Report Number: WUCSE-2004-17

2004-04-14

Automated Code Management for Service Oriented Computing in Ad Hoc Networks

Radu Handorean, Rohan Sen, Gregory Hackmann, and Gruia-Catalin Roman

Ad hoc networks are dynamic environments where fre-quent disconnections and transient interactions lead to de-coupled computing. Typically, participants in an ad hoc network are small mobile devices such as PDAs or cellu-lar phones that have a limited amount of resources avail-able locally, and must leverage the resources on other co-located devices to provide the user with a richer set of func-tionalities. Service-oriented computing (SOC), an emerging paradigm that seeks to establish a standard way of mak-ing resources and capabilities available for use by others in the form of services, is a useful model for engineering soft-ware that seeks to...

Read complete abstract on page 2.

Automated Code Management for Service Oriented Computing in Ad Hoc Networks

Radu Handorean, Rohan Sen, Gregory Hackmann, and Gruia-Catalin Roman

Complete Abstract:

Ad hoc networks are dynamic environments where fre-quent disconnections and transient interactions lead to de-coupled computing. Typically, participants in an ad hoc network are small mobile devices such as PDAs or cellu-lar phones that have a limited amount of resources avail-able locally, and must leverage the resources on other co-located devices to provide the user with a richer set of func-tionalities. Service-oriented computing (SOC), an emerging paradigm that seeks to establish a standard way of mak-ing resources and capabilities available for use by others in the form of services, is a useful model for engineering soft-ware that seeks to exploit capabilities on remote devices. This paper proposes an automatic code management sys-tem supporting SOC in ad hoc networks. The system is re-sponsible for ensuring that the binary code required to use a service on a remote machine is available on the local host only when required. To support this functionality, a local code base is maintained by discovering and installing code from remote hosts. Since the system is specifically designed for ad hoc networks, it incorporates additional features that help it withstand the inherent dynamism of the network. We present an architecture for our system supporting automatic code management and follow it with a discussion of a Java-based implementation.

2004-17

Automated Code Management for Service Oriented Computing in Ad Hoc Networks

Authors: Handorean, R.; Sen, R.; Hackmann, G.; Roman, G.-C.

April 14, 2004

Abstract: Ad hoc networks are dynamic environments where frequent disconnections and transient interactions lead to decoupled computing. Typically, participants in an ad hoc network are small mobile devices such as PDAs or cellular phones that have a limited amount of resources available locally, and must leverage the resources on other colocated devices to provide the user with a richer set of functionalities. Service-oriented computing (SOC), an emerging paradigm that seeks to establish a standard way of making resources and capabilities available for use by others in the form of services, is a useful model for engineering software that seeks to exploit capabilities on remote devices. This paper proposes an automatic code management system supporting SOC in ad hoc networks. The system is responsible for ensuring that the binary code required to use a service on a remote machine is available on the local host

Automated Code Management for Service Oriented Computing in Ad Hoc Networks

Radu Handorean, Rohan Sen, Gregory Hackmann, and Gruia-Catalin Roman
Department of Computer Science and Engineering
Washington University in St. Louis
Campus Box 1045, One Brookings Drive
St. Louis, MO 63130-4899, USA
Email: {radu.handorean, rohan.sen, ghackmann, roman}@wustl.edu

Abstract

Ad hoc networks are dynamic environments where frequent disconnections and transient interactions lead to decoupled computing. Typically, participants in an ad hoc network are small mobile devices such as PDAs or cellular phones that have a limited amount of resources available locally, and must leverage the resources on other co-located devices to provide the user with a richer set of functionalities. Service-oriented computing (SOC), an emerging paradigm that seeks to establish a standard way of making resources and capabilities available for use by others in the form of services, is a useful model for engineering software that seeks to exploit capabilities on remote devices. This paper proposes an automatic code management system supporting SOC in ad hoc networks. The system is responsible for ensuring that the binary code required to use a service on a remote machine is available on the local host only when required. To support this functionality, a local code base is maintained by discovering and installing code from remote hosts. Since the system is specifically designed for ad hoc networks, it incorporates additional features that help it withstand the inherent dynamism of the network. We present an architecture for our system supporting automatic code management and follow it with a discussion of a Java-based implementation.

1 Introduction

The utility and convenience of mobile devices have prompted a migration from traditional desktop systems to mobile devices. The result of this migration is that mobile devices are becoming more and more ubiquitous and users are becoming increasingly reliant on them. As society embraces mobile computing technology, there is a growing pressure to provide a rich set of software that is engineered to the same exacting quality and performance standards as those for traditional systems.

Providing a comprehensive library of software on mobile, resource constrained devices is a challenge due to lack of local permanent storage space and limited memory. The problem is further exacerbated in ad hoc networks, a special class of wireless networks where the network infrastructure is borne by member hosts. The software resources available in an ad hoc network are the sum total of the resources available on member hosts, as ad hoc networks are closed, peer-to-peer networks with no access to external resources. Thus, the provision of a rich set of services is dependent on hosts in the ad hoc network sharing their resources with each other.

