

An Architecture Supporting Run-Time Upgrade of Proxy-Based Services in Ad Hoc Networks

Rohan Sen, Radu Handorean, 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

Ph: +1-314-935-7536, Fax: +1-314-935-7302

Email: {rohan.sen, radu.handorean, ghackmann, roman}@wustl.edu

Abstract—In the proxy approach to Service Oriented Computing, a service advertises a proxy, which is searched for, retrieved and used by interested clients as a local handle to the service process that runs on a remote host. Due to software evolution, it becomes necessary at times to upgrade the service. Some of these upgrades may require an upgrade of the proxy software, in addition to the server itself. This paper addresses the issue of upgrading both the server and its proxy in a manner transparent to the client, and ensures only momentary interruption during the switching process. The model we propose is designed for ad hoc wireless networks, but can be used in other settings as well. We also describe a Java implementation of our model.

Keywords: Service oriented computing, proxy, live upgrade, ad hoc networks

Presenting Author: Radu Handorean

I. INTRODUCTION

Service Oriented Computing (SOC) is an emerging paradigm that seeks to promote interoperability of systems by providing a framework for seamless software-mediated integration of heterogeneous systems. In a SOC framework, a *service* represents some functionality that is advertised by a *provider*. The framework compares *requests* generated by *clients* with service advertisements to identify suitable matches. Most of the research to date has focused on developing SOC frameworks for wired networks such as the world wide web, where network topology changes and disconnections are infrequent. However, not much attention has been paid to developing SOC frameworks for wireless ad hoc networks.

Our previous work focused on developing a viable and reliable SOC framework for ad hoc wireless networks, a special class of wireless networks where the network infrastructure is supported by participating hosts. Ad hoc networks exhibit decoupled computing due to frequent disconnections and transient interactions. Software that is designed for ad hoc networks must meet the challenges imposed by the inherent volatility of the network. In [1] and [2], we outlined an architecture for SOC based on the proxy model introduced by Jini [3] but adapted to the nuances of ad hoc networks. In Jini, the provider augments the service advertisement with a

proxy object. Once the client requests and finds a candidate service, the proxy object is extracted from the advertisement by the client and used as a local handle to interact with the service which resides on a remote host. The proxy approach ensures that all details of communication between the client and server hosts remain hidden from the client application.

In the basic proxy model, once the proxy is installed on the client, it is assumed to continue to function with no changes to its code until the end of its lifespan. However, there are certain scenarios where the proxy software may need to be upgraded before its lifespan terminates. For example, if the server is upgraded to support secure communication with its proxies by encrypting the messages, the proxy software also needs to be upgraded to support the enhanced functionality. Performing such upgrades introduce certain technical challenges. Examples of such challenges are ensuring that the upgrade takes place in a manner transparent to the client while minimizing the downtime of the proxy and ensuring that when the server side software is upgraded, requests from proxies that have yet to be upgraded can still be handled (since it is unreasonable to expect that the server and all its proxies can be upgraded in a single atomic step). Additional constraints are imposed by the ad hoc networking environment, where hosts are generally resource-poor, requiring that all the above functionality be achieved in a lightweight fashion.

This paper augments the model described in [2] with a service update mechanism that can dynamically upgrade the server as well as its proxies on client hosts. We designed a lightweight mechanism that upgrades proxies on the client hosts by replacing them with newer versions. Transparency is achieved by employing a dynamically generated facade to temporarily hold calls from the client application while the old version of the proxy object is swapped for the new one. We ensure that there are no orphan calls as a result of swapping the service software by employing a tuple space based communication protocol which stores calls temporarily in a federated tuple space if it is determined that the server is off line. These calls are picked up by the new version of the server and since we require newer versions of the server to be backwards compatible, it can service the calls and return the

result to the client.

The rest of the paper is organized as follows: In Section II, we examine some of the related work. In Section III, we introduce and describe our architecture for proxy upgrade in ad hoc networks. We provide details of a Java based implementation of our architecture in Section IV. Design decisions and the merits of our approach are described in Section V. Conclusions appear in Section VI.

