

# A Component Deployment Mechanism Supporting 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  
{radu.handorean, rohan.sen, ghackmann, roman}@wustl.edu

**Abstract.** Ad hoc networks are dynamic, open environments that exhibit decoupled computing due to frequent disconnections and transient interactions. Reliable deployment of components in such demanding settings requires a different design approach for the mechanisms that perform these functions. Not only do the deployment mechanisms have to perform the traditional tasks of deploying, installing, integrating and activating components, they must also be robust enough to handle the nuances of an ad hoc network. This paper proposes a mechanism for component deployment that is adapted for use in ad hoc networks, and, as such, can cope with the effects of disconnection and transient connectivity. We present the general architecture for the mechanism and follow it with a discussion of a Java-based implementation of the model.

## 1 Introduction

The continually improving performance of mobile computing devices has stimulated a migration from traditional desktop systems to mobile devices. However, the expectations of performance of mobile devices is also rising at a similar, if not greater rate. This rising expectation has fuelled a demand for a range of functionality and quality of service on mobile devices that are commensurate with those available on wired networks.

Providing a wide-ranging set of services is especially challenging in ad hoc networks, a special class of wireless networks which are open environments where the network infrastructure is borne by member hosts. The total amount of resources in any ad hoc network is the sum of the resources of its constituent members. Hence to offer a diverse range of functionality to members of the network, each member must share its capabilities with other members. The sharing of resources requires all member hosts to advertise their individual capabilities in a uniform manner and to provide a uniform way of interacting with any capability that is available on a remote host.

In addition to requiring uniformity, sharing of resources takes on an added complexity in ad hoc networks due to the lack of standard protocols and due to

the physical mobility of hosts, i.e., frequent disconnections and transient interactions. In most cases, it is more practical to have the capability reside on the host that offers it (as opposed to migrating from host to host) and to provide a simple mechanism through which interested parties can interact with that capability. One way of achieving the twin goals of uniformity and protocol independence is to implement a service oriented framework using the proxy approach [1]. In this approach, the code for the service resides on the host that offers it and a software component in the form of a *proxy* is distributed to clients.

Proxies can be conceptualized as handles to remote services. They are advertised and deployed by the service provider and once installed and activated on the client side, behave like components of the client side application, representing the service locally. They accept calls from the client application and delegate them to the remote instance of the service, thus allowing usage of a remote service as if it were a local component of the client application.

A successful deployment of a proxy component entails deploying a primary proxy component and, often, other external components that are needed as support objects. They 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 components may not be found on the same host as the proxy component itself, requiring a mechanism to collate all required components.

The proxy model of using a component on the client side to control a service is not new in of itself. The novelty of our approach lies in the mechanism we use to deploy the initial proxy component, in the mechanism via which we discover dependencies that the proxy component requires, in the technique we use to install and integrate the proxy within the client application and in the manner in which we dispose of the proxy once it has fulfilled its function. This paper proposes a model for the deployment of proxy components supporting service oriented computing in ad hoc networks. The model we propose has features geared towards withstanding the dynamism of ad hoc networks while reliably fulfilling all the requirements of a successful proxy deployment. 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 component deployment and services in ad hoc networks. Section 3 outlines our model for deployment of proxies supporting 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

In this section, we present a selection of developments in the field of component deployment and highlight some research that focusses on non-traditional uses for

component-based application building. We observe that the models and systems reviewed are geared towards wired networks. Accompanying this observation is a summary of challenges we face in designing a Service Oriented Computing (SOC) framework for ad hoc networks and reasons why we chose to adopt the proxy approach for our solution to the problem.

## 2.1 Current Strategies for Component Deployment

Component oriented computing is a paradigm in which applications are assembled from a set of software components. The strategy of marshalling components into cohesive applications and deploying them is a mature one and a significant amount of research has been done in this field. Component oriented computing has been used in a variety of settings. Enterprise Java Beans (EJBs) [2] is a component technology developed by Sun Microsystems as part of the J2EE standard that has gained widespread acceptance for use in commercial enterprise-level applications. Web application servers like Weblogic [3], WebSphere [4], JRun [5] and ColdFusion [6] each provide their customized bean packaging and deploying tool that deploys components in the form of EJBs into bean containers where they may be accessed by various applications. Microsoft Corp.'s Component Object Model (COM) is another component technology that is popular in the enterprise application market.