For hosts in an ad hoc network to be able to share capabilities with each other, it is required that they advertise their capabilities in a standard format and provide a standard way for remote hosts to interact with these capabilities. This is a non-trivial task since ad hoc networks lack standardized application level protocols for inter-host communication. Additionally, frequent disconnections and transient interactions, due to host mobility, which are characteristic of devices that participate in ad hoc networks result in decoupled computing. This further complicates the seamless sharing of resources. One solution to the twin problems of the lack of standardized protocols and host mobility is to implement a service-oriented framework using the proxy approach, as proposed in the Jini [23] model. In the Jini model, the code for an advertised capability (called a *service* in service-oriented computing terms) resides on the host that offers it, and a proxy, which can be conceptualized as a remote handle to the service, is distributed to interested clients. Proxies are advertised by the provider of a service and fully encapsulate all details of communication between client and provider. Interested clients can discover the proxy and install it locally. Once the proxy is installed, it behaves like a component of the client application. It accepts calls from the client and delegates them to some remote instance of the service (where the instance may change over time - most likely due to host mobility), thus allowing usage of a remote service as if it were a local component of the client application.

A critical requirement of the proxy model is the availability of the proxy binary code on the client. Since proxies are discovered by clients at runtime as-needed, it is very likely that the binary code required to install and execute the proxy is not available on the client machine. This code must be fetched from the service provider, installed and started in a seamless and transparent manner.

This requires that code management for proxies be fully automated. In addition to automatically discovering and installing required proxy code, the system must also possess the capability to examine the proxy code and automatically discover and acquire other external dependencies that are needed as support objects. These support objects are components that are required for correct execution of the proxy code, e.g., a streaming media player can be considered a proxy to a music broadcasting service while the codecs to interpret file formats are external components that are not explicitly requested by the client but required by the proxy to correctly fulfil its functionality of playing streaming music. Often, the external dependencies may not be found on the same host as the proxy object itself, requiring a mechanism to collate all required parts. A further issue is that this entire interaction happens in an ad hoc network, where interactions between hosts are transient. The system must hence be robust enough to handle unpredictable disconnections while a critical activity is being performed.

The proxy approach to service oriented computing has been described extensively in Jini literature and is not new in of itself. The novelty of our contribution lies in the fact that by automating code management, proxies and their supporting code can be migrated to the client transparently, eliminating the need for Java RMI [19] to invoke remote services, while at the same time not increasing the complexity of the client application code. The model we propose has features geared towards withstanding the dynamism of ad hoc networks while reliably fulfilling all the requirements of a proxy code management system. We also describe a Java based implementation of our model.

The rest of this paper is organized as follows. Section 2 covers background material on automated code management, service oriented computing and proxy-based services in ad hoc networks. Section 3 outlines our architecture for automatic code management supporting proxy-based interactions with services on remote hosts. We provide implementation details and illustrate the concept via a demo application in Section 4. We highlight additional technical challenges in Section 5 and draw conclusions in Section 6.

2 Background

The original purpose of the Proxy pattern [3] was twofold - to provide controlled access to resources and to behave as a placeholder for resources that external entities could use to interact with the resource. Recent years have seen the application of the Proxy pattern to a wide range of scenarios. In GloMop [2], the authors use proxies to optimize the interaction between remote hosts in an ad hoc network, by intercepting document traffic and removing all the formatting information from the document (and thus reducing its size) under the assumption that it is more important for the user of a mobile device to have access to the text of the document rather than use valuable resources to display it with complete formatting. In [20], the author presents a framework where proxies are used to manage load balancing among multiple participants in a distributed computing session. In [12] proxy objects are automatically generated using the Java reflection mechanism to provide a solution for local customization of externally developed code by wrapping the obtained object in a proxy-object generated at run time.

Jini [23], a model for service oriented computing, proposed by Sun Microsystems is another notable application of the Proxy pattern. Jini uses proxies as service advertisements as well as remote handles to the service. Jini was conceived primarily for large, wired networks such as company LANs and the Internet. The advent of mobile computing and ad hoc networking has motivated efforts to tackle various problems associated with delivering service-oriented computing solutions for mobile, wireless devices. Proxy based systems such as Jini are especially suited for use in such settings because of the reasons given in Section 1. However, simply porting the Jini framework to these resource-constrained devices does not yield a viable solution. The reason for this is that the resources required to run Jini-based applications are too costly for the small, mobile devices. For example, in [10], the authors identify the Java RMI [19] mechanism, which is used by Jini to interact with remote services, to be a costly performance bottleneck. The Mini [8] system, their proposed solution, uses pure IP communication in place of RMI to interact with remote services.

Another restriction of the Jini approach is the fact that it uses centralized directories to store service availability information. This is not suited to ad hoc networks, where the availability of resources is mainly influenced by the direct connectivity between hosts and connectivity is not guaranteed for more than short intervals at a time. The Web Services model is even less flexible since it uses heavyweight ontologies and descriptors in addition to standardized markup languages such as WSDL [22], RDF [21] and

DAML [7] and standardized protocols such as SOAP [24]. Such a standards oriented approach is inappropriate in ad hoc networks that exhibit heterogeneous and constrained host capabilities and resources.

Despite certain limitations as described above, proxy based models such as Jini and Salutation [14] represent good starting points when developing a SOC model for mobile devices and ad hoc networks. Migrating SOC to ad hoc networks brings with it fresh challenges and imperatives, a representative list of which may be found in [18]. Current SOC models assume high-bandwidth, reliable wired networks and powerful server hosts. These assumptions fall apart in the dynamic and demanding environment of ad hoc networks. In the absence of these assumptions, the software is required to be engineered to be robust so as to withstand the implications of frequent disconnection and decoupled computing. Structures such as directories have to be re-engineered so as to not have a central point of failure. Mechanisms and protocols such as safe distance algorithms [9], group membership [13], and disconnected routing [6] are needed to facilitate or simulate reliable communication between the client and the provider during the duration of their interaction. These represent just a few of the new challenges when moving to the ad hoc environment.

