Continuous Remote Analysis for Improving Distributed Systems' Performance[*]

Antonio Carzaniga
Department of Computer Science
University of Colorado
carzanig@cs.colorado.edu

Alessandro Orso
College of Computing
Georgia Institute of Technology
orso@cc.gatech.edu

ABSTRACT

Engineering a highly distributed system requires the ability to evaluate and optimize the protocols that control the movement and processing of information throughout the system. Because the design of such protocols is often characterized by conflicting goals and trade-offs, the designer must calibrate the parameters of the protocols, seeking the best balance of performance in the most common usage scenarios. Unfortunately, fully testing these calibrations requires experiments conducted on large, expensive testbeds that are very difficult to deploy and maintain.

In this paper, we propose a new approach for the optimization of a highly distributed system's performance. The approach is based on leveraging data collected from fielded components to fine-tune the behavior of the system and its protocols. Captured data is "replayed" in simulations performed directly in the field during off-peak hours. The results of these simulations are then used to control the system directly in the field, and/or to report aggregate performance and behavior information to the system designer.

1. INTRODUCTION

In this paper, we investigate the use of execution data for tuning and optimizing highly distributed systems. We use the term *highly distributed system* to refer to a system capable of delivering a service to many clients through a large number of distributed access points. Examples of such systems are various forms of general-purpose communication systems, application-level overlays, networks of caching servers, peer-to-peer file-sharing systems, distributed databases, replicated file systems, and distributed middleware systems in general.

Engineering a highly distributed system is challenging in a number of ways. In particular, one activity that poses serious practical obstacles is the evaluation and optimization of performance. The performance of a highly distributed system is determined to a large extent by the protocols that control the movement and processing of information throughout the system, and the design of such protocols is often characterized by conflicting goals and trade-offs. In optimizing a highly distributed system, the designer must therefore calibrate the parameters of the protocols, seeking the best balance of performance in the most common usage scenarios. Unfortunately, fully testing these calibrations requires experiments conducted on large, expensive testbeds that are very difficult to deploy and maintain.

[*]This work was supported in part by National Science Foundation awards CCR-9988294, CCR-0096321, CCR-0205422, SBE-0123532 and EIA-0196145 to Georgia Tech, and by the State of Georgia to Georgia Tech under the Yamacraw Mission.

In current practice, because of these practical obstacles, protocol design decisions are tested using simulations, on the basis of assumptions about the behavior of the system, its clients, and its operating environment. While simulations greatly reduce the costs of running experiments, they also introduce unavoidable inaccuracies. Specifically, simulation suffers from two classes of problems:

- by its nature, it tests an abstraction of the system that may not realize a sufficiently accurate representation of the behavior of the actual system, and that may therefore hide problems or optimizations opportunities;
- it relies on the use of synthetic workloads that may or may not capture actual usage patterns and other characteristics of a live execution environment.

In essence, we observe a general trade-off between accuracy and cost in this design process, where testing the actual implementation in its actual environment gives the most accurate results at the expense of a complex, costly experimental setup, while simulation gives less accurate results but with the advantage of a low-cost process.

We also notice that these two approaches are complementary and that they are most useful in different phases of the design and development process. The use of simulations is most appropriate in the early stages of the design process, when several high-level design alternatives must be tested, and therefore when practicality is more important than accuracy. Conversely, working with the actual implementation in a live environment is most appropriate later on in the development process, when the design focuses on low-level implementation details, and when performance analysis and improvement requires the identification and optimization of hot spots for the most common cases.

We believe, and our previous research suggests, that software development can greatly benefit by augmenting analysis and measurement tasks performed in-house with analogous tasks performed on the software deployed in the field. We call this the *Gamma* approach [5]. There are two main advantages of the Gamma approach: (1) analyses rely on actual field data, rather than on synthetic user data, and (2) analyses leverage the vast and heterogeneous resources of an entire user community.

There are many scenarios in which the Gamma approach can be exploited, and numerous tasks that can benefit from applying it. In previous work, we investigated the use of the Gamma approach for collecting coverage information from deployed instances of the software, determining classes of users of the software, and assessing the costs and identifying the issues associated with collecting and analyzing field data [1, 4].