The Corba Component Model (CCM) [7] is part of the Corba 3.0 specification and is a component model for building server-side CORBA applications. Like EJBs, they exhibit standard interfaces that allow them to be connected seamlessly with other components. The CCM specification is simply a model though implementations such as OpenCCM [8] are available for use. In [9], the authors propose a deployment mechanism for CCM that uses a multi-step process to deploy components. First, the chosen components are assembled into an assembly archive. Next, the archive is read and an identifier generated for it. This is followed by installation of required libraries and connection of components. Finally, a reference to the completed component is returned.

Components have also been used in settings other than enterprise applications. In [10], von Laszewski et. al. suggest three deployment mechanisms—thick, thin and slender, depending on how much software is carried on the client—for computing grids, which are complex infrastructures that allow scientists to use distributed software, services, and components that access a variety of dispersed resources. Various local code repositories are integrated to form a virtual code repository from which components are selected and assembled into a meaningful application. In TACOMA [11], mobile agents deploy components for the purpose of updating software. An agent possesses a list of servers that it is responsible for keeping up to date. It collects information about updates required on the initial server and then migrates itself to another server on its list until all of them have been covered. After this is done, another agent deploys the components required for the update to the servers in a similar manner.

Rudkin and Smith [12] describe an architecture for deploying components for services, which are “functions delivered to users through the execution of

software components.” They describe an Application Description Transformer which combines a component list, a session description and a user profile to generate an application description which is then forwarded to an application builder that requests all the required components and dynamically assembles a customized application for the session in question.

EJB, COM and CCM are standards that are used in enterprise level systems. The assembly and deployment mechanisms for components obeying these standards is geared for use in wired networks with high bandwidth, permanent connectivity and on powerful servers with high computational capability. As such, they are heavyweight mechanisms. The work by von Laszewski et. al. on Computational Grids is of a similar flavor. Though they describe three different kinds of deployment, they focus on the amount of code carried by the client. However, the mechanism for deployment still remains fairly heavyweight. Similarly, Rudkin and Smith too suggest a complex architecture for deploying components for services.

All these approaches are tailored for use in wired networks. As such, the implementations are not appropriate for ad hoc networks where hosts are resource poor, bandwidth is limited and connectivity is not guaranteed for more than short intervals at a time. The TACOMA model addresses some of these issues by having the deployment carried out by a pair of lightweight agents. However, the model does not address the issues inherent in ad hoc networks. For example, if the state collecting agent migrated to a host that then went permanently out of range of all other hosts, the process of collecting state information would be permanently blocked and the state updates could not occur.

Our work focusses on designing a lightweight component deployment model supporting services which is engineered to deal with the nuances of an ad hoc network, i.e., frequent disconnection and transient connectivity. In the next subsection, we briefly discuss the challenges of providing services over ad hoc networks and justify our key design decision, the use of proxy components to control services on remote hosts in the network.

## 2.2 The Transition to Ad Hoc Networks

Service oriented computing is a paradigm that is fast gaining acceptance due to its promise of unhindered interoperability. In a SOC framework, a *provider* offers a *service*, which is some functionality wrapped in an interface familiar to the *client* that avails of the service. While numerous implementations of service oriented computing frameworks exist for wired networks, there has been relatively little attention being paid to designs and implementations of this framework for wireless networks, specifically ad hoc networks. Migrating SOC to ad hoc networks brings with it fresh challenges and imperatives, a list of which may be found in [13]. Assumptions that are valid in the wired setting fall apart in the dynamic and demanding environment of ad hoc networks. For example, in wired networks, standard application level protocols like HTTP exist to facilitate communication between hosts. However, such standards do not exist for ad hoc networks which are by nature very heterogenous.

Ad hoc networks are open environments comprising of member hosts. Hence, there are no standard application level protocols that govern communication between hosts, unlike the Internet where standards such as HTTP allow uniform communication between the client browser and a web server. Therefore, for a client to be able to communicate with a server, it must know *a priori* the protocol that the server uses. If the client needs to communicate with multiple servers, it must know the protocols for each of those servers. This is impractical because disseminating knowledge of the protocols requires an external channel and the storing protocol information could consume valuable space on the client device.

The solution to this problem is the proxy approach. A proxy can be conceptualized as a handle to a service that resides on a remote host. The proxy is designed and implemented by the service provider and deployed to clients that are interested in using the service. The proxy approach encapsulates details associated with the protocol used between the service provider and client. This is useful since it significantly simplifies the development effort and code base for client side applications and eliminates any need for a priori knowledge of the protocol, an important asset, given that the client is usually a resource poor device.