While the proxy approach mitigates many of the problems associated with engineering a SOC architecture in an ad hoc network, it adds complications in the form of systems that deploy and manage these proxies. Traditional deployment and management mechanisms are not suited for the dynamic environment of the ad hoc network and hence need to be built from scratch. For example, centralized directories for advertising the proxies need to be distributed across all hosts of the ad hoc network. Interactions between the proxy and its parent service must be handled in a seamless and efficient way. As an additional challenge, all these low level mechanics must function automatically, so that the application programmer does not have to deal with communication level issues.

In the next section, we present our architecture for automatic code management supporting SOC in ad hoc networks. The aim of this work is to propose an alternative to Java RMI in Jini for the purpose of interacting with remote services. We replace the functionality of RMI by supporting more sophisticated proxies that can accept calls from the client and tunnel them to its parent service using a specified coordination model for interaction in ad hoc networks. Allowing for these proxies means that the code for these proxies needs to be shipped on demand to the clients that require it. This could be done explicitly by the client application but this places a burden on the programmer. To simplify the task of programming client applications, we automated the entire process so that the programmer can write application code, secure in the knowledge that the system will collate and install all the required binary code as and when it is required.

3 System Architecture

In this section, we present our architecture for automatic code management supporting service oriented computing in ad hoc networks. We reiterate that in our architecture, a service consists of some application running on a *provider* host and a *service proxy* that the provider advertises. Interested clients may retrieve the service proxy and use it locally. We begin our presentation with a description of the software infrastructure required on the *provider*, followed by a description of the infrastructure required on the *client* side.

3.1 Provider Side Infrastructure

3.1.1 Proxy Advertisement

The proxy advertisement phase commences when a host explicitly advertises a capability it wants to share with other hosts. It does this by formulating a service advertisement, a detailed description of the service's functionality and performance parameters. In addition, the service advertisement contains a descriptor for the associated service's proxy class. The binary code for the service proxy is combined with the service advertisement to form a *proxy advertisement*. The proxy advertisement is registered with one or more *service directories*. Service directories are public entities that are open to all hosts. Interested clients may browse service directories for services that meet their needs.

Traditionally, service directories have employed centralized architectures with dedicated hosts in the network hosting and managing the directories. Centralized approaches are suitable for wired networks where powerful servers can host the directory, and reliable, permanent connectivity ensures prompt access to these directory hosts. However, the dynamic environment and opportunistic interactions found in ad hoc wireless networks make centralized structures ineffective. As an example, we highlight two scenarios in which a centralized directory architecture fails in ad hoc wireless settings. In the first scenario, a client may not be able to use a service offered on a nearby host because the client could not access the directory thus informing him of a candidate service's presence within his communication range. In the second scenario, a client could potentially discover the advertisement of a service which is no longer available because the host it is running on has moved away, leaving behind orphan advertisements. Due to such issues, alternate strategies are needed for ad hoc wireless networks.

To address issues introduced by the dynamic nature of the environment, our design entails a distributed architecture for service directories. Each host maintains its own local service directory. Hosts within communication range share these directories to form a *federated service directory*. A client's query spans the entire federated service directory, which is the conglomeration of local service directories of participating hosts. The content of the federated service directory is updated *atomically* with the arrival or departure of any host that has a local directory with service advertisements. The structure and content of the federated directory thus reflects any change in connectivity and *real* service availability (i.e., there are no orphan advertisements, each proxy in the service directory having a corresponding server to connect to). The mechanisms by which the service directories are merged, and by which the client query is made to span the merged service directories are implementation details and are covered at length in Section 4.

The proxy advertisement is divided into its two constituent parts (i.e., the service advertisement and the service proxy binary). The service advertisement is added as an entry in the service directory while the binary code for the service proxy is added to the code repository (a directory structure identical to the service directory but which holds binary code). When the binary code for the proxy is added to the code repository, the system inspects the file and automatically searches the provider host for any additional code (dependencies) required for the correct execution of the proxy. The dependencies are placed directly into the code repository by the system. If a dependency is not found by the system on the provider host, it is skipped under the assumption that the dependency will be available from another host in the ad hoc network. An argument can be made for packaging the proxy code as well as its dependencies into a single executable archive, e.g., JAR files. We chose not to adopt this approach for two reasons: a) In the interest of flexibility, we did not want to restrict a client to obtaining the dependencies of a proxy solely from the advertiser of the proxy. Returning to our example of the music service proxy, if the dependency is standardized codec which can be obtained from multiple hosts, the restriction is not required and is intrusive. b) Not using JAR files allows the client to request dependencies as required. If a certain branch of execution in the proxy does not require a dependency, then the client is not required to download it, thus saving valuable processor time. However, we do provide the support for packaging proxies as JAR files in case some providers wish to use this alternate approach.

3.1.2 Proxy Removal

When a provider no longer wishes to offer a service, it does so by removing its service advertisement from the service directory. This is a relatively straightforward operation because the server simply has to remove the advertisement from its own local directory. Here, we reiterate that though the service directory appears to be a single federated entity, it is in actuality the combination of all the local directories hence removing an advertisement from the federated registry is trivial if the advertisement lies in the portion of the directory that is local to the server that is removing it. Removing the advertisement ensures that no new clients can discover the service and use it. However, clients that are already using the service can continue to do so since they have a local copy of the proxy. The provider may choose to keep or remove the binary code that it placed in the code repository. However, leaving the code is not harmful as it could potentially be used by clients as dependency code.