II. RELATED WORK

The proxy object that is installed on and used by the client application as a handle to the service is analogous to a stand-alone component that fits modularly into a larger application. Hence, the technical problems associated with upgrading a proxy are similar to those encountered with upgrading components within an application. In [4], Hicks, Moore, and Nettles posit that it is a challenge to achieve a balance between flexibility, correctness, ease of use, and low overhead. Hence, it is necessary to place emphasis on those properties that are crucial in the context that an update framework is intended to function. For example, in large scale enterprise systems, where there are reliable, high bandwidth connections and large-scale servers, low overhead becomes less of a concern. Thus, approaches for component upgrading in wired networks have a distinctive heavyweight flavor.

The approach described in [5] proposes an upgrade server that holds all upgrades. When an upgrade is added to the upgrade server, it notifies an upgrade layer which in turn notifies an upgrade manager which downloads the upgrade and installs them as necessary. This approach works in wired networks where a centralized upgrade server can be easily accessed but falls apart in the ad hoc setting where no such centralized entity exists. In [6], Cook and Dage suggest maintaining both the old and new versions of the component concurrently and sending a call to the version that it applies to. The older version is destroyed only when it is verified that the new version correctly replaces the old version for all required functionality. This approach is not very scalable since multiple versions can run at the same time in an unbounded manner. In ad hoc networks where devices are resource poor, there is limited scope to run multiple instances of a service for extended periods of time. Flexible software connectors, as proposed in [7] do not use multiple servers. Instead, the connectors (called multi-versioning connectors) themselves determine correct points during execution when components may be swapped and subsequently swap them with newer versions.

There also exists a fair amount of infrastructure supporting component upgrading. Brada [8] suggests a mechanism for consistency checks to ensure that the new component works with all the other old components. Mencl, Petrova, and Plasil [9] propose an upgrade definition language to identify and keep track of updates. All such mechanisms, while useful can significantly detract from the ability to provide a lightweight framework, which is essential if to working on resource poor devices in ad hoc networks. Hence, it is our aim to develop

a model that provides as many of the features outlined above as possible, while still maintaining a small footprint. In some areas, we eliminated the need for some features by making a specific set of design decisions. Our model is described in detail in the next section.

III. ARCHITECTURE FOR SERVICE UPGRADE

In this section, we describe our model for upgrading services in ad hoc networks. We begin with an overview of the basic SOC model as proposed in [2]. We follow this overview with a discussion of the issues that arise when upgrading a service and the description of our service upgrade mechanism in two parts: the mechanism for upgrading the proxy, which represents the service locally to the client and the mechanism for upgrading the server, which delivers the functionality of the service.

A. Architecture Overview

In our framework, we conceptualize a service as an application that runs on a server host and a proxy object that the server advertises for clients. Interested clients retrieve the proxies and install them as local handles to the service, which resides on a remote host. In some cases however, the entire advertised functionality of the service can be delivered by the proxy itself, without the need to connect to the server that advertised it. However, all services need to advertise some incarnation of a proxy in order to be used by clients.

Traditionally, SOC frameworks have employed a centralized architecture with service directories running on dedicated hosts, whose sole purpose is to manage the directory. While the centralized approach is appropriate for wired networks where the risk of network failure is low, the dynamic environment and opportunistic interactions inherent in ad hoc 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.

We address the issues introduced by the dynamic nature of the environment by employing a distributed architecture for the service directory. Each host maintains its own local service directory. Services advertise their availability by registering themselves with their local service directory. The registration process consists of an entry in a service directory, which contains the proxy and a description of its performance parameters. Hosts within communication range share their service directories to form a logically federated service directory. 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).

When a client searches for a service, the query spans the entire federated service directory, which is the conglomeration of local service directories on participating hosts. Querying is done by providing an interface that the service proxy is expected to implement, and other additional performance parameters which may help choose from among multiple possible results.