While the proxy approach mitigates many of the problems associated with providing a service oriented computing framework in ad hoc networks, it does raise technical issues related to the deployment of the proxy. The environment of an ad hoc network requires a fresh approach to deploying these proxy components. Traditional directories and implementation repositories do not measure up to the task. The dynamism of the network dictates that directories and repositories be transiently shared. In addition, since ad hoc networks are open environments, it is essential to design the proxy component in such a way that it can be deployed, installed, instantiated and activated seamlessly within a wide range of potential client applications that may use it.

### 3 Architecture for Component Deployment

In our framework a service consists of an application running on a server host and a proxy object the server advertises for clients. The proxies are retrieved and used locally by the clients. They represent remote handles to the actual service. In some cases, the entire functionality of the service can be delivered by the proxy itself, without the need to connect to any server. However, all services need to advertise their proxy in order to be contacted and used by clients.

#### 3.1 Component Release

The release phase is the initial phase when the component is first made available. All service providers are required to register their proxies and the associated service's performance parameters with *service directories* since this is the only way the clients can contact them. Service providers register their proxies with

service directories and clients browse these directories in search of components that meet their needs.

Traditionally, centralized architectures have been employed with service directories running on dedicated hosts whose purpose is to manage the directory. While the approach is appropriate for wired networks, the dynamic environment and opportunistic interactions found in wireless networks require alternate strategies. As an example, we highlight two scenarios in which a centralized directory architecture fails in 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.

In our design, a proxy registration consists of an entry in a service directory and one or more entries in a binary code repository. The service directory entry contains the proxy and a description of its performance parameters. The code repository holds the binary code for the proxy and other components the proxy needs for its proper execution (dependencies). These dependencies represent additional code the proxies may depend on and may not be able to find on the clients host. To address issues introduced by the dynamic nature of the environment, our design entails a distributed architecture for the service directory as well as for the binary code repository. Since the model and the implementation are similar for the service directory and for binary code repository, we will only refer to the service directory hereafter.

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).

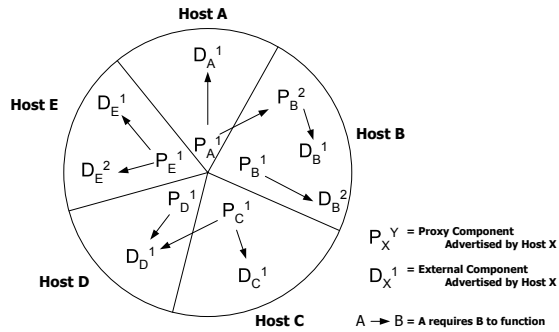
### 3.2 Component Install

The installation phase occurs on the client side, where our framework performs all the preparatory steps for the successful utilization of the component. When the client searches for a service it explicitly requests the proxy component, since the service directory is the only directory the client can access and knows about. The client browses the service directory for components which implement an interface known a priori by the client and which advertise performance parameters that meet clients requirements. When retrieving the proxy, the framework, in a manner transparent to the client, decides if the proxy needs any additional components not available on the clients host. These dependencies are fetched

from the binary code repository and installed before the proxy is made available for use by the client.

Though the dependencies are fetched after the proxy has been deployed to the client host, the mechanism employed is similar to the one used to fetch the initial proxy component. They are published in the same federated implementation repository but they are not subject to browsing performed by clients. Their availability is also updated in sync with real-time changes in connectivity.

Our framework design also provides mechanisms that support 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 on 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



**Fig. 1.** Federated component repository and its contents: proxies and their dependencies.

fetches and installed in the same manner as other dependencies. Figure 1 illustrates this feature. Each slice in the pie chart represents the implementation repository local to each host, and the entire pie represents the federated component 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 component 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 component which will be treated in a similar manner to  $D_A^1$ .