3.2 Client Side Infrastructure

3.2.1 Proxy Discovery & Installation

The discovery process is initiated by a client that is looking for a service. The first step in the process is formulating a request. A request has two parts – a) An interface description and b) Performance attributes. The interface description is a list of methods and their signatures that the client requires the service to implement. The performance attributes are attribute-value pairs that describe the minimum requirements of the client, e.g., for a printer service a performance attribute may be [resolution, 300]. The request formulated by the client forms the *template* for the discovery process. The next step is to search the service directory. This is done by using the formulated request as a template. Any service advertisement in the service directory that matches the template (i.e., matches all the requirements of the client) becomes a *candidate service*. The system chooses one service non-deterministically from the set of candidate services and returns the advertisement for that service to the client.

Upon receipt of the service advertisement, the client extracts the name of the corresponding proxy class from the advertisement. It then tries to instantiate the proxy locally. If the binary code for the proxy is not present on the client, this step raises an exception. The system catches the exception and launches a discovery protocol to find the required binary code. This process is identical to the process used to discover service advertisement except that the search is performed over the code repository rather than the service directory. Once the binary code has been discovered and retrieved by the client, it is installed on the client host.

The installation of the binary code for the proxy can be of two types – a) A light installation in dynamic memory, which exists as long as the application is executing and b) A permanent installation which involves the binary code being written to a file on permanent storage. The choice of light or permanent installation is dictated by the client application which specifies its choice by way of a command line flag to the system at startup. Currently, our system uses the choice provided at startup for all proxies. Allowing choices to be made on a per proxy basis is a capability which we will add in the future.

3.2.2 Discovering Dependencies

Once the proxy has been installed, it is loaded into the runtime environment on the client machine. At this time, the binary code for the proxy is parsed to determine if any additional code (dependencies) is required. If any dependencies are required, the system launches the discovery protocol in the code repository to discover the binary code for the dependencies and install them on the client machine. This process is identical to the one used for discovering the binary code for the proxy as described in the previous section.

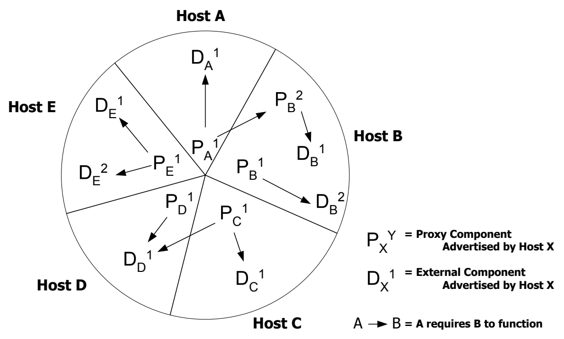

Host A
Host B
Host C
Host D
Host E
D_A^1
P_A^1
P_B^2
D_B^1
P_B^1
D_B^2
P_C^1
D_C^1
P_D^1
D_D^1
P_E^1
D_E^1
D_E^2
P_X^Y = Proxy Component Advertised by Host X
D_X^1 = External Component Advertised by Host X
A → B = A requires B to function

Figure 1. Federated code repository and its contents: proxies and their dependencies.

By its design, our architecture for discovering dependencies also supports service composition. One proxy can have among its dependencies another proxy, which could itself be a self sufficient service, offered by the same or a different provider and advertised by the same or on a different host. Though the dependency of the primary proxy may be another proxy, to the primary proxy, it is simply a dependency that is fetched and installed in the same manner as other dependencies. Figure 1 illustrates this feature. Each slice in the pie chart represents the code repository local to each host, and the entire pie represents the federated code repository. P_A^1 is the proxy of a service advertised on host A depending on D_A^1 and P_B^2. D_A^1 is a dependency that the P_A^1 proxy needs and is advertised by the server running on host A. P_B^2 is a stand-alone service which can be discovered and used independently by a client but, from P_A^1's perspective, it is just another dependency which will be treated in a similar manner to D_A^1.

3.2.3 Proxy Utilization

Once the proxy and all its dependencies have been fetched and installed on the client host, the proxy can begin executing. At this stage, the client can utilize the proxy as it would a locally available component. The client can call methods on the proxy which either resolves the request locally or tunnels the request back to the server on which its parent service resides.

While most of the time the proxy connects back to the instance of the server which published it, it is possible for one proxy to connect to another instance of the same server, running on a different host. This is particularly useful in ad hoc networks, when the client can be within proximity of different servers offering similar functionality. An example of such a scenario is the proxy for a printing service. While this could run on a user's PDA (embedded in a client application), the user could be next to different printers at different times, and the proxy could connect to the closest printer (the context awareness aspect of the problem is not within the scope of this paper, but the technical mechanism for delivering this behavior is presented).

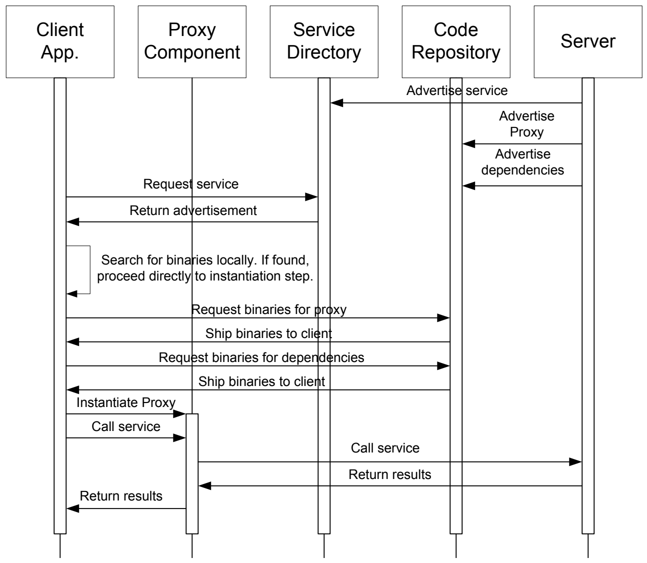


Figure 2. Proxy advertisement, discovery, installation and utilization.

