Report Number: WUCSE-2005-35

2005-08-02

SPAWN: Service Provision in Ad-hoc Wireless Networks

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

The increasing ubiquity of wireless mobile computing platforms has opened up the potential for unprecedented levels of communication, coordination and collaboration among mobile computing devices, most of which will occur in an ad hoc, on-demand manner. This paper describes SPAWN, a middleware supporting service provision in ad-hoc wireless networks. The aim of SPAWN is to provide the software resources on mobile devices that facilitate electronic collaboration. This is achieved by applying the principles of service oriented computing (SOC), an emerging paradigm that has seen success in wired settings. SPAWN is an adaptation and extension of the Jini model of... **Read complete abstract on page 2.**

SPAWN: Service Provision in Ad-hoc Wireless Networks

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

Complete Abstract:

The increasing ubiquity of wireless mobile computing platforms has opened up the potential for unprecedented levels of communication, coordination and collaboration among mobile computing devices, most of which will occur in an ad hoc, on-demand manner. This paper describes SPAWN, a middleware supporting service provision in ad-hoc wireless networks. The aim of SPAWN is to provide the software resources on mobile devices that facilitate electronic collaboration. This is achieved by applying the principles of service oriented computing (SOC), an emerging paradigm that has seen success in wired settings. SPAWN is an adaptation and extension of the Jini model of SOC to ad-hoc networks. The key contributions of SPAWN are (1) a completely decentralized service advertisement and request system that is geared towards handling the unpredictability and dynamism of mobile ad-hoc networks, (2) an automated code management system that can fetch, use and dispose of binaries on an on-demand basis, (3) a mechanism supporting the logical mobility of services, (4) an upgrade mechanism to extend the life cycle of services, and (5) a lightweight security model that secures all interactions, which is essential in an open environment. We discuss the software architecture, a Java implementation, sample applications and an empirical evaluation of the system.

2005-35

SPAWN: Service Provision in Ad-hoc Wireless Networks

Authors: Handorean, Radu; Sen, Rohan; Hackmann, Gregory; Roman, Gruia-Catalin

August 2, 2005

Abstract: The increasing ubiquity of wireless mobile computing platforms has opened up the potential for unprecedented
levels of communication, coordination and collaboration among mobile computing devices, most of which will occur
in an ad hoc, on-demand manner. This paper describes SPAWN, a middleware supporting service provision in ad-hoc
wireless networks. The aim of SPAWN is to provide the software resources on mobile devices that facilitate electronic
collaboration. This is achieved by applying the principles of service oriented computing (SOC), an emerging paradigm
that has seen success in wired settings. SPAWN is an adaptation and extension of the Jini model of SOC to ad-hoc
networks. The key contributions of SPAWN are (1) a completely decentralized service advertisement and request system
that is geared towards handling the unpredictability and dynamism of mobile ad-hoc networks, (2) an automated code

SPAWN: Service Provision in Ad-hoc Wireless Networks

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

Abstract

The increasing ubiquity of wireless mobile computing platforms has opened up the potential for unprecedented levels of communication, coordination and collaboration among mobile computing devices, most of which will occur in an ad hoc, on-demand manner. This paper describes SPAWN, a middleware supporting service provision in ad-hoc wireless networks. The aim of SPAWN is to provide the software resources on mobile devices that facilitate electronic collaboration. This is achieved by applying the principles of service oriented computing (SOC), an emerging paradigm that has seen success in wired settings. SPAWN is an adaptation and extension of the Jini model of SOC to ad-hoc networks. The key contributions of SPAWN are (1) a completely decentralized service advertisement and request system that is geared towards handling the unpredictability and dynamism of mobile ad-hoc networks, (2) an automated code management system that can fetch, use and dispose of binaries on an on-demand basis, (3) a mechanism supporting the logical mobility of services, (4) an upgrade mechanism to extend the life cycle of services, and (5) a lightweight security model that secures all interactions, which is essential in an open environment. We discuss the software architecture, a Java implementation, sample applications and an empirical evaluation of the system.

Index Terms

system integration and implementation, composite structures, storage/repositories, distributed systems, pervasive computing, middleware, network repositories.

I. INTRODUCTION

Rapid advances in technology have resulted in portable computing devices such as PDAs and cellular phones becoming increasingly more powerful, making them viable mobile computing platforms. In parallel, societal acceptance of mobile computing has resulted in more and more people owning such devices, thereby making them ubiquitous. Given this fact, the potential exists for unprecedented levels of electronic collaboration among mobile devices. However, this potential has thus far been realized only to a limited extent.

One of the key reasons why electronic collaboration among mobile devices has not become commonplace is because software available for portable devices is very limited in its scope. Currently, most portable devices function

This research was supported in part by the National Science Foundation under Grant No. CCR-9970939 and 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 reflect the views of the research sponsors.

like small stand-alone personal computers. Typical applications are address books, calendars, email, and simple word processors. Most of these applications usually synchronize themselves with a desktop PC or a server via a wireless network. Their usage of their networking capabilities as a mobile device is minimal. This occurs in part because the software resources for such collaboration are currently at a very rudimentary state. For example, PalmOS [1] relies on data transfer between two devices in a peer-to-peer fashion using the beam feature. Packaged applications must also be transferred in a similar manner–on being received by a host, these packages can be unpacked manually and the application installed locally. The shortcoming of this approach is that it is restricted to simple data transfers between two devices and there is no scope for running a collaborative distributed application with many-to-many interactions. The advantages of unimpeded collaboration among mobile devices are clear. If successful, it would add a completely new dimension to the exchange of information and capabilities, where collaboration will be possible with any other device with minimal effort. This is the grand challenge.

Service-oriented Computing (SOC) is a new computing paradigm that seeks to promote unhindered interaction among applications by wrapping each application with an interface which can be described and interacted with via the use of a standardized high level specification language. The application and the wrapping interface are collectively known as a *service*. The principles of SOC have been applied successfully in wired networks, e.g., in the form of Web Services [2] which among others, facilitates Business-to-business (B2B) type interactions between heterogeneous software packages used by various companies. Web services have been targeted towards powerful server units with wired Internet connections.

However, the design of the Web Services architecture is not targeted toward wireless ad-hoc networks such as those formed opportunistically between mobile devices. In this paper, we present SPAWN, a middleware for Service Provision in Ad-hoc Wireless Networks. SPAWN is designed to promote unhindered collaboration among heterogenous software and hardware available for *mobile devices* by applying the SOC paradigm to software engineered for mobile ad-hoc wireless networks. The aim is to allow an egocentric application running on a mobile device to expand its capabilities by exploiting services offered by other devices. Our focus is on *personalized* services, i.e., services that help a specific user complete his or her task.

SPAWN is based on Jini [3], a proxy-based model of SOC, which differs from the Web Services model. Like other SOC models, in Jini there is a *service provider* that offers a service and a *service recipient* or client that uses the service. What differs is the manner in which services are advertised and used. Unlike Web Services where services are described using an XML description and accessed by a Uniform Resource Identifier (URI) specified in the description, in Jini a service is advertised as a *proxy object*, which is at minimum a piece of code that offers an interface to the service provider but can also be a fully functional object delivering part or all of the offered service's functionality. Depending on the complexity of the proxy, the client uses the proxy as a local handle to a service residing on a remote service provider or as a local copy of the offered service. The advantage of the proxy-based approach is that the use of the proxy eliminates any need for the client to have knowledge of the application level communication protocol to interact with a given service. This in turn allows interaction with a large set of heterogenous services by simply making local method calls to their respective proxies (we assume that the client knows the interface of the service it needs a priori). A useful side effect of this is that clients can be relatively lightweight since all the communication logic is embedded in the proxy. However, Jini employs a centralized service directory where service providers offer services, which is not suitable for ad-hoc wireless networks because the loss of the host with the service directory (due to it not being in communication range or due to shutdown) can render the entire system non-functional. In addition, the dynamic nature of ad-hoc wireless networks creates further problems for middleware design which had to be addressed during the development of SPAWN.

The SPAWN system is an adaptation of the Jini model for ad-hoc wireless networks. Instead of a centralized service directory, SPAWN uses a transiently shared federated service directory. Hosts within communication range can contribute to and use services in the federated directory. Additionally, instead of using RMI for communication between the proxy and its parent services, the SPAWN system uses a tuple-space-based communication mechanism (described in Section II). The SPAWN system also implements some additional features. If the client requests a service that requires a source file that is not locally available, an automated code management system fetches the required file from the provider and installs it on the client in a transparent manner. If the file has other files as dependencies, those are also fetched as required. To accommodate the various constraints found on devices in an ad-hoc network, such as storage, battery power, and mobility. SPAWN provides a mechanism for services to be migrated from one host to another using a pause-transfer-resume mechanism. This is useful if for example, a service has to move to another host to stay in range of a client while it finishes execution, or if the battery on a device is low prompting a move to a device with more power. Another feature is the ability to upgrade services at runtime. SPAWN allows the service provider to upgrade the service software with minimal interruption to the client. Finally, a lightweight security mechanism secures all interactions between the client and the service provider.

The remainder of the paper is organized as follows. Section II describes the communication model for ad hoc networks upon which SPAWN is built. The basic mechanisms of service advertisement and discovery are covered in Section III, followed by a description of the automated code management feature in Section IV. Section V describes the strong migration capabilities of SPAWN. The facilities to upgrade services are discussed in Section VI and Section VII describes the lightweight security model. Section VIII describes the implementation details of the SPAWN system, followed by evaluations in Section IX. Related work is covered in Section X. Section XI gives further discussion of the paper before we conclude in Section XII.

II. COMMUNICATION & COORDINATION IN AD-HOC NETWORKS

SPAWN has been designed to work effectively in ad-hoc wireless networks. Before we begin a detailed description of SPAWN, we provide a brief summary of the characteristics of ad-hoc networks. We also describe Limone [4], a lightweight coordination middleware for ad-hoc networks, which provides basic communication primitives upon which the SPAWN system is built.

An ad-hoc network is a dynamic network where the network infrastructure is collectively supported by the devices that comprise the network. In mobile ad-hoc wireless networks, which are the focus of our work, devices are physically mobile. The physical mobility of devices results in transient connectivity leading to a decoupled style of computing. In general, devices in an ad-hoc network are resource poor in terms of battery power, processor speed, and possibly storage. Ad hoc networks can also be highly heterogenous, with devices from cellular phones to laptops participating in a given network. The inherent dynamism of the network results in frequent and unpredictable disconnections. Communication in such an environment cannot be conducted using traditional methods such as sockets and streams as they are not geared towards dynamic environments . Hence, we use Limone, a coordination model designed specifically for ad-hoc networks.