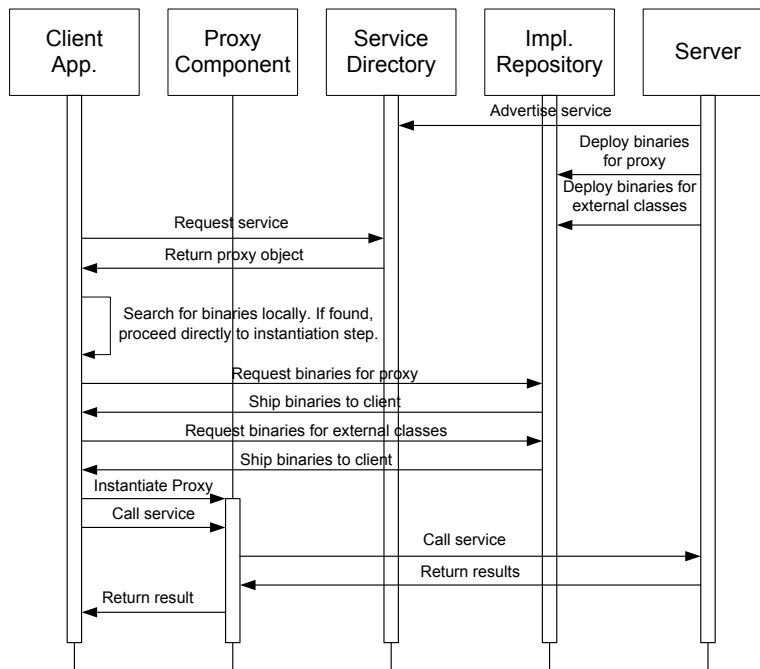
Due to the dynamic nature of the environment, we anticipate there is a low likelihood that a client will need and use a service for more than a short period of time. The devices that we designed our infrastructure for have limited resources and storage space is at a premium. Therefore it is imperative that we minimize its occupancy. In addition to this, the code needs to be loaded in memory before it can be used. We employ a lightweight installation of components, directly into the client device's memory. (Details are provided in the Implementation section.) Other benefits of this approach are discussed in a later subsection.

### 3.3 Component Utilization

The utilization of the component is the stage where the client calls methods the component implements. Once the proxy is fetched and installed, the client can use it as it would a locally available component. At this point, all the binary code needed for the correct utilization of the proxy (including the code the proxy

needs without client's explicit knowledge) has been correctly downloaded and installed. The client calls methods on the proxy and the proxy 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 will connect 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 of 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 in a department 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).



**Fig. 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. The only knowledge the client must possess is the interface offered by the proxy, which cannot change (see next section on component updating).

The proxy-server protocol needs to address a few issues specific to the ad hoc networking environment. Among these issues we mention 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 is indistinctive 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 client informs the user that the operation cannot be performed to completion.

Figure 2 illustrates the phases of the component's life cycle described thus far. The server publishes the proxy in the service directory and deploys the binary code to support proxy's execution in the implementation 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 framework in clients machine brings the needed binaries from the implementation repository and so that the client can instantiate and use the proxy locally. More details about the implementation can be found in Section 4.

### 3.4 Component Update

In the update phase, the service provider upgrades the components in a manner transparent to the client, thereby improving the quality of service the component offers. The maintenance process in the described framework addresses the issue of a live upgrade of software while hiding the extra complexity associated with this process from the client. Updates can occur on the providers side (i.e., the server is updated) or on clients side (i.e., the proxy is updated).

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 some newly deployed components on the server side by the provider), the proxy can mask the short disconnection from the client as a delayed return from 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. While the framework doesn't explicitly handle server updates, it offers support to the service designer to alleviate the consequences of the server update.

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. To overcome this, we propose the use of the interceptor design pattern [14] combined with the wrapper facade design pattern [15]. An extra layer, the wrapper-interceptor layer, between the proxy and the client, receives the calls from the client and normally

just forwards them to the proxy, which then handles the implementation of the method call. When the proxy needs to be updated, the interceptor will hold the client's calls until the new proxy is in place. Without this extra layer, it would be impossible to swap the proxies while in use by the client, without actively involving the client in the procedure. This feature is currently under further investigation and not supported in this version of our implementation.

It is again important to notice that the proxy's interface to the client (to the wrapper layer actually) cannot change upon 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 in after the upgrade but the client will call the methods the same way it used to before the update).

### 3.5 Component Remove

The removal phase is the last phase in component's lifetime and is related to the proper disposal of the component once it is no longer required. By default, the framework performs a light installation of components, i.e., in the client host's memory and not on permanent storage media. We took this decision as the short period of time that the service is used for does not warrant the expense of storage on permanent media. Our light installation allows for the already-in-use garbage collection mechanisms to remove unnecessary components like any other unused memory. Because of the limited storage space on the mobile devices, it is more convenient for the clients to search for and retrieve the proxy again than to use permanent storage space for a short-lived interest in a service proxy. This also discards the dependencies that were loaded along with the proxy.

On the server side, the advertisement is removed 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.

