UNIVERSITÀ DEGLI STUDI DELL'AQUILA
FACOLTÀ DI SCIENZE MATEMATICHE FISICHE E NATURALI

TESI DI LAUREA IN INFORMATICA

Co.M.E.T.A.

Mobility support in the *S*IENA publish/subscribe middleware

Candidato

Mauro Caporuscio

Relatori

Prof. Paola Inverardi

Prof. Alexander L. Wolf

ANNO ACCADEMICO 2000-2001

“*Da qui messere si domina la valle...*
Ciò che si vede, è.”

B.M.S.

Acknowledgement

I would like to thank my advisors, Prof. Paola Inverardi and Prof. Alexander L. Wolf, for making this experience possible and for their help and guidance through the entire course of this thesis.

I would like to thank Antonio Carzaniga for his patience correcting my "bad English", for his friendly teaching and because this work could not have been done without many discussions with him.

I would also like to thank the USENIX Association for supporting my work through the Research Exchange (ReX) grant.

I dedicate this work to my wonderful family: to my mom, my dad and my honey sister, for their love, help, support, confidence and guidance throughout my life... Thanks a lot... I love you so, so much!!

A special thanks to Maria Benigni for loving me and for her encouragement during all my study.

I would like to remember all my good friends: Michele "Falce" Mercuri and Francesco "Bugia" Troiani with whom I shared all the good and bad things of my (and their) life; Antonio "Lupo" Di Berardino, Romolo "Re" Salvi and Lorenzo "Rampyn" Felli for being my roommates for many years; "Il soccio" Alfredo "Freski" Navarra, Vincenzo "Biciu" Cesarini, Nicola "Alanghiro" Piccone, Emanuele "Oscar" and Alessandro "Soft" Asci, Simone "Scrigno" Scriboni, and Fabio "Xenon" Mancinelli, with whom I spent many beautiful days, for all smiles they gave to me; Armando "Bardone" Botticella for his marvelous diet; Marco "Egomet" Castaldi and Nathan "Peller Dude" D. Ryan, with whom I shared all America's "bullshit" (do you know what I mean?) and fun, for their friendship and help throughout my experience in Boulder (USA); and all the others, who do not appear in this list, each one important for a special thing... Thanks to all you guys! I'll carry you in my heart forever!!

Finally, but not less importantly, I would like to thank myself "Meskall" for my instability, my perseverance and my "hard head" in everything I did.

Boulder - March 21, 2002 *Mauro Caporuscio*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis is concerned with mobile applications that use a publish/subscribe infrastructure. In particular, this work consists of (1) a case study on the deployment of a publish/subscribe middleware on top of a wireless communication service, where mobility is supported at the network level, and (2) a design and initial implementation of a mobility support service realized within the publish/subscribe middleware.

The increasing size and performance of computer networks is generating a new phenomenon: networks are being pervasive and ubiquitous. While *pervasive* means that network connectivity is going to be a basic feature of any computing facility, *ubiquitous* refers to the ability of utilizing network connectivity independently of the physical location of the user. In this context (usually referred to as a *wide-area network*), applications are characterized by the fact that they are loosely coupled, asynchronous, and heterogeneous. This promotes a class of software system based on the abstract design called *event interaction* which in turn is supported by an emerging infrastructure called *Event notification service* [6].

Development in wireless technology are freeing application hosts from a constrained, fixed physical location in the network and enables the practical realization of the idea of *mobile computing.* In fact, portable computers (such as laptops and PDAs) are growing in popularity while they are shrinking in size. This process of miniaturization, combined with the emergence of high-speed wireless communications, allows users to use portable device with on-demand connections. In this scenario, mobile users can move together with their hosts across different physical locations, while remaining connected to the network through wireless links.

In addition to the mobility of hosts, new techniques based on code migration [10] have been developed to allow applications to move from host to host. These mobile applications, often referred to as “mobile agents”, aim to optimize information retrieval and other similar tasks by moving close to the data stores of interest, where they can execute their queries with low latency and network usage.

With the work described in CoMETA (*Co*mponent *M*obility using the *E*vent no*T*ification *A*rchitecture), we intend to combine the benefits of mobile applications (moving along with their host, or migrating from host to host) with the communication services offered by an advanced publish/subscribe service.

1.1 Contribution of This Thesis

In integrating publish/subscribe technology with mobile applications, we have two general choices of architecture: In one case we could simply attach the publish/subscribe system on top of a network that offers native support for mobility. In the opposite case, we could have the publish/subscribe system handle mobility without any direct support from the underlying network layers. In this latter case, the publish/subscribe service would implement an additional set of services designed to support mobile applications. These two alternative methods, detailed below, characterize the contribution of this thesis.

***S*IENA over a wireless network** We studied the integration of an event-based middleware on top of a wireless network, in a situation in which the mobility of clients is transparently managed at the network level. In particular, we focused on the *S*IENA distributed event-notification system [5], hosted over a General Packet Radio Service (GPRS) network.

In order to evaluate the behavior of Siena and its demands over the communication resources of the wireless network, we developed a test application (a distributed auction system) that we used in several simulated scenarios. Developing such an application required us to port the Siena client-side library to the Java^{TM} 2 Micro Edition, a platform specifically targeted at mobile devices such as cell phones and PDAs. The resulting application and library allowed us to run experiments on a simulated PDA, in combination with a highly configurable GPRS network emulator.

The primary goal of our experiments was to evaluate the impact of deploying Siena onto the wireless GPRS network. We did this from two different perspectives. The first was to gather data characterizing the performance of the three different low-level transport mechanisms (UDP, TCP, and "keep-alive" TCP that attempts to reuse TCP connection) on the wireless network. The second was to compare these results with baseline data collected on a local-area, wired network. The results of the experiments gave us an initial indication of whether a seamless integration of wired and wireless communication is feasible for a publish/subscribe communication service.

Mobility support in Siena We studied how to support mobile applications that use the Siena publish/subscribe system implemented over a wired-line network. We consider mobile applications that either move along with their host (e.g., because they execute on a laptop or a PDA) or move from host to host using mobile code technology. Regardless of the technology supporting mobility, we assume that application can detach from one Siena access point, travel to another network location, and reconnect to another Siena access point.

To support such applications, we designed and implemented a *mobility service* within the Siena publish/subscribe system. This service allows applications to receive notifications published while they are traveling to a new destination, and to restore the flow of notifications and their subscriptions when and where they reconnect to the Siena network at their destination.

As a basis for the mobility service, we implemented a persistent storage of notifications. We then implemented two additional functions, called **move-**

OutMaster and **moveInMaster**, that allow clients to switch from their current access point over to a new one, re-establishing their subscriptions as well as their flow of notifications. We implemented the moveInMaster facility in such a way that it can provide different levels of consistency for the switch-over function.

1.2 Structure of This Thesis

Section 2 introduces our starting-points and background. It presents in details the concepts of *mobility* and explains what SIENA is and how it works.

Chapter 3 describes how we put SIENA middleware on top of the GPRS Network and how we combined the concept of *Host Mobility* with the event-based architecture. It also explains the experimentation we made, in order to understand SIENA performances in a mobile environment, and presents our results.

Section 4 talks about our research in *Client Mobility* exploring the problems that it implies and describes how we allowed SIENA to support the mobility of its clients. This section also presents the algorithms we have been designing to solve the problems explained.

Finally in Section 5 we draw some conclusions summarizing our experience and discussing future developments.

Chapter 2

Background

In this chapter we would make an overview about our starting points: we briefly introduce the theory about *mobility*, illustrating different approaches and possible scenarios, and we give an high level presentation of the SIENA middleware and its principal characteristics. These concepts should be useful to understand the work explained through the next Sections in which we will examine how we put in touch the concepts of mobility and SIENA.

2.1 Mobility

Mobility breaks all bindings between hosts and software; the network structure may be mutable, nodes may come and go, processes may move between nodes, and programs may evolve and change structure. As some authors describe it, mobility is a "total meltdown" of the stability assumed by distributed systems [31]. From the software engineering perspective, *mobility* is defined as the study of systems in which computational components may change location, in a voluntary or involuntary manner, and move across a space that may be defined to be either *logical* or *physical*. This distinction is necessary to distinguish the different level in which mobility is handled.

2.1.1 Host Mobility

Host mobility (some authors refer to it as either *physical mobility* [31] or *mobile computing* [2]) entails the movement of mobile hosts (of all sorts and size) in the real world. It is assumed to be the next evolutionary step in the development of the worldwide communication infrastructure and the extension of wire-line networks. In fact, one can imagine a traditional static network, in which fixed hosts with static addresses exchange messages via the standard Internet infrastructure, with at the end-points some wireless-networks, made by an aggregation of base stations, that control message traffic from and to mobile devices (as showed in Figure 2.1). Mobile devices, even if physically detached from the fixed infrastructure, may interact with each other and with the fixed hosts, throughout wireless link. Sending data to and from

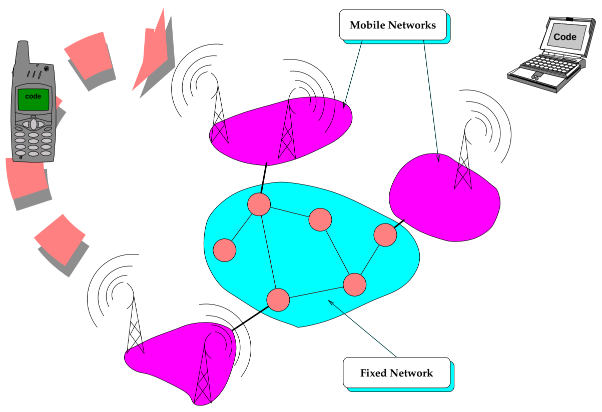


Figure 2.1: *Host Mobility*: hosts move in the real world.

a mobile unit requires the ability to find the current location of the device and to maintain the data flow as the unit moves from one place to another. This kind of mobility is managed at the network-layer and, therefore, the movement of the host is completely transparent at the application-layer. For example cellular phone system accomplish this through a combination of broadcast signals and hand-off protocols.

2.1.2 Code Mobility

Code mobility (or either *logical mobility* [31] or *mobile computation* [2]) instead, involves the movement of code (in all its forms) among hosts. At a level above the physical, there is a logical layer, called *Code Mobility*, that removes static bindings between the software components and the network hosts where they are executing. This allows components to be relocated to achieve flexibility and increase reconfiguration capability. In this scenario,

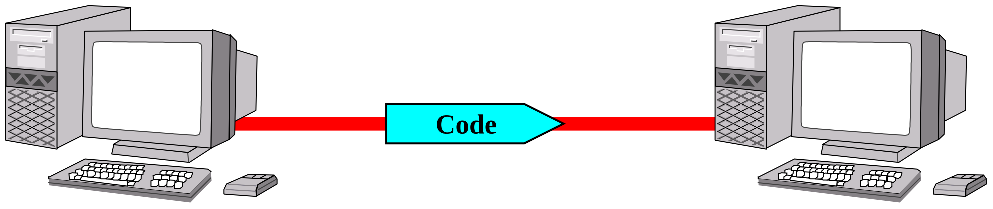


Figure 2.2: *Code Mobility*: code moves through hosts.

components are logical units that may move, together with their code fragments, from host to host across both wire-line and wireless network (see Figure 2.2). A system may be composed of several units some of which may be mobile. The overall architecture of the system can be considered to be independent of the location of each individual component.

Code mobility can be distinguished in two main categories depending on how the state of the execution of a mobile component is affected by the migration of that component from one host to the other. In particular, two types of mobility have been identified in the literature [4]:

weak mobility *Weak mobility* refers to the cases in which the code fragment is relocated by creating a fresh copy at the destination point or prior to start of its execution.

strong mobility By contrast, *strong mobility* refers to the movement of code that maintains the state of execution. The execution state is relocated along with the code thus allowing it to continue running even after the move.

Code mobility is viewed as offering designers a new set of conceptual and programming tools that seek to exploit the opportunities made available by the distributed computing infrastructure. An example of such technologies is a new family of programming languages, usually referred to as *mobile code languages* (MCL) [9], such as *Java*^{TM} by Sun Microsystem [35] and *Telescript*^{TM} by General Magic [18], that support mobility at various degree of sophistication.

An important aspect of mobility that is common to both *host mobility* and *code mobility* is the relation between a mobile component and other mobile or fixed components. These relations, that may obviously change during the lifetime of the component, are captured by the notion of *context*. Depending on the nature of the mobile system and on the nature of the mobility support, some context relations may be maintained through the migration, other may have to be temporary suspended and others may be discarded or changed after a migration. The first option is most common in settings involving physical mobility while the third is implemented when logical mobility takes place across connected sites, as in the case of Internet. This brief introduction to the *mobility* will result useful to understand problems, and proposed solutions, discussed in Sections 3 and 4.

2.2 The SIENA Middleware

A common approach to achieving loose coupling is a *event-based* design style. In a event-based system, component interaction are modeled as asynchronous occurrences of, and responses to, *events*. To inform other components about the occurrences of internal events, components emit *notifications* containing information about the events (i.e. to communicate a state changes). Upon receiving notifications, other components can react by performing action that, in turn, may result in the occurrence of other events and the generation of additional notifications. In general, the asynchrony, heterogeneity, and inherent high degree of loose coupling that characterize applications for wide-area networks suggest event interaction as a natural design abstraction for a growing

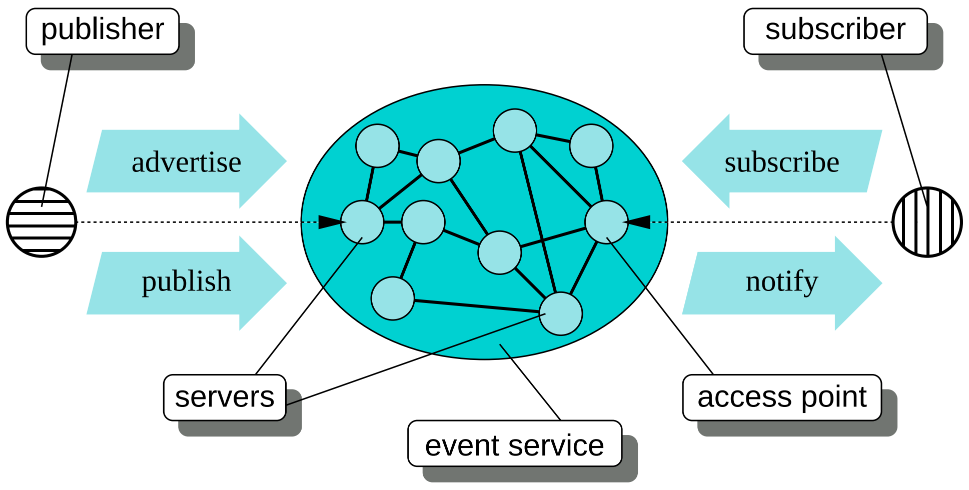


Figure 2.3: SIENA: Distributed Event Notification Service.

class of distributed systems.

SIENA (*S*calable *I*nternet *E*vent *N*otification *A*rchitecture) is an Internet-scale event notification service that is representative of capabilities for scalable event notification middleware. SIENA is implemented as distributed network of servers (as show in Figure 2.3) that provide clients with *access points* offering an extended publish/subscribe interface. The clients are of two kinds: *Object of interest*, which are the generators of notifications, and *interested parties*, which are the consumers of notifications; of course, a client can act as both of them. Clients use the access point of their local servers to *advertise* the information about notifications that they generate and *publish* the advertised notifications. Clients also use the access points to *subscribe* for individual notifications of interest. SIENA is responsible for *selecting* the notifications that are of interest to clients and then *delivering* those notifications to the clients via the access points.

SIENA is a *best effort* service in that it does not attempt to prevent race conditions included by network latency. This is a pragmatic concession to the realities of Internet-scale services, but it means that clients of SIENA must be resilient to such race conditions. For instance, clients must allow for the possibility of receiving a notification for a cancelled subscription.

2.2.1 A Brief Overview on SIENA API and Semantics

At a minimum, an event notification service has to export two functions that together define what is usually referred to as the *publish/subscribe protocol.* Interested parties specify the events in which they are interested by means of the function *subscribe.* Objects of interest publish notifications via the function *publish.* Figure 2.4 shows a filter on the top and a notification matching the filter on the bottom. SIENA extends the publish/subscribe protocol with

string	*dest* = *MXP*
int	*price* < *500*

string	*carrier* = *UA*
string	*dest* = *MXP*
int	*price* = *400*
bool	*upgradeable* = *true*

Figure 2.4: A Siena Filter and a Siena Notification

an additional interface function called *advertise*, which an object of interest uses to advertise the notifications it publishes. SIENA also adds the functions *unsubscribe* and *unadvertise.* Subscription can matched repeatedly until they are cancelled by a call to *unsubscribe.* Advertisement remain in effect until they are cancelled by a call to *unadvertise.*

Table 2.2.1 shows the interface function of SIENA. The expression given to *subscribe* and *unsubscribe* is a *pattern*, while the expression given to *advertise* and *unadvertise* is a *filter.* The parameter *identity* specifies the identity of the object of interest or interested party. The only requirement that SIENA imposes on identifiers is that they be unique.

Notification

An *event notification* is a set of typed attributes. Each individual attribute has a *type*, a*name*, and a *value*, but the notification as a whole is purely a structural value derived from its attributes. Attribute names are simply character strings. The attribute types belong to a predefined set of primi-

publish(notification *n*)
subscribe(string *identity*, pattern *expression*)
unsubscribe(string *identity*, pattern *expression*)
advertise(string *identity*, filter *expression*)
unadvertise(string *identity*, filter *expression*)

Table 2.1: Interface of SIENA

tive types commonly found in programming languages and database query languages, and for which a fixed set of operators is defined.

Filters

An *event filter* (or simply a *filter*) selects event notification by specifying a set of attributes and constraints on the values of those attributes. Each attribute constraint is a tuple specifying a type, a name, a binary predicate operator (i.e. $=$, $\neq$, $<$, $>$, etc) and a value for an attribute.

When a filter is used in a subscription, multiple constraints for the same attribute are interpreted as a conjunction (all such constraint must be matching): a notification *n matches* a filter f or equivalently that *f covers n.* Notice that the notification may contain other attributes that have no correspondents in the filter.

Patterns

While a filter is matched against a single notification based on the notification's attribute value, a *pattern* is matched against one or more notifications based on both their attribute values and on the combination they form. At its most generic, a pattern might correlate events according to any relation.

SIENA does not provide a complete pattern language, but a *pattern* is defined as a sequence of filters:

$$f_1 \cdot f_2 \cdots f_n$$

This is matched by a temporally ordered sequence of notifications, each one matching the corresponding filter.

Advertisements

The motivation for advertisements is to inform the event notification service about which kind of notifications will be generated by which object of interests, so that it can best direct the propagation of subscriptions. The idea is, that while a subscription defines the set of interesting notifications for an interested party, an advertisement defines the set of notifications potentially generated by an object of interest. Therefore, the advertisement is relevant to the subscription only if these two set of notifications have a nonempty intersection.

Unsubscriptions and Unadvertisements

Unsubscriptions and unadvertisements serve to cancel previous subscriptions and advertisements, respectively. Given a simple unsubscription **unsubscribe**(X, f), where X is the identity of an interested party and f is a filter, the event notification service cancels all simple subscriptions **subscribe**(X, g) submitted by the same interested party X with a subscription filter g covered by f. In analogous way, unadvertisements cancel previous advertisements. Note that an unsubscription (unadvertisement) either cancels previous subscriptions (advertisements) or else has no effect. It cannot impose further constraints onto existing subscriptions. For example, subscribing with a filter [price>100] and than unsubscribing with [price>200] does not result in creation of a reduced subscription [price>100, price≥200]. Rather, the unsubscription simply has no effect, since it does not cover the subscription. Note also that all subscription covered by an unsubscription are cancelled by that unsubscription.

Timing issues

The semantics of SIENA depends on the order in which SIENA receives and process requests (subscriptions, notification, etc.). For instance, in the subscription-based semantics, a subscription s is effective after it is processed and until an unsubscription u that cancels s is processed.

In the most general case, a service request R, say a subscription, is generated at time R_g, received at time R_r, and completely processed at time R_p (with $R_g \leq R_r \leq R_p$). SIENA guarantees the correct interpretation of R immediately after R_p. Notice that the *external delay* $R_r - R_g$ is caused by external communication mechanisms and is by no means controllable by SIENA. The *processing delay* $R_p - R_r$ is instead directly caused by computations and possibly by other communication delay internal to SIENA.

2.2.2 Architecture of SIENA

As show in Figure 2.3, the implementation of SIENA comprises a number of interconnected *servers*, each serving some subset of the clients of the service. In effect SIENA is a wide-area network of pattern matches and routers overlaid atop some other wide-area communication facility, such as the Internet. One reasonable allocation of such servers might be to place a server at each administrative domain within the low-level, wide-area communication network. A pair or interconnected servers use a server/server communication protocol that determines what kinds of information they can exchange, and in which direction. An interconnection topology and a protocol together define what we refer to as an *architecture* for SIENA. There are three basic architectures for SIENA: *Hierarchical client/server*, *acyclic peer-to-peer*, and *general peer-to-peer*.

Hierarchical client/server

In the *hierarchical client/server* architecture (see Figure 2.5), the servers form a hierarchical topology, with each server (except the root server) behaving like a SIENA client of the server one level up the hierarchy. The main problems exhibited by this architecture are the potential overloading of servers high in the hierarchy and the fact that each server is a single point of failure.