Limone abstracts the low level details of communicating with other hosts in an ad hoc network. By providing primitives that offer a higher level of abstraction, Limone simplifies application development for dynamic settings. Thus, events such as disconnections, network reconfiguration, etc. are all handled by Limone and the application is shielded from their effects. In Limone, each mobile device is referred to as a *host*. Each host may have one or more units of execution called *agents* running on them. Hosts are typically physically mobile, meaning that they move in space, while agents may be logically mobile, i.e., they can move from one host to another. When two hosts (and thus the agents running on them) are within communication range, they are said to be *engaged*. When they move out of communication range, the hosts are said to have *disengaged*. Communication can occur between two agents whose hosts are engaged. The following subsections describe how the Limone model works and how it can be used to communicate in ad-hoc networks.

A. Establishing Contact with Neighboring Hosts

The physical mobility of devices in an ad-hoc network coupled with the limited range of current 802.11b wireless LAN cards results in the set of neighbors of a given reference device changing rapidly over time. Hence, the first step towards effective communication is to track the devices that are within communication range. Each mobile device or *host* runs a single *LimoneServer*. A LimoneServer may have multiple LimoneAgents running on it, each analogous to an application. The LimoneServer periodically broadcasts beacons. Beacons contain profile information for each LimoneAgent running on the LimoneServer that broadcasted it. When a host receives a beacon from the reference host, it forwards the profile information contained in the beacon to each LimoneAgent running locally.

Every LimoneAgent has an *engagement policy* and an *acquaintance list*. The engagement policy dictates which agents can be added to the acquaintance list. By default, all agents are allowed to be added to the acquaintance list. The acquaintance list tracks the agents (and thus the hosts) that are in communication range and meet the requirements of the engagement policy. When the LimoneServer forwards a beacon to a LimoneAgent, it checks the policy information contained in the beacon to see which agents have policies consistent with its engagement policy. Those agents are added to the acquaintance list. The acquaintance list constantly monitors for beacons from all agents in the list (recall that the beacons are sent periodically by each host). If a beacon has not been received for a customizable period of time, the agent is removed from the acquaintance list and is considered not to be in communication range until another beacon is received from it. In this manner, LimoneAgents always have accurate snapshots of the hosts and agents within communication range.

<Name:String("Boat"), Size:Integer(25), Destination: String("Europe")>

<Destination: String("Europe"), Size:Integer.class>

Fig. 1. The first string in the figure represents a tuple. The second string represents a template which can be used as a parameter to the `in` and `rd` operations.

B. Communication Using Tuple Spaces

Once a host establishes who its neighbors are, the next concern is to actually have them communicate. In Limone, this occurs via a transiently shared data space called a *tuple space*, originally proposed in Linda [5], and adapted to ad-hoc networks for the first time in LIME [6]. Each agent owns a single physical *local* tuple space. Multiple logical tuple spaces can be created using a mechanism described later in this section. A tuple space can be conceptualized as a bag into which data can be placed. An agent can always access the data that is placed into its local tuple space. When two hosts are within communication range, the tuple spaces of its agents can be logically merged to form a *federated* tuple space. When the tuple spaces are merged, any agent can access data from any other agent whose tuple space is part of the logically federated tuple space. The merging (and demerging) of tuple spaces occurs in a single atomic step to ensure consistency.

Communication among agents occurs via the use of *tuples*. A tuple is an set of data fields. An example of a tuple can be seen in Figure 1. An agent can place a tuple containing some data in its local tuple space. Another agent can retrieve the tuple from the tuple space, resulting in communication between the agents. Three basic operations are provided for tuple space based communication. The `out` operation places a tuple in the local tuple space. The `in` operation removes a tuple from the local tuple space, while the `rd` operation performs a non-destructive read of the local tuple space. The `out`, `in`, and `rd` operations block until they are completed, meaning that further tuple space operations cannot occur while they are executing.

The `in` and `rd` take a *template* as their parameter. A template is a collection of named constraints, each defining a name and a predicate over the field type and value that is called the constraint function. A template matches a tuple if each constraint within the template has a matching field in the tuple. A constraint matches a field if the field's name, type, and value satisfies the constraint function. Multiple logical tuple spaces may be simulated by adding a field with the name of the tuple space, and then having the same name in a template to restrict the search to that particular logical tuple space. For convenience, Limone allows an agent to create a logical tuple space class which acts as a wrapper around the agent's local tuple space. When a tuple-space operation is performed on this class, it automatically adds the name of the logical tuple space to the tuple or template as the case may be.

In addition to the three basic tuple operations, Limone also provides advanced tuple space operations. The `inp` operation is similar to the `in` operation except for the fact that it is non-blocking. A null value is returned if no matching tuple exists in the tuple space. The `rdgp` operation is similar to the `rd` operation except for the fact that while the `rd` operation can only return a single tuple, the `rdgp` operation can return a vector of tuples if more than one matches the specified template.

C. Remote Operations & Reactions

For security reasons, an agent may not perform any of the three tuple space operations on another agent's tuple space. Instead, they must send a remote operation request. For example, Agent A may send a request for an `in` operation with template t on Agent B. Each agent has a *remote operation manager*. The remote operation manager handles all remote requests. When a request comes in, it executes the operation locally and sends the result back to the initiating agent. Optionally, the remote operation manager can be configured to reject certain requests to enforce access control or security policies. Note that to the application programmer, remote operations appear to be simple operations that span the federated tuple space. The sending of requests and receiving of results is handled entirely by the Limone middleware in a manner transparent to the application programmer.

The final feature of Limone we present is the reaction mechanism, initially proposed in [6]. Reactive programming constructs enable an agent to respond automatically to the existence of particular tuples in the tuple spaces of agents in its acquaintance list. Two state variables within each agent, the reaction registry and reaction list, support this behavior. An agent registers a reaction by placing it in its reaction registry. Once a reaction is registered, Limone automatically propagates the reaction to all agents in the acquaintance list that satisfy certain properties specified by the reaction (e.g., the agent's name or location). At the receiving end, the operation manager determines whether to accept the reaction. If accepted, the reaction is placed into the reaction list, which holds the reactions that apply to the local tuple space. When the tuple space contains a tuple satisfying the trigger for a reaction in the reaction list, the agent that registered the reaction is sent a notification consisting of a copy of the tuple and a value identifying which reaction was fired. If this agent receives this notification, it executes the code associated with the reaction atomically.

The use of Limone allowed us to design SPAWN without having to consider the complications of communication in an ad-hoc network. This greatly simplified the development of SPAWN. In the next few sections, we describe the various features of SPAWN and their role in the overall system.

III. SERVICE DEPLOYMENT & DISCOVERY IN THE AD-HOC ENVIRONMENT

We begin our description of the SPAWN system by describing the resources that support basic sharing of services. For services to be shared, a service provider must be able to advertise the service. A client or service recipient must be able to view advertisements from one or more service providers and select a service that most closely meets its needs. This interaction is facilitated by the *service directory*. Traditionally, the service directory is placed on a well known host which can be accessed by all other hosts in the network. An analogous example is the DNS system used on the Internet. While this centralized approach works well in reliable wired networks, it fails in ad-hoc wireless networks because hosts constantly move in and out of communication range and the directory moving out of range can compromise the entire system. This can also be seen pictorially in Figures 2 and 3. Figure 2 shows a scenario where orphan advertisements are present in the service directory while Figure 3 shows a scenario where lack of access to the service directory prevents use of services. To address these problems, SPAWN uses a distributed service directory, where each host is responsible for the portion of the directory that contains its own advertisements. The service directory is modeled as a Limone tuple space. Recall that the tuple space is a transiently shared federated data structure. Hence, if a host should get disconnected, only the portion of the federated service directory that was the responsibility of that host would be lost. Further, the loss of that portion of the directory does not affect the system as a whole since it only contains the advertisements of the host that just disconnected. Not having those advertisements accessible any more is actually desirable since the host is not within communication range to offer those services anyway. Thus, using the tuple space to model the service directory not only solves the problem of creating a decentralized directory, it also solves the crucial problem of consistency of the service directory (i.e., there are no orphan advertisements in the directory).

Having described the service directory, we now move on to the advertisement itself. In SPAWN, an advertisement has three parts: the proxy object, the descriptive profile of the service being offered, and the binary for the proxy object (including the class closure). The proxy is the serialized form of the object that becomes the local handle to the service on the client host. The descriptive profile is a set of attribute-value pairs that describe how well a service can perform a task. For example, for a service that offers access to a printer, a possible attribute-value pair could be `resolution:300dpi`. The binary is the machine code required to execute the proxy class. To advertise a service, the service provider passes the proxy object and attributes to SPAWN. The system automatically scans the proxy object and assembles a list of all the classes that are in the closure of the proxy's class. It then packages the serialized form of the proxy and the descriptive profile in a service advertisement tuple and places it in the local service directory using Limone's `out` operation. Additionally, for the proxy class and all its dependent classes, it generates a tuple with the binary for the class and a string representation of the name of the class. These tuples are placed in a separate directory called the *code repository*, which is also modeled as a Limone tuple space. The role and behavior of the code repository is described in greater detail in Section IV.

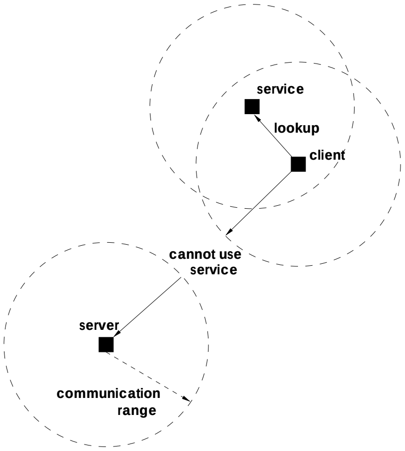


Fig. 2. The client accesses the service directory and discovers an orphan advertisement (an advertisement for a service that is not reachable by the client

When hosts are connected, the service directory is federated across all connected hosts. Hence, any other agent resident on a host that is connected can view the service advertisement. However, simply viewing the advertisement is not enough. A client must be able to request services that match its criteria. In SPAWN, a client requests a service by first formulating a template that describes the kind of service desired. A service request template has two fields– The interface of the service desired, and a set of minimum attributes that the service is required to meet. This template is passed to SPAWN which uses it as a parameter to Limone's `rd` operation which operates on the federated service directory. Limone's matching mechanism (described in Section II) compares service advertisement tuples in the service directory with the provided service request template. If a match is found, a copy of the service advertisement tuple that generated the match is sent back to the requesting client. It should be noted that if more than one match is generated, then the system non-deterministically selects one and returns it to the client. Optionally, the client can request to have all the matches returned and do the filtering itself (in this case, Limone's `rdgp` operation is used).