In this paper, we investigate the use of the Gamma approach to support the evaluation and tuning of systems and protocols as they are deployed in the field. Our approach is to use dynamically configurable, minimally intrusive instrumentation techniques with fielded systems to collect various forms of execution traces. These traces can then be used locally to perform a simulation that uses real environment settings and real workloads.

The result of the simulation can then be analyzed locally to optimize the distributed system. Examples of such analyses include computing various types of coverage metrics, identifying hot spots in algorithms and protocols, and identifying bottlenecks in the network. The results of these on-line analyses can then be fed back into the system by fine-tuning system-configuration parameters. Furthermore, the results of the analysis can also be sent back to the designers for further analysis.

In the next section we detail our approach in relation to a specific class of distributed systems. We then discuss open issues and implications of our approach, and conclude with some directions for future research directions.

2. APPROACH AND EXAMPLE

To develop our approach, we started by considering a distributed publish/subscribe system [2] that we use as a representative of a particular class of highly distributed systems.

A publish/subscribe service supports dynamic, many-to-many communications in a distributed environment by passing information from producers (publishers) to consumers (subscribers) through a subscription mechanism. With a publish/subscribe service, producers simply publish information while consumers declare their interests by sending subscriptions that define selection criteria over information. The service evaluates subscriptions against publications and delivers each publication to all the subscribers that declared subscriptions matching that publication.

A distributed publish/subscribe system is a highly distributed system that implements the publish/subscribe service. Distributed publish/subscribe systems are usually architected as an interconnection of servers (or dispatchers), where each individual server acts as an access point for a subset of publishers and subscribers, and as a "router" of publications and subscriptions for all the other servers. Distributed publish/subscribe systems implement specific "routing" protocols to exchange and process subscriptions and publications between servers in such a way that the entire network of servers behaves collectively as a single publish/subscribe service.

Publish/subscribe systems are very appropriate for our study because their behavior and performance depend on the particular choice of protocols, on the environment, and on the workload generated by clients. In particular, among other things, a publish/subscribe protocol determines which subscriptions are propagated from one server to which neighbors, how and how often the propagation occurs, and which subscriptions are combined during the propagation process and how. All these protocol parameters must be calibrated to obtain the best operational responsiveness and efficiency within the given environment—characterized by the number and type of nodes, by the topology of their interconnections, and by the latency, bandwidth, and reliability of the underlying communication links—and under the given workload—characterized by the number, content, and timing of subscriptions and publications posed by each client.

As stated in the introduction, our approach is to use field data to measure the quality parameters of the system and to tune it accordingly. We describe the approach with the help of Figure 1, which represents a high level view of a system implementing the approach. The figure shows the different instances of the system. For each instance, a *data-collection* module on the same network gathers execution data from the nodes in the instance. We collect two kinds of execution data: topological information and event traces.

Topological information consists of information about the number of nodes in the system, their type, and their connections. Event traces are traces of the observable events that occur in the distributed system. For the specific system considered, we collect the following types of events:

Subscribe event: A subscribe event is generated every time a host issues a subscription for a given content. We record subscribe events as a quadruple (s, h_g, h_r, t), where s is the actual subscription, h_g is the host that generated the subscription, h_r is the server that received the subscription from h_g, and t is the time at which the subscription was received by h_r.

Publish event: A publish event is generated every time a host publishes some information. Analogously to subscribe events, we record publish events as a quadruple (p, h_g, h_r, t), where p is the information published, h_g is the host that published p, h_r is the server that received p from h_g, and t is the time at which the publication was received by h_r.

Subscription-forward event: A subscription-forward event is generated every time a server forwards a subscription to another server, to spread the subscription information in the system. We record subscription-forward events as a quadruple (s, h_f, h_r, t), where s is the forwarded subscription, h_f is the host that forwarded the subscription, h_r is the server that received the subscription from h_f, and t is the time at which the subscription was received by h_r.

Receive event: A receive event represents the reception of some published information by a host. We record subscribe event as a triple (s, h, t), where p is the published information, h is the host that received p, and t is the time at which p was received.

At a minimum, both events and topological information can be collected by suitable network probes located in the proximity of servers. Alternatively, if the system itself supports some form of logging facility, events can be directly and more efficiently extracted from server logs. Collected events and topological information are then stored in a local database, as shown in Figure 1.

