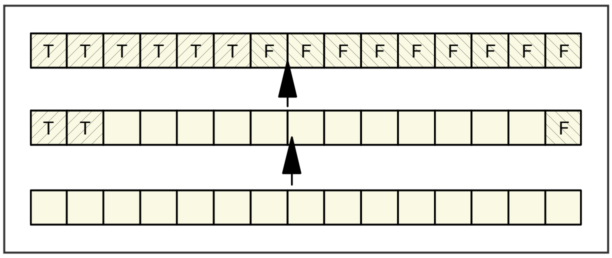
The fault-tolerance protocols presented here are derived from our earlier work on reliable content-based publish-subscribe [1] in two steps: (1) first we generalize the earlier protocols to apply not only to content filtering but also to any means by which monotonic knowledge is propagated from sources to destination by means of incremental monotonic transforms; (2) next we specialize the generalized protocol for particular monotonic transforms, exploiting the properties of the operators and the domains to compress information and reduce the number of messages and the amount of computing needed to recover. In this section we present the result of this first step – the common principles that underlie all specializations. We defer to the next section discussion of the tailored optimizations that make them efficient.

3.1. The Transform Graph

The compiler organizes the subscriptions into a *transform graph* – a directed acyclic hypergraph, where a node corresponds to a relation, and a hyperedge linking one or more relations to a view corresponds to a relational operator defining that view in terms of antecedent relations. Each node is compiled into a corresponding relation object, and each hyperedge is compiled into a corresponding transform object. In general, a hyperedge for a *k*-operand operator will have *k* inputs and one output. We group each transform object with the relation object fed from its single output to form a transform graph component. (Each base relation and the message stream that feeds it is also a transform graph component.) Our protocol allows for the worst case, in which transform graph components are assigned to separate nodes and communicate by asynchronous messages which might be lost, duplicated, or re-ordered. A component has *k* inputs, corresponding to the *k* hyperedge inputs, and may have arbitrary fan-out, depending on how many other transform graph components use their relation as an operand. Communication between neighboring transform graph components is bidirectional; the direction following the arrows of hyperedges is called *downstream*; the opposite direction *upstream*. Figure 1 illustrates a transform graph and sample relation contents for a system with 3 base relations and a subscribed relation Available.

3.2. Knowledge Propagation

Each transform object maintains the state of its relation together with any auxiliary state that may be useful in computing incremental state updates. (In the worst case, the auxiliary state can be a copy of each input relation.) All state is monotonic. A monotonic state can be conceptualized as a set of knowledge "bits" – cells that are either empty, or else contain an indelible mark of T or F. A monotonic type is an arrangement of such cells with rules governing under what circumstances cells may be filled in or not. For example, a monotonic type that reflects the sum of up to 5 integers from a base relation, each of which can be either *unknown, silent*, or an integer in the range 0 to 3 can be viewed as a collection of 15 cells with a possibly null prefix of T's and a possibly null suffix of F's. Initially all 15 cells are empty; this is the bottom state. If one of the summands moves from *unknown* to 2, the first two cells are marked with T, and the last with F, representing a sum of at least 2 and no more than 14. After all 5 integers become known, (e.g. 2, 1, S, 3, 0), the sum will evolve to a value which has no empty cells and represents the final value of 6.



In the absence of failures, transform graph components receive *knowledge* messages from upstream components, update the state of their views according to the transform rules by adding marks to some blank cells, and send new knowledge messages to all downstream components indicating which cells have received which new marks. Depending on the specialization, knowledge messages can either indicate the full value of a cell (e.g. the range [2..14]), or an incremental change (e.g., adding 2 T's on the left and one F on the right). These downstream components in turn receive, transform, and propagate knowledge messages, until subscribing views at the leaves of the transform graph are fully updated.

3.3. Handling duplicates or reversed messages

In a monotonic system, duplicate messages write over the same marks, and can be detected as redundant. Out-of-order messages containing full state can be detected based on the monotonic property of the state: if a message says the new value is a range [6..6], then a subsequent message saying the range is [2..14] is out-of-order and redundant. Out-of-order messages containing partial state are acceptable if the type defines no constraints on the order of marks; otherwise the missing message can be detected by an invalid pattern of marks, e.g. TT---------F might be followed by a message updating the sequence to TT--T----F-F. The presence of unfilled cells in the prefix and suffix indicates a *gap;* as shown below, this will trigger a protocol to request missing messages to fill the blank cells.

3.4. Guaranteeing liveness

It is possible for messages to be lost and yet not show up as gaps. Any relation with non-final columns is susceptible to the loss of messages that might potentially update these columns. Depending on system tuning parameters, either a push (publisher-driven) or a pull (subscriber-driven) protocol is used. The pull protocol involves an intermittent poll asking about the state of non-final cells. This protocol is also employed whenever a gap is detected and is not filled within a time threshold.

In the pull protocol, after a timeout expires, a message is sent from the downstream component containing unfilled cells to the upstream component or components that might be able to supply marks for these cells. The message is called a *curiosity message*; the sender is called the *curious component*; the recipient is called the *satisfying component*. The satisfying component determines whether it has the information requested. If not, the curiosity message is propagated further upstream – base relations as a last resort can satisfy curiosity from the log. If the satisfying component has the information, it retransmits what is needed.

In the push protocol, the satisfying component actively retransmits information until the no-longer-curious component *quenches* it by sending a *negative curiosity message.* If a downstream component detects that its state or some part of it has become final, it may be able to preemptively quench messages.

The techniques of exploiting monotonicity to detect duplicates and gaps, pulling when gaps are detected, and either retransmitting or pulling when state is stagnant, are sufficient to guarantee liveness under reasonable assumptions (namely that if one pulls infinitely often, eventually a retransmission will succeed). The safety requirement is automatically met provided that: (1) the transforms are sound and (2) effects of messages sent as a result of optimistic assumptions are not shown to end-users until those assumptions are validated.

3.5. Soft Checkpointing

Events are logged to stable storage when entering the system, unless the publishing client has already logged them. Logging must precede propagation; otherwise there is a risk that some subscribers will see their effects and (after a crash) others will not. However, in practice, it is advantageous to retain additional stable state in order to avoid the need to replay the entire message history when brokers fail. Such state saving is called *soft checkpointing,* because it is never needed for correctness. Checkpoints can be incrementally updated on an intermittent basis in the background. Once taken, the component taking the checkpoint no longer requires the messages on which the state depends, which may in turn permit upstream components to discard logs. Soft checkpointing is an optimization that permits reclamation of logs, and reduces worst-case time to recover. Soft checkpoints can also be used to relocate transforms between two brokers.

4. Tailored Fault-Tolerance Algorithms

The fault-tolerance algorithms generated by the SMILE compiler are all special cases of the general algorithm described above. Each tailored algorithm specializes the general algorithm by defining (a) the specific monotonic domain used; (b) a representation for that domain that captures the constraints; (c) the content of knowledge messages; (d) the algorithm for detecting gaps and issuing curiosity messages, and (e) the algorithm by which the satisfying relation decides what to retransmit.

We illustrate these specializations by means of the subscription in Figure 1, which uses most of SMILE's repertoire of relational operators. There are three published streams. Sellers posts items for sale, e.g. 550 tickets to World Series Game 6. Buyers West and Buyers East are two separate streams of accepted buys that are merged. The buys are grouped by item-id, then each group is summed over quantity. The Sellers and the summed Buys are then joined and the quantity available is computed. Then the relation is restricted with a **select** operator to show item-id, price, and quantity available for all items that have a nonzero quantity available.

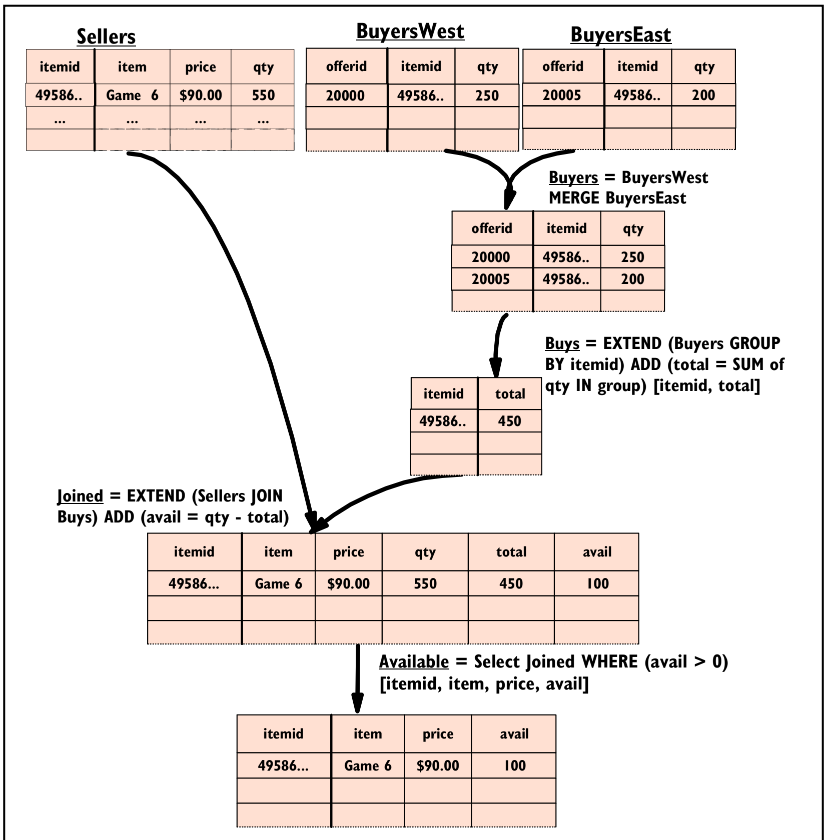
4.1. MERGE transform

To make merging more efficient, offer-ids are generated as timestamps in increasing order. To avoid timestamp conflicts, Buyers West generates events at even ticks and Buyers East at odd ticks. Both source relations contain silence values for all ticks known not to have events. The timestamps for the two streams need not be synchronized, but the algorithm performs better if they don't drift too far apart. Knowledge messages contain either a tuple with its tick, or a *silence range.*