Once the service advertisement tuple containing the proxy and the descriptive profile is returned to the client, the proxy is extracted from the tuple, deserialized and installed on the client. The installation is light-weight, i.e., the proxy is not written to permanent storage but is simply kept in memory. The reason we made the decision to not write the object to persistent storage is because we regard services as software resources that are used when and where needed and then discarded once their usefulness is at an end. Hence, saving them on a hard disk or flash memory is not necessary, i.e., it would be the same as always carrying all the needed applications on the portable device. We rely on the operating system to page out the binaries if available memory is low. Once the installation has been completed, the client can interact with the proxy as if it were a local object (recall that we assume that the client knows the interface of the proxy). The proxy accepts calls from the client and services them locally or delegates them to its parent service on a remote host.

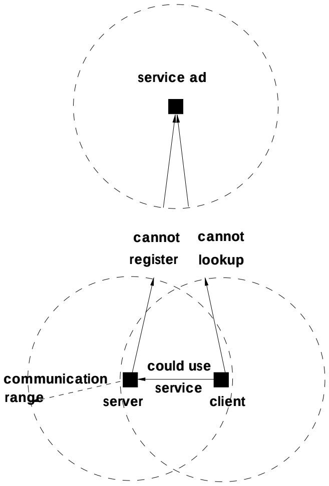


Fig. 3. While the client is within communication range of the service, it is not aware of this since the service directory, which facilitates the interaction is unreachable.

Though fully encapsulated, the proxy-service protocol needs to be resilient against temporary disconnections which can be caused by the two hosts moving beyond communication range or by having the proxy reconnect to a different server. This is addressed by the fact that we use tuple spaces for communication between the proxy and the server. A proxy on the client places a tuple in the client's local tuple space. The service provider typically registers a reaction or performs a remote `in` on the client's tuple space to retrieve the tuple. If the client moves out of range of the service provider or vice versa, the tuple will remain in the local tuple space. When the service provider reconnects, it will execute the remote operation or reaction and retrieve the tuple. Hence, no communication is lost due to disconnection. The behavior is identical for communication originating at the service provider. SPAWN also uses a timer to avoid infinite blocking during a method call made on a proxy that results in some communication with its parent service provider. When the timer expires, the system raises an exception which is caught by the proxy and propagated to the calling application.

The entire process of discovering and using proxies relies on a key assumption–that the binary of the proxy is already available on the client host. Without this binary, the proxy would not be recognized as a valid class on the client. This assumption is not reasonable since the entire purpose of SOC is to exploit services on other hosts, most of which are likely to have not been encountered before. Thus, to remove this constraint, we developed an automated code management system for SPAWN that ensures that the required binaries are fetched from the service provider and made available on the client when needed. This is described in the next section.

IV. Transparent Exploitation of Services Using Automated Code Management

SPAWN's automated code management system is responsible for ensuring that the bytecode for proxy classes is available on the client when required. It may be argued that such a system is not necessary if alternate models for downloading code or a completely different SOC architecture is used. However, we claim that those approaches are not as suited for ad-hoc wireless networks as the proxy-based approach. For example, Jini uses a centralized server to distribute code using HTTP, and RMI [7] to handle calls between the proxy and the client. Having a centralized code repository is not suitable for ad-hoc networks for the same reason that it is not suitable to have a centralized service directory. The use of RMI requires both the service provider and the client to run resource intensive RMI servers, which is not practical on resource constrained mobile devices. Other SOC models such as Web Services that operate without transferring any code are also not suitable as they rely on every resource being in a fixed location described by a static URI, which is not possible in a dynamic mobile setting as it can limit the potential for interaction between the client and the service provider.

While we could have left it to the client application developer to ensure that any required bytecode was prefetched, such a requirement makes the client application much more complex. Thus, we opted for a fully automated code management system that is completely transparent to the application developer and to the application on the client host, and which therefore significantly simplifies application development. The code management system is split into a provider side and a client side infrastructure.

A. Provider Side Infrastructure

The infrastructure on the service provider is responsible for facilitating the process by which the required code is made available to potential clients. The service provider is responsible for providing the classes for all its service proxies to SPAWN. When a proxy class is passed to SPAWN, it is put in a tuple and placed in the code repository. Additionally, reflection is used to determine the class closure of the primary proxy class. For each non-standard class in the closure (where non-standard is defined as a class that is not available in the basic libraries of the programming language being used), the local host is checked for the binary for that class. If found, the binary is placed in a tuple and placed in the code repository in a manner similar to the primary proxy class. If the binary is not found, it is skipped under the assumption that the dependency will be available from another service provider on another host. An argument can be made for packaging the proxy code as well as its dependencies in a single executable archive, e.g., JAR files. However, 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. Thus by default, archives are not used, but a service application programmer can force the use of archives if required.

B. Client Side Infrastructure

Upon receipt of the service advertisement after a successful lookup (see Section III), the client extracts the name of the service proxy class from the advertisement and tries to instantiate the proxy locally. If the binary code for the proxy is not present on the client, the instantiation 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 automatic and performed on the code repository in a manner entirely transparent to the client application developer. Once the binary code has been discovered and retrieved by the client, it is parsed to determine if any additional code (dependencies) is required. If any code is required, the system launches the discovery protocol in the code repository to discover the binary code for each dependence. This process is identical to the one used for discovering the binary code for the proxy itself. Once all dependencies have been fetched, the proxy and the additional code are installed on the client. Once the proxy has been installed on the client, it is loaded into the runtime environment. Figure 4 illustrates the entire process starting from the time a service is advertised to the time when it is installed and ready to use on the client.

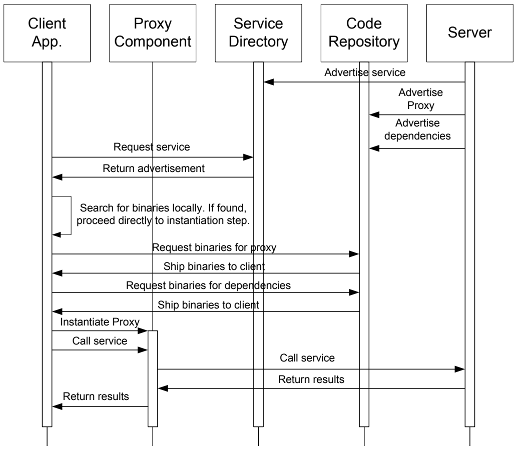


Fig. 4. Proxy advertisement, discovery, installation and utilization.

By design, our architecture for discovering dependencies also supports service composition, since one proxy can have among its dependencies another proxy, representing another self sufficient service. Figure 5 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 by a service provider 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 service provider 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 is treated in a similar manner to D_A^1.

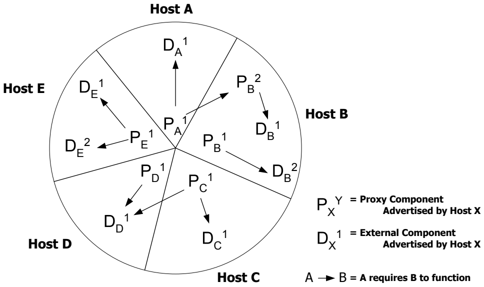

Host A
D_A^1
Host E
D_E^1
P_B^2
Host B
P_A^1
D_B^1
D_E^2
P_E^1
P_B^1
D_B^2
P_D^1
P_C^1
D_D^1
D_C^1
Host D
Host C
P_X^Y = Proxy Component Advertised by Host X
D_X^1 = External Component Advertised by Host X
A → B = A requires B to function

Fig. 5. Federated code repository - proxies and their dependencies.

The code management system is essentially a very weak form of code migration. The dynamism of ad-hoc wireless networks often requires stronger forms of migration, i.e., migration in which an executing process is paused, transferred to a new host and resumed from the point where it was paused. Strong migration helps processes achieve true logical mobility, which increases their flexibility. The advantages of this increased flexibility are described in the following section along with the description of the support SPAWN provides for strong process migration.

V. Mitigating the Effects of Mobility on Service Provision

Thus far, we have described how services are discovered and used in ad-hoc wireless networks. We have also shown how the effects of short periods of disconnection can be mitigated by the use of tuple-space-based communication. However, this is not sufficient for ad-hoc wireless networks since disconnections can be long-lived or even permanent. While we cannot control when disconnections occur, we can take steps to mitigate the effects of the disconnection. The solution we adopted in SPAWN was to enable services to be *logically mobile*. Logical mobility is when a program moves from one host to another. In addition to supporting logical mobility, we support *pause-transfer-resume* computing where an executing process is paused, its data and execution state transferred to another host and execution resumed at the point that it was stopped. To support such functionality, we developed *follow-me sessions*. A follow-me session refers to the interactions that occur during the time interval from when a client starts using a service until the time when the client is finished using the service. The interactions in a follow me session may span beyond a single interval of connectivity or a single host. During a follow-me session, the service can be logically moved to alternate hosts to help ensure that the client-service interaction can run to completion. At this point, we reiterate that the term “service” refers to a software process that runs on a host on which the corresponding service provider resides. A crucial observation is that a service process can reside on one or more service provider hosts or *servers* over a period of time. Hence, in this section, when we refer to a service, we refer only to the process, and not to the host on which it is resident.

Process migration is central to delivering follow-me sessions. To help maintain connectivity, a copy of the service can be migrated so that it remains within communication range of the client for as long as is required. It should be noted, however, that in some cases, the service cannot be migrated, possibly because it relies on some hardware, e.g., a GPS receiver, that is only connected to its primary host. To ensure connectivity, migration of processes is essential. The rest of this section describes how a process can be migrated from one host to another in SPAWN.

A. Checkpointing & State Saving

To support process migration, we have to place checkpoints in the software. Checkpoints are specific points in the code where the current state of a program and its data, including intermediate results, are saved to non-volatile storage so that if interrupted the program can be restarted at the last checkpoint. If a program’s run fails because of some event beyond the program’s control (e.g., hardware or operating system failure) then the processor time invested before the checkpoint will not have been wasted. In SPAWN, most of the failures will occur due to disconnection. If a client-service interaction does not run to completion during a window of connectivity between the client and the service provider’s host, the task has to be completed during a subsequent window of connectivity, or the service has to be moved to a host within communication range. Using checkpoints, the interaction can resume from an intermediary point (i.e., from the last checkpoint the execution flow went through before the disconnection), and does not have to be restarted from the beginning.