## 4 Implementation

The framework described in the previous section has been implemented in Java, using LIME [16] as a middleware to handle the implications of 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. 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 [17] 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` upon which 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 pattern describing 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). To provide asynchronous interactions, LIME offers a reaction mechanism. An agent can declare interest in a tuple by registering a reaction on a tuple space using an appropriate template. If multiple candidate tuples exist for a given reaction template, one is chosen non-deterministically from the set.

## 4.2 Implementation Details

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

The service directory tuple space contains service advertisement tuples of the form `<Attributes:attrib, ServiceProxy:proxy, ImageIcon:icon>`. The implementation repository tuple space contains tuples of the form `<String:class name, ClassFile:bytecode>`. The server generates these on initialization based on a list of classes passed to it. It is assumed that this list contains all classes used by the proxy that are not in the standard Java class library or the Lime class library, including the proxy class itself.

The client finds services by specifying an interface that the service is required to implement as well as a set of performance attributes, which the service must meet. This specification is translated into a pattern which is provided to LIME's reaction mechanism. A candidate advertisement is selected non-deterministically from the set of advertisement tuples that match the pattern. The proxy component contained in this advertisement is downloaded to the client. Since it is quite likely that the client does not have the bytecode for the proxy locally, it is

necessary that the client be able to treat the proxy as a self-contained component and dynamically download bytecode at runtime. This is accomplished 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. It then invokes our custom `LWClassLoader`, which attempts a **rd** operation on the implementation repository using the pattern `<class_name, ClassFile:bytecode>` with the purpose of retrieving the bytecode for the required class from the repository. If this **rd** operation succeeds, the class loader loads the bytecode contained in this proxy into memory and converts it into a standard Java class. An exception is thrown only if the bytecode cannot be found in the implementation repository.

By using our custom `ObjectInputStream`, 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. The net effect of this is that bytecode 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 bytecode in memory, which it consults before attempting a **rd** operation on the binary code repository. Though this helps minimize the number of repeated operations within a given client session, it cannot store retrieved bytecode across multiple sessions. We envision that future implementations of the classloader could use a fixed-size persistent cache much like a Web browser cache to save frequently-used bytecode across multiple sessions.

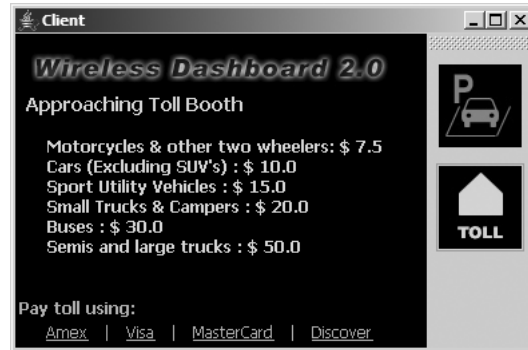
### 4.3 Demo Application

The demo application consists of a simple client shown in fig 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.

To ensure that the client is lightweight, the proxy rather than the client is responsible for providing all major functionality, such as GUI and communication protocol. However, a small number of hooks have been added from the lightweight client application to the client agent to allow interaction with tuple spaces (the design of LIME dictates that tuple spaces can only be created and manipulated within an agent). These hooks allow the proxy to make requests to the client agent to perform certain LIME level actions on its behalf. These hooks are entirely generic, allowing the implementor of the service and proxy to dictate

the protocol used for communication between client and server. There are also convenience hooks, e.g., certain hooks can allow the proxy to request a username and password pair, which can be used for authentication and/or encryption.

Figure 3 shows a sample execution of the client. It has discovered two roadside services, 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 into this tuple space indicating the client's payment. The server reacts to this tuple and responds to the client's payment which is displayed on the GUI as an acknowledgement.



**Fig. 3.** Demo application showing two discovered proxies.

## 5 Discussion

The LIME coordination model, which employs transient sharing of tuple spaces among mobile hosts is the basis of our model, which proposed distributed service directories and implementation repositories. The choice of Java as an implementation language allowed us to leverage off certain key capabilities such as reflection and dynamic class loading that is built into the standard libraries. The exchange of tuples via tuple spaces promoted location-agnostic communication protocols at the application level. This is achieved because tuples are read based on their content and 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.

The transient sharing of tuple spaces facilitates easy interactions among services offered by different providers, promoting service composition. For example, a fully deployed service can be a simple dependency for another service. This behavior can span across providers, each unaware of the location of the other. If the communication between components is also conducted via tuple spaces (e.g., the proxies and their servers use a tuple space-based protocol), migration of services and clients can occur in a manner transparent to the ongoing computation. As one end of a tuple space-based communication relocates, it will simply reconnect to the federated tuple space from the new location (if still mutually

reachable) and the communication can resume, as the location (e.g., IP address) never appears explicitly in the agent-tuple space interaction.