Access to services that are registered with this federated service directory is based on a matching mechanism applied between the service advertisements and client template specification. The advertisement entry in the service directory is encapsulated in a tuple. Clients provide a description of the tuple they need. A matching mechanism is applied between the template and every advertisement in the federated service directory. A single advertisement is chosen non deterministically from all advertisements which match the provided template and returned to the client. The client obtains the proxy object by extracting it from the advertisement tuple. Tuples can also be used to support communication between the proxy and its parent service. They can deliver proxies or encapsulate and transport any other type of information, e.g., method call parameters, results, etc. The advantage of using such a communication protocol is its location-agnostic characteristic. The tuple is not sent out to a specific host address, but rather put out for the target party to retrieve using a description of its content. Using transiently shared directories to implement communication protocols we achieve a high degree of decoupling between the two end nodes of the communication channel, which helps us manage mobility issues in ad hoc networking environments.

This paper presents an extension to the framework described above in the form of our solution to the problem of dynamically updating a service while it is being used by clients. Our approach can be divided into two distinctive parts: updating the proxy used by a client and updating the server that the proxy interacts with. When updating the proxy object, problems arise due to the fact that the client is actively using it when the server decides the proxy needs to be upgraded. We aim to swap the proxies in a manner transparent to the client. On the server side, the upgrade may also trigger the upgrade of the proxies or may not affect the proxies currently in use. In the second case, the infrastructure aims to replace the server with its newer version transparently even to its own proxies.

B. Updating the Proxy Object

While some server upgrades (e.g., minor changes in the server code) do not affect the proxies, others may entail changes to both the server and its proxies. An example of such a situation is an upgrade of the quality of service offered by a provider, entailing securing the communication protocol between the server and its proxies. Since the initial protocol did not encrypt the communication, the proxies currently in use lack this capability. Implementing such a change is not

as trivial as a feature that can be simply turned on by some configuration message from the server since the feature is absent on the client side. This additional feature can only be made available by replacing the old proxy with a new version.

For this version of our architecture, we impose the constraint that the external API advertised by the proxy cannot change from one version to the next. The reason for this is transparency. Recall that the interface was specified by the client during the lookup process and since we upgrade the proxy without notifying the client, the new proxy is required to provide the same interface to ensure that the client remains oblivious to the change. The changes can affect only the service (i.e., server plus proxy) and should not be visible to the client.

It is important to note that if the client is using the proxy we intend to substitute directly, we couldn't replace it in a manner that is transparent to the client. We solved the problem by adding a layer of indirection between the client and the proxy. Using a combination of the facade and interceptor [10] [11] design patterns, we developed an intermediary wrapper layer that isolates the client from the proxy and handles the proxy upgrade in a manner that is transparent to the client. This layer is generated automatically when the service publishes its proxy object. When the client searches for the proxy object, it receives and installs both the proxy as well as the wrapper.

The functionality this wrapper provides is essentially to decouple the client from the proxy, to forward the client's calls to the proxy, to monitor the server's decision to upgrade the proxy, and to manage the proxy upgrade process. An overview of the architecture is shown in Figure 1.

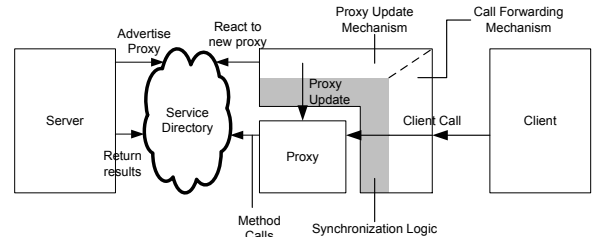


Fig. 1. Architecture supporting run-time service upgrades

During the normal mode of operation, the wrapper simply relays the client's calls to the proxy. The wrapper's facade is generated automatically, using reflection; it simply mirrors the client's calls from the client-wrapper interface to the wrapper-proxy interface. The results that are returned from the proxy are forwarded to the client via the same mirroring technique.