The communication between the proxy and a server is entirely designed by the service provider and the client does not need to be aware of the communication protocol. This is useful as it addresses one of the key problems of ad hoc networks, i.e., the lack of standardized application protocols such as SOAP [24]. The client is only required to know the interface offered by the proxy. Since the client specifies the interface the proxy must implement in the request (See Section 3.2.1), it is reasonable to assume that such knowledge exists on the client host. In addition, the proxy-server protocol needs to address a few issues specific to the ad hoc networking environment. Among these issues are temporary disconnections which can be caused by the two hosts moving beyond communication range or by having the proxy reconnect to a different server. The client will only need to wait longer for the result of a method call to be returned, which cannot be distinguished from a simple method call that takes a long time to complete. In certain cases the proxy needs to alert the user that it has reconnected to a new provider (e.g., the printing service proxy sends the first 10 pages of a document to a printer and the other pages to another printer, along the user's way towards a meeting room). The proxy object will need to use a timer to avoid infinite blocking. When the time expires, the proxy searches for a another, similar, server. If this is not found within a specified period of time, the application informs the user that the operation cannot be performed to completion.

Figure 2 illustrates the phases of the proxy's life cycle described thus far. The server publishes the service advertisement in the service directory and deploys the binary code to support proxy's execution in the code repository. The client searches for a service and if a matching advertisement is found, it retrieves the proxy component and verifies if the proxy has all that it needs locally. If not, the infrastructure on the client machine brings the needed binaries from the code repository so that the client can instantiate and use the proxy locally. More details about the implementation can be found in Section 4.

3.2.4 Proxy Upgrade

In certain cases, the provider of a service may choose to make changes to the service being offered. For example, the provider may choose to upgrade to encrypted communication between the proxy and the server. To achieve this upgrade, not only must the software on the provider be upgraded, but the proxies that are distributed to clients must also be upgraded. Since the proxy abstracts all details of communication and interaction with the actual software on the provider from the client, any change in the proxy software requires an automatic, transparent live upgrade of software.

Updating the server is easier since it does not affect software on the client's host in any way. If the server needs to go off line for a short period (i.e., needs to be restarted to run using a newer version of the software on the provider side), the proxy can mask the short disconnection from the client as a delayed return from a method invocation. The situation is similar to the server's host moving temporarily out of range. The proxy-server interaction can be designed such that short interruptions in communication or short disconnections do not cause crashes or influence the client's performance.

Updating the proxy is more challenging since it affects the code on the client side. The procedure entails replacing a piece of code the client has access to and is actively using without affecting the clients execution flow. This raises many issues, some of which are not within the scope of this paper. The proxy upgrade mechanism, along with related issues are presented in significant detail in [17]. Due to space constraints, we mention only selected points of interest.

The upgrade mechanism uses a dynamically generated *proxy wrapper* that wraps a proxy object so that the client application does not have direct access to the proxy. The proxy wrapper employs the interceptor design pattern [15] combined with the facade design pattern [16]. During normal operation, the proxy wrapper receives calls from the client and simply forwards them to the proxy, which then handles the implementation of the method call. When the proxy needs to be upgraded, the proxy wrapper holds the client's calls until the new proxy is in place. It is important to note that the proxy's interface to the client cannot change upon upgrade since such a change could result in the new proxy not having a method that was present in the original proxy, which would raise an exception the next time the client tried to access that method. This would compromise the transparency of the upgrade. Only internal functionality or the interface to the server delivering the advertised functionality can be affected (e.g., the communication protocol can be encrypted after the upgrade but the client still calls the same method which now internally has the capacity to encrypt communication).

The last issue we address with respect to proxy upgrades is the migration of user customizations from the old version of the proxy to the new. Any user customizations of the proxy are cached by the proxy wrapper when the original proxy is initialized. When the new proxy is installed, the customizations are transferred to the new proxy as the last step of the installation. It should be observed that all user customizations should consist of static data, i.e., data that cannot be mutated by the proxy. This is because if some data was mutated by the proxy being replaced, then the cached value is inconsistent with the value on the proxy and restoring the cached initial value to the replacement proxy may affect the semantics of operation of the new proxy.

3.2.5 Proxy Disposal

Once the proxy has served its purpose, it must be disposed promptly. This is because the devices we deal with in an ad hoc network have limited available memory at any given time. If a light installation is used, existing garbage collection mechanisms reclaim the memory automatically, and our system relies on these mechanisms to free up memory in an expedited fashion. If a permanent installation is used, then a disposal manager is also started at the time the application is started. The disposal manager tracks the usage of each proxy and its associated files that are on permanent storage. If a file has not been used for a user-defined period, the disposal manager deletes the file and frees up the storage space. The user may also define a frequency with which the disposal manager checks for files that have not seen recent usage.

4 Implementation

The framework described in the previous section has been implemented in Java, using LIME [11] as a middleware to handle the implications the an ad hoc wireless network, i.e., physical mobility of hosts. In this section we present a brief overview of LIME, followed by a description of the implementation of the automated code management system. We also show a proof of concept via a set of demo proxies running on our client.

4.1 LIME Overview

LIME is a Java implementation of the Linda [4] coordination model, designed for ad hoc networks, which masks details associated with coordination and communication from the application programmer. A host running LIME runs a `LimeServer` supporting run one or more LIME agents, which are analogous to applications. Coordination in LIME occurs via transiently shared *tuple spaces*. Every tuple space in LIME is identified by a name. Tuple spaces having the same name can be merged to form a federated tuple space when their hosts are within communication range.