SPAWN relies on the application developer to place checkpoints at points in the code where it can be paused safely, e.g., at points where it does not hold locks on shared resources. When the code is provided to SPAWN, the checkpoints placed by the application developer are replaced with subroutines in which the state of the thread is recorded, including its program counter and call stack, which consitute the execution state. We maintain an artificial program counter, which is updated at each checkpoint. The value of this artificial program counter is transferred to the destination host and is used to resume the execution of the service process. The data state is composed of the values of instance variables (live objects) which are transferred as serializable objects (we assume that all instance variables are serializable). When the service migrates from one host to another, the state recorded at the last checkpoint visited is transferred. Any further computing is lost as the session and state information resumes from that last checkpoint, e.g., if the checkpoint is placed just before a for loop, the loop will be started from the beginning when the execution is resumed at the destination host. If the checkpoint is added immediately inside the loop, the execution resumes with the last iteration of the loop executed on the initial host.

B. Process Migration

Process migration has traditionally been separated into weak and strong migration. Weak code migration requires that the process can move and restart, but not resume, on the destination host. This involves making the code available on the destination machine, loading it and restarting the process from the beginning, losing any progress the process may have made before migration. In some cases, some initialization data can be transferred along with the process but that does not account for execution state transfer. The process is still started from the very beginning, except that the memory is initialized to contain potential partial results.

Weak code migration is not sufficient to support follow-me sessions as it is loses any progress made (i.e., processes have to start executing from the beginning when transferred and any messages received during execution would be lost, thereby potentially violating some safety properties). Thus, we use strong code migration which entails the migration of the execution state as well. Obviously, strong mobility is more powerful but it is also more expensive to deliver. Migration might be made completely transparent to the process being transferred. However, this can be extremely dangerous. For example, such a process could be transferred at a moment when it holds locks on resources. Without support from the operating system, these locks may never be released, since the owner process does not operate under the supervision of the operating system on the current host. Hence, we cannot safely migrate a service in a manner that is completely transparent to the process itself, or to the programmer who writes the service. We instead give the programmer control over the places where the process is paused and transferred by marking such locations with checkpoints. This does not *guarantee* that the developer does not use the checkpoints in wrong places, e.g., while having a resource locked for exclusive access, but supports correct use.

During migration, the serialization process wraps only the content of an object (values of member variables), packages them in a tuple, and moves them to the destination host using Limone's standard tuple-space-based communication that was described in Section II. However, the bytecode from which the object (as well as all objects inside the initial object) were created is not transferred. Therefore, a separate mechanism is required to transfer the bytecode for each object (including its dependencies) to the destination host. This is handled by the automated code management system described in Section IV. Regardless of whether the migration is lightweight or heavyweight, the server continues to execute on the source host. When a service migrates, a copy of the service is created and the copy is migrated to the new host. The original copy of the service continues to run on the original host until it reaches the next checkpoint, where it checks a flag, which if set, triggers termination of the service process. This is done because stopping the service between checkpoints can result in undesirable behavior within the system, which is explained in greater detail in the Implementation section. Note that if a service has multiple clients, the original copy of the service can continue to run on the original host for as long as it is required to by all its other clients.

At a high level, process migration is fairly simple. Most of the complexity of strong process migration is in the actual implementation of the mechanism. Hence, we postpone further discussion of mechanism until the implementation section of this paper.

VI. Extending Service Life: Service Upgrade At Run Time

Software upgrades due to bug fixes or to provide enhanced functionality are commonplace. Since a service is at its core a piece of software, it is possible that upgrades may be required. The problem with an upgrade of the service software is that it could make the proxy incompatible. Hence, the proxies need to be upgraded at the same time as the service itself. Performing such upgrades introduces several technical challenges, including (1) ensuring that the upgrade takes place in a manner transparent to the client while minimizing the downtime of the proxy and (2) ensuring that when the server side software is upgraded, in-progress 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).

Spawn provides a lightweight service update mechanism that can dynamically upgrade the server as well as its proxies on client hosts. Transparency is achieved by employing a dynamically generated facade to hold calls from the client application temporarily while the old version of the proxy object is swapped for the new one. Orphan calls are avoided due to tuple-space-based communication which can hold communications for short periods of time without losing them (similar to when there is a temporary disconnection between client and service provider hosts). 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. 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.

A. Updating the Proxy Object

Also in the interest of transparency, we impose the constraint that the external API advertised by the proxy cannot change from one version to the next unless the new interface extends the old one and hence is backward compatible. Recall that the interface is specified by the client during the lookup process; since we upgrade the proxy without notifying the client, the new proxy is required to provide the same interface (or a subclass) to ensure that the client remains oblivious to the change. Also, if the client is using the proxy directly, the upgrade cannot be transparent since during the swap the reference to the proxy will not be valid. We solved the problem by adding a layer of indirection between the client and the proxy. Using a combination of the facade [8] and interceptor [9] 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 the proxy as well as the wrapper. The functionality this wrapper provides is essentially to decouple the client from the proxy, to forward the clients calls to the proxy, to monitor the servers decision to upgrade the proxy, and to manage the proxy upgrade process. An overview of the architecture is shown in Figure 6.

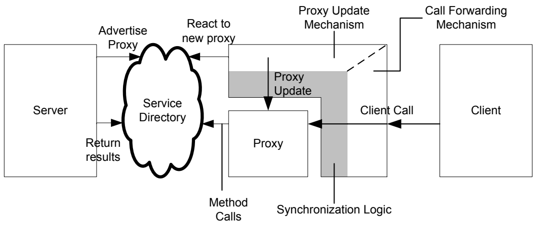


Fig. 6. Architecture supporting run-time service upgrades.

During the normal mode of operation, the wrapper is a simple pass through for calls from client to proxy and results from proxy to client. In parallel, the wrapper monitors the service advertisement in the directory service. If an upgrade is initiated on the server side, the old advertisement is removed and replaced with one containing the new version of the proxy. The wrapper reacts to the replacement of the advertisements and retrieves the new proxy. During the retrieval of the new proxy the wrapper continues to function as a pass through. 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. Note 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 clients 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. 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.

B. Updating the Server

Upgrades on the server side assume that the server needs to go off line temporarily and be restarted. Before the server goes off line, it 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 old version of the proxy. Not performing this step can delay the completion of the in-progress calls indefinitely(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. 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 the new proxy (if needed). The proxy wrappers on clients which have the old version of the proxy react to the new advertisement available in the service repository and upgrade the proxy if necessary. 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 servers 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 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 **??** shows the sequence of interactions between different parts of the system. In the initial state, the client 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 call return follows the same way back and releases the lock as it goes through the wrapper to the client.

The second call (shown in Figure **??** 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 therefore can only discover the new proxy and fetch it locally, but has to wait for 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.

VII. Secure Interactions in an Open Environment

The final component of SPAWN we present is its lightweight security mechanism. In a stationary setting, where all hosts access a wired (and therefore much more reliable) network, security issues are easier to deal with since the applications can rely on central databases containing information about users, passwords, credentials, and capabilities. Trust management in a distributed system operating in a stationary setting is easier when applications can always rely on the presence of a server-based service ready to authenticate requests and offers.

In ad-hoc wireless networks, security considerations face new challenges. When two devices come in contact, identifying the other party is not as easy as asking a trusted third party to verify the other's identity. While an authentication server may be available from time to time, the design of the applications cannot rely on this for their proper functioning. There is no guarantee that an authentication service is available at a given point in time or that it will become available any time soon. Thus, the two nodes need to take special safety measures to ensure secure interactions.

The challenge is to devise a system that does not depend on the behavior of a single node for secure interactions as well as to permit users and programs to be as effective as possible in this environment of uncertain connectivity, without changing their manner of operation (i.e., by preserving the semantics of interaction) while still offering security guarantees. Security enforcement is needed to protect the easily accessible service registries from tampering or unauthorized usage and the new model is required to address this issue to the maximum extent possible under the additional constraints imposed by ad hoc networking.

Addition of security to SPAWN required the addition of two components: a security *veneer* between SPAWN and Limone, and a security manager which maintains the access control rights, security keys, etc. Security is provided at two levels of granularity: by tuple-space and by tuple. Figure 8 shows the architecture of the system with its security veneer. The upper layer shows SPAWN, which handles all the service provision details. The security veneer is at the interface of SPAWN and Limone, where it acts as an interceptor and filters all incoming and outgoing traffic, and enforces security and access control policies. Finally, the bottom layer (Limone) handles all the communication between hosts and agents.

Recall that in Limone, each agent owns a local tuple space. This tuple space is always left unsecured, meaning that any entity can access its contents. This tuple space is used for general service advertisements and communication. In addition, the security veneer allows an agent to create a secure logical tuple space (see Section II for details on logical tuple spaces). A secure logical tuple space has a name and a password associated with it. To access the secure logical tuple space, an agent must possess a handle to the logical tuple-space (obtained when the logical tuple space is created or by providing the name to the Limone system) and in addition must specify the password when performing any tuple space operations (e.g.,

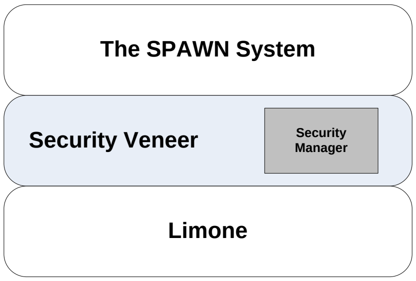


Fig. 8. Securing interactions in SPAWN.

`out`, `in`, `rd`) to ensure execution of the operation. This ensures that only those entities that possess the password can access the secure logical tuple space (we assume that the distribution of passwords is done by a third party key distribution software that is handled at the application level). The use of different names and passwords for logical tuple spaces enables creation of multiple (protected and/or unprotected) administrative domains where servers can publish services and clients can look them up, grouped by common interests or access rights. If a secure logical tuple space is created without a password, it functions the same way non-secure logical tuple spaces do, where only the handle (or the name to obtain the handle) is required to retrieve data.

Given the variety of data present in a repository and the multitude of users accessing it at any moment, often, the protection has to be extended to a finer grained level than the entire tuple space of service registry. The security veneer also provides secure tuples. A secure tuple is similar to standard Limone tuples but add two fields: a *read-password* which is required to read or copy the contents of the tuple, and a *remove-password* that is required to remove the tuple from the tuple space. The passwords are added when the tuple is placed in the tuple space. The *read-password* controls access to the content of the tuple, while the *remove-password* makes a tuple read-only to those entities that do not possess the password. Thus a service advertisement, which is essentially a Limone tuple, can be secured by using a secure tuple, which can be marked read-only by its provider protecting it with a *remove-password* that is not made public. This will prevent unauthorized (accidental or fraudulent) removal of the service advertisement or the replacement of the advertisement with a fake one. Additionally, each advertisement can be protected by a *read-password*, which enables a service provider to protect its services in a highly individualized manner. As with the passwords for the tuple space, if a password is not provided for a tuple, it behaves like an insecure tuple, e.g., if a *remove-password* is not provided, any entity can remove the tuple. Similarly, if a *read password* is not provided, any entity can read the contents. Note that in the case of both tuple-space-level and tuple-level security, the security manager is responsible for checking and verifying password protected access.