The tuple space-mediated communication simplifies software updates by offering a high degree of decoupling. The tuple space acts as an intermediary buffer between the two ends of a communication channel decoupling their interaction. This allows actions at one end without the implication/notification of the other, such as restarting a server without crashing the client because of broken socket level communication.

Another feature of LIME that proved useful in our model was the template-tuple matching mechanism. It allows clients to search for components by specifying their requirements at a high level (e.g., the clients can search for a service that implements an interface that they understand instead of searching for a components based on lower level details). The Java hierarchy of objects and polymorphism is used when comparing the client specified against the components available in the tuple space—the proxy objects in the service directory. Any proxy 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). The additional information provided in the template ensures that the framework filters out any results that do not meet the client’s criteria (e.g., pages per minute, in a printer service advertisement).

## 6 Conclusions

In this paper, we presented an framework for component deployment supporting service oriented computing in ad hoc networks. Special attention was given to making the framework robust enough to deal with the dynamism of the ad hoc network including host mobility and opportunistic interactions. We began by proposing a distributed approach to constructing service directories and implementation repositories. We followed this with a mechanism that can deduce the dependencies of a given proxy component and fetch the required binaries from remote locations. We also illustrated the model for a lightweight client as well as additional functionality such as on-the-fly component updating. Finally we showed how the proxy components can be disposed by existing garbage collection mechanisms.

### Acknowledgements

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

## References

1. Edwards, K.: Core JINI. Prentice Hall (1999)
2. Sun-Microsystems: Enterprise java beans product page. (<http://java.sun.com/products/ejb/>)
3. BEA: Bea weblogic service product page. (<http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/server/>)
4. IBM: Ibm websphere product page. (<http://www-306.ibm.com/software/inf01/websphere/index.jsp?tab=highlights>)
5. Macromedia: Macromedia jrun product page. (<http://www.macromedia.com/software/jrun/>)
6. Macromedia: Macromedia coldfusion product page. (<http://www.macromedia.com/software/coldfusion/>)
7. Object-Management-Group: Corba component model page. (<http://www.omg.org/technology/documents/formal/components.htm>)
8. OpenCCM-Project-Team: Opencm product page. (<http://opencm.objectweb.org/>)
9. Barros, M.C.B., Madeira, E.R.M., Sotoma, I.: An experience on CORBA component deployment. In: Proceedings of the Sixth International Symposium on Autonomous Decentralized Systems (ISADS'03). (2003)
10. von Laszewski, G., Blau, E., Bletzinger, M., Gawor, J., Lane, P., Martin, S., Russell, M.: Software, component and service deployment in computational grids. In Bishop, J., ed.: Proceedings of the 1st International Conference on Component Deployment. Number 2370 in LNCS, Springer-Verlag (2002) 244–256
11. Sudmann, N.P., Johansen, D.: Software deployment using mobile agents. In Bishop, J., ed.: Proceedings of the 1st International Conference on Component Deployment. Number 2370 in LNCS, Springer-Verlag (2002) 97–107
12. Rudkin, S., Smith, A.: A scheme for component based service deployment. In Linnhoff-Popien, C., Hegering, H.G., eds.: Proceedings of Third International IFIP/GI Working Conference, USM 2000. Number 1890 in LNCS, Springer-Verlag (2000) 68–80
13. Sen, R., Handorean, R., Roman, G.C., Gill, C. In: Service Oriented Software Engineering: Challenges and Practices. Idea Group Publishing (To appear in 2004)
14. Schmidt, D., Stal, M., Rohnert, H., Buschmann, F.: 2. In: Pattern-Oriented Software Architecture. Volume 2. John Wiley and Sons Ltd. (2000) 109–141
15. Schmidt, D., Stal, M., Rohnert, H., Buschmann, F.: 2. In: Pattern-Oriented Software Architecture. Volume 2. John Wiley and Sons Ltd. (2000) 47–75
16. Murphy, A., Picco, G., Roman, G.C.: LIME: A middleware for physical and logical mobility. In: Proceedings of the 21<sup>st</sup> International Conference on Distributed Computing Systems. (2001) 524–533
17. Gelernter, D.: Generative communication in Linda. ACM Transactions on Programming Languages and Systems **7** (1985) 80–112