In conjunction with its role of forwarding calls between the client and the proxy, the wrapper monitors the server's advertisement in the directory service. If an upgrade is initiated on the server side (as described in section III-C), the old advertisement is removed and replaced with one containing the new version of the proxy (if applicable). The wrapper reacts to the replacement of the advertisements and retrieves the new

proxy. Note that the retrieval of the proxy by the wrapper does not affect its normal function of forwarding calls.

Once the wrapper has retrieved the new proxy, it requests the synchronization logic module (which ensures consistency during the updating process) for a lock on the proxy. The synchronization logic ensures that the proxies are swapped when there is no activity from the client and no remote execution of some method in progress. A method call acquires a lock which guards the exclusive access to the proxy and will not release it until the result is returned from the proxy. During this time, even if the proxy has already been retrieved and is available on the client host, the swap cannot proceed. Symmetrically, if a proxy upgrade is in progress, a method call cannot complete and will be blocked by the same lock before it reaches the proxy. Once the swapping is finished, the lock is released and the method call is forwarded to the newly installed proxy.

Observe that there can be at most one call on hold. This is because there is only one client trying to access any given instance of the proxy. A client cannot initiate a second call before the previous one returns. We ensure that the wrapper does not return the flow of control back to the client, and keeps the client blocked until the call returns by forcing a synchronous behavior on the client side, even though the wrapper forwards the client's call to the proxy. This synchronization mechanism also ensures that it is not possible for a client to send out a call using the old proxy and receive the answer from the new proxy.

We reiterate that the new proxy implements the same interface as the old proxy and, therefore is compatible with the facade the wrapper uses to separate the client from the proxy (eliminating the need for the replacement of the facade). Once the wrapper is deployed, this facade is immutable and the entire process is transparent to the client, which continues to use the interface it used for the initial discovery of the service. The synchronization described in the previous paragraph ensures that the swapping will happen safely (e.g., no loose calls left unanswered) and transparently for the client (the client is unaware that its call was placed on hold while the wrapper swaps the two proxies).

C. Updating the Server

Upgrades on the server side can be divided into two distinct categories: those that require a parallel upgrade of the proxies and those that do not. Both types of upgrades assume that the server needs to temporarily go off line and be restarted. The possibility to update the server dynamically (i.e., without shutting the server down) constitutes a particular case, with challenges specific to each server's architecture and to each type of upgrade. The discussion of such scenarios is not within the scope of this paper.

Before the server goes off line, it goes through a series of steps to prepare for its downtime. First, the server removes the service advertisement it registered from the service directory. Clients interested in the functionality offered will not be able to discover the service during this stage, even though the

server may be running. At this stage of the process, the server ignores all incoming calls from clients. At the same time, it continues to process the calls *in progress*, which were generated by the older version of the proxy. Not performing this step can indefinitely delay the completion of the in-progress calls (as the set of in-progress calls can evolve over time - e.g., new calls come in from clients before the server finishes the calls it is currently working on) and thus defer the upgrade indefinitely. Meanwhile, other clients would also wait indefinitely to discover the service (unless some other provider offers a similar service), since the advertisement has already been removed. Once the response to the last in progress call from the old version of the proxy is serviced, the server can go off line.

When the new server starts up, it advertises itself in the service directory and it makes itself available to clients. The advertisement publishes a proxy (the same as the one published by the previous version of the service if no proxy upgrade was required or a new one if so needed). The proxy wrappers on clients which have the old version of the proxy react to the new advertisement available in the service repository. If the proxy has changed, the wrappers controlling the proxies' activities on client machines trigger the proxy update procedure. Otherwise, they continue forwarding calls from the client to the proxy which now directs the calls to the new server.