The merged relation is logically a map from a tick to either an unknown value, a silence value, or a tuple <itemid, qty>. Physically, what is stored is set S of tuples keyed by tick, and a *gap descriptor*. A gap descriptor consists of a pair of *horizon* timestamps t_1 and t_2, and a *gap list*. A gap list is an ordered set of tuples <R, f_1, f_2>, where R is a range of ticks [t_i..t_j], f_1 and f_2 are Boolean values (in general, *m* Boolean values when merging *m* source streams), with the following interpretation:

- All f_k = T: Ticks [t_i..t_j] are final because for each tick, either a message was received from one source or silence from all sources.
- Some f_k = T: Ticks [t_i..t_j] are *unknown* but silence has been received from source *k.* (Some other f_k = F.)
- All f_k = F: Ticks [t_i..t_j] are *unknown* because nothing has been received from any source.

The gap list contains consecutive ranges for all ticks after t_1 and before t_2. Ticks up to and including t_1 are known to be final – either an event if stored in S, or else a silence. Ticks at or beyond t_2 are known to be *unknown* (no data from either source). Provided messages are not lost and the timestamps advance at approximately the same rate, gaps are either non-existent or of very short duration. If gaps persist, the gap list is scanned and cu-



Sellers

itemid	item	price	qty
49586..	Game 6	$90.00	550
...	...	...	...

BuyersWest

offerid	itemid	qty
20000	49586..	250

BuyersEast

offerid	itemid	qty
20005	49586..	200

Buyers = BuyersWest MERGE BuyersEast

offerid	itemid	qty
20000	49586..	250
20005	49586..	200

Buys = EXTEND (Buyers GROUP BY itemid) ADD (total = SUM of qty IN group) [itemid, total]

itemid	total
49586..	450

Joined = EXTEND (Sellers JOIN Buys) ADD (avail = qty - total)

itemid	item	price	qty	total	avail
49586...	Game 6	$90.00	550	450	100

Available = Select Joined WHERE (avail > 0) [itemid, item, price, avail]

itemid	item	price	avail
49586...	Game 6	$90.00	100

Figure 1: Subscription with Transform Graph

riosity messages are sent to source *i* where f_i is false requesting reconfirmation of ticks in the range $[t_i..t_j]$. The satisfying relations use an indexed lookup and transmit the events or silences in the requested range.

4.2. GROUPBY + SUM transform

The Buys relation contains a total field, which, though displaying as an integer, has a very different monotonic behavior than the qty field of Buyers. Assuming that the possible range for qty is [0..Q], and the range on ticks is [0..T] (T is arbitrarily large, but it is only used to formulate the constraints), then the total field is a range that begins as [0..Q*T]. Each silence tick reduces the upper bound in all rows by Q; each non-silence event with itemid *j* and qty *k* increases the lower bound by *k* and decreases the lower bound by Q - *k* in row *j* and behaves like a silence for all other rows. We want to perform O(1) work on receiving an event or a silence range; therefore we represent Buys as a mapping from itemid to a single integer representing the lower bound (the display number). If a map entry is not stored, its lower bound is 0 by default. We infer the upper bound of a cell with lower bound *a* to be Q*T - Q*K + *a*, where K is a quantity that we maintain, representing the number of final (value or silence) ticks that have contributed to the sum. We represent K as a gap descriptor with *m=1*, so that we can track the specific ticks and not just the number of ticks, exploiting the fact that usually there will be no gaps and K will equal the horizon timestamp t_1.

When the transform receives a knowledge message, the gap descriptor is examined to determine whether this is a message for an already received final tick (in which case it is ignored), or whether it is a message for a new tick (in which case the sum is accumulated and the gap descriptor updated). As in the MERGE transform, if either a gap or non-progress endures for longer than a threshold, a curiosity message will be sent upstream indicating the ticks known or suspected to be missing.

The knowledge message propagated by Buys contains either: a silence range, or an update consisting of itemid, and total, represented as the pair (*a*, K) where K once again encodes the set of ticks that contributed to the total. In order to respond efficiently to curiosity messages from downstream, Buys additionally maintains a mapping M from itemid to hightick – the tick number of the tick that contributed the highest value of total. The protocol that results if a downstream relation sends curiosity to Buys is illustrated below.

4.3. JOIN transform

The JOIN transform combines information from Sellers and Buys. Ignoring the EXTEND for now, let's just discuss how qty is updated from Sellers and total from Buys. The qty column is a tick-keyed quantity exactly like the one in Buyers, maintained with a gap descriptor with *m=1*, using exactly the same curiosity protocol as between Buyers and Buyer West. The total column is more interesting, as it is a sum of grouped columns. The Joined relation tracks the gap descriptor by updating it from the K field of knowledge messages received from Buyers (computing the union of all ticks contributing to Joined). When there is a gap or non-progress of sufficiently long duration, Joined sends a curiosity message upstream to Buyers indicating a range of missing ticks. Relation Buyers responds as follows: it searches its mapping M for the itemids of items whose hightick values are in the requested range. It retransmits knowledge messages containing itemid and most recent total for those ticks; it transmits knowledge messages saying "don't care" for those ranges of ticks that it has processed, but which don't represent the most recent total for any value. Those ticks represent events that were either silent, or else were superseded by other events – e.g., if ticks 20000, 20003 and 20005 all contributed to the total, but tick 20005 to the latest value, a curiosity about ticks 20000-20005 can be responded to with "don't care" for 20000-20004. There is no need to send updates for 20000 and 20003 because they are superseded by the total based on tick 20005. By using this protocol, a downstream component that lost a whole sequence of messages can often retrieve its state with fewer messages compared to a protocol that retransmitted all lost messages. The Joined relation processes "don't care" by leaving its state alone but updating its K field to indicate that it is no longer curious about those ticks.

The field avail is a monotonic type that has not previously been discussed: its mathematical value can fluctuate from invisible to a positive value, and then downward. It is represented as a pair of numbers representing its separate monotonic positive and negative components.

4.4. SELECT transform

The SELECT transform is straightforward, since the outputs have the same shape as the inputs. Therefore the representation and the curiosity protocols for Available are the same as for Joined. The only difference is that the WHERE clause of each row is evaluated to determine whether the row is currently visible or not. Type analysis determines that for the WHERE expression of this example, the visibility transitions from "temporarily false" (avail unknown) to "temporarily true" (avail known but greater than zero) to "permanently false" (avail known and not greater than zero). Rows that are permanently invisible can of course be deleted from the representation. In future protocols, quenching messages can be sent upstream to prevent delivery of knowledge messages that would have updated fields in invisible rows.

5. Discussion

We believe that exploiting monotonicity allows us to weaken our service specification relative to ACID, or to continuous queries[5][9], while still not losing information subscribers need, and not requiring full message replay for recovery, as in our earlier work[1]. Other guarantees are discussed in [2], [4], and [8].

We have implemented our system for a subset of transforms, but have incorporated few optimizations beyond those discussed here. The protocols discussed here leave room for tradeoffs between eager and lazy propagation, between push-based and pull-based recovery, and among strategies for quenching by introducing dynamic filters. We need to extend our normal-operation performance measurements to include failure scenarios.

6. Acknowledgement

Yuhui Jin contributed significantly to the design and prototype implementation of these algorithms.

7. References

[1] Bhola, S., Strom, R., Bagchi, S., Zhao, Y., and Auerbach, J.: Exactly Once Delivery in a Content-Based Publish-Subscribe System, *Proc. Intl. Conference on Dependable Systems and Networks*, June 2002, Washington D. C.

[2] Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., Zdonik, S.: Monitoring Streams – A New Class of Data Management Applications. VLDB 2002.