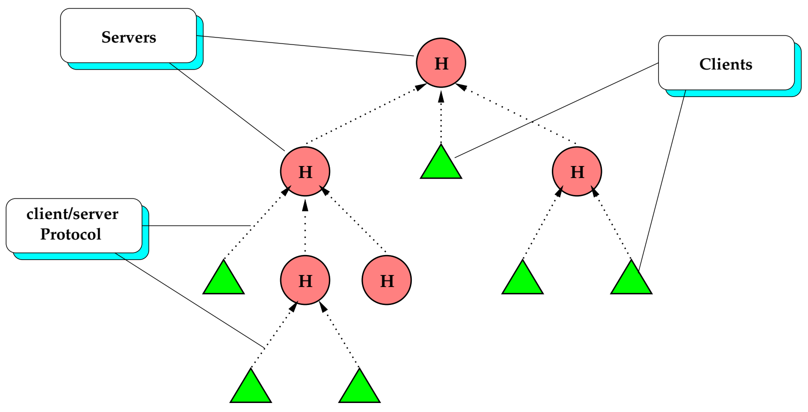


Figure 2.5: Hierarchical client/server architecture.

Acyclic peer-to-peer

In this architecture, servers communicate with each other symmetrically as peers in an acyclic undirected graph (as showed in Figure 2.6), adopting a protocol that allow a bi-directional flow of subscriptions and notifications. The configuration of the topology forms an acyclic undirected graph.

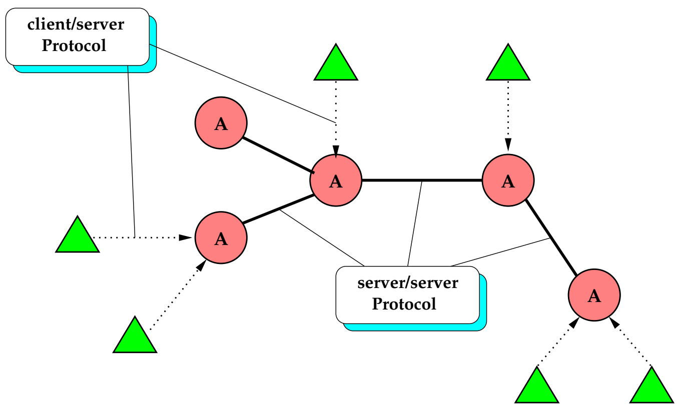


Figure 2.6: Acyclic peer-to-peer server architecture.

General peer-to-peer

Removing the constraint of acyclicity from the acyclic peer-to-peer architecture, SIENA network may be configured as a general peer-to-peer architecture. As depicted in Figure 2.7, a general peer-to-peer architecture can have multiple paths of bi-directional communication between servers. Allowing redundant connections makes it more robust respect to failures of a single servers. These three basic architectures can be combined to form hybrid ar-

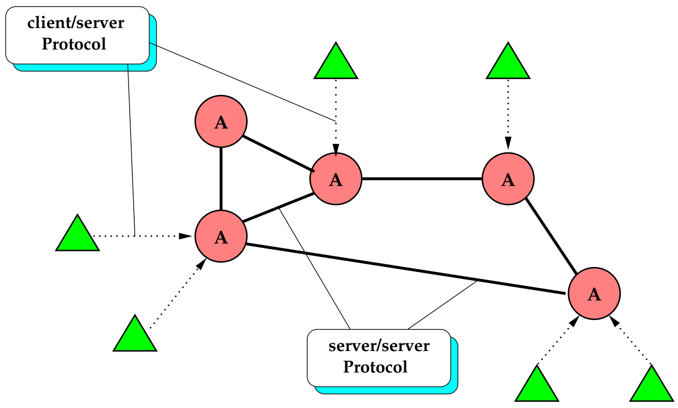


Figure 2.7: General peer-to-peer server architecture.

chitectures, such as an acyclic peer-to-peer topology of subnets, each subnet being hierarchy. Once topology of servers is defined, they must establish appropriate routing paths to ensure that notifications published by an object of interest are correctly delivered to all the interested parties that subscribed for them. In general, notifications must "meet" subscriptions somewhere in the network so that the notifications can be selected according to the subscriptions and then dispatched to the subscribers.

2.2.3 Processing Strategies

Once a topology of servers is defined, the servers must establish appropriate routing paths to ensure that notifications published by an object of interest are correctly delivered to all the interested parties that subscribed for them. In general, notifications must "meet" subscriptions somewhere in the network so that the notifications can be selected according to the subscriptions and then dispatched to the subscribers.

Routing strategies in SIENA hierarchical architecture

The main idea behind the routing strategy of SIENA is to send a notification only toward event servers that have clients that are interested in that notification, possibly using the shortest path. There are two simple principles that become requirements for the SIENA routing algorithms:

downstream replication: A notification should be routed in one copy as far as possible and should be replicated only downstream, that is, as close as possible to the parties that are interested in it.

upstream replication: Filters are applied, and patterns are assembled upstream, that is, as close as possible to the sources of (patterns of) notifications.

These principles are implemented by two classes of routing algorithms, the first of which involves broadcasting subscriptions and the second of which involves broadcasting advertisements:

subscription forwarding: In an implementation that does not use advertisements, the routing paths for notifications are set by subscriptions, which are propagated throughout the network so as to form a tree that connects the subscribers to all the servers in the network. When an object publishes a notification that matches that subscription, the notification is routed toward the subscriber following the reverse path put in place by the subscription.

advertisement forwarding: In an implementation that uses advertisements, it is safe to send a subscription only toward those object of interest that intend to generate notifications that are relevant to that subscription. Thus, advertisements set the paths for subscription, which in turn set the paths for notifications. Every advertisement is propagated throughout the network, thereby forming a tree that reaches every server. When a server receives a subscription, it propagates the subscription in reverse, along the path to all advertisers that submitted relevant advertisements, thereby *activating* those paths. Notification are then forwarded only through the activated paths.

Subscription-forwarding algorithms realize a *subscription-based* semantics, while advertisement-forwarding algorithms realize an *advertisement-based* semantics.

Chapter 3

Evaluating SIENA in a Wireless Network

As we explained in the previous sections, our interest is to study the integration of the benefits of mobile applications (moving along with their host or migrating from host to host) with the communication service offered by an advanced *publish/subscribe* middleware.

We defined the host mobility as the ability of devices (together with their application) to move around the real world. These physical components are generally referred to as *mobile hosts* and they come in different sizes from a laptop to a cellular phone or other wearable devices. It is reasonable to imagine some software-components running on them with the necessity to send (receive) messages to (from) other remote hosts, mobile or fixed. Of course, in order to allow the information exchange between components, the mobile hosts in which they are running need some form of wireless communication link.

In this chapter we will focus on the integration of *host mobility* with SIENA. We assume that a component is running on a mobile device (such as a PDA) and it uses a SIENA *client* in order to exchange messages with the external world. As we described in Section 2.2, a SIENA *client* that publishes or receives events must be connected to a SIENA *server* (access point). In particular we adopt a server running on a fixed Internet host.

This means that the wireless link acts as a "bridge" between the mobile

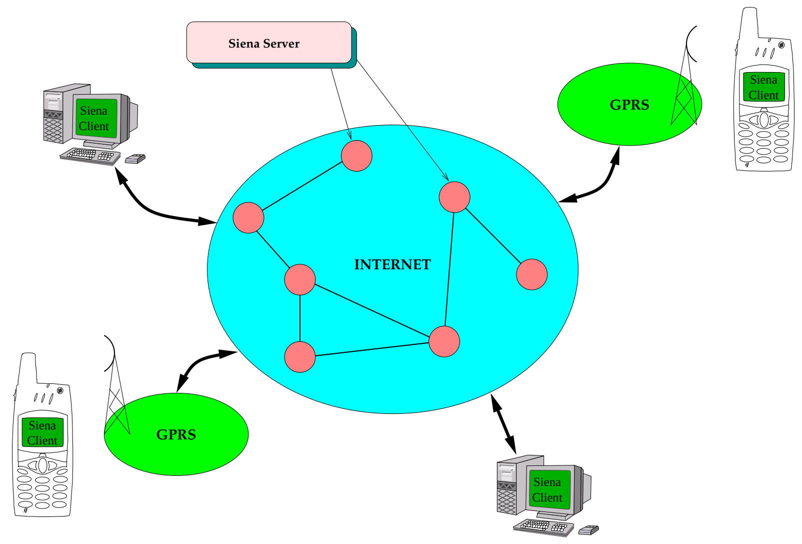


Figure 3.1: GPRS network makes a "bridge" between Client and Server.

client and the fixed server, handling mobility of the client in a way that is completely transparent to the server and the whole SIENA middleware.

In this scenario (depicted in Figure 3.1), we want to study SIENA's performance in combination with to different network protocols (TCP and UDP). In order to evaluate the behavior of SIENA and its demands over the communication resources of the wireless network, we developed a distributed test application called *auction system* (see Section 3.3) that we used in several simulated scenarios. Developing such an application required us to port the SIENA client-side library to the *Java^{TM} 2 Micro Edition* (discussed in Section 3.1), a platform specifically targeted at mobile devices such as cell phones and PDAs. We have also studied how to set up the graphic user interface, using the *J2ME^{TM} Wireless Toolkit* [37], in order to make the application user-friendly. We then used the resulting application to run experiments on a simulated PDA.

In the situation described above, a client may change its status quickly and often. Thus, it could connect (or disconnect) to the network in every moment. This may be a problem when using a publish/subscribe middleware like SIENA. In fact, since SIENA does not provide a mechanism for messages persistence, a mobile client could lose some notifications while it is disconnecting. We studied a simple solution (described in Section 3.2) for this problem, and we used it in the development of the *auction* System.

Finally, to establish wireless links, we choose a GPRS network (refer to Section 3.4) because it allows us to use IP-based protocols and because it represents the last step in path to UMTS Network.

In the following Sections we will introduce the tools we used for our experiments, we will discuss the experiment set up and the results we obtained.

3.1 Java 2 Platform, Micro Edition

J2ME [19, 36], a version of the Java™ 2 Standard Edition (J2SE™) [38], is aimed at the consumer and embedded devices market. It specifically addresses the rapidly growing consumer space that covers commodities such as cellular telephones, pagers, palm organizers, set-top boxes, and others. J2ME provides a complete set of solutions for creating state-of-the-art networked applications for consumer and embedded devices. It enables device manufacturers, service providers, and application developers to deploy compelling applications and services to their customers. J2ME defines the following set

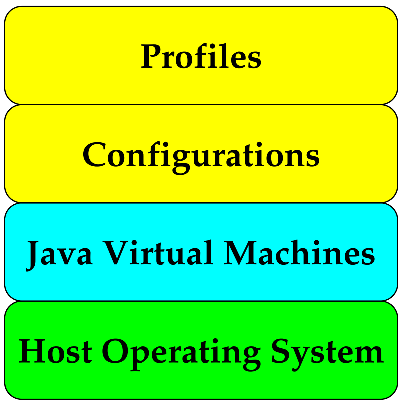


Figure 3.2: High-level view of J2ME.

of tools that can be used with consumer devices:

- A Java virtual machine

- Libraries and APIs that are suitable for consumer devices (configurations and profiles)
- Tools for deployment and device configuration

The first two components make up the J2ME runtime environment. Figure 3.2 shows how the different high-level layers of J2ME fit together.

3.1.1 Configurations

Cellular telephones, pagers, organizers, and so on, are diverse in form, functionality, and feature. For these reasons, J2ME supports minimal configurations of the JVM and APIs that capture the essential capabilities of each kind of device. At the implementation level, a J2ME configuration defines a set of horizontal APIs for a family of products that have similar requirements on memory budget and processing power. A configuration specifies:

- Java programming language features supported
- JVM features supported
- Java libraries and APIs supported

Currently there are two standard configurations: The Connected Limited Device Configuration (CLDC), and the Connected Device Configuration (CDC).

CLDC

The Connected Limited Device Configuration (CLDC) is intended for cellular phones, two-way pagers, and organizers. It targets devices with between 160 and 512 KB of memory. A reference implementation of the CLDC is available. A configuration, such as the CLDC or CDC, is more useful when used along with a profile.

CDC

The Connected Device Configuration (CDC) is intended for set-top boxes, Internet TVs, and in-car entertainment systems. The CDC targets devices

CLDC	**CDC**
Implements a subset of Java features and APIs	A full Java implementation
The Java virtual machine is KVM	The Java virtual machine is CVM
For limited devices	For more powerful devices
Processor: 16 or 32-bit	Processor: 32-bit
Targets devices with 160 - 512 KB of memory	Targets devices with at least 2 MB of memory

Table 3.1: CLDC vs. CDC

that have at least 2 MB of memory, and can support a complete implementation of the standard JVM, and Java programming language. A brief comparison of CLDC and CDC is shown in Table 3.1.1.

3.1.2 Virtual Machines

The CLDC and CDC configurations each define the set of Java and virtual machine features supported. Therefore, each configuration will have its own JVM. Clearly, the CLDC virtual machine will be smaller than the virtual machine required by the CDC since it supports less features. The virtual machine for the CLDC is the Kilo Virtual Machine (KVM), and the one for the CDC is the CVM.

KVM

The Kilo Virtual Machine (KVM) is a complete Java runtime environment for small devices. It is a true Java virtual machine as defined by the JVM Specification except for some specific deviations that are necessary for proper functioning on small devices. It is specifically designed from the ground up for small, resource-constrained devices with a few hundred kilobytes of total memory.

The KVM is derived from a research project called Spotless at Sun Microsystems Laboratories. The aim of the project was to implement a Java system for the Palm Connected Organizer.

CVM

Initially, the CVM used to stand for "Compact Virtual Machine". Sun Engineers however, realized that it might be confused with the KVM. So the C does not stand for anything now. It is just the C Virtual Machine or CVM. It is designed for consumer and embedded devices, and it supports all Java™ 2 Platform, version 1.3, VM features and libraries for security, weak references, JNI, RMI, and JVMDI.

3.1.3 Profiles

J2ME makes it possible to define Java platforms for vertical markets by introducing profiles. At the implementation level, a profile is a set of vertical APIs that reside on top of a configuration to provide domain-specific capabilities, such as user interfaces.

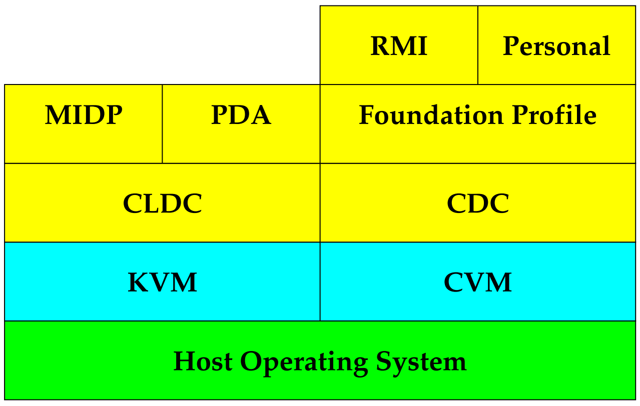


Figure 3.3: J2ME architecture.

Currently, reference implementations exist for two profiles: The Mobile Information Device Profile (MIDP), and the Foundation Profile (FP). MIDP is to be used with the CLDC and FP is to be used with the CDC. Other profiles in the works include: The PDA profile, RMI profile, and Personal Profile. The structure of the various J2ME configurations and profiles is depicted in Figure 3.3.

The MID Profile (MIDP)

The Mobile Information Device Profile (MIDP) extends the CLDC to provide domain specific APIs for user interfaces, networking, databases, and timers. MIDP is meant to target wireless phones and two-way pagers. A reference implementation is available, and an easy-to-use development environment (Wireless Toolkit [37]) is also available.

The PDA Profile

The Personal Digital Assistant (PDA) profile is based on the CLDC and will provide user interface APIs (which are expected to be a subset of the AWT) and data storage APIs for handheld devices. The PDA profile is still in the works and no reference implementation is available yet.

The Foundation Profile (FP)

The Foundation Profile extends the APIs provided by the CDC, but it does not provide any user interface APIs. As the name "foundation" implies, the Foundation Profile is meant to serve as a foundation for other profiles, such as the Personal Profile and the RMI profile.

The Personal Profile (PP)

The Personal profile extends the Foundation profile to provide GUIs capable of running Java Web applets. Since PersonalJava™ is being redefined as the Personal profile, it will be backward-compatible with PersonalJava 1.1. and 1.2 applications. No reference implementation for the Personal Profile is available yet.

The RMI Profile

The RMI profile extends the Foundation profile to provide Remote Method Invocation (RMI) for devices. It is meant to be used with the CDC/Foundation and not the CLDC/MIDP.

The RMI profile will be compatible with J2SE RMI API 1.2.x or higher. However, no reference implementation is available yet.

3.2 Service Discovery

In traditional client/server computing, a client that needs a particular service must known the address of the *Service Provider.* For example, a client that intend to use a Time-Synchronization service must know the address of a time server. *Service Discovery* is the process by a client finds out about one or more service providers for a specific service. The publish/subscribe

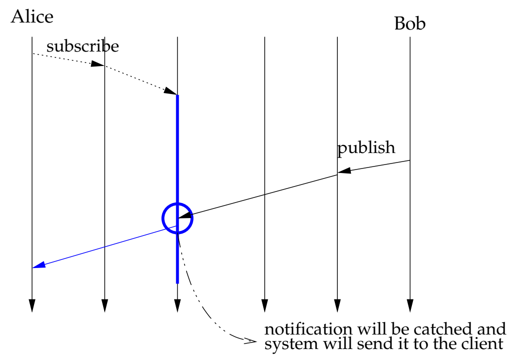

Alice
Bob
subscribe
publish
notification will be catched and system will send it to the client

Figure 3.4: Alice subscribes before Bob advises for a ticket.

architecture seem to offer a natural solution to the problem of service discovery. In fact, in this approach a user (the *Service Client*) subscribes for a service and, when it is available, he will receive a notification (see Figure 3.4). This simple protocol introduces new problems that we will describe in the following sections.

In the scenario depicted in Figure 3.4, Alice catches the notification published by Bob, which allows Alice to contact Bob. Unfortunately, the same protocol would fail in case Bob announced the availability of his service *before* Alice submitted her subscription, as show in Figure 3.5.

There are two possible solutions for this problem. One is to insert an additional component, a *Repeater*, to provide caching functions, in the system architecture. The other one is using a pair <request, offer>. We will describe the second one in the following. This solution is *application-level* in the

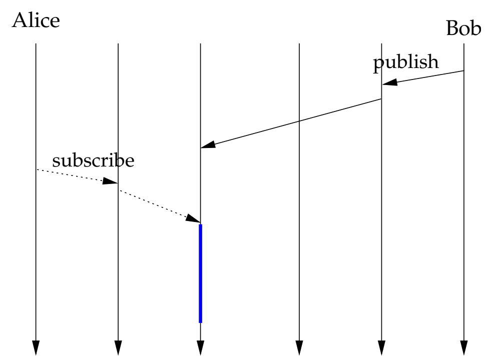


Figure 3.5: Bob advises for a ticket availability before Alice subscribes

sense that it does not change the publish/subscribe architecture, but instead combines its own features to put client and server in touch.

3.2.1 A Request-Offer Combination

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

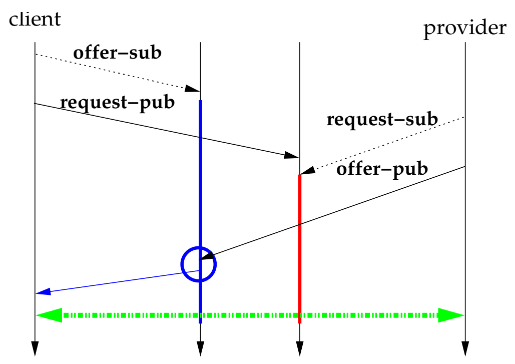


Figure 3.6: Both users send a pair <request, offer>.

actions explain how a pair <request, offer> works (see also Figure 3.6):

- provider-pair:

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

  step p2: publish "Service S" *Offer* Publication

- client-pair:

  step c1: subscribe for "Service S" *Offer* Subscription

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

In the rest of section we will refer to the pair made by the service provider as *provider-pair* and we will refer to the pair made by the client as *client-pair*.

Note that these are not *atomic actions* but there is a little time between the subscriptions and the notifications. Moreover, we assume that the publish/subscribe service is unreliable and messages could be delayed through the network. So, different cases are possible and we will describe these in the following subsections. We will show how the couple <*provider-pair*, *client-pair*> works in every one of these.

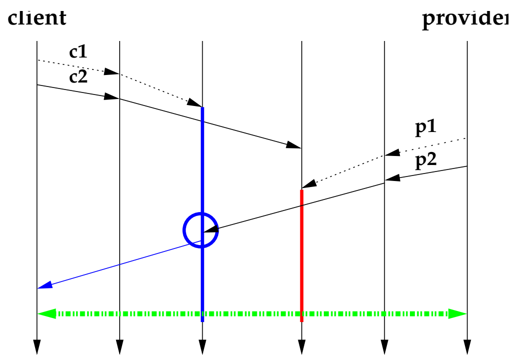


Figure 3.7: The *client-pair* is sent before the *provider-pair*.

case 1 We suppose that *client* subscribed for service before the *provider* publish its announcement. As depicted in Figure 3.7, the Alice's notification will be lost, but her offer-subscription will catch the Bob's offer-publication.

case 2 This case, in which we suppose that Alice (the *client*) subscribes for service after Bob (the *provider*) sends his notification (see Figure 3.5), represents the main problem. Since now we are using the pair <request, offer> (as depicted in Figure 3.8), Bob has been subscribing for interested party (action p1). This request-subscription will catch Alice's request-publication (action c2) and now Bob knows that a user needs his service. Therefore Bob can publish again his notification. This allows Bob to contact Alice and offers her the service.