It is important to note that the server is required to preserve backward compatibility with previous versions of proxies. The reason for this is twofold. In the first case, the old server may have ignored some calls from clients during its shutdown process. The new server, when it comes up, finds these calls waiting to be addressed. Until these calls are addressed, the wrapper on the client side keeps the client application blocked, waiting for the method to return. While the wrapper may react to the presence of a new proxy in the new server's ad in the service directory, the most the wrapper can do is fetch the new proxy on the client host and then block again, waiting for the above mentioned call to return. The server therefore has to be able to execute this call which necessitates that the server be able to read and understand the request, even if it was formulated by an older version of the proxy.

Figure 2 shows the sequence of interactions between different parts of the system. In the initial state, the client already has already discovered the service and installed its proxy. The first round trip of calls shows a complete path of interactions starting with the client issuing a call, intercepted by the wrapper which obtains the lock from the synchronization logic and then forwards the call to the proxy which sends it to its server. The return follows the way back and releases the lock as it goes through the wrapper to the client.

The second call (shown in the diagram below the dashed line) occurs at the same time that the server is upgraded. In the scenario depicted, the proxy update request arrives at the wrapper after the method call from the client already went through, towards the server. The wrapper can therefore only discover the new proxy, fetch it locally but has to wait for

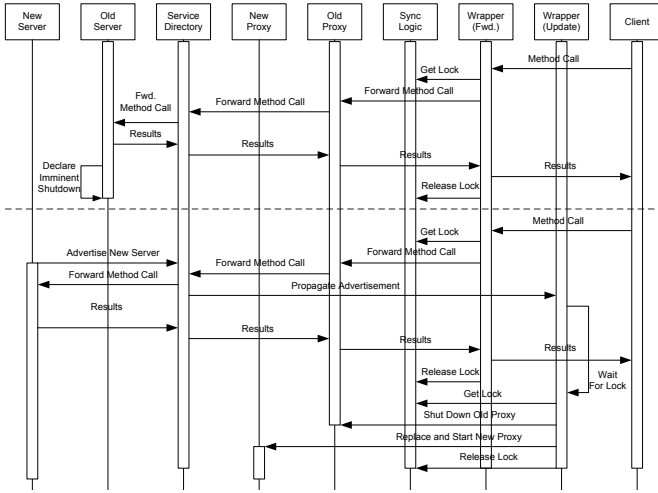


Fig. 2. Interactions between components of the service upgrade mechanism.

the method to return before it can proceed with the proxy replacement process. Once the result of the call is returned and the lock is released, the wrapper obtains the lock and swaps in the new proxy. Once the new proxy is in place, the wrapper releases the lock which guarded the proxy replacement and once again returns to the default operating mode of simply forwarding client calls and results.

IV. IMPLEMENTATION

We implemented the service upgrade architecture for ad hoc networks in Java, using LIME [12] a middleware for physical and logical mobility to handle the implications of ad hoc wireless networks, i.e., the dynamism of the network due to the physical mobility of hosts. In this section we present a brief overview of LIME, followed by details of the implementation.

A. LIME Overview

LIME is a Java implementation of the Linda [13] coordination model which is designed for ad hoc networks. LIME masks details associated with coordination and communication from the application programmer. A host offering LIME runs a `LimeServer` which supports one or more LIME agents, analogous to application modules.

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 (wildcard) 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 and by providing a callback function to be called when a matching tuple becomes available. If multiple candidate tuples exist for a given reaction template, one is chosen non-deterministically from the set.

B. Service Upgrade Extension

We implemented both the client and server as LIME agents which use tuple spaces for coordination and communication. Each agent creates local tuple spaces, which it shares with all other agents within communication range. We use a tuple space named `AdvertisementTupleSpace` as a standard tuple space that is used as an advertisement space for available services. All agents wishing to avail themselves of services are required to create this tuple space locally and share it with nearby agents. Service advertisement tuples are placed in this tuple space by servers using the `out` operation. These advertisement tuples are of the form `<ServiceDescription:desc, ServiceProxy:proxy>`. Clients find services that they are interested in by registering reactions on this tuple space using a template built from the interface they expect, for example `<ServiceDescription.class, PrinterInterface.class>`. If the proxy in the advertisement obeys the `PrinterInterface`, a match is returned. In order to keep the client lightweight, the proxies are expected to provide their own GUI (if required) as well as code supporting interactions with their servers.