[3] Carzaniga, A., Rosenblum, D.S., and Wolf, A.L.: Achieving Expressiveness and Scalability in an Internet Scale Event Notification Service. PODC 2000, Portland, OR, July 2000.

[4] Chandrasekaran, S., and Franklin, M., Streaming Queries over Streaming Data, In Proc. of the 28th VLDB Conference, Hong Kong, China, 2002.

[5] Chen, J., DeWitt, D., Tian, F., and Wang, Y.: NiagaraCQ: A scalable continuous query system for internet databases. In ACM SIGMOD, 2000.

[6] Darwen, H., and Date, C.J. Foundation for Object/Relational Databases: The Third Manifesto. Addison-Wesley. June, 1998.

[7] Jin, Y., and Strom, R.: Relational Subscription Middleware for Internet-Scale Publish-Subscribe, Proc. SIGMOD Workshop on Distributed Event-Based Systems, June, 2003.

[8] Van Renesse R., Birman, K., Vogels, W.: Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. ACM Trans. Comput. Syst. 21(2): 164-206 (2003)

[9] Terry, D., Goldberg, D., Nichols, D., and Oki, B., Continuous Queries over Append-Only Databases. In ACM SIGMOD, pp. 321-330, June, 1992.

Content-based Publish-Subscribe Over Structured P2P Networks

Peter Triantafillou and Ioannis Aekaterinidis
Research Academic Computer Technology Institute and
Department of Computer Engineering and Informatics, University of Patras, Greece
{peter,aikater}@ceid.upatras.gr

Abstract

In this work we leverage the advantages of the Chord DHT to build a content-based publish-subscribe system that is scalable, self-organizing, and well-performing. However, DHTs provide very good support only for exact-match, equality predicates and range predicates are expected to be very popular when specifying subscriptions in pub/sub systems We will thus also provide solutions supporting efficiently subscriptions with range predicates in Chord-based pub/sub systems.

1. Introduction

Publish/subscribe systems are becoming very popular for building large scale distributed systems/applications. The main functionality of pub/sub systems is the delivery of published notifications from every producer (publisher) to all interested consumers (subscribers). Publishers, who are completely unaware of the existence of the consumers, publish events (information) through the system by specifying the values of a set of well defined attributes. The consumers are expressing their interest through appropriate subscriptions and wait until they are informed about a matching event. A publish/subscribe infrastructure is responsible for matching events to related subscriptions and delivering the matching events to interested consumers.

Building a centralized publish/subscribe system has the advantage of having a global image of the system and thus making the matching algorithm much easier to implement. This approach suffers from scalability problems as the number of publications and subscriptions increases. Thus, the decentralized approach is more appropriate. The main challenge in a distributed environment is the development of an efficient distributed matching algorithm.

The peer-to-peer (p2p) paradigm is appropriate for building large-scale distributed systems/applications. P2p systems are completely decentralized, scalable, and self-organizing. A popular class of them is the “structured” p2p systems. The most prominent of these systems are built using a Distributed Hash Table (DHT [7],[8],[9]), which is a mechanism that provides scalable resource look-up/routing.

2. Related work

2.1 Publish/Subscribe systems

There are two types of publish/subscribe systems: i) *topic based* and ii) *content based*. Topic based systems are much like newsgroups. Users express their interest by joining a group (a topic). Then all messages related to that topic are broadcasted to all users participating to the specific group.

Content-based systems are preferable as they give users the ability to express their interest by specifying predicates over the values of a number of well defined attributes. The matching of publications (events) to subscriptions (interest) is done based on the content (values of attributes).

Publish/subscribe systems can be built in a distributed manner, avoiding the lack of scalability and fault-tolerance of centralized approaches. Distributed solutions are mainly focused on topic-based publish/subscribe systems [1], [2], [3]. Some attempts on distributed content-based publish/subscribe systems use routing trees to disseminate the events to interested users based on multicast techniques [4], [5], [15], [16]. Some other attempts use the notion of rendezvous nodes which ensure that events and subscriptions meet in the system [14].

Some approaches have also considered the coupling of topic-based and content-based systems. The authors in [6] used a topic-based system (Scribe [1]) that is implemented in a decentralized manner using a DHT (Pastry [7]). In their approach the publications and the subscriptions are automatically classified in topics, using an appropriate application-specific schema. A potential drawback of this approach is the design of the domain schema as it plays fundamental role in the system’s performance. Moreover, it is likely that false positives may occur.

2.2 Chord and DHTs

Distributed Hash Tables (DHTs [7], [8], [9], [11]) have been adopted to create peer-to-peer data networks. In a DHT each node has a unique identifier (nodeID) selected from a very large address space. Each message can be associated with a key which is a unique identifier of the same type as nodeID. The main functionality of a DHT is the following: giving a (message, key) pair the system locates (routes the message) to the node whose nodeID is numerically closest to that key. DHTs ensure that routing requires $O(\log(N))$ hops to locate/store a message (where N is the maximum number of nodes in the network).

Chord [9] is a fairly simple structured peer-to-peer network based on a DHT. Compared to unstructured peer-to-peer networks like Gnutella [12] and MojoNation [13] where neighbors of peers are defined in rather ad hoc ways, Chord is structured because of the way peers define their neighbors. Chord provides an exact mapping between node identifiers (nodeID) and keys associated with messages using consistent hashing **Error! Reference source not found.**. NodeIDs and keys are mapped to a large circular identifier space, e.g. $0\ldots2^{160}$ for 160-bit IDs. Values in this space can be viewed as positions in the ring defining the name/identifier space. Thus, given a key, Chord maps it to the (ring position) node whose nodeID is equal to the key. If this node does not exist, the key is mapped to the first successor of this node on the ring. This node is called the *successor* of the key.

Chord efficiently determines the successor of an identifier (key) in $\frac{1}{2}\log(N)$ hops on average (and in $O(\log(N))$ hops in the worst case). In the steady state each node maintains routing information of up to $O(\log(N))$ other nodes. Adding or removing a node from the network can be achieved at a cost of $O(\log^2(N))$ messages. Chord has become very popular and has been used as a building block for several large-scale distributed systems.

2.3 Contribution: Publish/Subscribe systems over DHTs

We choose to use Chord because of its simplicity and popularity within the peer-to-peer community. We leverage the advantages of Chord to build a content-based publish-subscribe system that is scalable, self-organizing, and well performing.

Furthermore, although DHTs provide very good support for exact-match equality predicates (i.e. find the node storing the item with id=itemID) they do not provide good support for range predicates (which are typically very popular when specifying subscriptions in pub/sub systems). We will show how to build Chord-based pub/sub systems which can support range predicates. We will first provide a startup solution and will then extend the Chord substrate to further improve the performance of matching events against subscriptions with range predicates. As far as we know, this is the first work that: (i) leverages DHT research to build large scale content-based pub/sub systems and (ii) while supporting subscriptions with range predicates efficiently.

Event 1					
Type	*Min* $v_{min}(\cdot)$	*Max* $v_{max}(\cdot)$	*Precision* $v_{pr}(\cdot)$	*Name*	*Value* $v(\cdot)$
string	-	-	-	Exchange	=NYSE
string	-	-	-	Symbol	=OTE
float	0.0	20.0	0.01	Price	=8.40
float	0.0	20.0	0.01	High	=8.80
float	0.0	20.0	0.01	Low	=8.22

Figure 1. An event example.

Subscription 1	
Name	*Value*
Exchange	=NYSE
Symbol	=OTE
Price	<8.70
Price	>8.30

Subscription 1	
Name	*Value*
Symbol	=OTE
Price	=8.20
Low	<8.05

Figure 2. Example with two subscriptions.

3. Publish/Subscribe over Chord

The Event/Subscription Schema.
The event schema of this model (Figure 1) is a set of typed attributes. Each attribute a_i consists of a type, a name and a value $v(a_i)$. The type of an attribute belongs to a predefined set of primitive data types commonly found in most programming languages. The attribute's name is a simple string, while the value can be in any range defined by the minimum and maximum ($v_{min}(a_i)$, $v_{max}(a_i)$) values along with the attribute's precision $v_{pr}(a_i)$.

The subscription schema is more general (Figure 2), allowing to express a rich set of subscriptions which contain all interesting subscription-attribute data types (such as integers, strings, etc) and all common operators (=, ≠,<, >, etc.). An event matches a subscription if and only if all the subscription's attribute predicates/constraints are satisfied. A subscription can have two or more constraints for the same attribute which can be thought as if we had two or more different subscriptions with unique constraints over their attributes. Finally, an event can have more attributes than those mentioned in the subscription attributes.