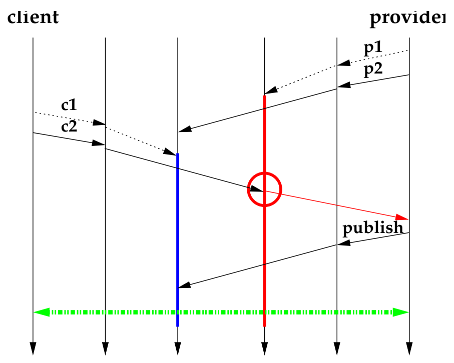


Figure 3.8: The *client-pair* is sent after the *provider-pair*.

case 3 Since the Publish/Subscribe architecture is unreliable, messages could be delayed through the network. Therefore we could have another possible scenario (such as depicted in Figure 3.9). In this case publications will intersect and both users, *provider* and *client*, will receive each other's publication. We suppose that the *provider* always publishes another notification. This will cause that the *client* will receive the same notification for two times. Anyway, we are sure that they will be able to establish a session and we can conclude the pair <request, offer> also works in this case.

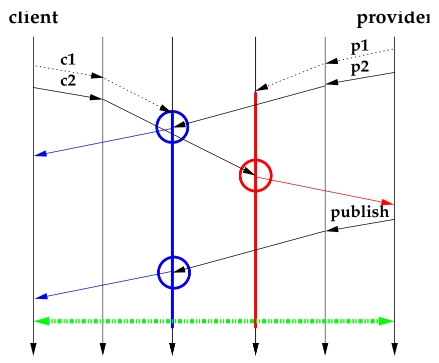


Figure 3.9: The *client-pair* and *provider-pair* are sent at the same time.

3.2.2 Observations

It is important to note that the final communication, between *provider* and *client*, should be established and conducted according to the specific service protocol (as depicted in Figure 3.10). For example, if the offered service is a printing service, the communication must be established using the appropriate printing protocol (e.g. using IPP [20]).

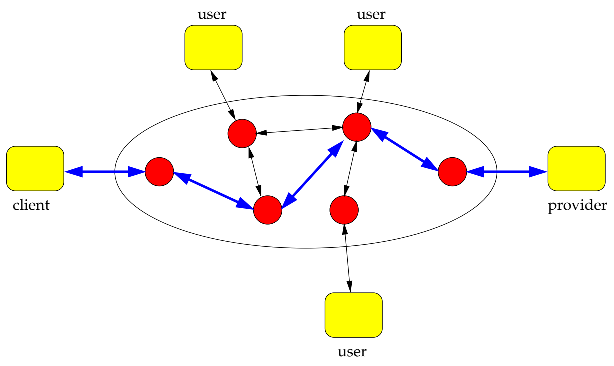


Figure 3.10: Client/server communication between *client* and *provider*.

We described how the pair <request, offer> works in different cases but, it is also important to note that this solution may not works if messages are lost through the network. In fact, if a notification does not reach the node (see Figure 3.11) where subscriber is connected, the client can not receive the message associated to the event.

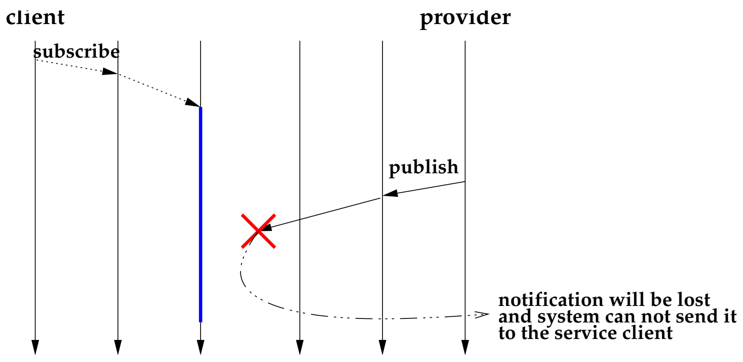


Figure 3.11: Publish/Subscribe is unreliable architecture.

Finally, it is important to note that pair must be issued in the exact sequence, that is with the notify message following the subscribe message. In fact there are cases in which the pairs formatted as <notification, subscription> does not work. As depicted in Figure 3.12 the pair sent by the client could not intersect the provider ones and none of them will know

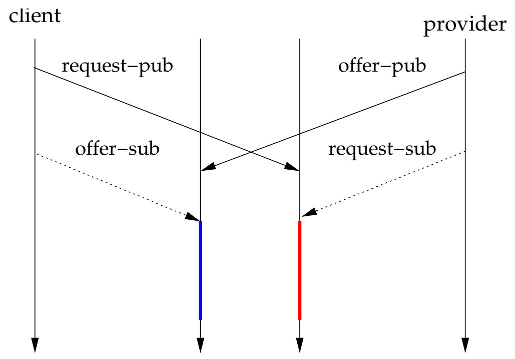


Figure 3.12: The pair formatted as <notification, subscription> does not work.

about the other.

As we explained above, this solution is *application-level* and it does not change the down-level architecture. This means that some problems such as the unreliability of the protocol cannot be fixed using the couple <*client-pair, provider-pair*>. Moreover if a number of clients and providers are using the *pair* <*request, offer*>, it could cause a traffic overload through the network and thus a denial of service could be happen.

3.3 Auction System

We decide to develop an *auction* system because it is a simple system but with high number of message exchange and real-time constraints. Thus, it allows us to study the Siena performances in the presence of low bandwidth and high error probability network such as a wireless network.

What we want is to develop a peer-to-peer application that allows clients to sell and buy items. Buyers and sellers, could be viewed as independent components of the system that use the event-based middleware to communicate with each other. The high level architecture of the system is showed in Figure 3.13. In particular, if a client is interested in buying a ticket (*buyer*), he will subscribe for events that advertise the availability of tickets. Conversely, a client that wants to sell a ticket (*seller*) emits an event to communicate the availability of tickets. When the *buyer* receive this notification, he can publish a bid for the ticket.

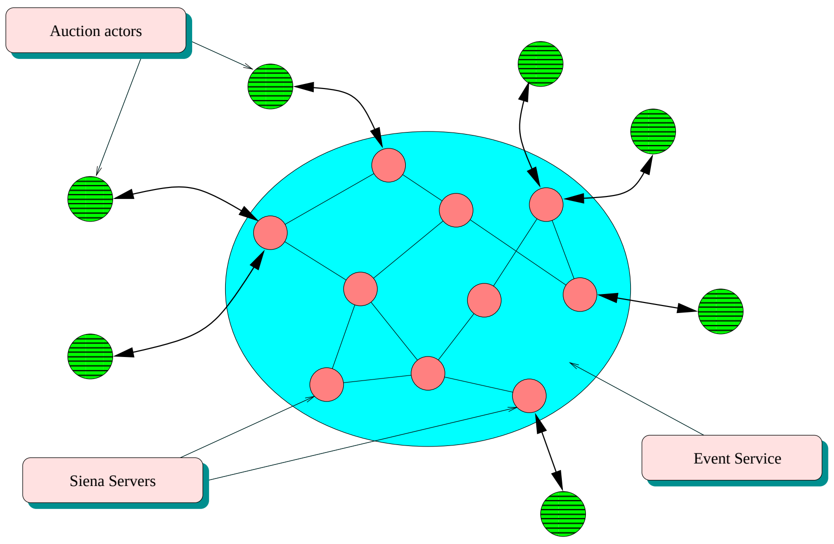


Figure 3.13: The *Auction System*'s architecture

3.3.1 Auction characteristics

In this section, we define the rules that determine the behavior of the auction system and the properties of the system.

Type of available auctions

We want to develop a system that allows users to choices different kinds of auctions. These can be:

close: A user can send the amount of his bid, but he can not check the Auction status.

open: A user can send the amount of his bid and he will be informed whether he is not currently the higher bidder. In this latter case, he can raise his bit.

Selling

Before advertising an item for sale, a seller must set up the following information:

kind of auction: An *Open* or *Closed* auction.

title/item Name: Brief description of the item he is selling.

category: *Category* in which the items will be listed.

description: A complete description of his item.

duration: The period of time during which the item can be auctioned.

starting price: If seller wants the bidding to start at a certain price, he needs to put it in there. This will set the *starting bid* at the price he specify.

reserve price: Setting a price, gives seller the option of not selling the item if the bids do not reach his *reserve price*.

bid increment: The amount seller wants the auction to increase after each bid is placed.

payment method: One or multiple methods of payment.

shipping: Seller must choose whether he will ship nationally only or internationally. He must also select who pays for shipping:

- seller
- buyer
- fifty-fifty

Buying

When user has found an item he would like to bid on, he can send the amount of his bid applying the *bid increment* established by the seller. After receiving a bid from a buyer, the system may notify the buyer of following events:

out-bidding: While bidder is waiting for the *open auction* finish, he will receive a notification informing him whether he has been out-bid by another bidder. Only in this case, the bidder can raise his bid. Instead, During a *closed auction*, the bidder will not receive any notification and therefor will not be able to raise his bid.

winning: If a user is the highest bidder when the *open/closed auction* ends, he will receive a notification, containing the personal information of the seller, that he can use in order to contact the seller. In the *closed auction*, a notification will also be sent to all users, that bid for that item, to inform them that the auction has been closed and to tell them who the winner is.

3.3.2 Quality of Service

The following quality of service apply to the auction system:

real-time delivery: It is important that every message is delivered in real time. In fact, high network latency may cause undesired effects such as bids switching and advertisement losing.

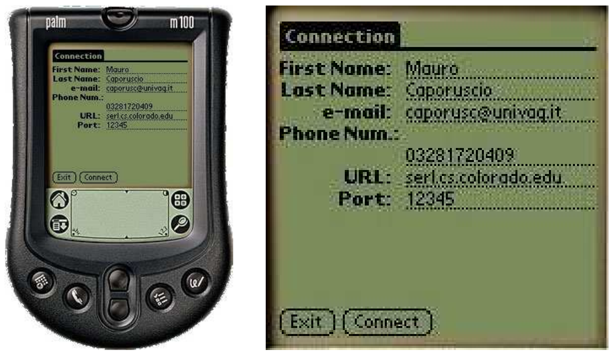
guaranteed delivery: Delivery of every message should be guaranteed.

message confidentiality: Only the interested parties may read the messages.

message integrity: Nobody can modify the messages.

3.3.3 Implementation

As explained above, we have developed this system using *Java*^{TM} *2 Platform, Micro Edition* and we designed it to run on limited resource devices. The application is composed of two distinguished sub-components: *Sell* and *Buy*. Each one of these, has one specific GUI as well as a shared GUI called *Connect*.



Connection

First Name: Mauro
Last Name: Caporuscio
e-mail: caporusc@univaq.it
Phone Num.: 03281720409
URL: serl.cs.colorado.edu
Port: 12345

normal size zoom

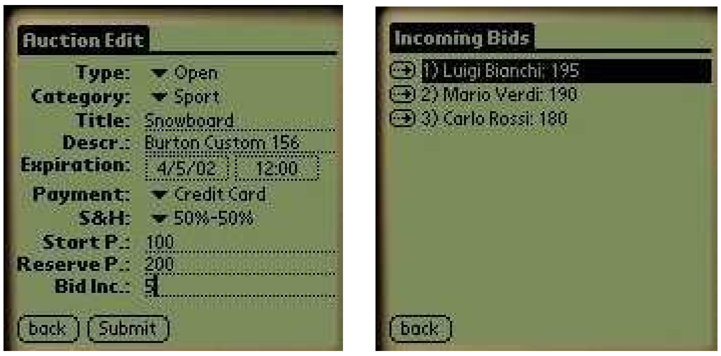
Figure 3.14: The *Connect* GUI.

Connect The *Connect* form implements is the first step to take part in the auction. It allows users to connect to a specified SIENA server located on a internet host. The user has to fill in the form (as show in Figure 3.14) in which he must write his personal data and the Internet address of the server. This information is used allow the buyer to get in touch with the *seller*.

This procedure simply stores user information in a data structure. The procedure also uses the ThinClient(*String* uri) class of the SIENA API to establish the connection with the SIENA master referred by uri (uri must have the *<schema>://hostname:port_number* format).

Sell This allows a seller to advertise an article available for sell, and starts a new auction for it. The procedure *Sell* is composed of two sub-procedures: *Edit Auction* and *Incoming Bids*.The first one allows the user to publish information about the auctioned item. The procedure gets the information from the GUI (see Figure 3.15.a) and creates a notification with them. This information are also used to create a unique *identifier* for this article. After the publication of this event, a subscription will be made, using the item *identifier* as filter. The purpose of this subscription is to catch the incoming bids related to this item. The *Incoming Bids* GUI (Figure 3.15.b), displays the received bid about the running Auction.

Internally (see Appendix A.1) a SIENA ThinClient, previously created by *Connect*, is used to send advertisements for new items, and to receive bids.



Auction Edit

Type:	Open
Category:	Sport
Title:	Snowboard
Descr.:	Burton Custom 156
Expiration:	4/5/02 12:00
Payment:	Credit Card
S&H:	50%-50%
Start P.:	100
Reserve P.:	200
Bid Inc.:	5

Incoming Bids

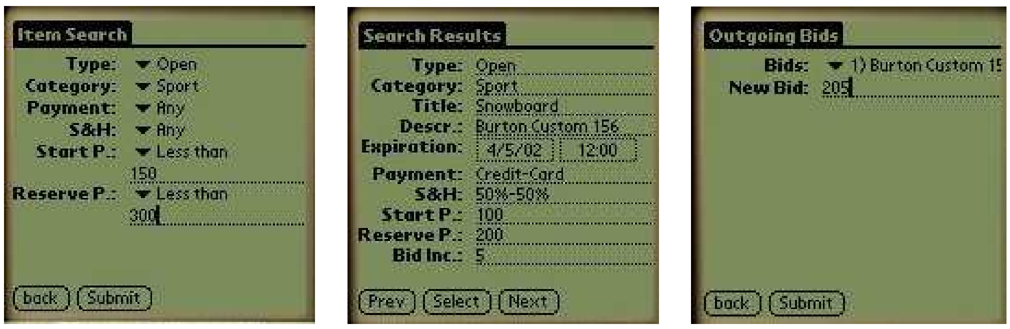
1) Luigi Bianchi: 195
2) Mario Verdi: 190
3) Carlo Rossi: 180

a) b)

Figure 3.15: The *Sell* GUI.

When the object *Bid_Update* receives an event representing a bid, it will update the appropriate data structure and the new bid will be displayed on the *Incoming Bids* form.

Buy After a connection has been established, in order to take part in an auction, the *Buyer* needs to know the *identifier* of a particular item. The *Item Search* form (see Figure 3.16.a) works like a search engine and enables the user to set filter constraints, and to creates the subscription corresponding to the given search criteria. When somebody publishes an event matched by the *Buyer* filter, this will be caught and stored in the appropriate data structure.



Item Search

Type:	Open
Category:	Sport
Payment:	Any
S&H:	Any
Start P.:	Less than 150
Reserve P.:	Less than 300

Search Results

Type:	Open
Category:	Sport
Title:	Snowboard
Descr.:	Burton Custom 156
Expiration:	4/5/02 12:00
Payment:	Credit-Card
S&H:	50%-50%
Start P.:	100
Reserve P.:	200
Bid Inc.:	5

Outgoing Bids

Bids:	1) Burton Custom 15
New Bid:	205

a) b) c)

Figure 3.16: The *Buy* GUI.

After an advertisement is stored, the buyer may use the *Search Results* form (Figure 3.16.b) to navigate through the data structure and choose the item he is interested. When an item has been selected, GUI *Outgoing bids* form will appear (refer to Figure 3.16.c) allowing the buyer to submit his bid for this article.

If during the Auction the user has been out-bid by another bidder, he will receive a notification and an *Alert* pop-up will appear. After receiving an out-bid alert, the buyer may open the *Outgoing bids* form and raise his bid.

Internally (refer to Appendix A.2) a siena.ThinClient() is used to submit bids and create search filters. Search results will be caught by the object *Search_Update* and stored in *Search_Results* object.

Observations

Since an advertise could be generated before the *buyer* has set up his filters or a bid could be submitted while the *seller* is momentary disconnected, advertisements, bids and searches are constructed with the *pair <request, offer>* paradigm explained in Section 3.2. As we showed in Section 3.2, the *pair <request, offer>* gives us an additional level of reliability using the functionalities of SIENA.

3.4 General Packet Radio Service

The *G*eneral *P*acket *R*adio *S*ervice (GPRS) is a new standard for wireless data that will be implemented in GSM and other mobile communication systems. The new technology provides effective utilization of the scarce radio resources and is therefore ideally suited for bursty packet transmissions. It enables instant and constant wireless access to IP based networks such as the public Internet and *L*ocal *A*rea *N*etworks (LAN).

GPRS facilitates new applications in wireless communication that have not been available previously, due to the limitations in GSM Circuit Switched Data (CSD) [28]. Through its packet switched (PS) nature, GPRS opens up for direct connectivity to the Internet with all its inherent user value. Examples of possible applications are Internet services such as Wireless Application Protocol (WAP), e-mail, web-browsing.

Increased capacity for data transmissions compared to GSM CSD is the obvious advantage that applies to the GPRS-system. Even so, much of the user-value lies in the possibility of obtaining immediate and constant connectivity to external networks such as Internet and Intranet, without repeatedly having to carry out a time-consuming setup procedure. Furthermore, the GPRS-system will incorporate new billing concepts, which includes paying for the volume of transmitted data, rather than the time of the data-connection as it is done today. This means that the user can stay connected and online to the networks even when nothing is transmitted, without paying excessive amounts for the duration of the data-connection.

3.4.1 Network Features of GPRS

GPRS as an overlay to the existing GSM-network may pose several paradigm-shifts to the and-users. In order to understand the inherent capacity issues of GPRS, some network features must be examined.

Packet switching

Most wireless data connections require the mobile user to go through a cumbersome setup procedure, resulting in a constant allocation of one timeslot during the entire length of the session. GPRS introduces fast access to networks through packed data technology. Rather than sending and receiving in a continuous stream as in the circuit switched (CS) world, data travels through routers for fast packet data transmission to and from the mobile subscribers. Packet switching means that GPRS radio resources are used only when are actually sending or receiving data [1]. Rather than dedicating a radio channel (timeslot) to a mobile data user for a fixed period time, the bursty nature of packet switched data allows the available radio channels for GPRS to be concurrently shared between several connection.

Coding Scheme	Data bits in radio block	Data rate per time slot kb/s on radio layer	Max data rate per8 timeslots kb/s
CS-1	181	9.05	72.4
CS-2	268	13.4	107.2
CS-3	312	15.6	128.8
CS-4	428	21.4	171.2

Table 3.2: Channel coding schemes parameters.

Channel coding schemes

Four different channel coding schemes are defined in the GPRS specifications [13]. Each coding scheme incorporates a different level of data integrity checks (error correction overhead) to data transmitted over the radio-interface. They are commonly labelled CS-1 to CS-4. Given fixed-channel capacity constraints, there is an inverse relation between the amount of actual data that can be transmitted and the amount of data integrity assurance. Basically, the channel can either be used to transfer data itself or error checks on the respective data. The different error coding procedure form varying size of the radio blocks, which produces four progressive data rates as listed in Table 3.2. It must be clear that these data rates are only valid for the radio-layer, and the data rates on the application layer will be somewhat less due to packet-overhead.

The higher the data rates, the higher the required signal to noise ratio (SNR). In good channel conditions with high SNR, any of the four schemes could be used. In this case the channel coding schemes with the least channel protection (CS-4) will yield the highest throughput. When interference is high on the other hand, the coding scheme with the highest amount of channel protection will achieve the highest throughput (CS-1), due to its extensive error coding which causes fewer retransmissions.

Technical limitation to the theoretical capacity

Although the system is awaited with high expectations from manufacturers and operators, the actual take-up of GPRS usage among subscribers is still an open issue. As explained in the previous section, the maximum theoretical data rate of 171.2kbps require an optimal coding scheme (CS-4). As such, the maximum speeds must be checked against the actual constraints in the network and terminals. The reality is that mobile networks are always likely to have lower transmissions speeds than fixed networks.

The increased data rates of GPRS are as result of two major aspects of the GPRS-system: improved coding schemes and the support of multiple timeslots. However, three main aspects prevent a user from ever achieving the maximum theoretical speed, namely the allocation of timeslots, restrictions in the terminals, as well as the actual availability of coding schemes.

Allocation of timeslots Because GPRS and GSM use the same radio resources, it is unlikely that a network operator would ever assign all eight timeslots to GPRS-traffic, since voice still will be a dominant service. In fact, how to allocate the timeslots to GPRS and GSM is supposedly an open issue among the operators. It seems clear, however, that GSM-traffic will have precedence over GPRS-traffic. Since GSM-traffic has precedence, GPRS-traffic will be offered a varying amount of capacity. The available timeslots will in turn be divided between all GPRS-users an the carrier at the given time. It should also be noted that among the carriers of one base station there will always be at least one signalling channel (mapped to the same amount of timeslots). The number of signalling channels depend on the number of carriers as well as the particular network environment.