The secure logical tuple spaces and secure tuples can be used for more than just securing access to advertisements. They can be used for secure communication between the proxy on the client host and the service on the service provider's host. Further details are provided in the next section. A last observation is that the transient sharing of tuple spaces supported by Limone eliminates single points of failure scenarios. By simply disconnecting, the stability of a system as a whole is not affected and interactions with other devices continue to function normally.

VIII. Implementation

In this section, we will present the implementation details of Spawn and discuss the decisions made during the development process. We have chosen to implement Spawn in the Java language, due to its use of platform-independent bytecode. We use a subset of the Java API that is supported by Java 2 Standard Edition, as well as the Foundation and Personal profiles of Java 2 Micro Edition. This allows Spawn to operate on many different classes of mobile devices, ranging from PDAs to full-fledged notebook computers.

A. Tuple Space Communication

We chose to implement SPAWN on top of the federated tuple-space primitives provided by Limone. Tuples are implemented in Limone by mapping field names stored as Java Strings to Java objects; the object type of each field is inferred using reflection. Tuple spaces are likewise implemented as sets of tuples.

Templates are implemented as a set of constraints. Each constraint contains a name specified as a Java String, the Java class describing the kind of object to match, and a constraint function. A field is said to match a constraint if the field and the constraint have the same name; the field's type polymorphically matches the class in the template; and the field's value satisfies the constraint function. The two constraint functions typically used are `DefaultConstraintFunction` (any value is acceptable, provided the field name and type match the template) and `EquivalencyConstraintFunction` (in addition to the name and type matching, the value must be exactly equal to some other value). For the sake of brevity, in the remainder of this paper we will use `(A:B.class)` to specify constraints in the form `(A:B.class, DefaultConstraintFunction)` and `A:B` to specify constraints in the form (A:B.class, EquivalencyConstraintFunction(B)).

A template is said to match a tuple if all the corresponding fields (i.e., all pairs of fields bearing the same name) match pairwise with its constraints. Note that templates can under-specify the tuple by describing only some fields in the tuples to be matched: matched tuples may contain additional fields not described by the template. If the application developer does not want to allow under-specification for some tuples, then the application can incorporate *"required fields"* into the tuple by using the special `addRequiredField(...)` method. All of these required fields must be described and matched exactly by name, type, and value in the template; if not, the tuple/template comparison immediately fails. For example, in Figure 1, the "Destination" field of the template requires an exact match with a String of value "Europe", whereas the "Size" field specifies an Integer whose value is not important. The "Name" field is unspecified, indicating that the type and value are not important.

As discussed in Section II, Limone agents must use a remote operation manager to perform operations on other agents' tuple spaces. A Limone agent can query its `AcquaintanceList` to find the `AgentID` of any other agents it has discovered, whether they reside on the same host or on another host across the network. It can then use this ID to obtain an `AgentProxy` from the `AcquaintanceList` that refers to the remote agent in which it is interested. This `AgentProxy` contains methods for performing the basic tuple space operations described in Section II:**in**, **out** , etc.

When invoked, these methods use the `RemoteOpManager` to forward the request as a message to the appropriate agent. Each agent has an operations queue for processing incoming requests. If the receiving agent resides on the same host, then the `RemoteOpManager` places the message directly on the corresponding queue. Otherwise, the `RemoteOpManager` stamps the message with a unique ID, serializes it, and forwards it to the recipient as a TCP or UDP packet; there it is deserialized and placed on the remote queue. The agent that created the request then blocks waiting for a response.

Each agent has a thread that repeatedly removes a request from the head of the queue and services it. Once a request has been serviced, the result is placed into a message along with the original request's unique ID. If the request is non-blocking, like **rdgp**, then the "result" is an acknowledgment that the request was received. The response is then serialized and sent back to the original agent in a packet. Once the original agent's `RemoteOpManager` receives the response, it cross-references its ID (to ensure that the correct request was serviced) and unblocks, returning the result of the transaction to the agent. In order to prevent infinite blocking due to lost messages, the `RemoteOpManager` will time out and throw an exception if no response is received within some configurable period of time.

Each proxy can only refer to a single agent. Therefore, reading from tuple spaces shared across multiple remote agents must be done one-at-a-time by sequentially performing the operation on each agent. The results of these sequential operations must then be aggregated manually. Since this task is unwieldy for application developers, Spawn includes a `NamedTupleSpace` veneer which automatically performs these operations on all agents in the acquaintance list and aggregates the results.

ServerServiceDirectory()

— creates a server service directory with the default tuple space name

ServerServiceDirectory(String name)

— creates a server service directory with the specified tuple space name

AgentTask advertise(String serviceName, Class proxyType, ServiceProxy proxy, boolean wrap)

— advertises a service of type `proxyType` with name `serviceName` and proxy `proxy`;
the proxy is wrapped before deployment if `wrap` is `true`.

AgentTask upgrade(String serviceName, ServiceProxy oldProxy, ServiceProxy newProxy)

— transparently upgrades the wrapped service named `serviceName` from the old proxy `oldProxy` to the new proxy `newProxy`.

void strongMigrate(AgentTask service, HostID host)

— migrates a service `service` to a new host `host` with strong migration

void weakMigrate(AgentTask service, HostID host)

— migrates a service `service` to a new host `host` with weak migration

Fig. 9. ServerServiceDirectory's public interface.

B. *Service Advertisement and Discovery*

As discussed in Section I, Spawn uses a proxy-based approach to service interaction. Each advertised service must have exactly one associated proxy; therefore service advertisement and discovery can be achieved by performing advertisement and discovery on the proxies the services use.

We have chosen to implement proxy advertisement and discovery using the existing tuple space semantics provided by Limone. Proxies are placed in a shared tuple space (named `Proxies` by default) in the form `<Proxy:ServiceProxy, Attribute1:Value1, ..., AttributeN:ValueN>`. Interested clients can

ClientServiceDirectory()

— creates a client service directory with the default tuple space name

ClientServiceDirectory(String name)

— creates a client service directory with the specified tuple space name

ServiceProxy request(Class template)

— locates a service that implements the interface `template`.

Fig. 10. ClientServiceDirectory's public interface.

discover services by performing queries or registering reactions using a template that describes the kind of service requested.

In order to enable this service discovery, we assume that proxies implement well-known interfaces that correspond to the service they provide. For example, assume printing services implement the well-known `PrinterInterface` class. A host providing a printer service advertises a proxy `PrinterProxy` that implements `PrinterInterface`; this advertisement is placed in the tuple space in the form `<Proxy:PrinterProxy>`. Clients can discover this printer service using the template `<Proxy:PrinterInterface.class>`, which will polymorphically match all proxies that implement `PrinterInterface`. Clients can further filter the services discovered by incorporating desired attributes into the service request. For example, the template `<Proxy:PrinterInterface.class, DPI:GreaterThanOrEqualConstraint(300)>` will match all services that implement `PrinterInterface` and have a `DPI` attribute of at least 300.

C. Code Deployment

In order to allow clients to dynamically fetch proxy bytecode at runtime, services place all required proxy bytecode in a shared code repository. Like the service registry, the code repository is implemented using a shared tuple space. Unlike the service registry, which contains a set of proxy objects, the code repository contains the binary code for the classes that compose these objects.

When the service provider calls the `advertise(...)` method to publish a service, the SPAWN middleware analyzes the bytecode of the proxy object. We cannot use reflection for this task, since Java's reflection API does not provide a list of temporary or local objects used within a method; so we perform this analysis by parsing the bytecode and looking for all object references. While parsing the bytecode, SPAWN recursively extracts every data type instantiated or referred to inside the proxy object in order to obtain a class closure, and then automatically generates tuples that are placed in the `Bytecode` tuple space in the form `<Names:class names, BinaryCodeFile:bytecode, isJar:boolean>`. The first field of this tuple contains a list of all the classes it contains. The second field either contains a single Java class file (if the tuple contains only a single class) or a JAR file (if the tuple contains multiple classes) in the form of a byte array. The final field indicates whether the

BinaryCodeFile field contains a single class or a JAR file. Note that such tuples are not generated for classes that are part of the standard Java runtime or the SPAWN distribution, since we assume these classes are available on every host.

When the client obtains a service proxy by calling the request(...) method, the deserialization mechanism will throw a ClassNotFoundException if the proxy requires bytecode that is not available locally. SPAWN uses the custom LWObjectInputStream deserializer that intercepts any failed attempts to resolve classes locally. If a class is missing, then the deserializer catches the exception raised. It uses this exception to determine the name of the missing class and in turn invokes our custom LWClassLoader, which attempts a **rdp** operation on the code repository using the pattern <Names:class name, BinaryCodeFile:byte[].class> with the purpose of retrieving the bytecode for the required class from the code repository. If this **rdp** operation succeeds, the class loader extracts the bytecode from the tuple and presents it to the Java system class loader as a standard Java class. We use a non-blocking read when searching for the bytecode to prevent the class loader on the client machine from blocking indefinitely; otherwise this would result in all agents on the host being blocked, not just the one that initiated the call. An exception is thrown only if this non-blocking read fails, i.e., the bytecode cannot be found in the code repository.

Currently, the class loader stores any retrieved class bytecode 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 bytecode across multiple sessions. In future versions of our class loader, we plan to include support for a customizable persistent cache which, much like a web browser cache, is designed to save frequently-used bytecode across multiple sessions.

D. Transparent Service Upgrades

If the clients automatically retrieve proxy bytecode at runtime, it follows that they should also automatically upgrade the proxies at runtime. As described in Subsection VI-A, this can be accomplished by "wrapping" the proxy in a wrapper class that maintains the wrapped proxy's API. SPAWN can optionally generate these wrappers at runtime upon service deployment.

Our middleware uses Java's reflection features to dynamically determine a proxy's API. This API is used to combine several pieces of template 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 [10] 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 service advertisement tuple space. Clients can then use these wrapper proxies as ordinary proxies, unaware of the fact that the underlying implementation of the proxy can be swapped out on demand.

The wrapper contains code that places a reaction in the form <OldProxy:this.class,

NewProxy:underlying interface.class> onto the service advertisement tuple space in order to look for newer versions of the proxy. When the service is upgraded, the server automatically places a tuple in the form <OldProxy:old proxy, NewProxy:new proxy> into that tuple space, causing the reaction to fire. When the reaction is fired, indicating that the proxy has been upgraded, the wrapper performs a **rd** on this tuple to get the new proxy, and then replaces it on the client as soon as the existing wrapped proxy is unlocked.