Tuple spaces are containers for tuples. Tuples are ordered sequences of Java objects which have a type and a value. An agent places a tuple in the tuple space, making it available to all other agents that are sharing the same tuple space. To read a tuple from the tuple space, an agent needs to provide a template, which is a description of the tuple that the agent is interested in. A template is a sequence of fields, each of which can contain a formal representing the required type for that field or an actual value that identifies the type and value of the corresponding field. A template is said to match a tuple if all the corresponding fields match pairwise.

An agent can access the tuple space via standard Linda operations (`rd` (read a tuple), `in` (remove a tuple), `out`(write a tuple)). The `in` and `rd` operations take a template as a parameter and return a tuple as the result or block until a match is found (the operations are synchronous). LIME offers probe variants of the traditional blocking operations (e.g., `inp`, `rdp`), and group operations (e.g., `outg`, `ing`, `rdg`, `rdgp`, and `ingp`). While the original calls return a matching tuple (if available) or null otherwise (if non-blocking), the group operations return all matching tuples (or null if none available).

To provide asynchronous interactions, LIME extends the basic Linda tuple space operations with a reaction mechanism $\mathcal{R}(s, p)$, defined by a code fragment s that specifies the actions to be executed when a tuple matching the pattern p is found in the tuple space. Blocking operations are not allowed in s, as they might prevent the program from reaching fixed point.

LIME protects applications from the complexity associated with sudden disconnection by using location information. The concept of safe distance [9] helps preserve the consistency of the system by predicting disconnections. When a host approaches a group, it is allowed to *engage* with the group only after it comes within safe distance of some member of the group. Once the safe distance is exceeded, an automatic *disengagement* protocol is triggered and the group is split, ensuring that no messages between group members are lost and that messages are always sent and received in the same configuration.

4.2 Implementation Details

In our implementation, we represent the client and the provider entities as LIME agents. The service directory and code repository are modelled as tuple spaces that are shared among client and server agents. The service directory contains a set of service advertisements encapsulating proxy objects while the code repository contains the binary code for the classes supporting the execution of the proxies on the client side.

When the service provider calls the `advertise(attributes, proxy)` method to publish a service, our middleware creates a tuple of the form <Attributes:attrib, ServiceProxy:proxy> and places it in the `ServiceAdvertisements` tuple space using LIME's `out` operation. The first field in the tuple contains a set of attribute-value pairs that describe the service's performance parameters and the second field contains the proxy object in serialized form. Our middleware also analyzes the code of the proxy object, recursively extracts every data type instantiated inside the proxy object, and automatically generates tuples that go in the `CodeRepository` tuple space. These tuples contain the byte code for the proxy object and for other classes that the proxy object may need when it is being instantiated. Note that such tuples are generated only for classes that are not part of the standard Java or LIME classes (which we assume are available on every host). These tuples are of the form <Names:class

names, BinaryCodeFile:bytecode>. The second field contains either a Java class file or a JAR file in the form of a byte array. The first field contains the set of names of the classes stored in the second field (the set has one element if the second field holds a class file or multiple elements if the second field contains a JAR file encapsulating multiple classes). While the byte code for the proxy object must always be advertised by the service provider, the dependent classes might be advertised by other providers.

The client searches for services by calling the `find(attributes, interface)` method. Internally this is translated to a reaction on the `ServiceAdvertisements` tuple space for the following pattern: <Attributes:attrib, ServiceInterface:interface> which is installed in LIME's reaction mechanism. A similar call is available for synchronous semantics, where instead of a reaction, the middleware passes the template to LIME's `in` operation which blocks until a matching service shows up. A candidate advertisement is selected non-deterministically from the set of advertisement tuples that match the pattern. The standard matching mechanism described in 4.1 had to be slightly altered to allow for set inclusion in tuple matching. That is, if the set of attributes required by the client is a subset of the set of attributes advertised by the provider in the first field of the advertisement tuple, a match has to be declared provided that the other conditions are met as well. The same applies for the tuples from the binary code tuple space, where the first field is a set of class names.

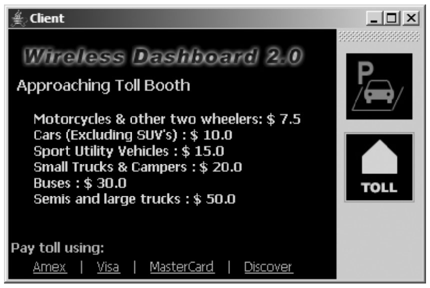
The client extracts the proxy object from the provider's advertisement and tries to instantiate it. If this step fails, the deserialization mechanism throws a `ClassNotFoundException`, and the mechanism for fetching the byte code is triggered. This is done by using a custom class loader, a custom `ObjectInputStream` that refers to this new class loader, and a slightly-modified version of LIME that uses this modified `ObjectInputStream` in place of the standard one for deserializing the proxy object. Our custom `ObjectInputStream` intercepts any failed attempts to resolve classes locally and invokes our custom `LWClassLoader`, which attempts a `rdp` operation on the code repository using the pattern <Names:class_name, BinaryCodeFile.class> with the purpose of retrieving the byte code for the required class from the code repository. The second field in the template is a formal specifying the type of data expected to be found in the second field in the tuple. If this `rdp` operation succeeds, the class loader loads the byte code contained in this proxy into memory, in the form of a byte array read out of a byte stream, and presents it to the class loader as a standard Java class. An exception is thrown only if the byte code cannot be found in the code repository. We chose to adopt non-blocking semantics when searching for byte code because blocking searched would block the class loader on the client machine indefinitely which would result in all agents on the hosts in addition to the one that initiated the call being blocked.