It is important to note that, while a client may have the bytecode for an interface that it wishes to match in the `AdvertisementTupleSpace` tuple space, it is not expected to have the bytecode of specific implementors of this interface, especially since different services may offer different implementations of the same interface. Hence, the server and the client share an additional tuple space named `BytecodeRepository` that is separate from the one containing the advertised proxies. This tuple space is used as a bytecode repository for any proxies that servers advertise. When a proxy is advertised, the server uses the `out` operation to place tuples of the form `<String:className, ByteCode:bytecode>` into this shared tuple space. On the client side, a custom classloader is used during the deserialization process that transparently fetches the needed bytecode from this tuple space and installs it in memory. This is done by performing a `rd` operation on the tuple space using a template containing the requested class's name and a wildcard for the bytecode field. The in-memory installation of bytecode allows

for lightweight clients that do not require persistent storage to contain bytecode for proxies they may need at runtime. Bytecode is automatically fetched when it is needed and freed by the garbage collector when it is no longer required. Further details of regarding the dynamic fetching of the bytecode may be found in [14].

We now describe the wrapper that shields the client from being aware of proxy updates generated by the server using standard Java reflection libraries. The server constructs the wrapper object by combining several pieces of code into a Java String object. This String contains the Java source code for the proxy upgrade mechanism and for the synchronization mechanism, which is common to all services, as well as call-forwarding code custom-generated for each individual proxy using reflection. Once the generic reaction code and the call-forwarding code have been combined into a single Java String, a compiler extracted from the Eclipse JDT Core project [15] is used to convert this String into Java bytecode on-the-fly. The server then converts this bytecode into a Java class and instantiates it, giving the original proxy object to its constructor. The server places the wrapper in the `AdvertisementTupleSpace` tuple space. Clients can then use these wrapper proxies as ordinary proxies, unaware of the fact that the actual underlying implementation of the proxy can be swapped out on demand.

This wrapping procedure necessarily adds some overhead to the proxy, both in terms of bytecode size and execution time. However, this overhead is negligible. As a test, our server wrapped a 14-kilobyte proxy object, which produced a 4-kilobyte wrapper. Though this wrapper is relatively large compared to the original proxy in percentage terms, such a ratio is not representative of the average case scenario. The reason is that all wrappers contain the code for synchronization and swapping. This code takes up a certain amount of space that remains constant regardless of the size of the original proxy. Hence, the larger the proxy, the better the ratio between itself and its wrapper since the cost of mirroring a method is negligible. It is also important to note that the wrapper need only be transferred to the client once during the client's interaction with a service, though it may in reality wrap more than one underlying proxy during this lifetime. Hence, even with a relatively poor ratio between the proxy and wrapper size, the total overhead becomes negligible over an extended usage scenario. The code contained within the wrapper is compact: apart from about 50 fixed lines of code dedicated to constructing the wrapper, registering the reaction, and handling proxy updates, the wrapper class contains two lines of code in each wrapped method to lock the wrapped proxy and pass along the method call. These two operations are relatively lightweight and may be considered negligible compared to the time any non-trivial underlying proxy spends servicing these method calls.

The implementation presented offers two advantages over other approaches such as mandating that clients or proxies contain code to support swapping. First, it saves considerable effort on the proxy developers' part, since they need not

specifically design the proxy with swapping in mind. Second, the class wrapping procedures offer a generic framework for adding functionality to objects at runtime; by augmenting or replacing the code used by these procedures, the server could add almost any functionality to proxies at runtime (as long as they do not change proxy's interface), not just restricted to swapping proxies.