Restrictions in terminals To take advantage of higher data transmission speed the GPRS-terminals will have to support several multiple timeslots simultaneously. In fact, in able to send and receive the theoretical maximum of 171.2 kbps the terminal must incorporate transmission and reception of timeslots (in both the downlink and uplink). This requires considerable amounts of processing and transceiver power in the terminal, adding great complexity to such a small device.

In reality, terminal manufactures are indicating that they will support a limited number of multislot classes, at least in the first stage of GPRS-terminal evolution. According to the representatives from the manufacturers, the terminals will initially support 1 timeslot uplink and 3 timeslots downlinks. Whether the evolution continues to improve further is not clear, but it is supposedly difficult to produce terminals that incorporate more than 4 timeslots in either direction.

3.4.2 GPRS System Architecture

As mentioned previously, the GPRS-system is built upon the existing GSM-infrastructure. So to enable GPRS, mobile network operators merely need to upgrade their GSM-infrastructure by introducing three new GPRS-elements, as well as updating a few of the existing GSM-nodes. Most importantly, this upgrade includes the GPRS Service Nodes (GSN), specifically the Serving GSN (SGSN) and the Gateway GSN (GGSN), but the upgrade also includes a new Border Gateway (BG) that provides access to other GPRS networks through a firewall. All new elements in GPRS system-architecture are illustrated in Figure 3.17.

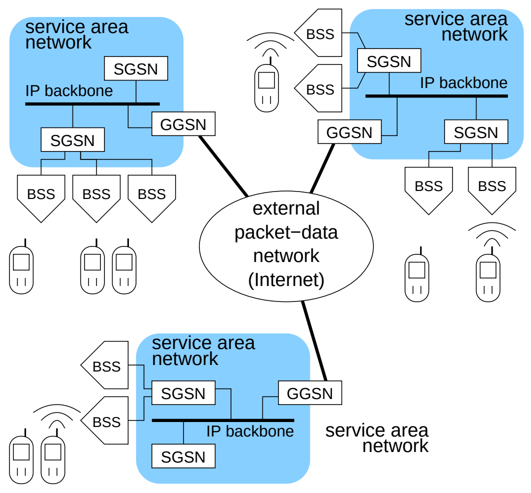


Figure 3.17: GPRS system architecture

Mobile Station (MS)

The Mobile Station (MS) is a combination of the Mobile Terminal (MT) and the Terminal Equipment (TE). It is important to be aware of that the MT and TE could be in the same device (such as a smartphone) or in separate devices like a regular GPRS-phone connected to a handheld computer or a laptop.

- The *Terminal Equipment* is the computer terminal that sends and receives end-user packet data.

- The *Mobile Terminal* communicates with the TE through cable or wireless technologies such as IrDA or Bluetooth. Over the air-link the MT communicates with the BTS. In order to be GPRS-capable, the MT must be equipped with specific software and hardware for the GPRS-system.

Mobile Stations developed for the GPRS-system will be differentiated in terms of their specific MS- and Multislot-class. The purpose of this definition is to enable the different needs of the various markets to be satisfied by a number of different MS types with distinct capabilities.

GPRS MS Class A Supports simultaneous attach, simultaneous activation, simultaneous monitoring, simultaneous invocation and simultaneous traffic. This means that the mobile user can simultaneously receive and transmit calls on the GPRS PS system and the GSM CS system. In order for GPRS and GSM to take advantage of the transceiver capacity at the same time, a minimum of one timeslot must be available to both services when required.

GPRS MS Class B Supports both GPRS and GSM connectivity, but the class B mobile cannot transmit and receive in GSM and GPRS mode simultaneously. However, signalling such "attach" and "activation" can be simultaneous. This means that a GPRS connection shall not be cleared down (deactivated), due to invocation of GSM traffic. The selection of the appropriate service is performed automatically (i.e. an active GPRS virtual connection is put on hold, if the user accepts an incoming CS call or establishes an outgoing CS call.It is worth noticing that precaution is needed when interrupting applications running over the GPRS-network. For instance, if the user establishes a CS session during an ongoing and time-consuming file transfer, the GPRS connection may abort due to a timeout.

GPRS MS Class C Supports both GPRS and GSM connectivity, but can only transmit and receive in one service at time. Furthermore, no simultaneous "attach" and "activation" is possible. The status of non-active service is always "detached" and the desired type of service is selected manually by the user.

Base Station Subsystem (BSS)

The Base Station Subsystem (BSS) consist of Base Station Controller (BSC) and Base Transceiver Station (BTS). All radio signals are transmitted and received by BSS, making it a shared resource between the CS GSM system and GPRS system. Specifically, a BSS upgraded for GPRS systems is provided with functionality adapted to a packet data. This includes packet data handling, GPRS information broadcast, resource administration, as well as new interfacing to the SSGN node.

Base Transceiver Station It is basically the receiving and transmitting facilities, including antennas and all the signalling related to the radio interface. When radio signals are received, the BTS separates GSM circuit switched data/voice from GPRS packet data and forwards both categories to the Base Station Controller (BSC) using standard GSM protocols for compatibility.

Base Station Controller Generically, the BSC has functionality to set up, supervise and disconnect CS and PS connections. These connections go to and from the BTSs on the radio side, as well as to and from one SGSN on the core network side. To manage this the BSC consists of a high capacity switch that provides functions such as hand-over cell configuration data and channel assignment.

Serving GPRS Support Nodes (SGSN)

The SGSN forwards incoming and outgoing IP packets addressed to and for a mobile station. It serves all GPRS-subscribers that are located and attached within the geographical SGSN service area. A subscriber may be served by any SGSN in the GPRS-network depending on location. The traffic is routed from the SGSN to BSC, via the BTS to the mobile station.

Gateway GPRS Support Nodes (SGSN)

Most importantly, the GGSN provides the interface towards the external IP packet networks. Actually, from the external IP network's point of view, the GGSN acts as a router for the IP-addresses of all subscribers served by the GPRS-network. To make this possible the GGSN exchanges routing information with the external networks and sets up connection towards external networks. Similar to the SGSN, the GGSN deals with session management, specifically the connection towards the external networks. Also, as many SGSN can connect to one GGSN, it has associate subscribers to the right SGSN.

3.4.3 GPRS Protocol Stack

The GPRS data communication architecture is based on the physical-layer of GSM [13]. It will continue to support the well-known principle of protocol layering according to the Open System Interconnection (OSI) communication architecture. The GPRS-system distinguishes between two protocols planes [1]:

- The transmission plane covers the protocols for the transmission of user information and the associated control procedures like flow control and error handling.

- The signalling plane consist of protocols that control and support the user transmission. GPRS-relevant functions in the signalling plane are connection control, routing and mobility management.

Transmission plane

The Transmission Plane, as show in Figure 3.18, illustrates the protocol layers of GPRS as well as the Internet data network.

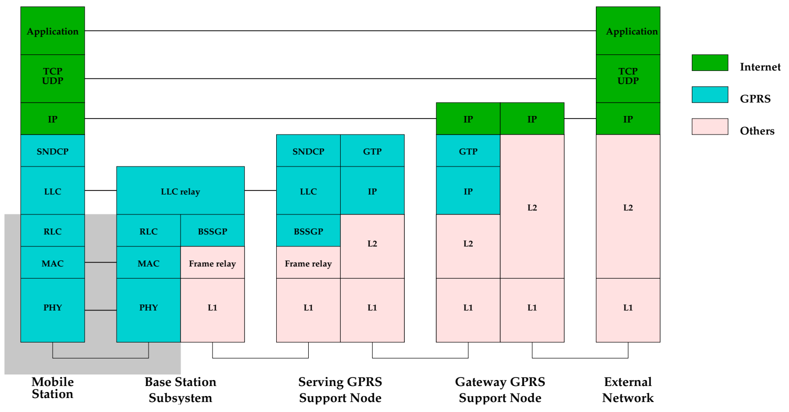


Figure 3.18: GPRS protocol stack

Application Layer The application layer is very broad in the sense that it incorporates several sublayers of functionality. It contains the logic needed to support various user applications. For each type of application, different protocols are needed that specifically manage the application sessions as well as the presentation of user data. These protocols are specific to the software and have no connection to the GPRS-architecture.

TCP/UDP The transport layer includes mechanisms for the exchange of user data on the end-to-end connection, which are essentially independent to the nature of the application. There exist two vastly different transport protocols, namely Transmission Control Protocol (TCP) and User Data Protocol (UDP) [8]:

- TCP providing a reliable data flow between two hosts.

- UDP instead provides a simple service to the application layer without reliability.

IP/X.25 The transport-layer may be carried on the network level by two types of Packet Data Protocols (PDPs), the Internet Protocol (IP) or the X.25-protocol.

IP user-addresses are located by (or via) the GGSN, but the pool of addresses are not necessary located there. It could be an external network such as an Internet Service Provider (ISP) or a corporate Local Area Network (LAN) that actually hand out the IP-addresses. Each external network has its own, unique, access point in the GGSN, containing functionality for handling network access and IP-address assignment.

SNDCP The Subnetwork Dependent Convergence Protocol (SNDCP) maps network-level characteristics onto the underlying radio-layers. This enables both IP and X.25 to be carried on top of the SNDCP-layer.

LLC The Logical Link Control (LLC) layer provides a highly reliable logical link. LLC shall be independent of the underlying radio interface protocols in order to allow introduction of alternative radio solution with minimum changes to the GPRS internal network.

RLC and MAC The Radio Link Control (RLC) and the Medium Access Control (MAC) are considered to be part of same layer. The RLC deals with segmentation of LLC data-packets into RLC data blocks. This RLC data block is given a MAC header and a Block Check Sequence (BCS) to form a radio block.

The Physical Radio-Interface The physical radio interface includes procedures for GPRS when it comes to channel coding, cell re-selection procedures and power regulation [13]. This layer also deals with frequency hopping and signal-modulation, improving the signal to noise ratio (SNR) through interface and frequency diversity.

BSSGP and Frame Relay On the reliable interface between the BSC and SGSN, the Base Station Subsystem GPRS Protocol (BSSGP) transmits packets and routing-information. To make the interface open it is standardized through Frame Relay (FR) [14]. The frame relay communications standard enables high data rate.

GSN interconnection Between the GSNs, the GPRS Tunnel Protocol (GTP) tunnels the PDUs through the GPRS backbone network by adding routing information [15]. Below the GTP, the usual TCP/UDP and IP/X.25 are used as transport and network layer protocols. The latter combination of protocols will be most common on reliable and over-dimensioned connection of the GPRS-backbone network. Ethernet, Integrated Service Digital Network (ISDN) and Asynchronous Transfer Mode (ATM) base protocols may be used below IP depending on the operators network architecture [1].

3.4.4 Signalling Plane

The signalling plane of the GPRS-system consists of protocols for control and support of the transmission plane functions [12]. This includes:

- controlling the GPRS network access connections, such as attaching to add detaching from GPRS network;
- controlling the attributes of an established network access connection, such as activation of a Packet Data Protocol (PDP) address;
- controlling the routing path of a network connection, in order to support user mobility
- controlling the assignment of network resources.

3.4.5 Survey of GPRS Tools

In this section we briefly survey several GPRS tools. The tools range from those for doing network planning to those for doing performance evaluations.

They also range from those developed by commercial vendors of GPRS technology to those developed by research institutions. While this survey is almost certainly not complete, the tools are all those for which we were able to obtain some amount of documentation.

Nokia NetAct™ Planner

Nokia's NetAct Planner [30] is an integrated set of tools for planning radio-based voice and data networks, including those based on GPRS technology. The tools allow one to "plan" in the sense of designing how the network will be deployed to satisfy usage and physical constraints. For example, there is a tool called the Rollout Planner that supports the process of site acquisition and project tracking. Another tool is the Transmission Planner, which supports the planning of the transmission and datacom network, including dimensioning and network architecture comparisons. A third tool supports an analysis of the placement and strength of microwave links.

Motorola GPRS Emulator

Motorola's GPRS emulator [27] is designed to help developers understand how their applications can be expected to behave over a typical GPRS connection. The emulator runs on a standalone Linux computer, with application clients and servers connected to that computer over a normal IP link. In essence, the standalone computer acts as a monolithic GPRS network. The emulator provides communication effects that reflect the performance of client/server interaction over the GPRS network under a variety of conditions, including normal loads, heavy ("busy hour") loads, and both short and long interruptions in signals.

Ericsson GATE II

Ericsson's GATE II [11] is another Linux-based emulator of a GPRS network. It emulates typical properties of a GPRS network, including varying bandwidths, loads, latencies, and radio conditions. The emulator is made available in a rather unusual way: Rather than being available for installation and use in the evaluator's environment, it is provided as a service to which one brings an application for evaluation. The evaluation itself is carried out by trained personnel at designated service centers.

University of Helsinki Seawind

In cooperation with Nokia Mobile Phone and Sonera Corporation, the University of Helsinki has developed Seawind [24], a Linux-based emulator of wireless networks. The emulator can be used to study network flow and congestion control, as well as other properties of an application communicating over a GPRS network. Like the Motorola GPRS emulator, it based on the use of a normal wireline local-area network. Link characteristics are emulated by delaying, dropping, and modifying the flow of packets according to a set of simulation parameters.

Network Simulator

NS-2 (Network Simulator) [16] is a general-purpose discrete event simulator for networks. The architecture of the simulator is designed to allow the specifics of a given network to be provided as a pluggable module. Recently, a module for simulating a GPRS network has become available [23], but we have not yet had an opportunity to fully study its capabilities. What we do understand at this point is that it is more suited to studying the internal behavior of the GPRS network than it is to studying the interaction of an application with the network.

Selecting a tool

In order to carry out our evaluation, we needed to select from among the available GPRS tools. In a sense, our choice was easy. The Nokia NetAct Planner is targeted at network planning, not performance evaluation. The Motorola emulator, while it appears extremely well suited for our evaluation, is simply not yet available. The Ericsson GATE II emulator might also be suitable, but the fact that it is available only as a second-hand service makes it very inconvenient to iteratively develop experiments.

We selected Seawind because of its combination of reasonable functionality and immediate availability. Nevertheless, as we detail in the next section, Seawind is limiting in the kind of information that we can gather, specifically in regard to the effect of deploying and operating *S*IENA servers in the GPRS network. NS-2 might well be an alternative worth exploring in the future, but it too has its limitations. In fact, Seawind and NS-2 appear to be complementary, since Seawind concentrates on the interaction of an application with the (monolithic) network, while NS-2 combined with the GPRS module concentrates on the performance of the network itself.

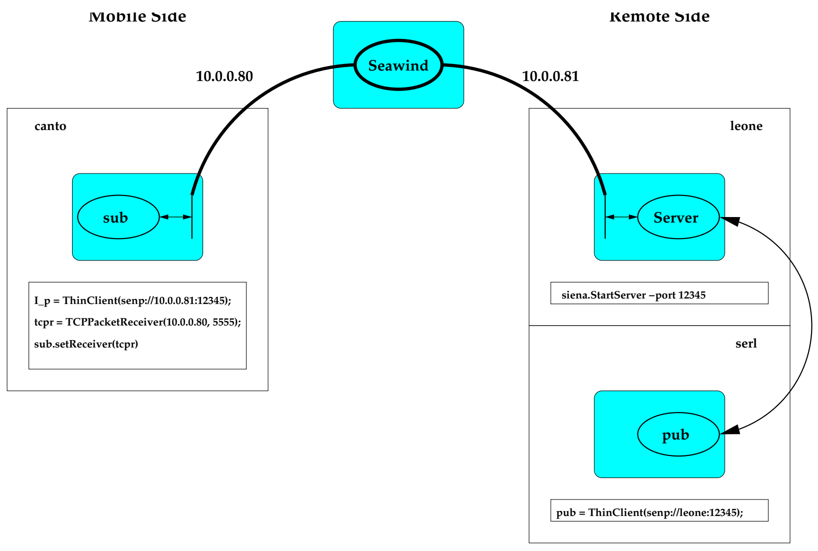
3.5 Experimentation

In this section, we will explain our experimentation and the results we obtained. We imagined a scenario in which users engage an auction using wireless devices (as explained in previous sections). This means that *buyers* and *sellers* are using an *Auction System* application installed on mobile devices and a GPRS Network as wireless link to connect their clients to an Internet host.

Since the impossibility to use a real GPRS network, and thus real mobile devices, we simulated this scenario using the Wireless Network Emulator SEAWIND [24] and a *J2ME Wireless Toolkit* [37].

Seawind emulates a point-to-point communication channel extending over a GPRS network. One end of the channel represents the mobile station, while the other endpoint represents the remote host. The mobile station and the remote host act as workload generators for the GPRS network. A *network protocol adapter* binds a workload generator at each endpoint of the emulated channel. The traffic produced by one workload generator is fed into the Seawind emulation process through one adapter. It is then processed by Seawind and passed on to the workload generator at the other end through the corresponding adapter. In processing through-traffic, Seawind emulates the behavior of a GPRS network according to its configuration parameters, thereby introducing characteristic delays, errors, and packet loss.

The current version of Seawind comes with a protocol adapter for the point-to-point protocol [32] that can be used to redirect IP traffic though Seawind. In practice, running Seawind amounts to running the main Seawind emulation process connected with two PPP adapters (running as separate processes). Each adapter creates a PPP interface configured with a given IP address, and with a "peer" address corresponding to the IP address of the other adapter. A workload generator is implemented by an ordinary network application, appropriately configured to direct some of its traffic to the IP address of one of the PPP adapters bound to Seawind. Seawind produces a traffic trace in *tcpdump* format [22] that can be analyzed by a variety tools [7].



```
I_p = ThinClient(senp://10.0.0.81:12345);
tcpr = TCPPacketReceiver(10.0.0.80, 5555);
sub.setReceiver(tcpr)
```

```
siena.StartServer –port 12345
```

```
pub = ThinClient(senp://leone:12345);
```

Figure 3.19: A SIENA mapping onto Seawind

Seawind has two significant limitations for the studies that we would like to perform. First, it models the GPRS network as a simple tunnel, capable only of moving data between a mobile station and the external packet-data network. In particular, Seawind does not model workload generators deployed *within* the GPRS network, which for us means that it cannot be used to study the performance of multiple, distributed SIENA routers. Second, Seawind focuses on a single pair of workload generators, not taking into account the interactions among multiple mobile stations sharing the same pool of radio links and base-station resources. While Seawind does in fact model the effect of other applications in the same cell, it does so by simulating generic, static "background" traffic. Such an approach captures some conflicts in resource allocation, but it does not reveal potential destructive dynamics resulting from the combination of interrelated applications.

Despite these two shortcomings, we can still extract some useful data using Seawind. For our experiments, we used a *S*IENA subscriber and a *S*IENA server as workload generators. The experiment setup is depicted in Figure 3.19. The subscriber plays the role of the mobile station. The server plays the role of the remote host. Notifications are produced by a publisher connected to the server directly on the remote host. Each experiment is defined by the sequence of subscriptions and notifications exchanged between subscriber and server, by the configuration of the connections between the subscriber and the server, and by the configuration of the GPRS network.

The workload that we used in our experiments consists of one subscription posted by the subscriber, followed by a number of matching notifications sent from the server to the subscriber.

*S*IENA uses a generic message-based communication mechanism that is realized in the current implementation by three specialized connectors. The configuration of the server-subscriber connection is obtained by selecting a specific connector. In particular, the choices include a UDP connector, a basic TCP connector, and what we refer to as a "keep-alive" TCP connector. A UDP connector sends messages through UDP packets, a basic TCP connector uses one TCP connection per message, and a keep-alive TCP connector attempts to use the same TCP connection for multiple messages.

For the configuration of the GPRS network, we experimented with a subset of the rich set of parameters offered by Seawind. In particular, in accordance with the GPRS CS-1 specification, we emulated a mobile station capable of using one uplink channel and up to three downlink channels. This setting is shown in the parameters of Table 3.3.

The *ms_max_rate* parameter defines the capabilities of the mobile station. A value of 3 selects the most advanced class of mobile stations, capable of

Parameter	**Uplink**	**Downlink**
ms_max_rate	3	3
available_rate	0-1	0-3
rate_base	9050 bps	9050 bps

Table 3.3: GPRS CS-1 simulation parameters

handling data communications (GPRS) and normal calls (GSM) at the same time. The *rate_base* is the bandwidth of an individual channel. *available_rate* determines the range of channels available to the mobile station. The actual number of channels allotted to the mobile station at any time depends on the presence of other GPRS or GSM users in the same cell.

In addition to the parameters of Table 3.3, which serve to characterize the connectivity of the mobile station to its base station, we must set other parameters that determine the quality of the communication channel. These parameters are listed in Table 3.4.

Parameter	**Value**
error_rate_type	BIT
error_probability	static 10^{-3} static 10^{-4}
error_handling	DELAY_ITERATE FORWARD DROP
error_delay_function	uniform distribution 40–50ms
delay_drop_threshold	static 10s

Table 3.4: GPRS error simulation parameters

The effect of noise is to introduce transmission errors or delays. Errors occur with a probability determined by the *error_rate_type* and *error_probability* parameters. In our experiments, errors are set to occur at the level of individual bits with a probability of 10^{-3} and 10^{-4}.

The *error_handling* parameter determines how the GPRS network handles transmission errors. With "DELAY_ITERATE" the network provides a reliable delivery service by simply forcing retransmission, which in turn introduces a delay for end-to-end communications. Alternative modes are "FORWARD", in which errors are simply ignored and passed on to higher levels in the communication stack, and "DROP", which causes the network to drop packets that contain errors. In the case of "DELAY_ITERATE", *error_delay_function* determines the interval before retransmission and *delay_drop_thresholds* defines an upper bound for the total retransmission delay, after which a packet is simply dropped.