The Subscription Identifier.
A subscription id is the concatenation of three parts:

1. c1: The id of the node receiving the subscription (i.e., where the subscription "belongs"). The size of this field is m bits in a Chord ring with an m-bit identifier address space.
2. c2: The id of the subscription itself. The size of this field in bits is equal to the rounded-up base-2 logarithm of the maximum number of outstanding subscriptions a node can have (e.g. if each node needs to manage 1,000,000 of subscriptions, c2 will be 20-bits long).
3. c3: The number of attributes on which constraints are declared. The maximum value of this field is equal to the total number of attributes supported by the system.

1	0	0	0	1	1	1	0	1
← c_1=2 →			← c_2=2 →			← c_3=5 →		

Figure 3. An example subscription id (subID).

Assume an example Chord ring with a 3-bit identifier address space. Each node can support 8 outstanding subscriptions with an attribute schema including 7 attributes. The subscription id depicted in Figure 3 identifies subscription 3 (c_2=3), belonging to node 4 (c_1=4), comprised of constraints on 5 attributes (c_3=5).

We should note that for every subscription there is a node in the network storing metadata information for it. That node is identified by the c_1 field of the subscription id and it keeps metadata information about the subscription (for example the IP address of the user that generated the subscription etc.).

3.1 Processing subscriptions

Consider, for simplicity, that there is an example pub/sub system supporting only one attribute (a_1). In general, subscriptions specify a single value or a range of values for the attribute a_1. The main idea behind our approach is to store the subscription ids into those nodes of the Chord ring that were selected by appropriately *hashing the values of the attributes* in the subscriptions. The matching of an incoming event can be performed simply by asking those nodes for stored subscription ids.

3.1.1 Storing subscriptions. Storing subscription is done using the hash function provided by Chord (later we will change this to improve performance). Consider that this hash function h() (e.g., SHA-1) returns an identifier uniformly distributed in the address space used for node identifiers. Thus, the result of this hash function h() for the value v(a_i) of the attribute a_i is k (k=h(v(a_i))).

In the case where a subscription contains an equality operator on the single attribute of the example schema, we place the subscription id at the node whose id is the least id which is equal or greater to k (that is *successor*(k) from the Chord API). Therefore, the subID will be placed at the following node: *successor*(*h*(*v*(a_i))).

Things are slightly more complicated with ranges of values. In this case, we map the range on the Chord network storing the subID to all the mapped nodes. Suppose that there is a subscription declaring a range of values over the attribute, a_i which is defined to be between $v_{low}(a_i)$ and $v_{high}(a_i)$. Since all values between $v_{low}(a_i)$ and $v_{high}(a_i)$ are finite, e.g. n (remember that the attribute a_i was declared with a specific precision $v_{pr}(a_i)$), we follow n steps and at each step we store at a Chord node, which is chosen by hashing the previous value incremented by the precision *step*, the subID of the given subscription (the algorithm can be seen in Figure 4).

subID : subscription id
a_i : attribute i , ***L_{ai}*** : List of subIDs for attribute a_i
$v_{pr}(a_i)$: precission of attribute a_i
$v(a_i)$: value of attribute a_i, ***h()*** : Chord hash function

```
For every attribute $a_i$ in subscription
    if $a_i$ has equality constraint
        store subID in node = successor(h(v($a_i$)))
        in the $L_{ai}$ list.
    else if $a_i$ has a constraint in [$v_{low}(a_i)$, $v_{high}(a_i)$]
        $v = v_{low}(a_i)$
        while $v \le v_{high}(a_i)$
            store subID in node = successor(h(v))
            in the $L_{ai}$ list.
            $v = v + v_{pr}(a_i)$
```

Figure 4. The procedure of storing subscriptions

If our schema consists of many attributes, we follow the above procedure for each attribute in every subscription. The only difference is that we keep a different list of subscription ids at each Chord node for every attribute in our schema. For example, consider the case of subscription 1 of Figure 2. The attributes are being processed one at a time starting with *Exchange*. The subscription id of subscription 1, say subID$_1$ will be placed at *successor*(*h*("NYSE")) node in the list dedicated for attribute *Exchange* and at *successor*(*h*("OTE")) node in the list dedicated for attribute *Symbol*. As you can see, there is a range constraint over the *Price* attribute, 8.30<*Price*<8.70. Since the precision of the attribute *Price* is defined to be 0.01, the subID$_1$ will be placed in 39 Chord nodes defined by *successor*(*h*(v_j(*Price*))) for the following sequence of values 8.31, 8.32, …, v_j(*Price*), …, 8.68, 8.69.

3.1.2 Updating subscriptions. Updating a subscription involves a procedure during which the values of all attributes contained in the subscription are updated using the standard API of the Chord system. In the case of equality only two nodes are affected. First, the node that is mapped to the old, stale, value is forced to delete the subID for the attribute that belongs to the subscription with identifier subID. Second, a new node is going to store the subID, depending on the id returned from the Chord's hash function passing the new updated value. In other words, we delete the subID from the node with nodeID=successor(h($v_{stale\_value}(a_i)$)) and then we add it to the node with nodeID = successor(h($v_{updated\_value}(a_i)$)).

***subID**$_i$* : subscription id of subscription **i**
***N**$_{a\text{-}event\text{-}i}$* : number of attributes defined in the event **i**
***N**$_{a\text{-}sub\text{-}i}$* : number of attributes defined in a subscription **i**
***N**$_{list\text{-}sub\text{-}i}$* : number of collected lists that subID$_i$ is stored
***a**$_i$* : attribute **i**
***L**$_{ai}$* : List of subIDs for attribute a_i
v(a**$_i$**) : value of attribute a_i
h() : Chord hash function
Delivery List: the list of subIDs that the event is going to be delivered.

```
for every event_j arriving at the system
                    subIDs lists collection phase
    for every attribute a_i in event_j with value v(a_i)
        go to node successor( h( v(a_i) ) )
        retrieve the list L_ai of subIDs found there
                    Matching phase
    for every list L_ai of subIDs
        for every subID_k
            retrieve the number of attributes defined
            from the c3 field: N_a-sub-k
            count the number of collected lists that subID_k
            is stored: N_list-sub-k
            if N_a-sub-k equals N_list-sub-k
                we have a match
                remove subID_k from all lists
                store subID_k in the Delivery List
                    Delivery phase
    for every subID_i in the Delivery List
        contact the node that keeps the subscription. Its
        nodeID is the c1 field of the subID_i
        retrieve the IP of the client that generated the
        subscription with id : subID_i
        deliver the event to the client
```

Figure 5. The matching algorithm.

As we said before, ranges are spread all over the Chord ring. Thus, updating a range (in other words updating the $v_{low}(a_i)$ and $v_{high}(a_i)$ values) results in following the above procedure for all Chord nodes that store the subID for the given range of values. The procedure we follow depends on the new values of the range bounds ($v_{low\_NEW}(a_i)$ and $v_{high\_NEW}(a_i)$) compared to the old ones. If $v_{low\_NEW}(a_i) < v_{low}(a_i)$ we store the subID to a number of nodes that are going to cover the [$v_{low\_NEW}(a_i)$, $v_{low}(a_i)$) range. The same procedure holds when $v_{higb\_NEW}(a_i) > v_{high}(a_i)$ resulting in covering the range ($v_{high}(a_i)$, $v_{higb\_NEW}(a_i)$]. In the case where $v_{low\_NEW}(a_i) > v_{low}(a_i)$ or $v_{higb\_NEW}(a_i) < v_{high}(a_i)$ we delete the subID form the nodes covering the range [$v_{low}(a_i)$, $v_{low\_NEW}(a_i)$)and ($v_{high\_NEW}(a_i)$, $v_{higb}(a_i)$] respectively.

Deleting subscriptions is done as explained above since the updating includes the deleting procedure.

3.2 Processing events: The matching algorithm

The distributed matching algorithm should be able to cope with expected very high loads, determined by high event arrival rates.

Suppose that an event arrives at the system with N$_{a\text{-}event}$ attributes defined. The matching algorithm starts by processing each attribute separately. It first tries to find the node which stores subIDs for the value v(a_i) of the attribute a_i. This node is the n=*successor*(*h*(v(a_i))). The algorithm, then, stores the list of unique subIDs found to be stored in node n in the list L$_{ai}$ designated for a_i. After processing all attributes, N$_{a\text{-}event}$ lists of subIDs will be stored. Suppose, now, that a subID$_k$ presented in at least one of those lists consists of N$_{k\text{-}sub}$ attributes (N$_{k\text{-}sub}$ can be easily derived from the field c$_3$ of the subID defined in section 3). Then, the subID$_k$ matches the event if it appears in exactly N$_{k\text{-}sub}$ lists collected from the Chord ring. The matching algorithm can be seen in Figure 5.

Example: Matching events with subscriptions.

Suppose that we have the subscriptions of Figure 2 generated by two clients connected to a Chord node and the event of Figure 1. First, the algorithm will collect all the subIDs lists in which the values of the event attributes, satisfy the corresponding constraints of the subscriptions.