The implementation is showcased in a demo of our lightweight client, shown in the top half of Figure 3. It currently contains only one interface which describes a basic roadside service. Servers can pick from any one of a number of predefined roadside services, such as a toll service, parking meter, etc. When a client comes in communication range, it automatically discovers the advertised services, downloads their associated proxies' bytecode, and displays the proxy's GUI. Since these proxies are wrapped before they are placed in the shared tuple space, the servers can upgrade their offered services at any time, such as encrypting the communication between the proxy and the server. The wrapped proxies running on the client will automatically discover the change and replace their underlying proxies at the nearest opportunity, as shown in the bottom half of Figure 3.

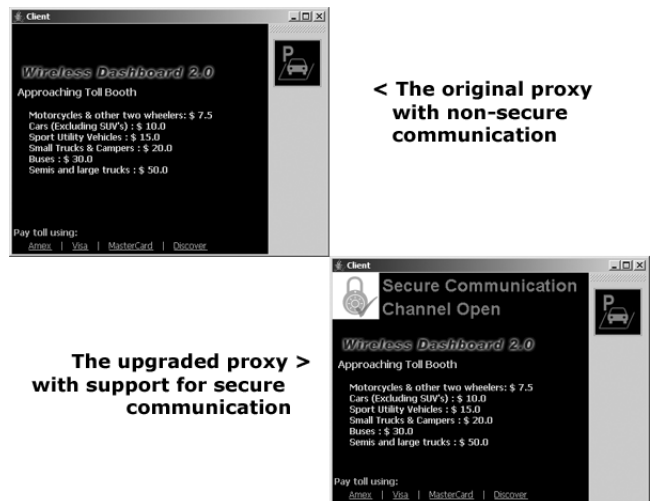


Fig. 3. Screenshots of the application before and after the proxy upgrade.

V. DISCUSSION

The design of our architecture for service upgrading was shaped by our desire to keep the architecture lightweight, so that it could be used in ad hoc settings, where client applications generally run on resource poor devices. In the first part of this section, we explain how using a set of existing artifacts simplified our development process. In the second part, we discuss general issues related to upgrading services and how our architecture might handle them. Finally, we discuss plans for future work.

In designing our model, we chose a tuple space-based communication model for several reasons. Firstly, the tuple space based model allows the decoupling of two interacting

entities (in our case the server and the proxy). Hence, the server and the proxy can each be individually updated without affecting the other one. For example, if a server is shut down, remaining entities in the network can still put tuples in their local tuple space, which can be picked up by the server when it is restarted. The second reason for choosing tuple space-based communication is that the federated tuple space is a transiently shared global directory which contains only tuples from hosts which are reachable (See explanation in Section IV). This eliminates the need for having garbage cleaning mechanisms, which supports the lightweight nature of our architecture. Finally, tuple space-based communication has been shown to be suitable to ad hoc networks in LIME [12], which we use as a base for the implementation of our model. Our model also implicitly provides support for the client application to run multiple proxies from the same service. The model just treats each instance of the proxy as coming from a different service (though this is not really the case). Hence, the client application can have multiple access to the service (which may be needed for multithreaded programs etc.).

We now turn our attention to some general issues associated with service upgrading and how we chose to handle them in our architecture. Recall that in Section III, we made the assumption that the server is backwards compatible. At the model level, this assumption is unnecessary, since some mechanism could be designed to service all the old calls and queue the new calls until the server is upgraded. However, at the implementation level, the challenge is greater. This is in part due to the fact that the proxy needs to simulate synchronous and atomic calls between itself and the server using LIME, which uses asynchronous communication. Hence, the code for simulating the required behavior becomes very extensive. The problem can be solved by imposing certain design constraints on the server. However, that falls outside the scope of this paper.

Another pertinent issue is that of ownership of the service and the right to upgrade a service. In our opinion, any upgrades for a service should come from its original owner. Even if the service is replicated on multiple hosts of an ad hoc network, the upgrades for the service should come from a single host. The reason for this is consistency. By having the upgrades come from a single source, it can be ensured that there are no conflicts due to different hosts issuing simultaneous upgrades that may cause version conflicts (akin to those seen when using CVS to merge different versions of the same file.)