3.6 Sample Results

This section presents some sample results that we were able to obtain using Seawind to evaluate the configuration described in the previous section. The primary goal of these experiments was to evaluate the impact of deploying SIENA onto the wireless GPRS network. We did this from two different perspectives. The first was to gather data characterizing the performance of the three different low-level connectors (UDP, TCP, and keep-alive TCP) on the wireless network. The second was to compare these results with baseline data collected on a local-area, wired network. By doing this we should get an initial indication of whether a seamless integration of wired and wireless communication is feasible for a publish/subscribe communication service.

error_probability = 10^{-3}						
	DELAY_ITERATE		FORWARD		DROP	
	notif.	*IP packets*	*notif.*	*IP packets*	*notif.*	*IP packets*
Keep Alive	79	875	17	350	8	285
TCP	100	1173	64	1205	62	1571
UDP	79	82	73	78	66	97
error_probability = 10^{-4}						
	DELAY_ITERATE		FORWARD		DROP	
	notif.	*IP packets*	*notif.*	*IP packets*	*notif.*	*IP packets*
Keep Alive	82	855	72	458	70	443
TCP	100	1153	100	1156	99	1152
UDP	100	106	84	95	76	91

Table 3.5: SIENA behavior in the wireless GPRS network.

Table 3.5 shows the network usage corresponding to the three low-level connectors under two different error probabilities. In essence, this table captures data on the cross product of the parameter values of Table 3.4. We collected counts of application-level notifications received by the subscriber and the resulting counts of IP packets. The counts shown in each cell are the average taken from five runs of the simulation. In all cases, there were 100 notifications published. The data give an indication of the circumstances that lead to different notification loss rates. For example, as we would expect, the highest loss rate occurs at an error probability of 10^{-3} under the DROP error-handling mode. We can also see that the keep-alive connector is the most sensitive to increasing error rates and decreasing quality of error-handling service.

	notif.	**IP packets**
Keep Alive	100	437
TCP	100	828
UDP	100	102

Table 3.6: Siena behavior in a local-area, wired network.

Table 3.6 shows the baseline behavior obtained by running the application on a local-area, wired network. The data characterize the relative overhead of each of the low-level connector protocols. For instance, UDP encounters no overhead (The two extra packets are used to carry the subscription and unsubscription messages). On the other hand, approximately eight packets, on average, are required by TCP to deliver a single notification. We can compare the baseline overhead to that experienced in the wireless network. The overhead of TCP in the wireless case is approximately twelve packets per notification, considerably higher than in the local-area, wired case.

Chapter 4

Mobility Support in SIENA

As we explained in Section 2.1 while host mobility is concerned with the physical movement of hosts, *Code Mobility* is the ability to transfer data and/or code from one host to another by using a network. *Data mobility* is a very common mechanism and is often used to exchange or spread information among different hosts distributed on a network. At a level above this, *code mobility* allows the migration of executable code. Data mobility can be

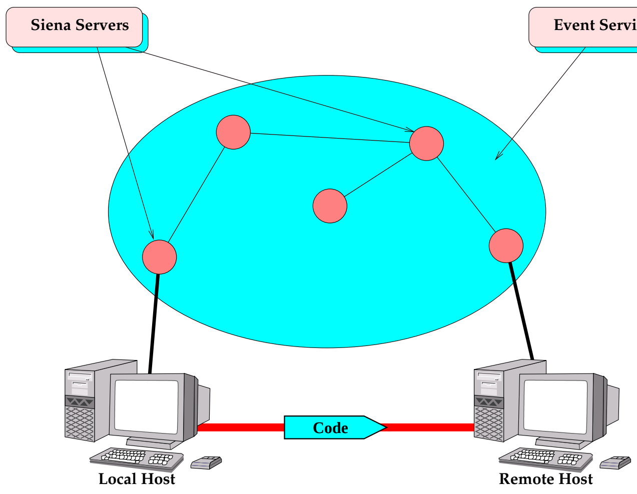


Figure 4.1: A code fragment moves from host to host and changes its server master after the motion.

achieved with simple transport protocols, or with higher-lever protocols such as RPC by passing parameters to a remote procedure. A simple example of code mobility consist in a WEB browsers loading an applet from a remote site. For instance, data e code mobility in Java are supported through object serialization and class loading. The status of objects can be serialized and transfered from one host to another while the class loading strategies can vary, depending on the application. The class of the moved object can migrate onto the new host or it can be fetched from a remote server.

Two more sophisticated mobile code paradigms are classified as *remote evaluation* and *mobile agents* [17]. Remote evaluation allows the proactive shipping of code to a remote host to be executed [33]. Mobile agents [39] are autonomous objects carrying their state and code that actively move across the network. Agent mobility requires the migration of both code and state of the agent at the same time and they can move actively performing tasks on be half of users. In this scenario, it is reasonable to think about mobile agents that use a publish/subscribe system to communicate with each other and with other non-mobile components.

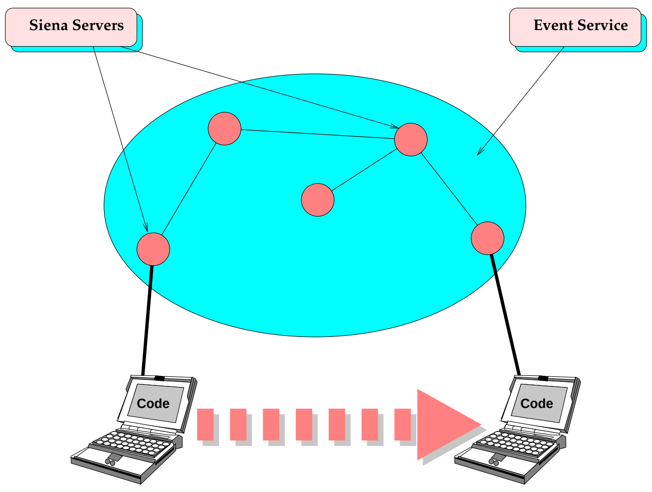


Figure 4.2: Code moves together with its host and changes its server after the motion.

This introduces some problems that must be solved. In fact, in a publish/subscribe system such as SIENA the state of a mobile agent is non completely stored within the agent, but it is partially maintained by the event-service. Specifically subscriptions are maintained by SIENA-servers, and possibly spread across the network (refer to Section 2.2.3). Therefore, an agent that moves to a new location must inform SIENA about its movement, to allows an appropriate re-routing of notifications of interest. In this chapter we will describe the solutions we propose to manage the problem arising with the mobility of clients in SIENA.

We suppose to have scenarios (depicted in Figures 4.1, and 4.2) in which a client wants to switch, after its migration, from a *local master server* to a *remote master server*. Notice that from the point of view of SIENA, the way in which a client performs the movement is completely transparent. In fact, whether the client moves together with its host or it migrates using some mobile code technology, the problems related to its disconnection, and reconnection remain the same.

Since in a publish/subscribe architecture part of the client's status is stored in its access-point (such as its subscriptions or its location) we need some new operations to manage the switching. In fact, during the switching, a client could lose some events or get duplicates.

To avoid these problems we propose two solutions that differ from each other in the *quality of service* they offer. The first solution favors speed over quality of service, while the second one offers better service guarantees at the cost of a slower, and more complex process. In next Sections we will

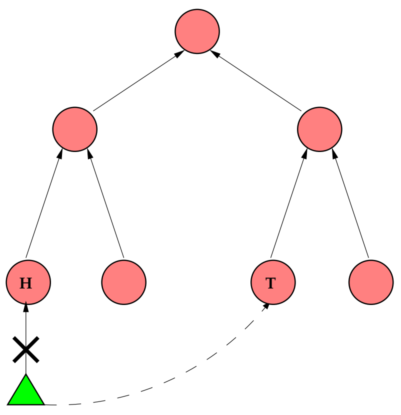


Figure 4.3: Access-points switching actions.

describe which solutions we developed, and how they work. As first instance we examine the simple case in which a fixed client wants to switch from a *local master server* to a *remote master server* (see Figure 4.3). Then we will extend this particular case in order to manage the more general case in which a client changes its master server at the end of the motion.

4.1 Mobile Dispatcher

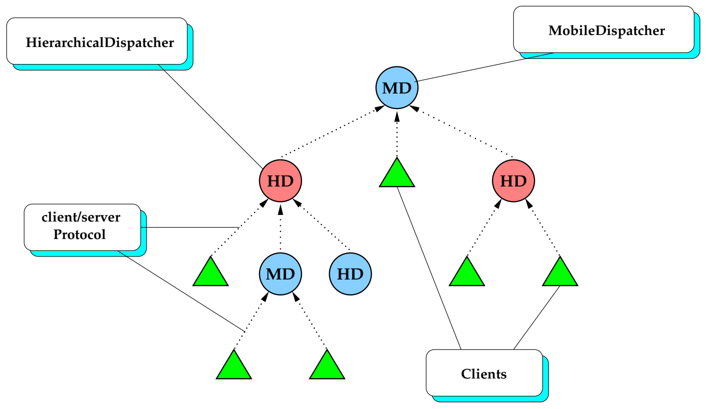


Figure 4.4: *HierarchicalDispatcher* and *MobileDispatcher* work together.

We created a new kind of dispatcher, called *Mobile Dispatcher* with the ability to manage client mobility. This dispatcher is based on the *HierarchicalDispatcher* class provided by SIENA, and adds new features oriented to mobility management. It is possible to use *MobileDispatcher* in combination with old dispatchers in order to create hybrid networks. Of course we may have networks made by only *MobileDispatchers* in which fixed clients still can use the old services. In this case every server is able to achieve clients mobility.

4.2 Notification Persistence Service

The first feature we added in *MobileDispatcher* is the *persistence* of notifications. This enables a client to be disconnected for a while, and to receive all events it is interested in when it will reconnect (see Figure 4.5). In order to do this, we developed two new actions. The first one, called **moveOutMaster**(), puts the client in a *suspended* mode, and asks the master server to store

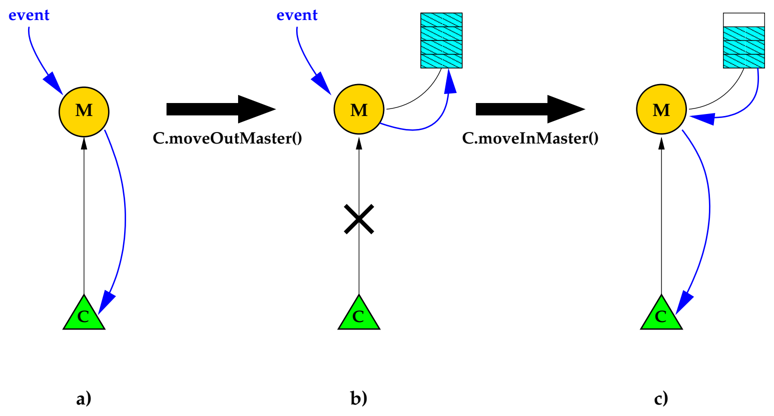


Figure 4.5: Notification persistence service.

all events that match its filters. After the disconnected period, the client will use the dual operation of **moveOutMaster**() (called **moveInMaster**()) to reconnect. This operation allows the server to put the client back in *active* mode, and to dispatch all stored events to it. This feature results useful if client *leaves*, and *returns* to the same master server. Of course, the client could change its position during the movement, but its master server will remain the same. To communicate this change of position to the server, the client can use the **setReceiver**(*new_coord*) method provided by SIENA. We extend this action, by creating a new one called **moveInMaster**(*new_coord*) that sets up new position, and then acts as **moveInMaster**().

4.2.1 Implementation

The implementation of **moveOutMaster**() is really simple: when a remote client invokes the **moveOutMaster**(), a SENP.MVL massage, formatted in according to SIENA protocol [1] [3], will be sent to the master server to inform it about the Client request. Then, the server will search the client's id into its "contacts" switching it to a *movingON* mode (see Figure 4.5.a). Since this moment, every notification addressed to this client, will be stored in a private queue (see Figure 4.5.b).

[1]A SIENA server uses a number of codes to identify requests from clients or other servers. Those requests are defined by the SIENA Event Notification Protocol (SENP). For example a publication is defined by the SENP.PUB code.

To return active, the client has to call the function **moveInMaster**(). This will send a SENP.MVI message to the master server that puts the client in *movingOFF* mode, and delivers every stored message to it (see Figure 4.5.c). If client position is changed during the disconnection periods (client could be moved to another host), but it would still use the same server, client may invoke **moveInMaster**(*new_coord*) where *new_coord* is a PacketReceiver class (for more information refer to the SIENA API [3]). **moveInMaster**(*new_coord*) will set up the new position in the server, by sending a SENP.MAP request, and the it will act as the **moveInMaster**(). For more implementation details refer to Appendix C.1, and C.2.

4.3 Event Re-routing

It is reasonable to assume that a mobile client wants to move from an actual master server to another one. In fact, during the motion, it may decide to switch to a new access-point considered better than the old. In this scenario, the *notification persistence service* represent only the first step of a new set of possible solutions. As we explained above, since part of the client state is stored inside the *event service*, if a client moved from a master server to another, it must inform the *event service* in order to modify its state, and configuration. These include events routing path, client's position, and subscriptions. In fact, if a client changed a server without inform the *event service*, the latter will still send the notification at the old address.

In the current implementation of SIENA [3] there is an available action, called **setMaster**(*new_master*), that allows a client to change its master server. The single operations which it performs are:

1. Unsubscribe all filters from *old_master*
2. Disconnect from *old_master*
3. Connect to *new_master*
4. Subscribe all filters

But, since **setMaster**(*new_master*) is not an atomic action, some problems may occur. In fact, some *events* of interest for the client may be generated while the client is still disconnected, or before it re-subscribes its filters. This means that the client will lose some *notifications*.

In order to solve these problems, our idea is to store the client's notifications, and subscriptions on the old master server throughout the switching procedure. During this time the status of the client should be *suspended*, so that the notification persistence service is active, on both old and new master server. This prevents lost events, but may create redundancy. In fact the same events could be stored in both master servers. Moreover, after the motion the client should also be able to receive all notifications stored on the old master server. In the following Section we detail our solution.

4.4 Event Downloading

One obvious solution is to download the events stored on the old master server (called H) from the new one (called T), and then send them to the client (called C). To do this, we add a new action, called **moveInMaster**(*dest*), which connects the client to the new server, subscribes all client's filter, downloads stored events from the old master server, and finally it disconnects the client from the server H (refer to Figure 4.6). Of course, this procedure requires some synchronization between master servers to avoid losing or replicating events.

In order to solve these problems, we implemented the **moveInMaster**(T) procedure with the following sequence of operations. Notice that the procedure requires that the client calls the **moveOutMaster**() method before leaving the old server H:

1. Connect(T);
2. Store(T);
3. Subscribe(*filters*, *id*);

4. Download(H, T);
   MergeEvents();

5. Disconnect(H);

where *filters* are filters subscribed by the client, and *id* is the identity of the client. After downloading the events (4), T has two queues of stored

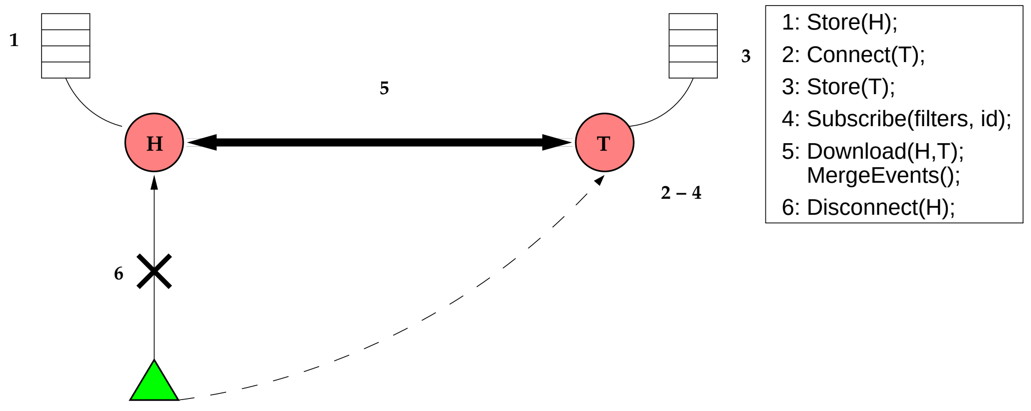


Figure 4.6: T Downloads the events stored in H

events, many of which could be replications of the same notification. In order to remove duplicated events, T will merge the two queues, and sends the resulting queue to the client. The merge operation uses a simple comparison function based on the exact match of all attributes and values.

4.4.1 Implementation

We assume the client has invoked the **moveOutMaster**() method before it starts to move. When the client decides to return to an active status, thereby changes its master server, the client may call the **moveInMaster**(*uri*) (where *uri* is the address of the new server). This function will perform the following actions:

1. The client C sends a connection request (SENP.WHO) to the new master server T referred to by the *uri* parameter. T receives the request and creates a new MobileSubscriber in its *contacts*.

2. C sends a MoveOutMaster request (SENP.MVL) to the new master T. T receives it and puts C in *movingON* mode.

3. C re-subscribes all its *filters* to the new master server T.

4. C sends an Event-download request (SENP.DWL) to T. T receives it and sends an upload request (SENP.UPL) to H. H receives it and sends all events it has stored (using SENP.PRV messages) to the new master server T. When the download is completed, the server T sends a disconnect request (SENP.BYE using the C's *id*) to H in order to cancel all of C subscriptions from the contacts of H. Thus, the new master server T merges the downloaded events with the locally stored events and sends the result to the client C. Finally, T put the C in the *movingOFF* mode.

If also the position of the client C is changed during the disconnection period (it could be moved to another host), C may invoke **moveInMaster**(*new_coord*, *uri*) where *new_coord* is a PacketReceiver object (see the *S*IENA API [3]). This operation will set up the new position of C in the master T sending a SENP.MAP request, and then it will act as the **moveInMaster**(*uri*). For more implementation details refer to Appendix C.2.

4.4.2 Observations

This solution seems solve every problem, but since *S*IENA delivers all filters subscribed by C throughout the master server's hierarchy (refer to Section 2.2.3), there is a time gap between the filters subscription and the filters activation. This time gap may be long and, if during this time T downloads the client's events stored in H (and thus disconnects C from H), there is a high probability to lose events. In fact some notifications could be generated before the filters activate but after the events download. During this interval, the notifications cannot reach T. Furthermore, in H the events persistence service is no longer active and this implies that these notifications will be lost. The duration of the activation time may depend on a number of factors, such as network congestion, *S*IENA workload, and others out of our control.

4.5 Event Downloading With Path Test

In order to solve problems explained in the previous section, we need to synchronize the event downloading (and then the client's disconnection from the old master server) with the filters activation.

The main idea is to send a *ping* message (from the new master server T to the old server H) throughout the $\mathcal{S}$IENA network and wait for a *ping_ack* reply. When T catches the *ping_ack* message, T can download the events and disconnect the client C from H (the actions sequence is showed in Figure 4.7).

The only way to send a message through the $\mathcal{S}$IENA master server's hierarchy is to build the *ping* message as a notification. Of course H must be subscribed for the *ping* and T must subscribe a filter for the *ping_ack*.

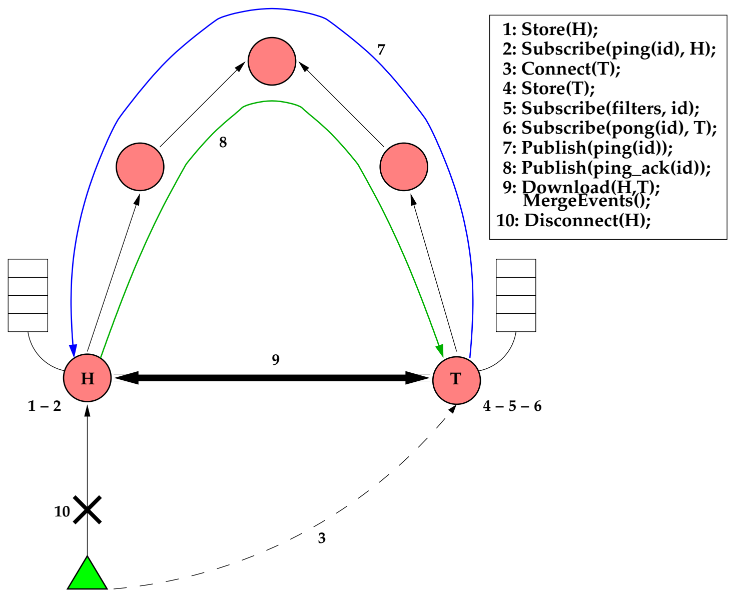


Figure 4.7: Download events stored on the old *server* with synchronization.

We implemented a new version of the **moveInMaster**(T) procedure with the following sequence of operations:

1. Connect(T);

2. Store(T);

3. Subscribe(*filters*, *id*);

4. Subscribe(*ping_ack(id)*, T);

5. Publish(*ping(id)*);

6. Download(H, T);
   MergeEvents();

7. Disconnect(H);

Also in this case the procedure requires that the client C calls the **moveOutMaster**() method before leaving the old master server H. Notice that in this case, the **moveOutMaster**() method must be modified in order to enable H to catch the *ping* message and reply to it with:

1. Store(H);

2. Subscribe(*ping(id)*, H);

3. Wait(*ping(id)*);

4. Publish(*ping_ack(id)*);

This solution, based on "*ping-pong*" synchronization, seems to secure us from losing packets. Of course, since we are talking about a scalable network (such as the Internet), we have to use the expression "highly probable" instead of "secure". In fact, *S*IENA is not reliable, and thus some packets could be lost along the path between two consecutive *S*IENA master servers.