Every given time interval, the execution data are retrieved by the *simulator* module, which replays the execution of the system simulating it with real topological information and real workloads. The result of such simulation, together with

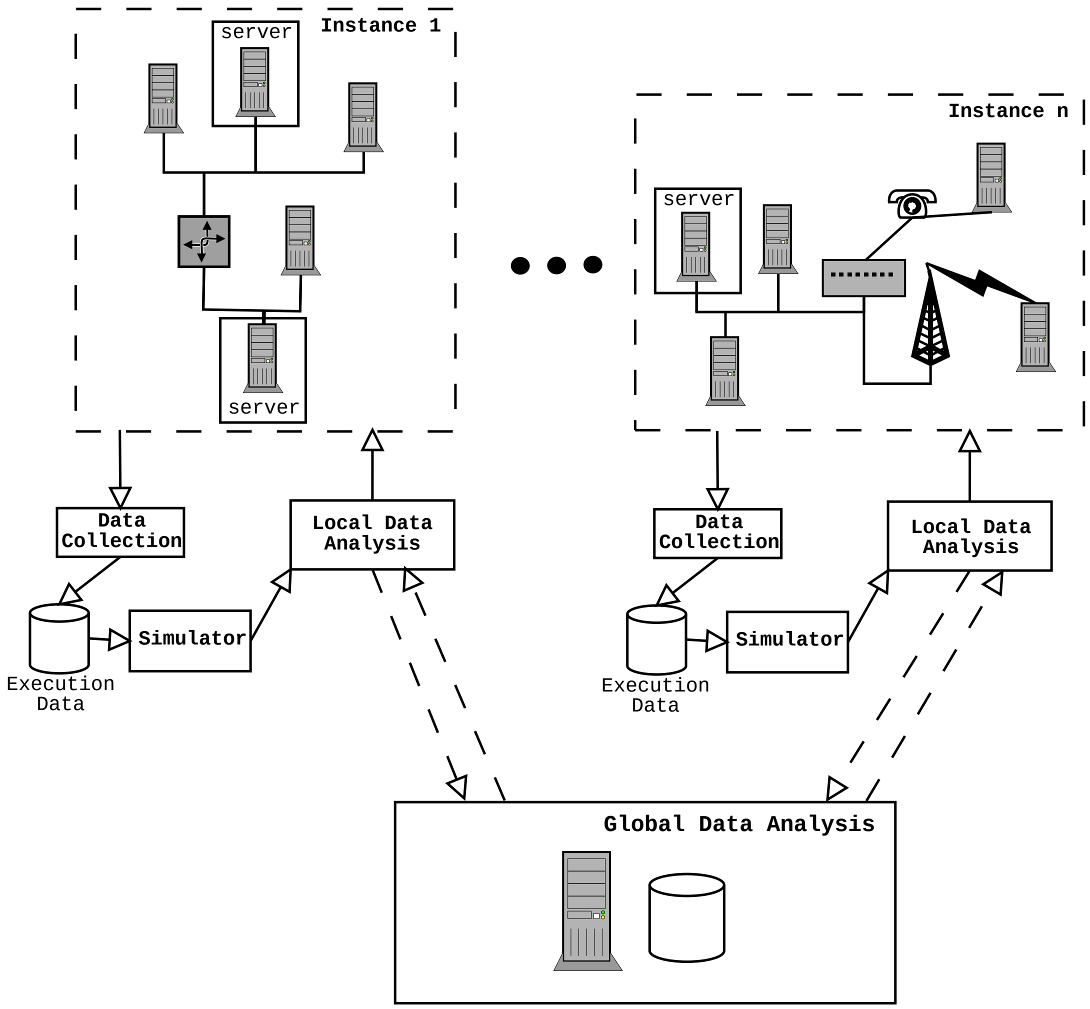

Figure 1: High-level view of a system that implements the approach.

the execution data, are then passed to the *local data-analysis* module, which measures quality parameters of the system as follows.

- Communication latency: the communication latency is expressed as $T_{ric}-T_{pub}$, where T_{ric} is the time at which a given published information is received and T_{pub} is the time at which the information was published. Using the event traces collected, the communication latency can be computed for each publish event, and the average, standard deviation, maximum, and minimum values for such latency can be computed.