For this to be done, the algorithm starts with attribute *Exchange* and retrieves the subID list (L$_{Exchange}$) from node *successor*(*h*("NYSE")). This list contains only the subID$_1$. Hence, we have L$_{Exchange}$→**subID$_1$**. For the attribute *Symbol* the corresponding list is L$_{Symbol}$→**subID$_1$, subID$_2$**, since both subscriptions are satisfied for the specific event. For the attribute *Price* only subscription 1 is satisfied and, thus, the list is L$_{Price}$→**subID$_1$**. Finally, for the attribute *Low* only subscription 2 is satisfied and the list is L$_{Low}$→**subID$_2$**.

After this phase of the matching process the collected subscription ID lists are:

L$_{Exchange}$……	→**subID$_1$**
L$_{Symbol}$………	→**subID$_1$, subID$_2$**
L$_{Price}$……….	→**subID$_1$**
L$_{Low}$………..	→**subID$_2$**

Subscription 1 was found in three lists while subscription 2 was found in two lists. By processing appropriately the subIDs of subscriptions 1 and 2 (the c_3 part) we can find out that both subscriptions have constraints over three attributes. Since subscription 1 was found in three lists, a match is implied and so we keep the $subID_1$ in order to inform the node which generated the subscription about the matched event. This is done by consulting the node storing the subscription (with nodeID equal to the c_1 field of the $subID_1$) and holding metadata information for $subID_1$, in order to locate the IP address of the client that generated the subscription. Then, the event is delivered to the interested client. Subscription 2 on the other hand is dropped since the number of lists that the $subID_2$ was found in is 2 while the number of attributes defined in it is 3.

3.3 Expected performance

In a Chord network with N nodes and 2^m-bit address space the average number of nodes that must be contacted to find a *successor* is ½log(*N*) hops.

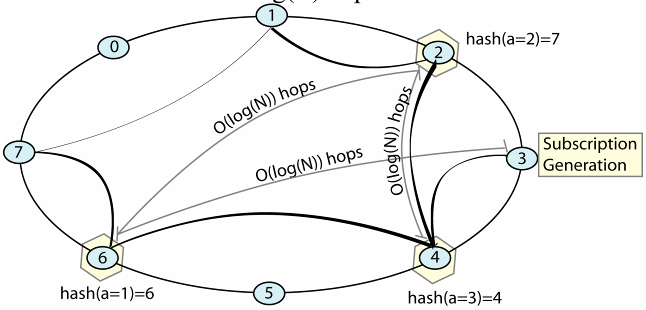


Figure 6. Storing range values with Chord.

During the subscription storage procedure, the average number of hops needed to store a subID depends on the type of constraints over the attributes. In equality constraints, the average number is ½log(*N*), since the subID is stored in a single node, i.e. the *successor*(*h*(v(a_i))). When the constraint is a range of values (e.g. [$v_{low}(a_i)$, $v_{high}(a_i)$]) over the attribute a_i with precision $v_{pr}(a_i)$ (in Figure 1 the $v_{pr}(Price)$ of attribute *Price* is 0.01) then $r = \frac{v_{high}(a_i) - v_{low}(a_i)}{v_{pr}(a_i)}$ nodes are affected leading to r·½log(*N*) hops on average in order to store the subID.

The update/deletion of subscription again depends on the type of constraints over the attributes. For an equality constraint, an update can be performed by contacting 2·½log(*N*)=log(N) nodes. For ranges the number of nodes is k·log(*N*) on average, where k depends on whether the new range is smaller or wider compared to the previous one.

The event-processing (matching) process involves contacting $N_{a\text{-}event}$·½log(*N*) nodes to collect the subscription ids lists. Thus, we see that, by design, our proposal leads to fast and scaleable event matching.

4. Improving performance

When trying to store a subscription over the Chord ring with attributes defined by a range of values, we need perform r·½·log(*N*) hops on average for every attribute (note that r depends on the precision of the value as well as the $v_{low}(a_i)$, and $v_{high}(a_i)$, values of the range interval). In this section we extend the Chord's functionality so that range attributes will require r+½·log(*N*) hops.

4.1 OPChord : Order Preserving Chord

We use a 2^m-order preserving hash function (OPHF) in order to store the sequential values of a range interval in sequential nodes over the Chord ring.

Expected performance.
We need to perform ½·log(*N*) hops on average to locate the node which will store the minimum value of the range (that is $v_{low}(a_i)$ for the attribute a_i). Then, we have to perform r hops to store the remaining values in the range. This approach leads to r+½·log(*N*) hops in total.

The Order Preserving Hash Function.
Suppose, now, that every attribute a_i is characterized by $v_{min}(a_i)$: the minimum value that a_i can take, $v_{max}(a_i)$: the maximum value that a_i can take, and $v_{pr}(a_i)$: the precision of a_i . If $v_j(a_i)$ is defined to be any value in the interval [$v_{low}(a_i)$,$v_{high}(a_i)$], the OPHF is:

$$h(v_j(a_i)) = \left(s_o(a_i) + \frac{v_j(a_i) - v_{\min}(a_i)}{v_{\max}(a_i) - v_{\min}(a_i)} \cdot 2^m \right) \bmod 2^m$$

The $s_o(a_i)$ is defined to be:

$$s_o(a_i) = hash(attribute\_name(a_i))$$

and is used to randomize the node on the Chord ring where the minimum values of different attributes will be stored, leveraging thus different subsets of the Chord network. *Hash*() is the base hash function used by Chord (e.g. SHA-1). Note that there is a different OPHF for every attribute.

4.2 Subscription and event processing with OPHF

The algorithms are generally the same as the ones presented earlier. The only main difference is the use of OPHF instead of the base hash function of Chord.

Example: Storing subscription.
Consider a Chord ring with 3-bit identifiers and 8 nodes and a subscription of a single integer attribute *a* arriving at node 3 with constraint: 0< v(*a*) <4. Using Chord (Figure 6) would require *O*(r·log(*N*)) hops to store the subID at three

nodes (in this example r equals 3, as there are 3 distinct values in the interval (0,4)), Hashing the first value (a=1) returns node 6 requiring to access $O(\log(N))$ nodes to reach node 6 (½·log(N) on average). Repeating the previous step, the other nodes that will store the subID are 2 and 4, requiring overall $O(r\cdot\log(N))$ hops at most (in our example, 6 hops).

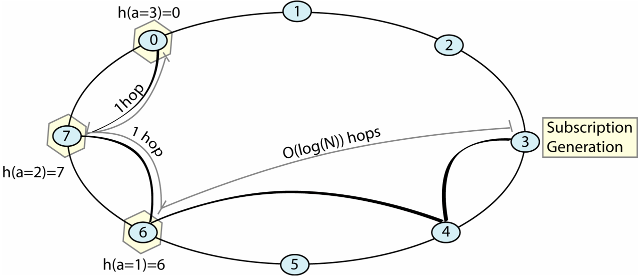


Figure 7. Storing range values with OPHF/Chord.

Suppose, now, that we use OPHF/Chord (Figure 7). We need to perform $O(\log(N))$ hops only once at the very first time when trying to reach the first node (node 6). Then, storing the subID at nodes 7 and 0 requires two more hops.

4.3 Discussion

Load balancing in the Chord system is based on the randomness guarantees of the consistent hashing function. We have investigated load balancing within the OPHF architecture, but it is beyond the scope of this paper.

We should also briefly note the "small domain problem": when the number of nodes in the network is much greater than the domain of attribute values, this could lead to have k "useless" nodes between two consecutive (ring positions) values in the range. In this case we need to pay an overhead of extra hops in order to store subIDs for a range of values, in the OPHF design. We have developed solutions that alleviate the extra-hop problems; however, they are beyond the scope of this paper.

5. Concluding remarks

In this work we have shown how to leverage a popular DHT, Chord, towards building scalable, self-organizing, well-performing, content-based pub/sub systems. We have shown how to support subscriptions that involve equality and range predicates and the associated performance benefits. Our proposal favors fast and scaleable event processing. This is achieved by essentially turning the task of event processing into a DHT lookup operation. To our knowledge this is the first work that meets these goals.

6. References

[1] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron., *Scribe: A large-scale and decentralized application-level multicast infrastructure.* Journal on Selected Areas in Communication, vol. 20, Oct. 2002.

[2] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. *Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination.* 11th ACM NOSSDAV, pp. 11-20, '01.

[3] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. *Application-level multicast using content-addressable networks*. In Proc. 3rd International Workshop of NGC, vol. 2233, pages 14–29. LNCS, Springer, 2001.

[4] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. *An efficient multicast protocol for content-based publish-subscribe systems*. In Proceedings of the 19th ICDCS, pp. 262–272, 1999.

[5] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. *Design and evaluation of a wide-area event notification service*. ACM Transactions on Computer Systems, 19(3):332–383, 2001.

[6] D. Tam, R. Azimi, H. Jacobsen, *Building Content-Based Publish/Subscribe Systems with Distributed Hash Tables*, Int. Workshop on Databases, Information Systems and Peer-to-Peer Computing, September 2003.