The main idea behind this implementation of the event downloading is the following: In order to receive the *ping-acknowledgement*, T must be subscribed for the *ping_ack* event and the related filter must be active through the *S*IENA network. The idea is that T subscribes the filter for the *ping_ack*, then C re-subscribed all its filters. When T receives the *ping_ack* message, the others filters are also probably active. If this is true, T is able to catch all events in which C is interested and store them in the appropriate queue. Thus, T can download the events stored in H and can merge them with the events locally stored. Only at this point T can disconnect the client C from the old master server H and make C active. The synchronization time-steps performed by the download procedure are shown in Figure 4.8.

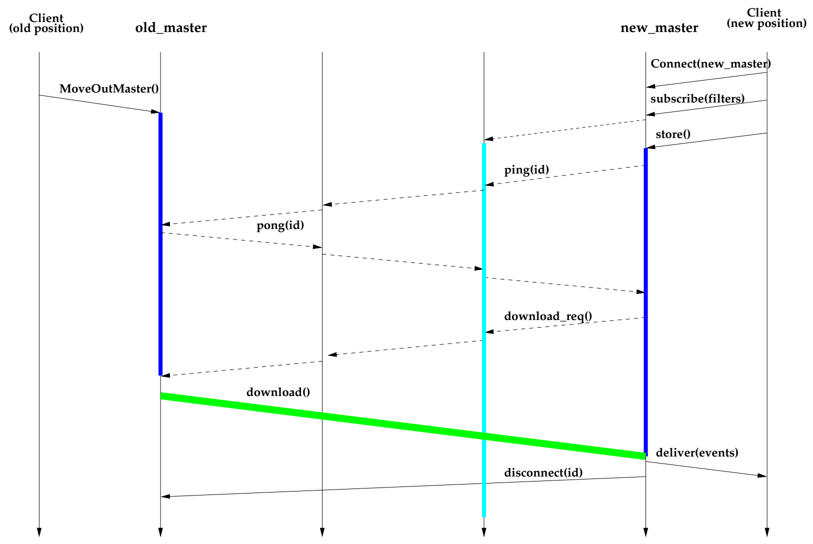


Figure 4.8: Events downloading: time-steps synchronization

4.5.1 Implementation

We assume the client has invoked the **moveOutMaster**(**true**) before it starts to move. The new parameter **true** forces the master server T to use the Hierarchical-path test before it downloads the events. When the client calls the **moveOutMaster**(**true**) function, the server H creates a listener that waits for a *ping* message.

When the client C decides to return to an active status, it may call the **moveOutMaster**(*uri*, **true**) (where *uri* is the address of the new master server and **true** means that the C wants to use the Hierarchical-path test). The **moveOutMaster**(*uri*, **true**) function will perform the following action:

1. Client C sends a connection request (SENP.WHO) to the new master server T referred to by the *uri* parameter. T receives it and creates a new MobileSubscriber in its *contacts*.

2. C sends a MoveOutMaster request (SENP.MVL) to T. The master server T receives it and puts C in *movingON* mode.

3. C re-subscribes all its *filters* to the new master server.

4. C sends an Event-download request (SENP.DWH) to the new master server T. T receives it, subscribes for the *ping_ack*, sends a *ping* message through the SIENA dispatcher hierarchy, and waits for a reply. When the server H catches the *ping*, H will reply with the *ping_ack* message. After T catches the *ping_ack* it will send an upload request (SENP.UPL) to the master server H. H receives it and will send all events it has stored (using SENP.PRV messages) to the master server T. When the download is finished, T sends a disconnect request (SENP.BYE using the C's *id*) to H in order to cancel all of the C's subscriptions from the contacts of H. Thus, T merges the downloaded events with the locally stored events and sends the result to the client C. Finally, the master server T puts C in the *movingOFF* mode.

In order to manage both a change of location and master server switching, we also added the function **moveInMaster**(*new_coord*, *uri*, **true**) where *new_coord* is a PacketReceiver class (refer to the SIENA API [3]). This operation will set up the new position in the master server, sending a SENP.MAP request, and then it will act as the **moveInMaster**(*uri*, **true**). For more implementation details refer to Appendix C.2.

4.6 Mobile Server Discovery

A usual problem to be solved in mobility management is how discover service servers during the motion. For instance, in wireless networks each server has a physical dedicated channel (typically a radio frequency) used for continually sending the server identifier and location. When a Mobile Station (MS) enters the zone served by a specific server, the MS catches this signal and MS can perform the server switching using information it has read. In the Internet environment, this solution is not applicable (see Figure 4.9) because a disconnected Mobile Agent does not have the possibility to receive any kind of message.

A simple solution we deployed is to offer a *server discovery* service to the client. The idea is that, before its disconnection, the client can ask

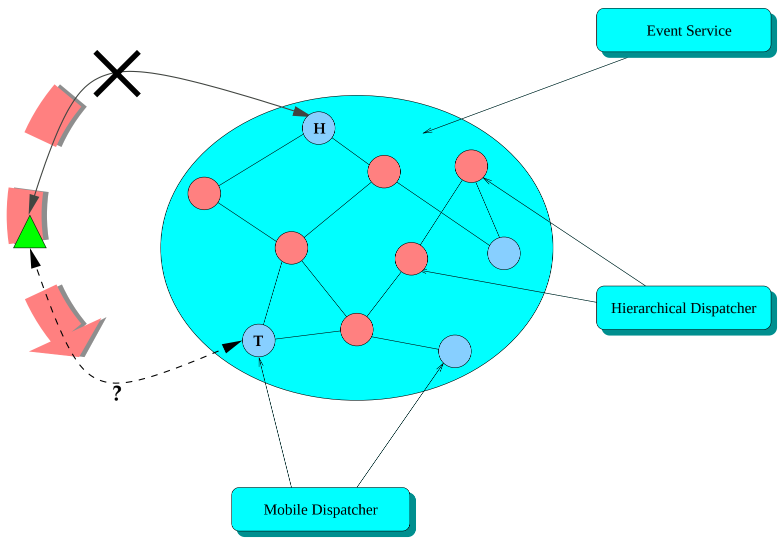


Figure 4.9: How Discover another *MobileDispatcher*

the SIENA network for a list of available hosts which are able to manage the client's mobility. This new functionality, invoked by **addMobileDispatcherFinder**(Notifiable n), sends a public message from the actual client's server through the network, and every MobileDispatcher will reply to it with their own location and information. Every reply packet, caught by the source master server, will be delivered to the interested client (refer to Figure 4.10).

Of course, the *server discovery* service may be stopped at any moment by the client by invoking the **removeMobileDispatcherFinder**(Notifiable n). After this operation, the mobile client has a list of available *MobileDispatchers* and it can choose one of them to reconnect itself after the motion.

4.7 Implementation

This service is implemented using the standard features offered by SIENA. In fact, when a SIENA mobile server is started, it simply creates a *Notifiable* object subscribed for the filter

$$f_s : service\_\_ = Mobile\_Server\_Request$$

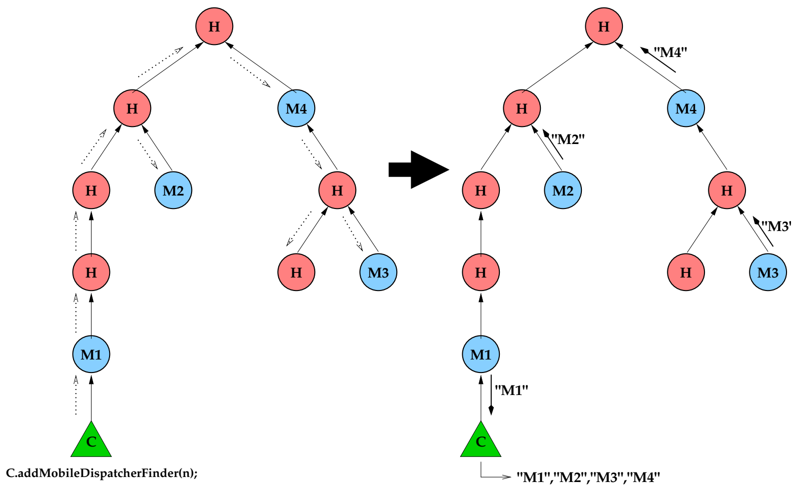


Figure 4.10: How Discover another *MobileDispatcher*

. This filter remains active during the life cycle of this master server.

When a client wants to know where the servers that offer the mobility service are located, it may invoke **addMobileDispatcherFinder**(*rec*), where *rec* is the the object that will receive the notifications. This function subscribes *rec* for

$$f_r : service\_\_ = Mobile\_Server\_Reply$$

and publishes the

$$e_s : service\_\_ = Mobile\_Server\_Request$$

event. the event e_s will match with f_s and, at this point, the master servers that caught this event will reply to it by generating the event

$$e_r : service\_\_ = Mobile\_Server\_Reply, uri\_\_ = localuri, info\_\_ = localinfo$$

where *localuri* is the address of the replyer's master server, and *localinfo* could represent some useful information about this server. This information may be specified when the server is starting up by setting the "-info *string*" option in the siena.StartMobile command. For more information refer to the

moveOutMaster()
moveOutMaster(boolean *QoS*)
moveInMaster()
moveInMaster(PacketReceiver *pr*)
moveInMaster(string *uri*)
moveInMaster(string *uri*, boolean *QoS*)
moveInMaster(PacketReceiver *pr*, string *uri*)
moveInMaster(PacketReceiver *pr*, string *uri*, boolean *QoS*)
addMobileDispatcherFinder(Notifiable *rec*)
removeMobileDispatcherFinder(Notifiable *rec*)

Table 4.1: Interface SIENA Mobility Support

SIENA implementation [3].

After that, a client may stop the search by calling the **removeMobileDispatcherFinder**(*rec*) method. This method will simply unsubscribe the *rec* object for the filter f_r.

4.8 Observations

In Table 4.8 there are listed the APIs we added to SIENA in order to support the client's mobility. Note that the parameter *QoS* represent the Quality of Service guaranteed by the referred functions. The default value of *QoS* is **false**. This means that the default kind of download does not test the hierarchical-path. If a client wants to use this specific downloading mode, it must invoke the relevant function with "*QoS*=**true**". Obviously, the client has to choose the same QoS for the **moveOutMaster** and **moveInMaster**. For example, if it called **moveOutMaster**(false) at the old site, it must invoke **moveInMaster**(pr, uri, false) from the new location.

Chapter 5

Conclusions

In Section 3.6 we have described our attempt at performance evaluations of a distributed application deployed over a wireless network. The application is characterized by the interaction of multiple clients residing at the periphery of the network, as well as by the need to deploy elements of the application deep into the network.

We evaluated the impact of deploying SIENA onto the wireless GPRS network from two different perspectives. The first was to gather data characterizing the performance of the three different low-level connectors (UDP, TCP, and keep-alive TCP) on the wireless network. The second was to compare these results with baseline data collected on a local-area, wired network. The data shown in Table 3.5 gives an indication of the circumstances that lead to different notification loss rates. For example, as we would expect, the highest loss rate occurs at an error probability of 10^{-3} under the DROP error-handling mode. We can also see that the keep-alive connector is the most sensitive to increasing error rates and decreasing quality of error-handling service. By comparing these results with the baseline overhead (showed in Table 3.6) one simple thing we can note is the high overhead of TCP. In fact in the wireless case it is approximately twelve packets per notification, considerably higher than in the local-area, wired case.

To our disappointment, we were not able to find tools capable of supporting a full evaluation of this application. We were limited to the narrow evaluation of a single client interacting across the network with a single server. Nevertheless, our experience should not be taken as a criticism of Seawind, the tool that we decided to use for our evaluation. In fact, we found Seawind to be a reasonable and useful tool for its purpose.

Clearly, a need exists for a different kind of tool for wireless-network performance evaluation. Before embarking on the development of such a tool ourselves, we first plan to study the capabilities of NS-2 and its GPRS module, which hold some promise for modeling and evaluating services deployed deeply into a wireless network. We might in fact be able to extend them to also allow modeling and evaluation of client interaction over the network.

As we explained in Section 4, our intent was to deploy a mobility support in the *S*IENA publish/subscribe middleware. This introduced some problems, such as messages persistence during the client motion, notification re-routing after the movement, events downloading from the new client's destination, that we have studied and solved. We extended the set of available operations in *S*IENA adding new actions specifically oriented to manage the mobility of the clients. These allow the client to relocate from host to host updating its information maintained by the *event-service*. The basic operations we developed are **moveOutMaster** and **moveInMaster**. **moveOutMaster** allows a client to declare its intention to move and causes the *event-service* to suspend the delivery of notifications to that client. Of course, all events addressed to this client will be stored by the *event-service*. When the client reaches its new location, it can use the **moveInMaster** operation to reconnect to the *event-service* and retrieve all notification stored while the client was disconnected (see Section 4.8). Finally, we also added some operations that allow the client to discover other mobility-service-enabled servers available throughout the network.

The API extension we presented in Section 4.8 provide specific services for mobile clients. However we do not consider them as a definitive solution, but rather as a basis for future works. In fact they allow us to perform additional case studies. One important aspect that we would like to study is the level of reliability provided by the new services. In particular, we would like to quantify the probability of losing or duplicated messages, or of changing the their ordering [26]. As another further development, we would like to perform additional tests which we believe are very important due the probabilistic nature of the errors affecting the system.

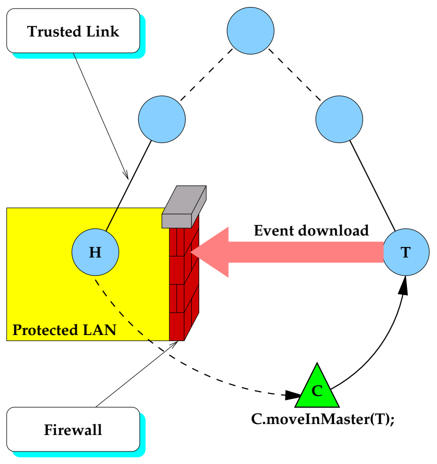


Figure 5.1: Dynamic reconfiguration using Mobile Support

It is also important to note that this solution could not work in the presence of network security constraint. In fact, the downloading process could be restricted in case in which one of two masters involved in the downloading process is located behind a network firewall (as in the scenario of Figure 5.1). We would have to study alternative solutions for cases such as this.

Since a component is unreachable during its disconnection, it cannot receive events and thus it cannot perform any kind of operation in replying to it. This may be an undesired behavior in presence of real-time constraints in the system.

Finally, since in SIENA clients and masters are built using the same class and architecture, a client can also act as server. This allows us to change master for dynamically reconfiguring the SIENA network (see Figure 5.2). In fact, even if a master is usually fixed in the network, we may use the mobile capabilities (such as **moveOutMaster**() and **moveInMaster**(*uri*)) to momentary disconnect the master, reconfigure the SIENA network topology and reconnect it to another master avoiding lost packets.

Next milestones in these directions should be to finish the testing phase

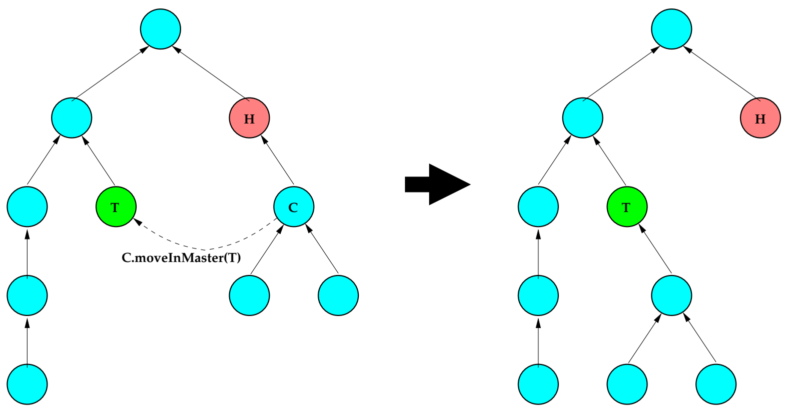


Figure 5.2: Dynamic reconfiguration using Mobile Support

and evaluate the results. Furthermore, we should understand if reliability is indeed a critical non-functional requirement in the context of mobility. After that, we would also like to experiment with mobile agents using the SIENA mobility support in order to study the impact of the publish/subscribe architecture in this context. Finally we would like to combine the mobility support we deployed in SIENA with the *host mobility* for example using an ad-hoc network [21]. In fact, since this is completely composed by mobile hosts, its topology (and then the relations between the masters which composed it) changes quickly over the time. In a situation like this, we imagine a SIENA MobileDispatcher running in every mobile hosts and using the mobile features to manage its relation to the other components (this is usually referred by the term *Context Management* [29]). It would be useful to validate our solution in this scenario and possibly study alternative solutions.

Appendix A

Auction Class Diagrams

A.1 Seller Class Diagram

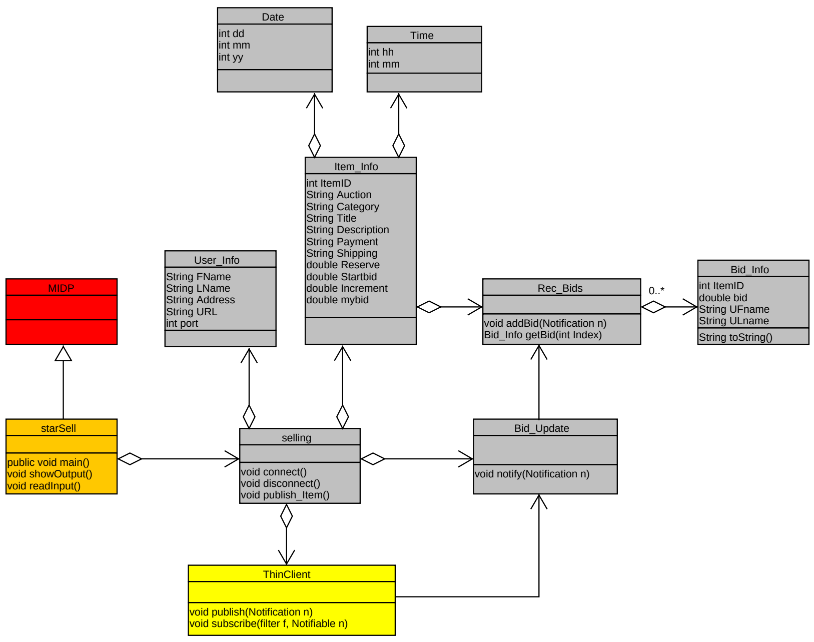


Figure A.1: *Seller* classes interaction.

A.2 Buyer Class Diagram

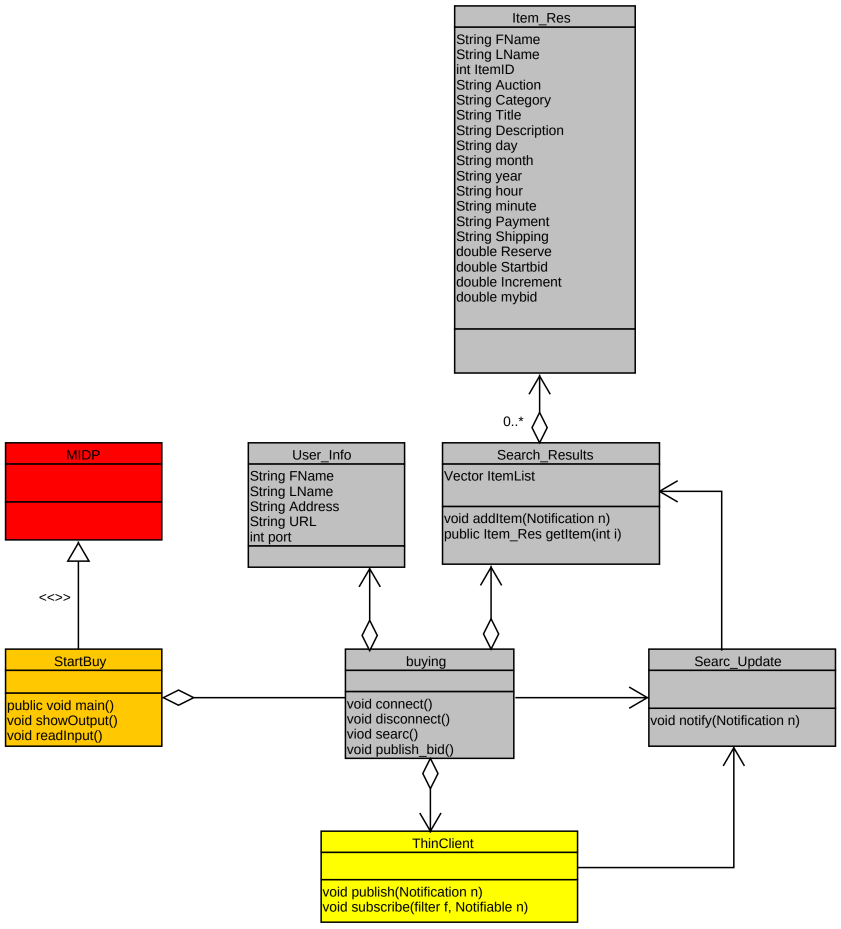


Figure A.2: *Buyer* classes interaction.

Appendix B

Seawind v3.0

Seawind enables researchers to emulate the behavior of wireless network using a common wireline local area network. The emulator allows examination of data transfer of wireless network like GSM and GPRS. The ability to emulate a wireless data network gives the possibility to find enhancements in transport protocol and network parameters.