- Protocol latency: although the protocol latency cannot be directly measured, we can use the number of false negatives in the system as an indicator of such latency. A false negative consists of some published information that should have been delivered to a node, according to the node's subscriptions, and was not. Although false negatives may occur because of faults in the protocol and because of network-related problems, they are usually the consequence of latencies in the propagation of the subscription information. False negative can easily be measured in our approach: in the replayed execution, we can assume instantaneous network communication and instantaneous subscription propagation, and thus identify all the receive events that should have occurred, but did not occur in the actual execution.

- Ratio of control information to actual content: we can measure this ratio by leveraging the execution data collected. The ratio r is given by the simple formula:

$$\text{r} = \frac{\sum_{s|(s,h_s,h_r,t)\in\{\text{subscription-forward events}\}} size(s)}{\sum_{p|(p,h_g,h_r,t)\in\{\text{publish event}\}} size(p)} \tag{1}$$

Once this information is computed and collected at each individual server, it can be used to control some parameters of the routing protocol for each server. For example, assuming a "heart-beat" routing protocol that propagates subscriptions at periodic intervals, a simple control algorithm could decide to slow down the heart-beat rate whenever the ratio of control information to actual content exceeds a given threshold value. Similarly, because the heart-beat rate is a general control parameter for the responsiveness of the service, a complementary control algorithm could decide to increase the heart-beat rate whenever the protocol latency raises above a given limit.

Obviously, the information collected on-site may also be shipped back to the protocol designer for further analysis. In fact, the use of the kind of close-loop control mechanism used in the previous examples may introduce dangerous instabilities in the protocol. So, before activating such automatic feedback controls, the designer might want to analyze the behavior of the system and either set the heart-beat rate at a fixed value, or set the automatic triggers to some safe threshold values.

3. DISCUSSION

We realize that the success of this approach depends on the wide-spread adoption and use of instrumented applications and their corresponding infrastructure in real settings. It is therefore extremely important that this approach be minimally intrusive in the normal operation of the system as perceived by the end-user. In particular, we see the following requirements:

- *Light-weight operation*: the instrumentation must introduce minimal overhead. The overhead must be adaptable to the overall load of the system so as to exploit periods of low activity to perform low-priority analyses and/or to communicate partial results or entire traces for off-site or off-line analyses.

  Our approach has the advantage that many kinds of information can be collected by simply monitoring the network. Those monitoring techniques impose no overhead on the nodes. However, some instrumentation may be necessary for collecting specific kinds of information (e.g., traditional coverage information).

- *Security and privacy*: the data collected by instrumented applications is likely to reveal sensitive information about the end-user or its operating environment. End-users must therefore be able to apply selection policies to determine exactly what may and may not be exposed. In any case, collected data must be protected against unwanted access both on the instrumented site, and in other communications between cooperating sites.

  As far as privacy is concerned, our approach is based on the idea of using sensitive information, such as subscription contents, only for the local data analysis; only filtered and summarized information, according to the users' preferences, is sent back to the system's designer. As for the security aspect, we have not yet considered any specific mechanism do define and enforce security policies for collected data. However we are confident that we will be able to leverage existing mechanisms, such as private/public key cryptography and digital signatures.

The effectiveness of our approach depends also on the type and complexity of the metrics that can be computed by on-site low-priority analysis modules. Here we envision a spectrum of levels of complexity. At one end of the spectrum, there are analyses that are computed entirely by self-contained and completely localized modules (i.e., by algorithms that use only locally-collected information). At the extreme opposite, there are analyses that themselves require intense information exchange and coordination between analysis modules at different sites. Our current focus is clearly on self-contained analyses, although future development of our approach might include more complex, distributed analyses.

4. REFERENCES

[1] J. Bowring, A. Orso, and M. J. Harrold. Monitoring deployed software using software tomography. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Ana lysis for Software Tools and Engineering (PASTE 2002)*, pages 2–8, Charleston, SC, USA, Nov 2002.

[2] 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.

[3] A. Carzaniga and A. L. Wolf. A benchmark suite for distributed publish/subscribe systems. Technical Report CU-CS-927-02, Department of Computer Science, University of Colorado, Apr. 2002.

[4] A. Orso, J. Jones, and M. J. Harrold. Visualization of program-execution data for deployed software. In *Proceedings of the ACM symposium on Software Visualization*, San Diego, CA, USA, June 2003 (to appear).

[5] A. Orso, D. Liang, M. Harrold, and R. Lipton. Gamma system: Continuous evolution of software after deployment. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'02)*, pages 65–69, Jul 2002.