This wrapping procedure saves considerable effort for proxy developers, since they need not specifically design the proxy with swapping in mind. Proxies are automatically wrapped if wrap is true when the ServerServiceDirectory's advertise(...) method is called to deploy the service. The proxy can later be upgraded with a single call to the ServerServiceDirectory's upgrade(...) method.

The code generated by wrapping services is inherently specific to the kind of user interface toolkit used by the proxy. SPAWN's current implementation includes support for proxies written using the Swing toolkit and the Standard Widget Toolkit [11]. The service wrapper is designed in a modular way, so support for proxies written in other UI toolkits can be added as required by proxy developers, or for proxies that do not have a graphical interface.

E. Service Migration

Service migration in SPAWN is equivalent to thread migration, since the advertise(...) method in ServerServiceDirectory returns an AgentTask object, which extends Java's Thread object. Once a service thread is encapsulated as described below, it can be placed in the shared Tasks tuple space. Willing recipients install reactions of the form <Task:AgentTask.class, HostID:local ID> onto this tuple space; once the reaction fires, they remove the tuple from the tuple space and continue the service's execution.

As discussed in Section V-B, there are two forms of thread migration: *strong* migration and *weak* migration. The latter form is simple to implement in SPAWN, since it only requires serializing the AgentTask object (which can be done using Java's built-in serialization routines) and placing it in a tuple of the form <Task:thread, HostID:recipient ID>. This tuple is placed in the shared Tasks tuple space. Willing recipients install reactions in the form <Task:AgentTask.class, HostID:local ID> on this tuple space. Once the reaction fires, they remove the tuple from the tuple space and continue its execution by invoking its start(...) method.

However, weak migration imposes serious performance penalties on the service. Restarting the service on the recipient host effectively destroys the service's state. If the service is not entirely stateless, then whatever computations were performed on the originating host are lost. Further, there is no safe way to stop a thread: stopping threads is unsupported or deprecated in many threading libraries, because it can stop the thread's execution at any arbitrary point, including points where the thread may have an exclusive lock on system resources. Therefore the thread must continue execution on the original host until it runs to completion, even after it has been migrated. If the service is no longer needed on the original host, then this is wasteful of resources, especially if the migrated service performs expensive calculations.

Strong migration migrates the thread's execution state along with the thread itself, which allows it to be resumed on the recipient host with little or no loss in computational progress. However, it is impractical to save and restore a thread's state directly in a platform-independent fashion for several reasons. First, there is no platform-independent way to save or restore the program counter, and many languages (including Java) prohibit access to it altogether for security reasons. Second, as discussed above, once an application has been migrated, it should stop running on the original host; but arbitrarily stopping threads in the middle of execution is inherently unsafe. Finally, saving the complete state of an application involves saving its local variables, which cannot be accessed at runtime by an external library.

We approached these problems by choosing to rewrite the bytecode of applications rather than trying to manipulate them at runtime. This rewriting process adds bytecode to applications to add support for strong code migration, including code to work around these technical limitations.

As noted in Subsection V-A, the state of the server application is saved at specific checkpoints. The application programmer creates mobile applications by extending the `MobileThread` class, which adds several methods and fields to Java's standard Thread class as described below. The programmer defines checkpoints by calling the `addCheckpoint()` method. Though this appears to the programmer to be an ordinary method call, it simply serves as a placeholder in the bytecode to indicate when the partial progress and state information are recorded. After compiling the Java source code, the resulting bytecode is passed to the bytecode rewriter. The rewriter loops through all the methods in the class and modifies them to allow strong code migration.

To do this, the rewriter first collects a list of all the local variables in the current method. It then adds a field for each of these local variables; these fields will be used later to store the state of the local variables. The rewriter also inserts a field to store an artificial program counter. The rewriter then searches for all calls to `addCheckpoint()`. At each checkpoint, the rewriter inserts code to check the `do_pause` field, which indicates whether or not the application thread is being paused so it can be migrated. If this field is set, then the method immediately returns. If it is not, then the method copies all of the in-scope local variables to the fields described above and then sets the artificial program counter to some unique value. Finally, the rewriter removes the call to `addCheckpoint()`, since it only serves to mark the bytecode. The rewriter also appends code at the end of the method to copy these fields back into the corresponding local variables and jump to the checkpoint; these "restoration points" provide a place for the thread to restore its state and return to the last checkpoint it passed before being migrated. The bytecode rewriter then adds code to the beginning of the method to see if the `paused` field is set. If it is, then the application jumps to the appropriate restoration point based on the contents of the artificial program counter field. This has the indirect effect of restoring the thread's local variables and the JVM program counter.

The `MobileThread` class adds two important methods to the standard `Thread` class: `pause(...)` and `unpause(...)`. The `pause(...)` method sets the `do_pause` and `paused` fields to `true`; the former tells the thread that it should stop execution as soon as it reaches the next checkpoint, and the latter tells the thread that it should restore its state when it is restarted. The `unpause(...)` method simply resets the `do_pause` field to false and restarts the thread; since the `pause(...)` call set the `paused` flag, the thread will jump to the appropriate restoration point and return to the last checkpoint passed before pausing. This way, rewritten applications can be migrated across hosts by pausing the application thread, serializing it on the original host, deserializing it on the new host, and unpausing it. Thus, strong migration is performed in much the same way as weak migration. The difference is that sender invokes the thread's `pause(...)` method before placing it in the tuple, and the recipient invokes its `unpause(...)` method rather than `start(...)` after extracting it from the tuple.

F. Limone Security Concerns

For the sake of this discussion, we assume that Java's built-in security policies completely protect against improper field access (e.g., an object cannot use a security flaw in the Java VM to access another object's private fields). Though we assume that the network medium is insecure and prone to sniffing, we do not consider physical-level attacks like wireless signal jamming.

Our first security concern is tuple space-level security: i.e., specific agents should be able to communicate privately using a shared tuple space without other agents being able to access the tuple space. This protection is implicit in Limone's named tuple space system, since agents must know a tuple space's name to perform operations on it. Limone does not expose the names of the available tuple spaces to applications; therefore agents may privately negotiate a secret tuple space name which other agents cannot determine. This negotiation may be done ahead-of-time or by using the tuple-level security measures described below.

Our second security concern is tuple-level security: i.e., agents should be able to restrict access to specific tuples, even those contained in unsecured tuple spaces. To implement this security policy, the `SecureTuple` class contains two optional passwords accessible only by the tuple itself. These passwords, known as the *read-password* and *remove-password*, respectively restrict **rd** and **in** operations to templates that contain the respective passwords. The password-checking is performed in the `SecureTuple` class as a part of the template-matching mechanism, ensuring that the passwords cannot be exposed outside of the middleware to malicious agents. These passwords are not interchangeable: the read-password cannot be used to perform an **in** operation, and vice-versa.

While tuple space-level and tuple-level security protect tuple access on a single host, they do not provide adequate protection for inter-host transactions. Limone's shared tuple spaces are implemented using a message-passing mechanism, which leaves tuples and templates being shipped across the network vulnerable to eavesdropping. In order to alleviate this problem, we incorporated message-level encryption using the interceptor design pattern. The encryption interceptor contains a `SecurityTable` structure that allows agents to assign passwords to individual tuple spaces. As with the tuple space names and tuple passwords, access to these passwords is constrained to the `EncryptionInterceptor` class, ensuring that they are not exposed to agents. When messages referring to a protected tuple space leave or enter a host, the interceptor encrypts or decrypts the message with the AES encryption algorithm, using a key generated from the password. If a host receives a message for which it does not have a password, then it immediately rejects the message. Thus, when tuples are placed in password-protected tuple spaces, their contents are protected against link-level eavesdropping attacks.

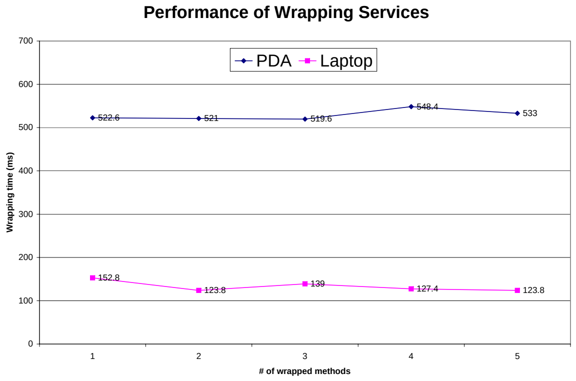


Fig. 11. An evaluation of the time required to wrap proxies.

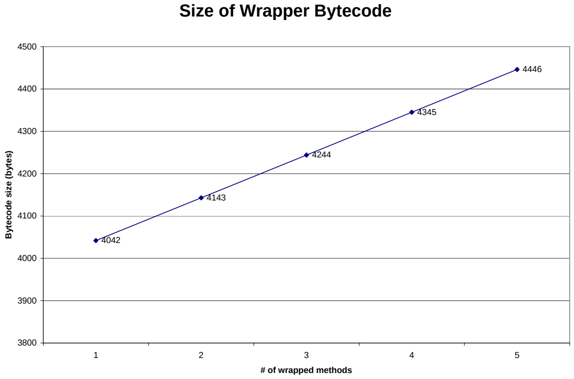


Fig. 12. An evaluation of the size of the resulting wrappers.

IX. Evaluation

In this section, we discuss the performance of Spawn. We performed these evaluations on two testbed computers. The first, a 744 MHz Pentium III-based laptop running Linux 2.4.20, represents a typical device that might consume or provide services in an ad hoc network, such as an ultraportable laptop. The second, a 624 MHz PXA270-based PDA running Windows Mobile 2003, represents the kind of device we expect to consume (but, due to resource constraints, not to provide) services. These devices were equipped with Spawn running on the Sun JRE 1.4.2 and IBM WebSphere Everyplace Micro Environment 5.7.1 Java runtimes, respectively. Communication between the two computers occurred over an ad-hoc Wi-Fi network, with the PDA using an integrated Wi-Fi chipset and the laptop using an Orinoco Gold PC Card adapter.

A. Service Wrapper

As was discussed in Subsection VIII-D, the wrapping portion of Spawn is specific to the user interface toolkit library used for implementing the proxy. For the sake of these tests, we wrote proxies using the SWT toolkit, since the Swing library is not present on our PDAs.

In order to measure the performance of wrapping services on-the-fly, we generated five “dummy” proxies designed for the clients using SWT toolkit. Each of these proxies contains from one to five methods, along with a stub constructor and stubs for the four standard methods required for proxies written with SWT. These proxies were each wrapped five times, with the average time needed to wrap each proxy shown in Figure 11. The size of the generated proxies is shown in Figure 12.