B.1 Components of Seawind

The emulator acts like a black box that takes information in, handles it, and sends it out. Seawind produces output which can be investigated graphically. Figure B.1 present the logical architecture of the system. This configuration

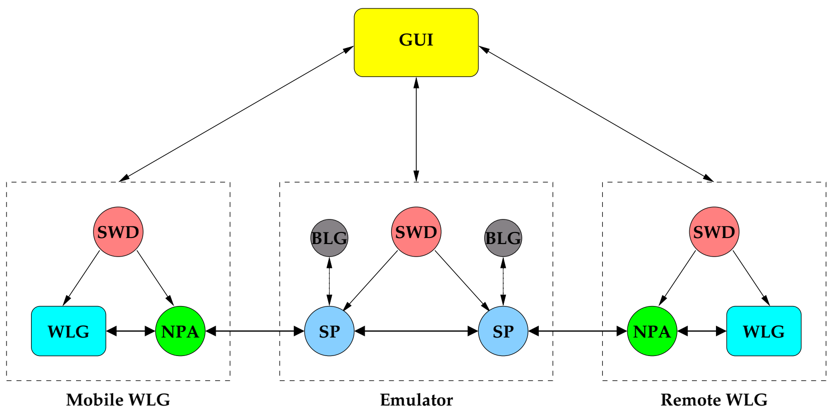


Figure B.1: *Seawind* Architecture.

sets up two *Simulation Psrocess* (SP) and they are located in the same network node.

B.1.1 Graphical User Interface (GUI)

The GUI interface interacts directly with the user and consist of a few windows in which the user can control the system.

B.1.2 Seawindd (SWD)

The *seawindd* is the Seawind daemon and runs in every machine where Seawind components work. It starts other components after getting coherent messages from the GUI.

B.1.3 Workload Generator (WLG)

The WLG generates the workload used in tests run. Seawind provides for two types of WLGs: unidirectional and bidirectional *ttcp* [34]. Other external WLGs can be used but those need to be controlled outside Seawind.

B.1.4 Network Protocol Adapter (NPA)

If Seawind own WLG is used, the NPA is used to encapsulate the data and forward it to the SP. Vice versa on the other end the NPA gets data from the SP and decapsulates the data before forwarding it to the receiving WLG. In this version of Seawind there is only one type of NPA defined:the type PPP (Point-to-Point protocol) [32].

B.1.5 Simulation Process (SP)

The simulation Process is the heart of Seawind. It affects the communication between workload generators by delaying and dropping packets according to given parameters. In addition, the SP produce output information which describes the current communication flow.

Befor a test can be run the user needs to define the parameters and thus behavior for all the mentioned components. The SP parameters, like spped and error distributions, can be defined independently for the Uplink [1] and downlink [2]. After the test, the user can utilize third party tools (like *tcpdump* [22] and *ethereal* [7]) for analyzing the transfer.

B.1.6 Bacground load (BGL)

The background load simulates the real communicationsystem's problem that also other users utilize network resources and affect data transmitting of primary users.

B.2 Parameters of Seawind

Setting up a test with Seawind requires a number of parameters to be set in the graphical user interface (GUI). The numerous parameters of Seawind are distributed among the different emulator system components. Every component has its own parameters.

The user starts the setting up by choosing different *replication sets*. A replication set defines the workload to be used, the network setup, and the number of replications. A test run can include several replication sets, which are run one after the other. The results of all replication sets are written in log files.

The network setup defines the location of different Seawind components and the network subsystem parameters for both directions of the transfer. The connection between the WLGs and the emulator kernel can either be a TCP connection or a serial link through the computers communication ports. PPP can be used to carry the workload from *ttcp* sender through Seawind to the *ttcp* receiver. Also background load parameters are part of SP parameters set. For more information, refer to the Seawind User Manual [25].

[1]Uplink is the direction from the mobile station to the network server.

[2]Downlink is the direction from the network server to the mobile station.

Appendix C

MobileDispatcher.java

```
//
//  This file is part of Siena, a wide-area event
//   notification system.
//  See http://www.cs.colorado.edu/serl/siena/
//
//  Author: Mauro Caporuscio <caporusc@cs.colorado.edu>
//
//  Copyright (C) 1998-2002 University of Colorado
//
//  This program is free software; you can redistribute
//  it and/or modify it under the terms of the GNU
//  General Public License
//  as published by the Free Software Foundation;
//  either version 2 of the License, or (at your option)
//  any later version.
//  This program is distributed in the hope that it will
//  be useful, but WITHOUT ANY WARRANTY; without even the
//  implied warranty of MERCHANTABILITY or FITNESS FOR A
//  PARTICULAR PURPOSE.  See the GNU General Public License
//  for more details.
//
// $Id: MobileDispatcher.java,v 1.00 2002/03/05 18:51:31
//      based on HierarchicalDispatcher.java, v 1.50
//

package siena;

import siena.comm.*;

import java.util.Collection;
import java.util.Set;
import java.util.HashSet;
import java.util.Map;
import java.util.Map.Entry;
import java.util.HashMap;
import java.util.List;
import java.util.LinkedList;
import java.util.Iterator;
import java.util.ListIterator;
import java.util.Vector;

import java.io.IOException;
import java.io.*;
import java.net.InetAddress;
import java.net.ServerSocket;
import java.net.Socket;
```

C.1 Class MobileSubscriber

```
//
// this is the abstraction of the subscriber used by the
// MobileDispatcher.  It represents remote as well as local
// notifiable objects.  In addition to that, this o
// bject keeps track of failed attempts to contact the notifiable
// object so that MobileDispatcher can periodically clean up
// its subscriber tables.
//

class MobileSubscriber implements PacketNotifiable {
    public  short           failed_attempts = 0;
    public  long            latest_good     = 0;

    private boolean         suspended       = false;
    private Notifiable      localobj        = null;
    private PacketSender    remoteobj       = null;
    int                     refcount        = 0;
    private SENPPacket      spkt            = new SENPPacket();

    private boolean         moving          = false;
    private Vector          storedEvents    = null;
    private Vector          dwlEvents       = null;
    private int                aspectedSize    = -1;
    public final byte[]        identity;

    public Filter              pfilter         = null;
    public ping_pong           pingpong;
    public boolean             pingack;

    synchronized public Vector getStoredEvents(){
        return storedEvents;
    }

    synchronized public void movingON(){
        moving = true;
    }

    synchronized public void movingOFF(){
        if (isLocal())
           while(!storedEvents.isEmpty())
               {
                 try {
                      localobj.notify((Notification)
                                          storedEvents.remove(0));
                     }
                 catch (Exception ex) {
                       handleNotifyError(ex);
                       return;
                      }
               }
        else
           while(!storedEvents.isEmpty())
               {
                 try {
                       SENPPacket pkt = (SENPPacket)
                                           storedEvents.remove(0);
                       remoteobj.send(pkt.buf, pkt.encode());
                     }
                 catch (Exception ex) {
                       handleNotifyError(ex);
                       return;
                      }
               }
        moving = false;
    }
```

```
synchronized public void movingOFF(Vector stored){
    if (isLocal())
      {
        while(!stored.isEmpty())
           {
             try {
                  localobj.notify((Notification)
                                                 stored.remove(0));
                 }
             catch (Exception ex) {
                   handleNotifyError(ex);
                   return;
                  }
           }
      }
    else
      {
        while(!stored.isEmpty())
           {
             try {
                   SENPPacket pkt = (SENPPacket)
                                          storedEvents.remove(0);
                   remoteobj.send(pkt.buf, pkt.encode());
                 }
             catch (Exception ex) {
                   handleNotifyError(ex);
                   return;
                  }
           }
      }
    moving = false;
}

synchronized public boolean dwlEvent(SENPPacket pkt){
    int pos = pkt.ttl - 10;
    if (pkt.event == null)
          aspectedSize = pos;
    else
        {
          if (dwlEvents == null) dwlEvents = new Vector();
          pkt.method = SENP.PUB;
          dwlEvents.add(pkt);
        }
    int actualsize = dwlEvents.size();
    if (actualsize == aspectedSize)
        {
          mergeEvents();
          return true;
        }
    else return false;
}

synchronized public void mergeEvents(){
    moving = false;
    if ( (dwlEvents == null) && (storedEvents.size() == 0))
        return;
    if (dwlEvents == null)
       {
          movingOFF();
          return;
       }
    if (storedEvents.size() == 0)
       {
          movingOFF(dwlEvents);
          return;
       }
    for(int i = 0; i < dwlEvents.size(); i++)
       {
         SENPPacket objd = (SENPPacket) dwlEvents.get(i);

         for(int j = 0; j < storedEvents.size(); j++)
            {
              SENPPacket objs = (SENPPacket)
                                        storedEvents.get(j);
              String str1 = objd.event.toString();
              String str2 = objs.event.toString();
              if (str1.compareTo(str2) == 0)
                 storedEvents.remove(j);
            }
       }
    dwlEvents.addAll(storedEvents);
    movingOFF(dwlEvents);
}

synchronized public boolean notify(SENPPacket pkt) {
        if (suspended) return true;
        try {
            if (localobj != null) {
                if (moving) storedEvents.add(new
                                         Notification(pkt.event));
                else localobj.notify(pkt.event);
            } else {
                if (moving) storedEvents.add(pkt);
                else remoteobj.send(pkt.buf, pkt.encode());
            }
            failed_attempts = 0;
            return true;
        } catch (Exception ex) {
            handleNotifyError(ex);
            return false;
        }
    }

    synchronized public void notify(Notification n,
                                              byte[] our_id) {
        if (suspended) return;
        try {
            if (localobj != null) {
                if (moving) storedEvents.add(n);
                else localobj.notify(n);
            } else {
                spkt.init();
                spkt.id = our_id;
                spkt.method = SENP.PUB;
                spkt.event = n;
                spkt.to = identity;

                if (moving) storedEvents.add(spkt);
                else remoteobj.send(spkt.buf, spkt.encode());
            }
        } catch (Exception ex) {
            handleNotifyError(ex);
        }
    }

    synchronized public void notify(Notification [] s,
                                              byte[] our_id) {
        if (suspended) return;
        try {
            if (localobj != null) {
                //
                // here I purposely do not duplicate the
                // sequence for efficiency reasons.
                // Clients should never modify
                // objects passed through notify().
                //
                if (moving)
                   {
                     for(int i=0; i < s.length; ++i)
                         storedEvents.add(s[i]);
                   }
                else localobj.notify(s);
            } else {
                spkt.init();
                spkt.id = our_id;
                spkt.method = SENP.PUB;
                spkt.events = s;
                spkt.to = identity;

                if (moving) storedEvents.add(spkt);
                else remoteobj.send(spkt.buf, spkt.encode());
            }
        } catch (Exception ex) {
            handleNotifyError(ex);
        }
    }
}
```