[7] A. Rowstron and P. Druschel. *Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems*. In Proc. 18th IFIP/ACM Int. Conference on Distributed Systems Platforms (Middleware 2001), pages 329-350, November 2001.

[8] S. Ratnasamy,P. Francis, M. Handley, R. Karp, and S. Shenker. *A scalable content addressable network*. In proceedings of ACM SIGCOMM 2001, 2001.

[9] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. *Chord: A scalable peer-to-peer lookup service for internet applications*. In proceedings of ACM SIGCOMM 2001, pages 149-160, 2001.

[10] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. *Achieving scalability and expressiveness in an Internet-scale event notification service*. Proc. ACM PODC, pp 219–227, 2000.

[11] Zhao, Y. B., Kubiatowitcz, J., Joseph, A.: *Tapestry: An infrastructure for fault-tolerant wide-area location and routing*. Tech. Rep. UCB/CSD-01-1141, Univ. of California at Berkley, Computer Science Dept. (2001)

[12] Gnutella: http://gnutella.wego.com

[13] Wilcox, B., Hearn, O.: *Experiences Deploying a Large-Scale Emergent Network*. In 1st International Workshop on Peer-to-Peer Systems, IPTPS'02 (2002)

[14] P. R. Pietzuch and J. Bacon, *Peer-to-Peer Overlay Broker Networks in an Event-Based Middleware*, in proceedings of the DEBS'03 conference, 2003.

[15] P. Triantafillou, A. Economides, *Subscription Summarization: A New Paradigm for Efficient Publish/Subscribe Systems*, In IEEE, ICDCS 2004.

[16] W. W. Terpstra, S. Behnel, L. Fiege, A. Zeidler, and A. P. Buchmann, *A Peer-to-Peer Approach to Content-Based Publish/Subscribe,* in DEBS 2003.

A Transformational Framework for Testing and Model Checking Implicit Invocation Systems[1]

Hongyu Zhang, Jeremy S. Bradbury, James R. Cordy, Juergen Dingel
School of Computing, Queen's University
Kingston, Ontario, Canada
{hellen, bradbury, cordy, dingel}@cs.queensu.ca

Abstract

With the growing size and complexity of software systems, software verification and validation (V&V) is becoming increasingly important. Model checking and testing are two of the main V&V methods. In this paper, we present a framework that allows for testing and formal modeling and analysis to be combined. More precisely, we describe a framework for model checking and testing implicit invocation software. The framework includes a new programming language – the Implicit Invocation Language (IIL), and a set of formal rule-based transformation tools that allow automatic generation of executable and formal verification artifacts. We evaluate the framework on several small examples. We hope that our approach will advance the state-of-the-art in V&V for event-based systems. Moreover, we plan on using it to explore the relationship between testing and model checking implicit invocation systems and gain insight into their combined benefits.

1. Introduction

With the growing size and complexity of software systems, software verification and validation (V&V) is becoming more and more important. Testing and model checking belong to the two categories of software V&V: testing/inspection and formal methods. While testing focuses on the actual behavior of the program, model checking focuses on the mathematical model. Testing and model checking are complementary: testing is lightweight but incomplete while model checking is more heavyweight but complete. A major problem with testing and model checking is that they usually require different software artifacts. In fact, there is often a large semantic gap between the software developer artifacts that are tested and the artifacts that are accepted by model checkers. The gap between artifacts typically has to be bridged by humans with little tool support. Thus, there is a possibility for spurious results when the finite-state model does not correspond exactly to the software system.

To alleviate this problem, we have developed a transformational framework for the testing and model checking of implicit invocation (II) or publish-subscribe systems. II systems are event-based and have two primitives. First, components can announce or publish events. Second, other components can listen or subscribe to events that are announced. A centralized message server or event dispatcher receives announced events and uses them to invoke the appropriate subscriber methods. We have chosen to focus on II systems for several reasons. In the context of testing, II systems feature a lot of non-determinism due to concurrent execution of components and the event dispatcher. In the context of model checking, this non-determinism often causes the model to be excessively large. Additionally, II has become increasingly popular as an event-based architecture.

Our framework includes a new programming language – the Implicit Invocation Language (IIL). IIL is a special-purpose language that is designed specifically for software systems that use the II architectural style. The primary advantage of IIL for programming II systems is that it leverages our knowledge about II and provides a notation with a level of abstraction that is convenient to read and write.

Rather than write a compiler, we have chosen to implement IIL for simulation and testing by source transformation to an existing executable language. We have chosen to translate II into Turing Plus, a concurrent programming language [11].

Model checking systems written in IIL involves the use of an existing II model checking system originally developed by Garlan and Khersonsky [6, 7] that we have extended in [2]. This system involves representing an II system in an XML intermediate representation that is translated into a finite state machine which is analyzed by a standard model checker. The model checker allows for the analysis

[1] This work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Ontario Graduate Scholarship Program (OGS).

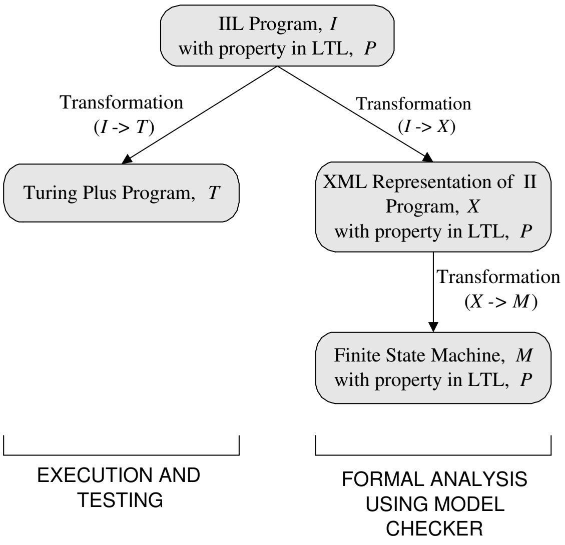

IIL Program, *I*
with property in LTL, *P*
Transformation
(*I* -> *T*)
Transformation
(*I* -> *X*)
Turing Plus Program, *T*
XML Representation of II Program, *X*
with property in LTL, *P*
Transformation
(*X* -> *M*)
Finite State Machine, *M*
with property in LTL, *P*
EXECUTION AND TESTING
FORMAL ANALYSIS USING MODEL CHECKER

Figure 1. Our transformational framework

of Linear Temporal Logic (LTL) properties. We integrate our previous model checking approach with IIL by providing an automated transformation from IIL into the XML intermediate representation.

Figure 1 shows the overall structure of our transformational framework. All of our transformation tools are implemented using TXL, a programming language and rapid prototyping system specifically designed to support rule-based source to source transformation [5]. Each tool is fully automated and is based on formal rewriting rules expressed in terms of the syntax of the source language and the target language. The framework allows for a powerful combination of two complementary V&V techniques. We hope that our automated approach will allow us to explore the relationship between testing and model checking and gain insight into the possible benefits of their combined use.

We will next provide an overview of the II architectural style in Section 2 before introducing IIL in Section 3. In Section 4 we will discuss the execution and testing of II systems using Turing Plus and in Section 5 we will discuss model checking II systems. Section 6 describes our evaluation of the framework and Section 7 discusses how our framework relates to existing research work. Finally, we outline our conclusions as well as possible future work in Section 8.

2. II systems

An II system is characterized by six parameters: components, events, event-method bindings, an event delivery policy, a shared state, and a concurrency model.

Events are the primary method of communication between components. The components in the system can announce events. Upon receiving events from the components, the event dispatcher sends the events out to all subscriber components that have requested to receive that particular type of event (Figure 2). The correspondence between events that are announced and the methods in a component instance that are invoked in response to these announcements is defined in the event-method bindings. Event-method bindings instruct the dispatcher where to send events. The event delivery policy, a set of conditional delivery rules, instructs the dispatcher when and how to send them.

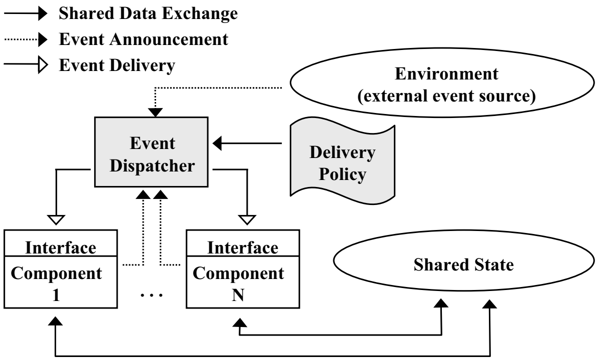


Figure 2. II system structure [6]

3. An II language (IIL)

IIL is a programming language designed especially for expressing II systems. IIL includes the following special features: component declarations, event declarations, announcement statements, dispatcher declaration, delivery statements, event-method bindings, and property declarations. We will now discuss these features in the context of a Set-Counter example [17]. The Set-Counter example in IIL is presented in Figure 3. Due to space limitations, this example has been elided and is provided primarily to show the main features and overall structure of an IIL program.