It is worth noting that there is no apparent correlation between the number of methods in the proxy and the time required to wrap it. This indicates that the majority of the time required to wrap a proxy is spent on overhead (e.g., initializing the compiler) independent of the proxy itself. The time required to wrap a proxy before deployment ranges from 123.8 ms to 152.8 ms on the laptop computer, and from 519.6 ms to 548.4 ms on the PDA. It is unrealistic to deploy useful services on devices with as limited computational resources as our PDA, so we consider 500 - 600 ms to be a conservative upper bound on the time required to wrap a service.

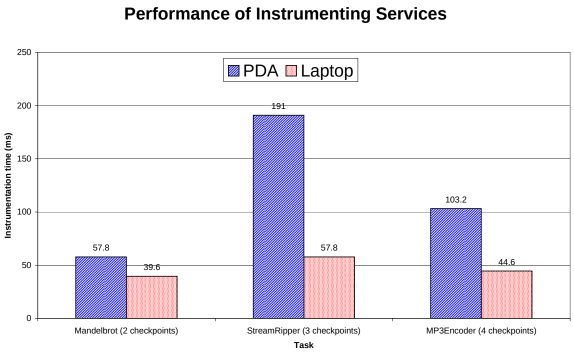


Fig. 13. An evaluation of the time required to instrument services.

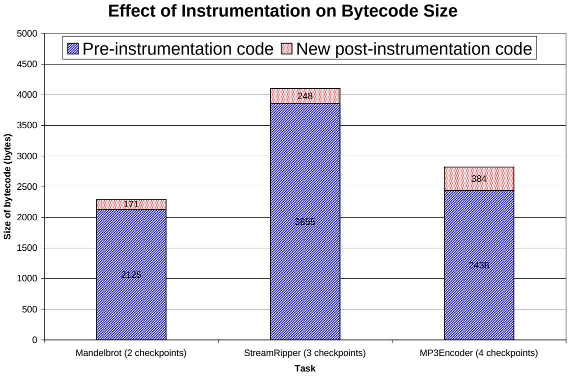


Fig. 14. An evaluation of the size of the services, pre- and post-instrumentation.

Unsurprisingly, the size of the generated proxies is linear with respect to the number of methods wrapped. The wrappers are on the order of 4 to 5 kilobytes, with a cost of approximately 100 bytes per each additional method. We feel that this is entirely acceptable, since the size of the wrapper will be dwarfed by the size of any non-trivial proxy.

Note that the methods we wrapped have no return values or parameters. Adding return values or parameters would increase the size of each method in the wrapper, since these values would have to be pushed or popped off the stack. However, we consider this insignificant, since these instructions would only add a few bytes to methods with reasonable numbers of parameters.

B. Thread Rewriter

To study the performance effects of instrumenting service bytecode, we implemented three simple services. The first, `Mandelbrot`, generates a bitmap image of a Mandelbrot fractal. The second, `StreamRipper`, connects to an Internet radio station and saves the sound to a buffer for later listening. The final, `MP3Encoder`, encodes an MP3 file from an input waveform. These three tasks were chosen because they contain loops with many iterations, which reflects the kinds of tasks that are most likely to be instrumented to support migration. Checkpoints were inserted in each task where appropriate; the three tasks respectively contained two, three, and four checkpoints.

To evaluate the time required to instrument a service's bytecode, we instrumented each of the services five times. The averages of these five runs are shown in Figure 13. Even on our PDA, the rewriter took less than 200 ms to instrument each task; this was further reduced to under 60 ms on the laptop. Interestingly, instrumenting the `StreamRipper` task was slower than the `MP3Encoder` task, even though the latter had more checkpoints. This occurred because the rewriter must perform some analysis on all local and temporary variables, and the former task had many more local variables than the latter.

Since the instrumentation process must necessarily increase the size of the bytecode, we also examined the size of the resulting bytecode. This data is shown in Figure 14. The services increased by approximately 6% to 16%

due to code and fields added during instrumentation. We do not consider this to be a significant increase, especially since these tasks would presumably be executed on hosts where storage is not a major constraint.

Finally, we must consider the runtime performance of the instrumented bytecode versus the original bytecode. To evaluate this potential overhead, we ran the uninstrumented `Mandelbrot` task five times each on the PDA and the laptop. We repeated this test using an instrumented version of the task to see if the instrumented code carried any significant overhead. We did not evaluate the `MP3Encoder` in this fashion since its performance can vary significantly based on the data being encoded. Likewise, we did not evaluate the `StreamRipper`'s runtime performance since it is not CPU-bound.

On average, the uninstrumented task completed in 3424.4 ms on the PDA and 85.0 ms on the laptop. In comparison, the instrumented version completed in 3421.2 ms on the PDA and 84.8 ms on the laptop on average. This difference is statistically insignificant, which suggests that instrumentation has little impact on the task's performance.

C. Tuple Space Performance

To test the performance of tuple-space operations, we wrote a service that creates a shared tuple space and installs a reaction onto this tuple space. When this reaction fires, the service places a second tuple in the tuple space that echoes the first tuple. On the client, the proxy places a tuple in this tuple space and installs a reaction that fires when the server echoes back the original tuple; it repeats this process when the reaction fires. By measuring the time between the arriving tuples, the service provider can estimate the "round-trip time" on typical tuple-space operations.

We deployed the service on the laptop and let the proxy run on the PDA for 40 iterations. The RTT for these tuple-space operations ranged from 259 ms to 525 ms, with an average of 339.6 ms. Thus, the service user can expect a latency on the order of 300 - 500 ms from the time he issues a command to the time he receives a response from the server, plus any time the server needs to perform computations. This latency should be more than acceptable for most services, though it may be more noticeable in interactive applications where the proxy must frequently communicate with the client.

D. Service Discovery and Upgrades

Like the previous service, in this evaluation the service creates a shared tuple space and installs a reaction onto it. When the proxy starts on the client, it places a tuple into the tuple space. When the service's reaction fires from this new tuple, it shuts down and upgrades itself to a second service, along with an associated new proxy. This new proxy places a second tuple into the tuple space, which the new service echoes.

As with the last service, the service was deployed on the laptop and the proxy was deployed on the PDA. We ran the proxy five times with the proxy and wrapper code downloaded on-the-fly from the service provider, and five times with the bytecode predeployed on the client. We started the service each time the proxy was restarted.

When the bytecode was predeployed on the client, it took an average of 934.8 ms from the time the service was started to the time the first service's reaction fired. This increased to an average of 3239.4 ms when the proxy was dynamically deployed at runtime. Since tuple space operations have a RTT of 340 ms, we can assume that the reaction fired about 170 ms after the client discovered the service. So, the discovery process required about 800 ms, with another 2000 ms to dynamically deploy the proxy code. We chose to deploy the proxy, the wrapper, and the wrapper's two inner classes in four separate tuples. Had we packed them together in a single .JAR file, the deployment would have taken roughly a second less, since three fewer round-trip tuple space operations would have been needed.

When the proxy was pre-deployed on the client, the service provider then spent an average of 150.4 ms from the time the reaction fired to the time the service was fully upgraded. This process instead took an average of 162.8 ms if the proxy was not pre-deployed. The difference in these averages is not statistically significant, which is to be expected, since the service provider's operation is not affected by availability of bytecode on the client.

After the service upgrade completed on the service provider, it took an average of 1706.0 ms for the new proxy to place the second tuple in the shared tuple space when the new proxy was predeployed on the client. This increased to an average of 2491.6 ms when the proxy bytecode was downloaded on-the-fly.

X. Related Work

A. *Service Oriented Computing*

Universal Description, Discovery and Integration [12] was formulated jointly by IBM Corp., Ariba Inc., and Microsoft Corp. UDDI technology is aimed at promoting interoperability among Web Services [2]. It specifies two frameworks, one to describe services and another to discover them. UDDI uses existing standards such as Extensible Markup Language (XML), Hypertext Transfer Protocol (HTTP) and Simple Object Access Protocol (SOAP). The UDDI model employs a central repository which is called the UDDI Business Registry (UBR). A simple XML document is used to specify the properties and capabilities of a service. The UBR acts as a mediator and assigns a unique identifier to each business and each service. Marketplaces, search engines and enterprise level applications query the UBR for services, which they use to integrate their software with other business entities.

Jini [3] is a Java-based service oriented architecture developed by Sun Microsystems. The platform independence of Java enables code mobility to support service distribution. Jini clients search for services in directories managed by lookup services, where providers register proxy objects (at least one such service must exist in a Jini community). The use of proxy objects allows for the separation of the service implementation from its interface. The user of a proxy only needs to understand the interface the proxy exposes and nothing about the how the proxy delivers the functionality (e.g., whether the proxy does the entire computation or connects to a server, what communication protocol proxy uses with such a server, etc.) The preferred communication protocol is Java RMI, but is not enforced as the only acceptable protocol. The code of the proxy object is transferred using a web server and HTTP.

Service Location Protocol [13] is a language neutral protocol for IP networks, with interfaces available for C and Java. User Agents search for services on behalf of the clients. Service Agents advertise services on behalf of service providers. User Agents and Service Agents interact directly or with the help of Directory Agents (if at least one Directory Agent is available this is the preferred mode of operation). Directory Agents are similar to lookup services in Jini. They only contain service contact information, but no code. The interaction between the two (e.g., communication protocol) is not a concern. SLP employs a standard set of service types and a standard set of descriptive attributes. User Agents search for services by type and then narrow the result set using the values of descriptive attributes.

Universal Plug and Play [14] is a set of protocols developed by an industry consortium led by Microsoft. UPnP does not entail any mobile code. It instead standardizes the protocols the clients and servers use to interact. Service descriptions are expressed in XML. The advertisement and discovery are handled in SSDP [15] style. Clients and servers advertise their needs and/or presence directly using unicast (when the location of the party is known), or multicast. The announcements happen when the process starts up and periodically thereafter. For invocation, the clients and the servers use SOAP [16]. The GENA [17] event mechanism is also part of the UPnP group of protocols.

Salutation [18] is an open standard service discovery and utilization model formulated by the Salutation Consortium. Salutation aims to promote interoperability among heterogeneous devices in settings such as corporate LANs where there is permanent connectivity from either wired networks or wireless gateways and disconnection is not an issue. In addition, Salutation strives to be platform, processor, and protocol agnostic. Salutation has two major components: (1) the Salutation Manager (SLM) which presents a transport independent API (SLM-API) and (2) The Transport Manager (TM) which is dependent on network transport and presents an interface (SLM-TI) between the Salutation Manager and the Transport Manager. Service discovery occurs when SLMs find other SLMs and the services registered with them. Capability exchange is done by type and attribute matching. The SLM protocol uses Sun’s ONC RPC [19].

B. Code Management

Binary code deployment and reuse has been around for a long time and gave birth to the component-oriented computing research. While a migration towards the component-oriented software seems inevitable [20], the complexity of the technology remains a significant challenge [21].