C.2 Class MobileDispatcher

```
public class MobileDispatcher implements Siena,  Runnable {
    private MPoset         subscriptions   = new MPoset();
    private Map            contacts        = new HashMap();
    private MSenderManager pqueue          = null;
    private byte[]         master_id       = null;
    private byte[]         master_handler  = null;
    private PacketSender   master          = null;
    private PacketReceiver listener        = null;
    private byte[]         my_identity     = null;
    private List            matchers        = new LinkedList();

    private SENPPacket      spkt = new SENPPacket();
    private byte []         sndbuf = new byte[SENP.MaxPacketLen];

    private PacketSenderFactory         sender_factory;
    static private PacketSenderFactory  default_sender_factory
                                      = new GenericSenderFactory();

private void processRequest(SENPPacket req) {
        Logging.prlnlog("processRequest: " + req);
        if (req == null) {
            Logging.prlnerr("processRequest: null request");
            return;
        }

        if (req.ttl <= 0) return;
        req.ttl--;
        try {
            switch(req.method) {
                case SENP.NOP: break;
                case SENP.PUB: publish(req); break;
                case SENP.SUB: subscribe(req); break;
                case SENP.BYE: req.pattern = null;
                               req.filter = null;
                case SENP.UNS: unsubscribe(req); break;
                case SENP.WHO: reply_who(req); break;
                case SENP.INF: get_info(req); break;
                case SENP.SUS: suspend(req); break;
                case SENP.RES: resume(req); break;
                case SENP.MAP: map(req); break;
                case SENP.CNF: configure(req); break;
                case SENP.OFF: shutdown();
                                //
                                // BEGIN_UNOFFICIAL_PATCH
                                try { Thread.sleep(500); }
                                catch (Exception ex) {};
                                System.exit(0);
                                // END_UNOFFICIAL_PATCH
                                //
                                break;

            //BEGIN_MOBILITY_PATCH
```

```
                case SENP.MVL: moveoutLOW(req); break;
                case SENP.MVH: moveoutHIGH(req); break;
                case SENP.MVI: movein(req); break;
                case SENP.DWL: downloadNotificationLOW(req);
                               break;
                case SENP.DWH: downloadNotificationHIGH(req);
                               break;
                case SENP.UPL: uploadNotification(req); break;
                case SENP.PRV: req.ttl++; privateEvent(req);
                               break;
            //END_MOBILITY_PATCH

                default:
                   Logging.prlnerr("processRequest:
                                       unknown method: " + req);
                   //
                   // can't handle this request (yet)
                   // ...work in progress...
                   //
            }
        } catch (Exception ex) {
            Logging.exerr(ex);
            //
            // log something here ...work in progress...
            //
        }
    }

//================================================================
// MOBILITY SUPPORT PATCH
//================================================================

//----------------------------------------------------------------
//Local Requests
//----------------------------------------------------------------
     /** suspends the delivery of notification to the
      *  given subscriber n and allows the master to store
      *  all notification addressed to this dispatcher .
      *
      *  This causes the <em>master</em> server to stop sending
      *  notification to this subscriber and to store
      *  them in a queue.
      *  The master correctly maintains all the existing
     *  subscriptions so that the flow of notification can be
     *  later resumed
     *  (see moveIn(Notifiable n)).
     *  This operation can be used when this
     *  dispatcher, that is this virtual machine, is going to be
     *  temporarily disconnected from the network or somehow
     *  unreachable from its master server.
     *
     **/
    synchronized public void moveOut(Notifiable n)
                             throws SienaException {
        if (n == null) return;
        MobileSubscriber s;
        s = (MobileSubscriber)contacts.get(n);
        if (s != null) s.movingON();
    }
     /** resumes the delivery of notification to the given
     *subscriber n.
     *
     *  This causes the <em>master</em> server to resume sending
     *  stored and new notifications to this subscriber.
     *
     *
     *  @see #moveOut(Notifiable n)
     *  @see #suspend(Notifiable n)
     *  @see #resume(Notifiable n)
     **/
    synchronized public void moveIn(Notifiable n)
                             throws SienaException {
        if (n == null) return;
        MobileSubscriber s;
        s = (MobileSubscriber)contacts.get(n);
        if (s != null) s.movingOFF();
    }

//-------------------------------------------------------------
//Remote Requests
//-------------------------------------------------------------
    synchronized private void disconnectMaster(
        byte[] omaster_handler, PacketSender omaster, String id) {
        if (omaster != null) {
            try {
                spkt.init();
                spkt.method = SENP.BYE;
                spkt.id = id.getBytes();
                spkt.to = omaster_handler;
                omaster.send(spkt.buf, spkt.encode());
            } catch (PacketSenderException ex) {
                Logging.prlnerr("error sending packet to "
                    + master.toString() + ": " + ex.toString());
                //
                // well, what would you do in this case?
                // ...work in progress...
                //
            }
            //master = null;
            //master_handler = null;
        }
    }

   synchronized private void moveoutLOW(SENPPacket req) {
        if (req.id == null || req.ttl == 0) return;
        String id = new String(req.id);
        MobileSubscriber s = (MobileSubscriber)
                                    contacts.get(id);
        if (s != null) s.movingON();
    }

    synchronized private void moveoutHIGH(SENPPacket req) {
        if (req.id == null || req.ttl == 0) return;
        String id = new String(req.id);
        //System.out.println(id);
        MobileSubscriber s = (MobileSubscriber)
                                    contacts.get(id);
        if (s != null)
           {
             s.movingON();
             try{
                  s.pingpong = new ping_pong(this, s);
                  s.pfilter = new Filter();
                  s.pfilter.addConstraint("id__",id);
                  s.pfilter.addConstraint("type__",
                                       "SYNC_PING");
                  this.subscribe(s.pfilter, s.pingpong);
                }
             catch (SienaException ex)
```

```
               {
                 Logging.prlnerr("error subscribing ");
                 Logging.exerr(ex);
               }
          }
    }

    synchronized private void movein(SENPPacket req) {
        if (req.id == null || req.ttl == 0) return;
        String id = new String(req.id);
        //System.out.println(id);
        MobileSubscriber s = (MobileSubscriber)
                                      contacts.get(id);
        if (s != null) s.movingOFF();
    }

    synchronized private void downloadNotificationLOW(
                                        SENPPacket req){
        if (req.id == null || req.ttl == 0) return;
        String id = new String(req.id);

        MobileSubscriber s = (MobileSubscriber)
                                       contacts.get(id);
        if (s != null)
            {
              //ServerSocket server;
              PacketSender oldmaster;
              try {
                    //send the server address to the OLD master
                    oldmaster =
                      sender_factory.createPacketSender(
                                     new String(req.to));
                    spkt.init();
                    spkt.method = SENP.UPL;
                    spkt.id = req.id;
                    spkt.to = oldmaster.toString().getBytes();
                    spkt.handler = listener.uri();
                    oldmaster.send(spkt.buf, spkt.encode())
                  }
              catch (Exception ex) {
                    Logging.prlnerr("error
                      sending packet to " + master.toString());
                    Logging.exerr(ex);
```

```
                }
            }
    }

    synchronized private void downloadNotificationHIGH(
                                                SENPPacket req){
        if (req.id == null || req.ttl == 0) return;
        String id = new String(req.id);

        MobileSubscriber s = (MobileSubscriber)contacts.get(id);
        if (s != null)
            {
              try{
                   s.pingack = false;

                   s.pingpong = new ping_pong(this, s);
                   s.pfilter = new Filter();
                   s.pfilter.addConstraint("id__", id);
                   s.pfilter.addConstraint("type__", "SYNC_PONG");
                   this.subscribe(s.pfilter, s.pingpong);

                   Notification n = new Notification();
                   n.putAttribute("id__",id);
                   n.putAttribute("type__","SYNC_PING");
                   this.publish(n);

                   int tent = 0;
                   while ((!s.pingack) && (tent < 60)){
                     try{
                          Thread.sleep(1000);
                          tent++;
                        }
                     catch (java.lang.InterruptedException ex) {
                             System.out.println("interrupted");
                     }
                   }

                   if (tent == 60 )
                      {
                        Logging.prlnerr("error dowloading");
                        return;
                      }
                this.unsubscribe(s.pfilter, s.pingpong);

              }
            catch (SienaException ex)
              {
                Logging.prlnerr("error subscribing ");
                Logging.exerr(ex);
              }


            //ServerSocket server;
            PacketSender oldmaster;
            try {
                  //send the server address to the OLD master
                  oldmaster =
                    sender_factory.createPacketSender(
                                         new String(req.to));
                  spkt.init();
                  spkt.method = SENP.UPL;
                  spkt.id = req.id;
                  spkt.to = oldmaster.toString().getBytes();
                  spkt.handler = listener.uri();
                  oldmaster.send(spkt.buf, spkt.encode());

               }
            catch (Exception ex) {
                  Logging.prlnerr("error sending packet to " +
                                           master.toString());
                  Logging.exerr(ex);
               }
          }
    }

    synchronized private void uploadNotification(SENPPacket req){
        if (req.id == null || req.ttl == 0) return;
        String id = new String(req.id);
        MobileSubscriber s = (MobileSubscriber)contacts.get(id);
        if (s != null)
            {
              PacketSender soc;
              try {
                   //Connection
                   soc = sender_factory.createPacketSender(
                                   new String(req.handler));

                  //Downloading stored Events
                  Vector storedEvents = s.getStoredEvents();
                  byte pcknum = 9;
                  while(!storedEvents.isEmpty())
                       {
                         pcknum ++;

                         SENPPacket prv = (SENPPacket)
                                        storedEvents.remove(0);

                         prv.method = SENP.PRV;
                         prv.ttl = pcknum;
                         prv.id = my_identity;
                         prv.to = id.getBytes();
                         prv.handler = listener.uri();
                         soc.send(prv.buf, prv.encode());
                       }

                  //Download is finished
                  pcknum++;

                  spkt.method = SENP.PRV;
                  spkt.ttl = pcknum;
                  spkt.id = my_identity;
                  spkt.to = id.getBytes();
                  spkt.handler = listener.uri();
                  spkt.event = null;
                  soc.send(spkt.buf, spkt.encode());

                 }

             catch (Exception ex) {
                  Logging.prlnerr("error sending packets to "
                                 + new String(req.handler));
                  Logging.exerr(ex);
                 }
           }
    }

    synchronized private void privateEvent(SENPPacket req){
        if (req.id == null) return;
        String id = new String(req.to);
        MobileSubscriber s = (MobileSubscriber)contacts.get(id);
        if (s != null)
            if (s.dwlEvent(req))
               {
                 try {
                       PacketSender old_master =
                       sender_factory.createPacketSender(
                                      new String(req.handler));
                       disconnectMaster(req.handler,
                                                   old_master, id);
                     }
                 catch (Exception ex) {
                       Logging.prlnerr("error sending packet to "
                                       + new String(req.handler));
                       Logging.exerr(ex);
                     }
                }
    }

    //-----------------------------------------------
    //Public Methods


    /** suspends the connection with the master server of
     *  this dispatcher and allows the master to store all
     *  notification addressed to this dispatcher.
     *
     *  This causes the master server to stop sending
     *  notification to this dispatcher and to store them
     *  in a queue.
     *  The master correctly maintains all the
     *  existing subscriptions so that the flow
     *   of notification can be later resumed
     *  (see moveInMaster()).
     *  This operation can be used when this
     *  dispatcher, that is this virtual machine, is going to be
     *  temporarily disconnected from the network or somehow
     *  unreachable from its master server.
     *
     *  @param  QoS is the Quality of Service
     *          if true high reliability
     *          if false low reliability
     *
     *  @see #suspendMaster()
     *  @see #resumeMaster()
     *  @see #moveInMaster()
     **/

    synchronized public void moveOutMaster(boolean QoS) {
        if (QoS)
           try {
                 spkt.init();
                 spkt.method = SENP.MVH;
                 spkt.to = master_handler;
                 spkt.id = my_identity;
                 spkt.handler = listener.uri();
                 master.send(spkt.buf, spkt.encode());
           } catch (Exception ex) {
                 Logging.prlnerr("error sending packet to "
                                            + master.toString());
                 Logging.exerr(ex);
                 //
                 // of course I should do something here...
                 // ...work in progress...
                 //
           }
        else
           try {
                 spkt.method = SENP.MVL;
                 spkt.to = master_handler;
                 spkt.id = my_identity;
                 spkt.handler = listener.uri();
                 master.send(spkt.buf, spkt.encode());
           } catch (Exception ex) {
                 Logging.prlnerr("error sending packet to "
                                            + master.toString());
                 Logging.exerr(ex);
                 //
                 // of course I should do something here...
                 // ...work in progress...
                 //
           }
    }

    /** suspends the connection with the master server of
```

```
     *  this dispatcher and allows the master to store all
     *  notification addressed to this dispatcher.
     *
     *  see moveInMaster(boolean QoS).
     *
     *  @see #suspendMaster()
     *  @see #resumeMaster()
     *  @see #moveInMaster()
     **/

    synchronized public void moveOutMaster() {
        moveOutMaster(false);
    }

     /** resumes the connection with the master server.
     *
     *  This causes the master server to resume sending
     *  stored and new notifications to this dispatcher.
     *
     *  @see #moveOutMaster()
     **/
    synchronized public void moveInMaster() {
        try {
              spkt.init();
              spkt.method = SENP.MVI;
              spkt.to = master_handler;
              spkt.id = my_identity;
              spkt.handler = listener.uri();
              master.send(spkt.buf, spkt.encode());
        } catch (Exception ex) {
              Logging.prlnerr("error sending packet to "
                                          + master.toString());
              Logging.exerr(ex);
              //
              // of course I should do something here...
              // ...work in progress...
              //
        }
    }

    /** resumes the connection with the master server.
     *
     *  This causes the master server to resume sending
     *  stored and new notifications to this dispatcher.
     *
     *  This also sets the new packet receiver for this server.
     *
     * This method simply calls setReceiver(PacketReceiver, int)
     *
     *  @param pr is the receiver
     *
     *  @see #moveOutMaster()
     *  @see #setReceiver(PacketReceiver)
     **/
    synchronized public void moveInMaster(PacketReceiver pr) {
        try {
              setReceiver(pr);

              spkt.init();
              spkt.method = SENP.MVI;
              spkt.to = master_handler;
              spkt.id = my_identity;
              spkt.handler = listener.uri();
              master.send(spkt.buf, spkt.encode());
        } catch (Exception ex) {
              Logging.prlnerr("error sending packet to "
                                              + master.toString());
              Logging.exerr(ex);
              //
              // of course I should do something here...
              // ...work in progress...
              //
        }
    }

    /** resumes the connection with the master server.
     *
     *  This causes the master server to resume sending
     *  stored and new notifications to this dispatcher.
     *
     *  This also sets the new server for this dispatcher.
     *
     *  @param uri is the external identifier
     *             of the master dispatcher
     *             (e.g., * senp://host.domain.edu:8765")
     *
```

```
 *   @param  QoS is the Quality of Service
 *          if true High reliability
 *          if false low reliability
 *
 *  @see #moveOutMaster()
 *  @see #setReceiver(PacketReceiver)
 *
 **/

synchronized public void moveInMaster(String uri, boolean QoS)
    throws InvalidSenderException, java.io.IOException {

    byte[] old_master_handler = null;
    PacketSender old_master   = null;

    //Backup old Master
    old_master_handler = master_handler;
    old_master = master;

    //Create a new Master
    PacketSender new_master =
                      sender_factory.createPacketSender(uri);

    boolean new_listener = false;
    if (listener == null) {
        setReceiver(new TCPPacketReceiver(0));
        new_listener = true;
    }

    master_handler = uri.getBytes();
    master = new_master;
    //
    // sends a WHO packet to figure out the identity of
    // the master server.
    // This dispatcher uses the "to" field of the SENP
    // packet to tell the master server the handler used
    // by this server to reach the master server.
    // (see reply_who())
    //
    try{
         spkt.init();
         spkt.method = SENP.WHO;
         spkt.ttl = 2;                      // round-trip
```

```
          spkt.to = master_handler;
          spkt.id = my_identity;
          spkt.handler = listener.uri();
          master.send(spkt.buf, spkt.encode());
          //
          // perhaps I should sit here waiting for the
          // INF response
          // of the server
          //
          // ...to be continued...
          //
     } catch (Exception ex) {
          Logging.prlnerr("error sending packet to "
                                  + master.toString());
          Logging.exerr(ex);
          master = null;
          master_handler = null;
          if (new_listener) {
             try {
                   listener.shutdown();
             } catch (PacketReceiverException pex) {
                   Logging.exerr(pex);
             }
         }
         //
         // of course I should do something here...
         // ...work in progress...
         //
     }

     //Store the notification at the NEW Master
     moveOutMaster();

     //
     // sends all the top-level subscriptions to the new master
     //
     for(Iterator i = subscriptions.rootsIterator();
                                            i.hasNext();) {
         MSubscription s = (MSubscription)i.next();
         try {
               spkt.init();
               spkt.method = SENP.SUB;
               spkt.ttl = SENP.DefaultTtl;
```

```
                spkt.id = my_identity;
                spkt.handler = listener.uri();
                spkt.filter = s.filter;
                master.send(spkt.buf, spkt.encode());
            } catch (Exception ex) {
                Logging.prlnerr("error sending packet to "
                                           + master.toString());
                Logging.exerr(ex);
                //
                // of course I should do something here...
                // ...work in progress...
                //
            }
        }

        //Download the events stored at the OLD Master
        if (QoS)
          try {
              spkt.init();
              spkt.method = SENP.DWH;
              spkt.ttl = 2;                  // round-trip
              spkt.to = old_master_handler;
              spkt.id = my_identity;
              spkt.handler = listener.uri();
              old_master.send(spkt.buf, spkt.encode());
            }
          catch (Exception ex)
            {
              Logging.prlnerr("error sending packet to "
                                             + master.toString());
              Logging.exerr(ex);
          }
        else
            try {
              spkt.init();
              spkt.method = SENP.DWL;
              spkt.ttl = 2;                  // round-trip
              spkt.to = old_master_handler;
              spkt.id = my_identity;
              spkt.handler = listener.uri();
              old_master.send(spkt.buf, spkt.encode());
            }
          catch (Exception ex)
```

```
              {
                Logging.prlnerr("error sending packet to "
                                             + master.toString());
                Logging.exerr(ex);
            }
    }

     /** resumes the connection with the master server.
      *
      *  This causes the master server to resume sending
      *  stored and new notifications to this dispatcher.
      *
      *  This also sets the new server for this dispatcher.
      *
      *  @param uri is the external identifier
      *             of the master dispatcher
      *             (e.g., * senp://host.domain.edu:8765")
      *
      *  @see #moveOutMaster()
      *  @see #setReceiver(PacketReceiver)
      *
      **/

    synchronized public void moveInMaster(String uri)
        throws InvalidSenderException, java.io.IOException {

        moveInMaster(uri, false);
    }

    /** resumes the connection with the master server.
     *
     *  This causes the master server to resume sending
     *  stored and new notifications to this dispatcher.
     *
     *  This also sets the new server and the new
     *  packet receiver for this dispatcher.
     *
     * This method simply calls setMaster(String)
     * and setReceiver(PacketReceiver, int)
     *
     *  @param pr is the receiver
     *
     *  @param uri is the external identifier
```

```
               of the master dispatcher
 *             (e.g., * senp://host.domain.edu:8765")
 *
 *   @param  QoS is the Quality of Service
 *          if true High reliability
 *          if false low reliability
 *
 *  @see #moveOutMaster()
 *  @see #setReceiver(PacketReceiver)
 *
 **/

synchronized public void moveInMaster(PacketReceiver pr,
                                     String uri, boolean QoS)
                               throws InvalidSenderException,
                                         java.io.IOException {


    // Change Client Location
    try {
          setReceiver(pr);
    } catch (Exception ex) {
          Logging.prlnerr("error sending packet to "
                                         + master.toString());
          Logging.exerr(ex);
    }

    byte[] old_master_handler = null;
    PacketSender old_master   = null;

    //Backup old Master
    old_master_handler = master_handler;
    old_master = master;

    //Create a new Master
    PacketSender new_master =
                   sender_factory.createPacketSender(uri);

    boolean new_listener = false;
    if (listener == null) {
        setReceiver(new TCPPacketReceiver(0));
        new_listener = true;
    }
```

```
master_handler = uri.getBytes();
master = new_master;
//
// sends a WHO packet to figure out the identity of the
//master server.
// This dispatcher uses the "to" field of the SENP
// packet to tell the master server the handler used
// by this server to reach the master server.
// (see reply_who())
//
try {
      spkt.init();
      spkt.method = SENP.WHO;
      spkt.ttl = 2;                      // round-trip
      spkt.to = master_handler;
      spkt.id = my_identity;
      spkt.handler = listener.uri();
      master.send(spkt.buf, spkt.encode());
      //
      // perhaps I should sit here waiting for the
      // INF response
      // of the server
      //
      // ...to be continued...
      //
} catch (Exception ex) {
      Logging.prlnerr("error sending packet to "
                                     + master.toString());
      Logging.exerr(ex);
      master = null;
      master_handler = null;
      if (new_listener) {
         try {
               listener.shutdown();
         } catch (PacketReceiverException pex) {
               Logging.exerr(pex);
         }
      }
      //
      // of course I should do something here...
      // ...work in progress...
      //
}


//Store the notification at the NEW Master
moveOutMaster();

//
// sends all the top-level subscriptions to the new master
//
for(Iterator i = subscriptions.rootsIterator();
                                            i.hasNext();) {
    MSubscription s = (MSubscription)i.next();
    try {
          spkt.init();
          spkt.method = SENP.SUB;
          spkt.ttl = SENP.DefaultTtl;
          spkt.id = my_identity;
          spkt.handler = listener.uri();
          spkt.filter = s.filter;
          master.send(spkt.buf, spkt.encode());
    } catch (Exception ex) {
          Logging.prlnerr("error sending packet to "
                                   + master.toString());
          Logging.exerr(ex);
          //
          // of course I should do something here...
          // ...work in progress...
          //
    }
}

//Download the events stored at the OLD Master
if (QoS)
  try {
        spkt.init();
        spkt.method = SENP.DWH;
        spkt.ttl = 2;                  // round-trip
        spkt.to = old_master_handler;
        spkt.id = my_identity;
        spkt.handler = listener.uri();
        old_master.send(spkt.buf, spkt.encode());
      }
  catch (Exception ex)
```

```
                {
                  Logging.prlnerr("error sending packet to "
                                             + master.toString());
                  Logging.exerr(ex);
                }
            else
              try {
                  spkt.init();
                  spkt.method = SENP.DWL;
                  spkt.ttl = 2;                   // round-trip
                  spkt.to = old_master_handler;
                  spkt.id = my_identity;
                  spkt.handler = listener.uri();
                  old_master.send(spkt.buf, spkt.encode());
                }
              catch (Exception ex)
                {
                  Logging.prlnerr("error sending packet to "
                                             + master.toString());
                  Logging.exerr(ex);
                }
        }

        /** resumes the connection with the master server.
         *
         *  This causes the master server to resume sending
         *  stored and new notifications to this dispatcher.
         *
         *  This also sets the new server and the new
         *  packet receiver for this dispatcher.
         *
         * This method simply calls setMaster(String)
         * and setReceiver(PacketReceiver, int)
         *
         *  @param pr is the receiver
         *
         *  @param uri is the external identifier
         *             of the master dispatcher
         *            (e.g., * senp://host.domain.edu:8765")
         *
         *  @see #moveOutMaster()
         *  @see #setReceiver(PacketReceiver)
         *
```

```
     **/

    synchronized public void moveInMaster(PacketReceiver pr,
                                                      String uri)
                                  throws InvalidSenderException,
                                            java.io.IOException {

        moveInMaster(pr, uri, false);
    }


    /** starts the Mobility server search.
     *
     * This method simply subscribes the ``Notifiable n''
     * for a special filter.
     *
     *  @param n is the object will receive the search results
     *
     **/
    synchronized public void addMobileDispatcherFinder(
                             Notifiable n) throws SienaException {

        Filter f = new Filter();
        f.addConstraint("servive__","Mobile_Server_replay__");

        subscribe(f,n);

        Notification e = new Notification();
        e.putAttribute("servive__","Mobile_Server_request__");

        publish(e);
    }

     /** stop the Mobility server search.
     *
     * This method simply unsubscribes the ``Notifiable n''
     * for a special filter.
     *
     *  @param n is the object receiving the search results
     *
     **/
    synchronized public void removeMobileDispatcherFinder(
                             Notifiable n) throws SienaException {
```

```
        Filter f = new Filter();
        f.addConstraint("servive__","Mobile_Server_replay__");

        unsubscribe(f,n);
    }

     /** start info sending about this dispatcher.
      *
      * This method simply subscribes this dispatcher
      * for a special filter representing a service request.
      *
      *  @param s is the information about this dispatcher.
      *
      **/

    public void StartAvailability(String s)
                                       throws SienaException {

        ServiceReplay sr = new ServiceReplay(this, s);

        Filter f = new Filter();
        f.addConstraint("servive__","Mobile_Server_request__");

        subscribe(f, sr);
    }
}

//==============================================================

class ping_pong implements Notifiable{

    MobileSubscriber ms;
    MobileDispatcher  md;

    public ping_pong(MobileDispatcher d, MobileSubscriber s){
      ms = s;
      md = d;
    }

    public void notify(Notification[] s) throws SienaException {
    }
    public void notify(Notification n) throws SienaException {

        String type = n.getAttribute("type__").toString();

        if (type.compareTo(new
                     AttributeValue("SYNC_PONG").toString()) == 0)
            ms.pingack = true;

        if (type.compareTo(new
                     AttributeValue("SYNC_PING").toString()) == 0)
           {
             Notification e = new Notification();
             e.putAttribute("id__",new String(ms.identity));
             e.putAttribute("type__","SYNC_PONG");

             md.publish(e);
           }
    }

}

class ServiceReplay implements Notifiable{

    String             info;
    MobileDispatcher   md;

    public ServiceReplay(MobileDispatcher m, String s){
      md   = m;
      info = s;
    }

    public void notify(Notification[] s) throws SienaException {
    }

    public void notify(Notification n) throws SienaException {

        Notification e = new Notification();
        e.putAttribute("servive__","Mobile_Server_replay__");
        e.putAttribute("uri__",new String(md.getReceiver().uri()));
        e.putAttribute("info__",info);

        md.publish(e);
    }
}
```

Bibliography

[1] G. Brasche and B. Walke. Concept, services and protocols for the new GSM Phase 2+ General Packet Radio Service. Technical report, IEEE Communications Magazine, 1997.

[2] L. Cardelli and A. D. Gordon. Mobile Ambients. *Theoretical Computer Science*, 2000.

[3] A. Carzaniga. Siena 1.3.0 Api Documentation.
www.cs.colorado.edu/∼carzanig/siena/.
Copyright ©2000-2002 University of Colorado.

[4] A. Carzaniga, G. Picco, and G. Vigna. Designing Distributed Applications with Mobile Code Paradigms. In *Proceedings of the 19th International Conference on Software Engineering*, pages 22–32, Boston, Massachusetts, May 1997.

[5] 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.

[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*, 19(3):332–383, Aug. 2001.

[7] G. Combs. The Ethereal Network Analyzer. UNIX manual.
Available from www.ethereal.com.

[8] D. E. Comer. *Internetworking with TCP/IP*, volume Volume I - Principles, protocols and architecture. Prentice Hall, third edition edition, 1995.

[9] G. Cugola, C. Ghezzi, G. P. Picco, and G. Vigna. Analyzing Mobile Code Languages. In *Mobile Object Systems: Towards the Programmable Internet*, pages 93–110. Springer-Verlag: Heidelberg, Germany, 1997.

[10] E. Durocher and I. Filotti. Native Code Migration over a Heterogeneous Network - An Efficient Approach to Process Migration.

[11] Ericsson Mobility World. GATE II. www.ericsson.com/mobilityworld/.

[12] ETSI. GSM 03.60: Digital cellular telecommunications system (Phase 2+); General Packet Radio Service (GPRS); Service Description; Stage 2.

[13] ETSI. GSM 03.64: Digital cellular telecommunications system (Phase 2+); General Packet Radio Service (GPRS); Overall description of the GPRS radio interface.

[14] ETSI. GSM 08.18: Digital cellular telecommunications system (Phase 2+); General Packet Radio Service (GPRS); Base Station System (BSS) - Serving GPRS Support Node (SGSN) - BSS GPRS Protocol (BSSGP).

[15] ETSI. GSM 09.60: Digital cellular telecommunications system (Phase 2+); General Packet Radio Service (GPRS); GPRS Tunneling Protocol (GTP) across the Gn and Gp Interface.

[16] K. Fall and K. Varadhan. *The* ns *Manual*. The VINT Project, November 2001. A Collaboration between researchers at UC Berkeley, LBL, USC/ISI and Xerox PARC.

[17] A. Fugetta, G. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transaction on Software Engineering*, 24(5), 1998.

[18] General Magic. Telescript Language Reference, Oct 1995.

[19] E. Giguere. *Java^{TM} 2 Micro Edition.* Professional Developer's Guide. John Wisley & Son, release 1.0 edition, 2001.

[20] Internet Engineering Task Force. Internet Printing Protocol (IPP). www.ietf.org/html.charters/ipp-charter.html.

[21] Internet Engineering Task Force. Mobile Ad-Hoc Networks (MANET) WG Charter. www.ietf.org/html.charters/manet-charter.html.

[22] V. Jacobson, C. Leres, and S. McCanne. tcpdump - dump traffic on a network. UNIX manual. Available from www.tcpdump.org.

[23] R. Jain. *GPRS Simulations using ns-Network Simulator.* PhD thesis, Department of Electrical Engineering, Indian Institute of Technology - Bombay, June 2001.

[24] M. Kojo, A. Gurtov, J. Manner, P. Sarolahti, and K. Raatikainen. Seawind: a Wireless Network Emulator. University of Helsinki, Finland.

[25] M. Kojo, A. Gurtov, J. Manner, P. Sarolahti, and K. Raatikainen. *Seawind v3.0 User Manual.* University of Helsinki, Finland, September 2001.

[26] L. Lamport. Time, clocks, and the ordering of events in a distributed system, 1978.

[27] Motorola Wireless Development Centre. The Motorola GPRS Emulator. developers.motorola.com/developers/wireless/global/uk/emulator.htm.

[28] M. Mouly and M. Pautet. Current Evolution of the GSM Systems. Technical report, IEEE Pers. Commun., 1995.

[29] A. L. Murphy, G.-C. Roman, and G. P. Picco. Coordination and Mobility. In A. Omicini and F. Zambonelli and M. Klusch and R. Tolksdorf, editor, *Coordination of Internet Agents: Models, Technologies, and Applications*, pages 254–273. Springer, 2000.

[30] Nokia. Nect Act Planner.
www.nokia.com/networks/services/netact/netact_planner/.

[31] G.-C. Roman, G. P. Picco, and A. L. Murphy. Software Engineering for Mobility: A Roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*, pages 241–258. ACM Press, 2000. Invited contribution.

[32] W. Simpson. The Point-to-Point Protocol (PPP). Request for Comments, July 1994. RFC 1661.

[33] J. Stamos and D. Gifford. Remote Evaluation. *ACM Trans. on Programming Languages and System*, pages 537–565, October 1990.

[34] R. Stine. FYI on a network management tool catalog: Tools for monitoring and debugging TCP/IP internets and interconnected devices. Request for Comments, Apr. 1990. RFC 1147.

[35] Sun Microsystem. The Java Language Specification, Oct 1995.

[36] The Source for Java Technology. Java™ 2 Platform Micro Edition. Available from java.sun.com/j2me/.

[37] The Source for Java Technology. Java™ 2 Platform Micro Edition, Wireless Toolkit. Available from java.sun.com/products/j2mewtoolkit/.

[38] The Source for Java Technology. Java™ 2 Platform Standard Edition. Available from java.sun.com/j2se/.

[39] D. Wong, N.Paciorek, and D. Moore. Java-based Mobile Agents. *Communication of the ACM*, pages 92–102, 1999.