The Set-Counter system involves two examples of component declaration: a `Set` and a `Counter`. The `Set` component contains a set of objects and the `Counter` component keeps a count of the objects in the set. Figure 3 shows the IIL representation of the `Set` component. All components in IIL can have variables which describe component properties and methods which describe actions.

The Set-Counter example contains four event declarations. Two of the declared events (`EnvAdd`, `EnvRemove`) are external events (also called environment events). The declarations contain an event name and event announcement properties. Environment events represent external be-

```
system SetAndCounter {
   external event EnvAdd {1..N}, EnvRemove {1..N};
   event Insert(int {1..2} numElements);
   event Delete(int {1..2} numElements);

   dispatcher delivers Insert, Delete {
      if (Insert.count > Delete.count) {
         deliver Immediate Insert;
         deliver Random Delete;
      }
      else {
         deliver Random Insert;
         deliver Immediate Delete;
      }
   }

   int {0..3} setSize;

   SetAndCounter() {
      Set s = new Set();
      Counter c = new Counter();

      bind EnvAdd to s.Add();
      bind EnvRemove to s.Remove();
      bind Insert to c.CountIns(Insert.numElements);
      bind Delete to c.CountDel(Delete.numElements);

      property AlwaysCatchesUp =
          (G F (setSize = c.counter));
      property ...
    }

   component Set
      announces Insert, Delete
      accepts EnvAdd, EnvRemove {
      int {0..2} value;

      Add() {
         value = {1,2}; //nondeterministic choice
         if ((setSize + value) < 4) {
            setSize = setSize + value;
            announce Insert(value);
         }
      }

      Remove() {
         ...
      }
   }

   component Counter
      accepts Insert, Delete {
      int {0..3} counter = 0;

      CountIns(int {1..2} number) {
         counter = counter + number;
      }

      CountDel(int {1..2} number) {
         ...
      }
   }
}
```

Figure 3. The Set-Counter example in IIL

havior that can affect the II system. The other two declared event are local events (`Insert`, `Delete`) which are declared with an event name and optional event data.

Components in an IIL program can contain announcement statements which define the announcement of locally declared events. For example, in the component `Set` the `Insert` event is announced in the `Add` method.

In addition to components and events, a dispatcher is declared. The dispatcher is responsible for event delivery and defines the delivery policy. Environment events will be delivered immediately by the dispatcher while local events will be delivered according to the delivery policy, which is composed of delivery statements. In our Set-Counter example we have included a delivery policy which states that if there are currently more `Insert` events waiting to be delivered than `Delete` events, then an `Insert` event is delivered immediately and a `Delete` event is delivered randomly (i.e., delivered sometime in the future). Otherwise the opposite occurs.

Event-method bindings are needed to register the methods to the events for event delivery. For example, in the Set-Counter example we see that the `EnvAdd` event is bound to the `Add` method in the Set component `s`. That is, when an `EnvAdd` event is announced the `Add` method in `s` will be invoked.

In addition to the special language features used to construct an II system, IIL also allows for LTL property declarations. The properties that are declared can be verified using our model checking process. For example the property `AlwaysCatchesUp` in the Set-Counter example states that the global variable `setSize` will always eventually be equivalent to the `counter` variable in the Counter component `c`.

4. Translating IIL programs into executable Turing Plus programs

As previously mentioned, IIL has no compiler and requires transformation to an executable language for testing and simulation. We have chosen to transform IIL into Turing Plus, an extension of the programming language Turing [10]. We decided to use Turing Plus for execution of II systems because Turing Plus, as a concurrent programming language [11], has a simple and general concurrency model.

Before implementing our automated transformation from IIL to Turing Plus we first had to develop a model of II in Turing Plus that captured the semantics of an IIL program. The two main design issues in developing this model were: implicit method invocation and the concurrency model.

4.1. Implicit method invocation

Turing Plus does not support II directly. The first problem we need to solve is to find a mechanism to carry out II in Turing Plus.

According to Garlan and Scott, “implicit invocation supplements, rather than supplants, explicit invocation” [8]. In our Turing Plus model, we take this approach and use three steps of explicit invocation to implement II (see Figure 4). That is, an explicit method call is used in event announcement, event delivery and bound method invocation. The three main elements of our implementation of II in Turing Plus are:

1. A system event warehouse, a set of queues, is built to receive all the announced events. When components announce an event or an environment event is generated, it will be sent to the system event warehouse.
2. The dispatcher removes the events in the system event warehouse and delivers them. Environment events will be delivered immediately. Local events will be delivered according to the delivery policy. The dispatcher delivers the events in the system event warehouse by calling the bound component to receive the event.
3. Each component has a component event warehouse to receive the events delivered by the dispatcher. The component will invoke the bound method after it receives the delivered events.

Our modeling of II in Turing Plus thus divides event-method bindings into two parts: the event-component binding information contained in the dispatcher and the event-method binding information contained in the components.

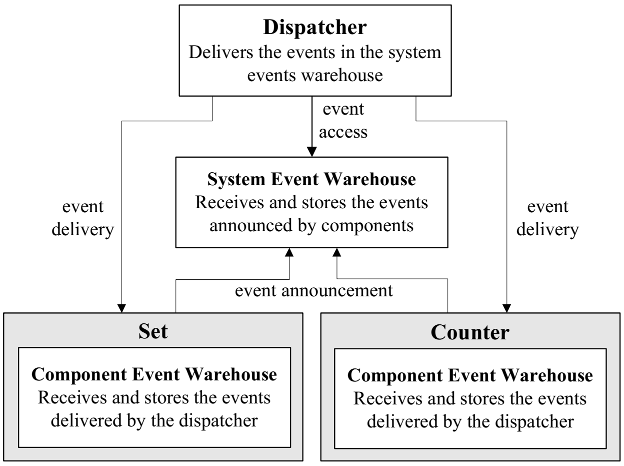

Dispatcher
Delivers the events in the system events warehouse
event access
System Event Warehouse
Receives and stores the events announced by components
event delivery
event delivery
event announcement
Set
Component Event Warehouse
Receives and stores the events delivered by the dispatcher
Counter
Component Event Warehouse
Receives and stores the events delivered by the dispatcher

Figure 4. Implicit method invocation for the Set-Counter example in Turing Plus

4.2. The concurrency model

The concurrency model determines how to assign and manage threads in the system. In [7], Garlan and Khersonsky propose several models of concurrency including a single thread of control for all components and separate threads of control for components.

In our implementation, we fix the concurrency model to use a single thread for each component. Each component, the event dispatcher, and the system have a thread defined by a “run process”. For example, in Figure 5 we see that the Set-Counter example has 4 threads. To ensure that the execution semantics of an IIL program in Turing Plus matches its model checking semantics in SMV, all of the threads in an Turing Plus implementation of an II system are synchronized by the `Rendezvous` monitor.

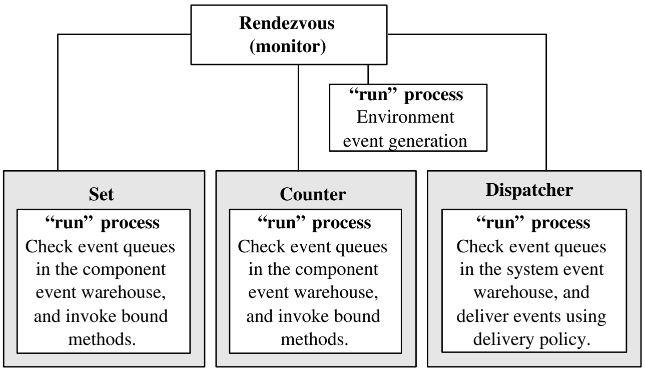

Rendezvous
(monitor)
“run” process
Environment event generation
Set
“run” process
Check event queues in the component event warehouse, and invoke bound methods.
Counter
“run” process
Check event queues in the component event warehouse, and invoke bound methods.
Dispatcher
“run” process
Check event queues in the system event warehouse, and deliver events using delivery policy.

Figure 5. The II concurrency model for the Set-Counter example in Turing Plus

4.3. Transformation of IIL to Turing Plus

The structure and syntax of Turing Plus programs is very different from IIL programs. Figure 6 provides a summary of how information in an IIL program is used in generating each part of a Turing Plus program.

Our automated tool for transforming IIL to Turing Plus consists of a set of formal transformation rules written in TXL, a popular source-code transformation language that has been used in numerous industrial and academic projects over the past 10 years. Unfortunately, due to space restrictions we cannot include these rules in this paper. For examples of TXL rules see [18].