For future work, we aim to develop an architecture that decomposes the proxy such that a proxy is no longer a monolithic piece of code, but is modular so that only parts of it need to be swapped rather than the entire object. Another feature that we wish to support is a versioning system that is responsible for managing the different versions of a service and ensuring compatibility. Finally, we wish to provide a matching mechanism that supports searching at finer granularity (e.g., at the method level rather than the interface level). The results of this work will help provide even lighter wrappers and also provide support for service composition.

VI. CONCLUSIONS

In this paper, we presented a lightweight mechanism to upgrade services without completely shutting them down. We began with an overview of a SOC framework that we developed for ad hoc networks. We followed this with descriptions of a model for upgrading the server and the proxy. For swapping the proxy, we proposed the use of a wrapper-interceptor that temporarily holds calls while the proxy and/or the server are swapped. We showed how the tuple space based communication protocol can allow for a server to shut down and restart without any perceived interruption in service. We described the implementation of our architecture built on top of the LIME coordination model, we justified our design decisions, and we presented future work plans.

VII. 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] Radu Handorean and Gruia-Catalin Roman, "Service provision in ad hoc networks," in *Proceedings of 5th International Conference on Coordination Models (COORDINATION 2002)*, 2002, number 2315 in LNCS, pp. 207–219, Springer-Verlag.
- [2] Radu Handorean and Gruia-Catalin Roman, "Secure service provision in ad hoc networks," in *Proceedings of The First International Conference on Service Oriented Computing (ICSOC 03)*, Springer Verlag, Ed., 2003, number 2910 in Lecture Notes in Computer Science, pp. 367–383.
- [3] Keith Edwards, *Core JINI*, Prentice Hall, 1999.
- [4] Michael Hicks, Jonathan Moore, and Scott Nettles, "Dynamic software updating," in *Proceedings of the ACM SIGPLAN Workshop on Types in Compilation*, September 2000.
- [5] Sameer Ajmani, Barbara Liskov, and Liuba Shrira, "Scheduling and simulation: How to upgrade distributed systems," in *Proceedings of the Ninth Workshop on Hot Topic in Operating Systems*, May 2003.
- [6] Jonathan Cook and Jeffrey Dage, "Highly reliable upgrading of components," in *Proceedings of the 1999 International Conference on Software Engineering*, 1999, pp. 203–212.
- [7] Marija Rakic and Nenad Medvidovic, "Increasing confidence in off-the-shelf components: A software connector-based approach," in *Proceedings of the 2001 Symposium on Software Reusability*, 2001, pp. 11–18.
- [8] Premysl Brada, "Component change and verification in sofa," in *Proceedings of SOFSEM 1999*, 1999, number 1725 in LNCS, Springer.
- [9] Vladimir Mencl, Zuzana Petrova, and Frantisek Platil, "Update description language," June 1999.
- [10] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann, *Pattern-Oriented Software Architecture*, vol. 2, chapter 2, pp. 47–75, John Wiley and Sons Ltd., 2000.
- [11] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann, *Pattern-Oriented Software Architecture*, vol. 2, chapter 2, pp. 109–141, John Wiley and Sons Ltd., 2000.
- [12] A.L. Murphy, G.P. Picco, and G.-C. Roman, "LIME: A middleware for physical and logical mobility," in *Proceedings of the 21st International Conference on Distributed Computing Systems*, 2001, pp. 524–533.
- [13] D. Gelernter, "Generative communication in Linda," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 80–112, January 1985.
- [14] Radu Handorean, Rohan Sen, Gregory Hackmann, and Gruia-Catalin Roman, "A component deployment mechanism supporting service oriented computing in ad hoc networks," Tech. Rep. WUCSE-2004-02, Washington University, 2004.
- [15] JDT-Core-Project, "The jdt-core project homepage," <http://dev.eclipse.org/viewcv/index.cgi/%7Echeckout%7E/jdt-core-home/>.