By using our custom class loader, we also solve the problem of class dependencies on the client. Java will note at runtime that a non-standard class loader was used to deserialize the proxy object, and it will continue to use this custom class loader whenever the proxy object refers to a class that is not already loaded into the runtime class space. Note that the middleware on the client side does not know if the server packaged the classes in a JAR or a class file and therefore has to ask for each class separately (i.e., if the server publishes all support code in tuples containing each a single class and the client is looking for one tuple containing the set of dependent classes in one JAR file, the request will never return the desired result). Note that if multiple classes are packaged in the same JAR file, when this JAR file is downloaded because of one of the classes, it will remain available locally and the other classes will not trigger the downloading mechanism since their code will be readily available. This eliminates the repeated downloading of the same JAR file. The net effect of this is that byte code fetching is done whenever a missing class is first used by the proxy, and the fetching is entirely transparent to the developer and the user. This allows the proxies to be deployed as whole components without any extra effort required on the developer's part.

Currently, the class loader stores any retrieved class byte code in memory, which it consults before attempting a `rdp` operation on the binary code repository. Though this helps minimize the number of repeated operations within a given client session, it cannot store retrieved byte code across multiple sessions. We plan to include support for a customizable persistent cache which, much like a web browser cache, is designed to save frequently-used byte code across multiple sessions in future implementations of our class loader.

4.3 Demo Application

The demo application consists of a simple client shown in Figure 3 and several services that simulate roadside services, like a tollbooth and a parking meter. When the client discovers one of these services, it adds the service's proxy (which contains its own GUI) to its main window and adds an icon to its toolbar allowing the user to switch to the newly-found proxy. In our example, the client has discovered a tollbooth and a parking meter, and placed icons representing them in its toolbar. The toll booth GUI is currently displayed in the window. Both of these proxies communicate back to the server in a similar fashion: when the user clicks one of the proxies' payment links, the proxy prompts the user for a username and password, which it uses to share an encrypted tuple space with the server. It then places a tuple in this tuple space indicating the client's payment. The server reacts to the tuple and responds to the client's payment which is shown on the GUI as an acknowledgement.


Client

Wireless Dashboard 2.0

Approaching Toll Booth

Motorcycles & other two wheelers: $ 7.5
Cars (Excluding SUV's) : $ 10.0
Sport Utility Vehicles : $ 15.0
Small Trucks & Campers : $ 20.0
Buses : $ 30.0
Semis and large trucks : $ 50.0

Pay toll using:
Amex | Visa | MasterCard | Discover

Figure 3. Client application with two proxies.

In our example, the functionality provided by the service could have been packed in one single proxy object. Since user interaction is required and the scene happens in a running vehicle, the GUI is a necessary addition to the proxy object, which expanded the set of files to be downloaded to more than the simple proxy that handles the payment. Note that the GUI described is hard coded for this service and is not related to the work presented in [1], which is related to automatic provision of GUIs in Jini services. The screen shot does not show the inner functionality of the service. It is intended to demonstrate the applicability of automated code management in SOC settings.

5 Discussion

In designing the system, we were able to leverage the capabilities of certain existing systems. The LIME coordination model, which employs tuple space-based coordination among mobile hosts in an ad hoc network provided the low level communication primitives for our system. In addition to providing communication capabilities, the `LimeTupleSpace` proved to be the ideal data structure for modelling a distributed directory, and by its semantics of containing data from hosts that were connected, we saved the overhead of having to implement a process that ensured the consistency of the directory.

Using tuple spaces for communication allows for location-agnostic protocols at the application level. As long as the recipient maintains connectivity with the ad hoc group, it can receive communications intended for it regardless of its physical location and without the need for an explicit addressing scheme such as IP address. This is achieved because tuples are read based on their content while the actual local tuple space in which the tuple is located is irrelevant. When a local tuple space is shared with others, every agent perceives only a change in the content of the tuple space it was already accessing. This behavior can span across multiple providers and clients, each unaware of the location of the other.

The transient sharing of the tuple spaces allowed for a cooperative workspace giving different providers to an opportunity to offer services, which could then be composed into larger services. For example, a fully deployed service from host A can be a simple dependency for another service offered by host B. The tuple space-mediated communication also allows for easy software updates because of the nature of the interactions it employs. The tuple space acts as an intermediary buffer between the two ends of a communication channel decoupling their interactions. This allows actions on one end without the implication/notification of the other, such as restarting a server without crashing the client because of broken socket level communication issues.

Lights [5] is another piece of software that proved useful in the implementation of our system. Lights provides the `tuple` and `template` primitives as well as matching mechanisms. Three levels of matching are provided – a) Value matching, b) Matching based on type, and c) Polymorphic matching based on type and hierarchy, which is the most powerful of the three. The polymorphic matching mechanism allows clients to search for components by specifying their requirements on a high level (e.g., interface level). The polymorphic algorithm uses Java hierarchy of objects when comparing the client specified pattern against the tuples available in the tuple space (i.e., service advertisements and proxy binaries). Using the polymorphic mechanism, any instance of a class that implements the specified interface or any other class that extends such a class, will qualify as a candidate (e.g., a `Printer` class may implement the `PrinterInterface` the client uses but `LaserPrinter` and `InkjetPrinter` can both extend the `Printer` class and thus polymorphically implement the interface and therefore qualify as candidates). Finally, the choice of Java as an implementation language allowed us to leverage certain key capabilities such as reflection, code mobility, and dynamic class loading that are built into the standard libraries.