5. Translating IIL programs into SMV models

To model check systems written in IIL we use the existing approach we previously presented in [2]. This approach is an extension of an II model checking system originally developed by Garlan and Khersonsky in [6, 7]. This approach focuses on the automatic analysis of II by representing an II system in an XML parameterized representation.

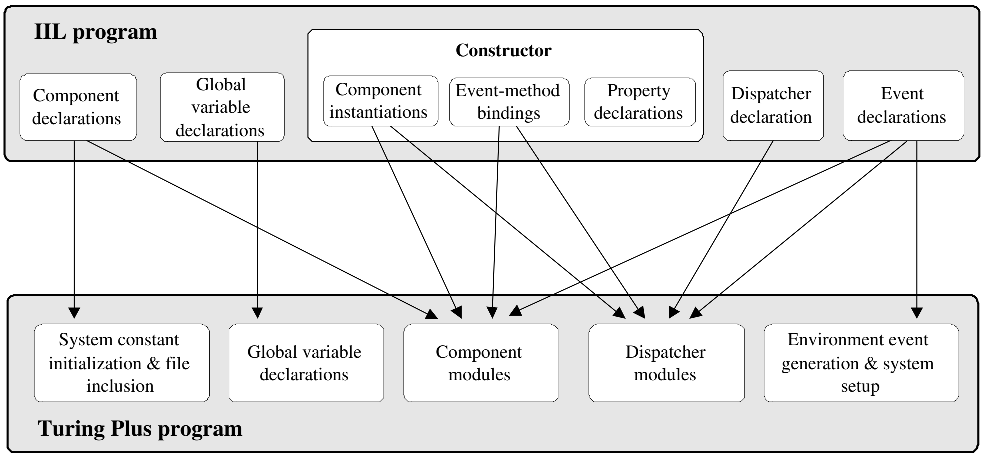


Figure 6. Information integration in IIL to Turing Plus transformation

Once the XML input has been created, a Java transformation tool converts the information into a set of finite state machines. This set of state machines can then be checked using the Cadence SMV model checker [14]. We currently check properties written in LTL.

A major drawback to our model checking work presented in [2] was that it was not completely automatic since user interaction was required in developing the XML representation. The approach presented in this paper overcomes this deficiency and completely bridges the gap between artifacts by automating the process of generating finite state models for software systems written in IIL.

5.1. Transformation of IIL to XML

Originally, due to the verbose nature of XML, IIL started out as a replacement to the XML intermediate language that would be easier to read and write. In addition to improving the syntax, IIL evolved into a special-purpose language that includes notational conveniences such as variable declarations in methods, the use of for loops and the use of switch statements.

As with the transformation to Turing Plus, our transformation from IIL to XML is done using an automatic transformation tool written in TXL. The transformation from IIL to XML involves two steps. In the first step the notational conveniences of IIL are removed. For example, for loops are unrolled. The program is also reorganized to follow the program order required by the XML representation. After the first step the IIL program is a statement-by-statement match to the XML representation. The second step of the transformation involves the syntax translation from IIL to XML.

6. Evaluation

To evaluate our transformational framework we use three examples: the Set-Counter example, the Active Badge Location System (ABLS), and the Unmanned Vehicle Control System (UVCS) [2]. Our evaluation of each example involved modeling the example in the IIL language and verifying that our transformation tools from IIL to Turing Plus and from IIL to XML worked correctly.

Our evaluation shows that IIL programs are substantially smaller in size than the corresponding Turing Plus implementation used for testing and both the XML and SMV representations used for model checking. Table 1 summarizes the results and illustrates the advantage of using a special-purpose language convincingly. Note that the ABLS and UVCS values in the table are the average of several example systems implemented in IIL. Our evaluation of our automatic transformation tools also demonstrated that the semantics was well preserved during all of the transformations.

Example	IIL (KB)	TP (KB)	TP (% IIL)	XML (KB)	XML (% IIL)	SMV (KB)	SMV (% IIL)
Set-Counter	2	13	650%	9	450%	24	1200%
ABLS	3	13	444%	8	278%	23	767%
UVCS	8	16	200%	21	263%	38	469%
Overall	**5**	**14**	**315%**	**13**	**281%**	**28**	**622%**

Table 1. File size comparison

7. Related work

Rapide [13] and Eventua [15] are two existing special-purpose languages for event-based systems. Rapide is an executable architecture definition language. It is intended for modelling the architectures of concurrent and distributed systems. Eventua is an object-oriented language that includes native support for events by including classes, fields, a self keyword, and parameter passing for both methods and events. An Eventua program can be transformed to the $\varrho\varpi\varsigma$-calculus, the underlying formalism, for execution.

Bandera [4] and the Spin model checker [12] provide automatic translation from a general purpose programming language to a standard model checker. Our approach differs in that we limit ourselves to a special-purpose II language.

In the Cadena project at Kansas State University [9], the model checker Bogor was used to analyze BoldStroke – an event-based real-time middleware architecture developed by Boeing. BoldStroke was modeled using CORBA's Interface Definition Language. The model construction was only partially automatic.

As an alternative to our approach, it would be interesting to explore the use of Java to represent event-based systems (e.g. using the Message-Driven Thread API for Java [1], or publish/subscribe infrastructures like Elvin [16] or Siena [3]) and to use Bandera for automatic model extraction and analysis.

8. Conclusion

We have presented a framework for specifying, testing, and model checking II systems. It consists of a high-level language for specifying II systems and two fully automatic, formally specified translations: one into the Turing Plus language for execution and testing, and one into the input language of a standard model checker [18]. The framework demonstrates how automatic source code transformation can be used to combine the convenience of a special-purpose language with the benefits of two complementary V&V techniques: testing and model checking.

We believe that our work provides a useful test bed for studying the relationship and possible synergies between testing and model checking. In particular, it might allow us to investigate the following questions: To what extend can parallel testing be used to increase confidence in model checking results and in the correctness of the model checker? How can testing be used to simplify or optimize the model checking? Can model checking be used to evaluate the coverage offered by the test suite? Would it be useful to integrate temporal logic properties into the testing effort through, for instance, run-time safety analysis?

In addition to studying the relationship and possible synergies between testing and model checking, another future direction of research is the extension of our framework for use with more general forms of publish-subscribe systems. For example, II systems that support dynamic bindings and additional concurrency models.

References

[1] Message-driven thread API for the Java programming language. Web page: `http://www.mdthread.org`.

[2] J. S. Bradbury and J. Dingel. Evaluating and improving the automatic analysis of implicit invocation system. In *Proc. of ESEC/FSE 2003*, pages 78–87, Sept. 2003.

[3] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. on Computer Systems*, 19(3):332–383, Aug. 2001.

[4] J. Corbett, M. Dwyer, J. Hatcliff, et al. Bandera: Extracting finite-state models from Java source code. In *Proc. of the Int. Conf. on Software Engineering*, pages 439–448, June 2000.

[5] J. Cordy, T. Dean, A. Malton, and K. Schneider. Source transformation in software engineering using the TXL transformation system. *Journal of Information and Software Technology*, 44(13):827–837, 2002.

[6] D. Garlan and S. Khersonsky. Model checking implicit-invocation systems. In *Proc. Int. Work. on Software Specification and Design*, Nov. 2000.

[7] D. Garlan, S. Khersonsky, and J. Kim. Model checking publish-subscribe systems. In *The Int. SPIN Work. on Model Checking of Software*, May 2003.

[8] D. Garlan and C. Scott. Adding implicit invocation to traditional programming languages. In *Proc. of the Int. Conf. on Software Engineering*, pages 447–455, 1993.

[9] J. Hatcliff, X. Deng, M. B. Dwyer, G. Jung, and V. P. Ranganath. Cadena: An integrated development, analysis, and verification environment for component-based systems. In *Proc. of the Int. Conf. on Software Engineering*, pages 160–173, May 2003.

[10] R. Holt and J. Cordy. The Turing Plus report, CSRI, University of Toronto, 1987.

[11] R. Holt and D. Penny. The concurrent programming of operating systems using the Turing Plus language, University of Toronto, 1988.

[12] G. J. Holzmann and M. H. Smith. An automated verification method for distributed systems software based on model extraction. *IEEE Trans. Softw. Eng.*, 28(4):364–377, 2002.

[13] D. C. Luckham and J. Vera. An event-based architecture definition language. *IEEE Trans. Softw. Eng.*, 21(9):717–734, 1995.

[14] K. L. McMillan. *The SMV Language*. Cadence Berkeley Labs, Mar. 1999.

[15] J. S. Patterson. An object-oriented event calculus. Technical Report TR02-08, Computer Science, Iowa State University, 2002.

[16] B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings AUUG97*, Sept. 1997.

[17] K. Sullivan and D. Notkin. Reconciling environment integration and software evolution. In *Proc. of SIGSOFT '90: Symp. on Software Development Environments*, Dec. 1990.

[18] H. Zhang. An implicit-invocation language and its implementation. Master's thesis, Queen's University, 2004.