For future work, we will address certain streamlining features for our system. Among these, we will implement a mechanism for disposing proxies that are cached on permanent storage. Currently, we provide hooks for byte code disposal, user customizable policies based on timeout, least used, availability in the network, etc. While all these are relatively simple to implement, our interest focusses on a strategy that anticipates future use of the service based on the motion profile of the client's and provider's hosts (e.g., if the motion profile indicates that the two hosts will not come in contact again, the proxy will not be able to connect to its server and therefore will become a candidate for disposal; while the same proxy may connect to a different provider offering a similar service, the future availability of such a provider may not be available to the decision algorithm). We will also investigate the implications of migration (i.e., logical mobility of a service or the client application from one host to another) and the guarantees that we can provide for uninterrupted service under those conditions, and whether such migration can be transparent to an external viewer. Our ultimate goal is to integrate this piece of software into a comprehensive middleware for mobile devices supporting service oriented computing in ad hoc networks.

6 Conclusions

In this paper, we presented an architecture for automatic code management supporting service oriented computing in ad hoc networks. Our presentation covered various aspects of the software infrastructure required on the provider and client hosts such as advertisement, discovery, installation, utilization, upgrade and disposal of binary code for service proxies. Special attention was given to making the architecture robust enough to deal with the dynamism of ad hoc network such as host mobility and resultant opportunistic interactions. The result is a system that significantly simplifies the task of a mobile application programmer by abstracting all details of code management. This allows the programmer to develop applications at higher levels of abstraction, leveraging the power of the code management system to handle the low level mechanics of remote service usage in ad hoc wireless environments.

Acknowledgements: This research was supported by the Office of Naval Research under MURI contract N00014-02-1-0715. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not represent the views of the research sponsors.

References

[1] Aritma Software. ServiceUI. http://www.artima.com /jini/serviceui.

[2] Armando Fox and Steven D. Gribble and Eric A. Brewer and Elan Amir. Adapting to Network and Client Variability via On-demand Dynamic Distillation. In *Proceedings of ASPLOS VII*, pages 160–170. ACM press, 1996.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*, chapter 4, pages 207–219. Addison Wesley, 1994.

[4] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[5] Gian Pietro Picco. LighTS Webpage. http://lights.sourceforge.net/.

[6] R. Handorean, C. Gill, and G.-C. Roman. Accommodating Transient Connectivity in Ad Hoc and Mobile Settings. In *Proceedings of The Second International Conference on Pervasive Computing (Pervasive 04)*, 2004.

[7] I. Horrocks. DAML+OIL: A Description Logic for the Semantic Web. *IEEE Bulletin of the Technical Committee on Data Engineering*, 25(1), March 2002.

[8] P. Huang, V. Lenders, P. Minnig, and M. Widmer. Mini: A Minimal Platform Comparable to Jini. In *International Symposium on Distributed Objects and Applications*, 2002.

[9] Q. Huang, C. Julien, and G.-C. Roman. Relying on Safe Distance to Achieve Partitionable Group Membership in Ad Hoc Networks. *Accepted to be published in IEEE Transactions on Mobile Computing*.

[10] V. Lenders, P. Huang, and M. Muheim. Hybrid Jini for Limited Devices. In *Proceeding of the International Conference on Wireless LANs and Home Networks*, pages 27–34, 2001.

[11] A. Murphy, G. Picco, and G.-C. Roman. LIME: A middleware for physical and logical mobility. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, pages 524–533, April 2001.

[12] K. Renaud and H. Evans. JavaCloak: Reflecting on Java Typing for Class Reuse Using Proxies. In S. M. A. Yonezawa, editor, *Proceedings of REFLECTION 2001*, volume 2192. Springer-Verlag Heidelberg, 2001.

[13] G.-C. Roman, Q. Huang, and A. Hazemi. Consistent Group Membership in Ad Hoc Networks. In *Proceedings of 23rd International Conference on Software Engineering*, pages 381–388, 2001.

[14] Salutation Consortium. Salutation web page. http://www.salutation.org.

[15] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture*, volume 2, chapter 2, pages 109–141. John Wiley and Sons Ltd., 2000.

[16] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture*, volume 2, chapter 2, pages 47–75. John Wiley and Sons Ltd., 2000.

[17] R. Sen, R. Handorean, G. Hackmann, and G.-C. Roman. An Architecture Supporting Run-Time Upgrade of Proxy-Based Services in Ad Hoc Networks. In *To appear in the Proceedings of the International Conference on Pervasive Computing and Communications PCC-04*, 2004.

[18] R. Sen, R. Handorean, G.-C. Roman, and C. Gill. *Service Oriented Software Engineering: Challenges and Practices*. Idea Group Publishing, To appear in 2004.

[19] Sun Microsystems. Java Remote Method Invocation. http://java.sun.com/products/jdk/rmi.

[20] D. Thissen. Flexible Service Provision Considering Specific Customer Resource Needs", booktitle =.

[21] W3C Semantic Web Activity. Worldwide Web Consortium Page on Resource Description Framework. http://www.w3.org/RDF.

[22] W3C XML Activity On XML Protocols. W3C Recommendation: Web Services Description Language 1.1. http://www.w3.org/TR/wsdl.

[23] J. Waldo. The Jini Architecture for Network-Centric Computing. *Communications of the ACM*, 42(7):76–82, 1999.

[24] XML Protocol Working Group. W3C Recommendation: SOAP version 1.2 Parts 0-2. http://www.w3.org/TR/SOAP.