Enterprise JavaBeans (EJB) [22] is one of the widely accepted architectures for the deployment of component-based applications. EJB addresses the entire software life cycle, including application assembly and deployment. EJB components (beans) are assumed to run in virtual environments, exclusively on the server side (different from the proxy object approach where processing on the client’s host is possible). The model of interaction is synchronous where all client’s calls are tunneled to the server where they are addressed while the client application remains blocked.

Binary Component Adaptation [23] modifies the binary code of a component in an effort to increase component integration and support interface evolution in situations where the interface offered by the component does not match the interface expected by the application using the component.

C. Process Migration

Most of the related work in this area comes from the mobile agents research community. Some of the platforms that support weak migration are [24] and [25]. Strong mobility is more powerful but it is also more expensive to deliver. Strong migration is supported by [26] and [27]. Ideally, the migration occurs in a manner completely transparent to the subject process. A system that achieves transparent process migration, in cooperation with the operating system is [28]. For an excellent presentation of related work in code mobility we point the reader to the survey paper by Fuggetta et al [29].

More recent work is μCode [30], a Java implementation of a code migration mechanism. μCode resembles TACOMA [24], Sumatra [31], and Mole [25] [32], as it figures the transitive closure of the classes needed during migration but, unlike our implementation, does not handle the transfer of binaries, and does not deliver strong process migration. It transfers some state information in the form of some data used to initialize the newly migrated thread, which is started from scratch and does not resume execution from the point where it was before migration.

D. Runtime Software Upgrade

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 while it is running. Hence, the technical problems associated with upgrading a proxy are similar to those encountered with upgrading components within an application. In [33], the authors posit that it is a challenge to achieve a balance between flexibility, correctness, ease of use, and low overhead. 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 to component upgrade in wired networks have a distinctive heavyweight flavor.

The approach described in [34] 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 upgrades 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 [35], the authors suggest maintaining both the old and new versions of the component concurrently and sending each call to the version to which it applies. The older version is destroyed only when it is verified that the new version correctly replaces the old version for all required functionality. Flexible software connectors, as proposed in [36] do not use multiple servers. Instead, the connectors (called multi-versioning connectors) themselves determine correct points during execution when components may be swapped.

There also exists a fair amount of infrastructure supporting component upgrade. [37] suggests a mechanism for consistency checks to ensure that the new component works with all the other old components. [38] proposes 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 working on constrained devices in ad hoc networks.

E. Security

Security issues in distributed systems are addressed in [39], along with discussions about threats and protection mechanisms. A capability-based security system is presented in [40]. The authentication mechanism is similar to ours, in that the capabilities can be verified locally, as opposed to an access control list approach where a central server is needed (e.g., Lampson's access matrix [41]). In [42] the authors describe an infrastructure for secure service discovery which offers privacy and authentication at the expense of a loaded infrastructure and centralized architecture allowing single points of failure. Proxy-based security protocols for mobile devices are presented in [43], but these also rely on a relatively centralized architecture for accomplishing some key tasks. Security is accomplished by adapting SPKI/SDSI for proxy-server and/or proxy-proxy interactions. In [44] the authors use public keys as authentication certificates for Jini services, manually managed in local databases as initial trust relationships. In the Service Location Protocol, authentication is done using public key encryption and having trust relationships between directory agents and service agents defined by the network administrator [45].

XI. Discussion

In designing our model, we chose a tuple space-based coordination model for several reasons. First, the tuple space based model allows the decoupling of interacting entities (in our case, the server from client (or proxy) during service utilization, or the client and server from the service registry during advertisement/lookup, etc.) 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. 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 for use in ad hoc networks in Limone, which we use as a basis for the implementation of our model.

Limone offers support for implementing the service model in the mobile ad hoc networking environment. The transient sharing of tuple spaces used in Limone enables the atomic update of the service registry, maintaining its consistency across connected hosts. A single interface allows access to the entire federated tuple space as if it were local. However, some changes in the functionality of Limone were required. The initial public release of Limone has typical Linda-style pattern-matching capabilities. The actuals in the template fields had to match exactly the type of the corresponding fields in the tuple being examined. It turns out that we needed additional flexibility in our implementation. We changed the matching algorithm to allow polymorphism in pattern matching, i.e., a field containing a formal in a template will match the corresponding field in a tuple if the latter implements the interface, or subclasses the pattern type. This is necessary in order to enable clients to use services they discover for the first time and for which they know only an interface that the service implements. The stricter pattern matching would have required the client to already have the class file of the proxy it receives, thus reducing the usability of the services.

The semantics of the lookup operation can vary from implementation to implementation. One could choose to block the client until the lookup operation returns successfully. Another implementation may allow the client to continue execution if an attempt to use a service fails. A third case may allow the client to announce its interest in a service and its desire to be notified when the service becomes available. In some cases, the client may need more than one service of a given type, an implementation of the lookup primitive that handles groups of tuples. Other situations may permit a client to use a service whenever the latter becomes available. All these implementations of the lookup primitive are easily constructed on top of Limone.

In comparison to Jini, our middleware can be thought of as a richer implementation of Jini for ad hoc networks. While never explicitly stated, Jini assumes a certain degree of network stability. The centralized design Jini employs is inappropriate for ad hoc networks. As described, a client and a server can interact only if they are "introduced" to each other by the lookup server which usually runs on a different host, acting as a bottleneck and single point of failure. A client may also discover orphan advertisements in lookup repositories after the host offering the service has left the community. Jini employs a leasing mechanism that collects such advertisements but there is no relation between the lease expiration time and the moment the server that advertised a service in a certain lookup service disconnects. Our transiently shared tuple spaces and the service advertisement repositories developed on top of such tuple spaces eliminate both problems.

Jini uses the Java RMI mechanism for remote proxy-server interactions, although this protocol is not mandatory. In RMI all processing is done on the server side. This may be too expensive in a dynamic environment where connections may break down easily and any processing done by the proxy on the client machine can save important communication overhead. The proxy model used both by Jini and our model can help deliver part of the functionality locally. If a connection breaks, our tuple space-based implementation does not jeopardize the applications since it does not raise exceptions or lose messages. Messages sent while the two ends of a communication session are disconnected are stored and delivered when they reconnect. However, messages sent over RMI while the recipient is disconnected are lost without the senders notification, leading to inconsistencies in applications behavior. Eventually, an I/O exception is raised, but it is possible to send messages over RMI (initiate methods calls) between the disconnection and the moment the exception signals the communication problem to the client. Applications have no means to determine when the disconnection occurred and which messages have been lost.

Another major RMI drawback is the need to know *a priori* the URL of every single object that is used remotely. This URL has the form [*IPaddress:port/ObjectName*], where *IPaddress:port* identifies the location where an RMI server runs and *ObjectName* identifies an object registered with that RMI server. There is no discovery process that allows a client to learn about such objects on the fly. A DHCP-based network will create problems for applications as they need to be reconfigured (manually) every time a server obtains a new address. First-time encounters are completely out of reach for this protocol (since it does not entail a discovery phase, the parts must have a priori knowledge of each other). Both our approach and Jini solve this problem via discovery. In addition, our communication protocols are tuple space-based, which makes them independent of the networking protocol. The communication is content-based (a template is matched against the tuples content) and the location of the two is irrelevant. This allows collaboration with services that change their IP addresses over time (e.g., software migration).

One final drawback is the way RMI (and Jini) handles the byte code of the object(s). The byte code that needs to be transferred between hosts has to be added manually to a repository. For the transfer itself RMI (and Jini) uses an HTTP server, with which the byte code repository has to be registered at startup. Since the process of adding class files to this repository is manual, forgetting one class file can crash an application at runtime (a costly error, undecidable at compile time). In addition to this manual configuration, all classes can only come from one single host. In contrast, our approach handles all byte code deployment automatically, by code inspection. We also support gathering code from multiple sources if available, supporting service composition, and partial code upgrade.

We now turn our attention to general issues associated with service upgrade and how we handle them in our architecture. Recall that in Section VI, 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 simulates synchronous and atomic calls between itself and the server, while it actually uses Limone, which entails 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 a versioning control system - such as CVS - to merge different versions of the same file.)

For future work, we aim to develop an architecture that decomposes the proxy such that it 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.

Service mobility is a completely new feature we introduced with SPAWN. The tuple space coordination we used and the automatic code management mechanism we developed support service mobility effectively. There are, however, certain limitations to our mobile services. There are situations, for example, when the service needs specific hardware that is attached to a host and is not available on any of the other nearby hosts. In this case, a service can still migrate to follow the client host (maybe even carried by the client host), looking out for the opportunity to jump on a host that has the needed hardware, and resume execution. For example, imagine a printing service that sends documents to printers ahead of the user on his way through a department. While not all computers are attached to printers, the print manager service can follow the PDA running the user's client application and do work only when in proximity to a printer or a computer with a printer. Dynamic binding, also supported by tuple space-based interactions, holds much potential for future development. For now, the only thing that triggers dynamic rebinding (assuming an alternative is available) is the imminent disconnection from the current host. The rebinding policy currently used is geared towards choosing the host that offers the longest estimated period of connectivity among all hosts that offer that service. This policy can be improved in the future by taking into account the computational power of hosts, security concerns, etc.

The security extensions we provided for Limone are targeted towards the safety of remote interactions. There still are concerns related to the safety of downloading and running code from some other party. The can be tackled form two directions: how to protect the downloaded code from being inspected or modified by a malicious host, and how to protect a host from malicious code. Efforts toward solving the first problem include computing with encrypted functions [46]. The solution to the second problem relies on the authentication of the party from which the code is being downloaded and used and on restricting the types of operations the code can perform (run the code in a sandbox with access control) on the new host [47]. Our security extensions provide a foundation for secure public key advertisement. Once this mechanism is available, a multitude of protocols can be run to establish secure interaction channels among remote parts of an application. We provide the basic mechanisms for such protocols in case the application wants to develop its own protocol, as well as a support infrastructure for secure interactions that are ready and available for use.

XII. Conclusions

In this paper we presented a technique that supports consistent service advertisement and discovery in ad hoc networks. A key feature of our approach is that service registry updates are atomic with respect to changes in service availability due to mobility and disconnections. We also introduced security features that can provide for safe service advertisement and utilization in ad hoc networks. The model presented provides the necessary support for safe distribution of public keys in ad hoc networks. Once this mechanism is in place, it enables either the development of algorithms for establishing session keys or simply the distribution of secret keys. The entire architecture has been geared to deal with the dynamism of ad hoc networks and the resultant opportunistic interactions between hosts. The outcome is a system that significantly simplifies the task of a mobile application programmer by abstracting all details of code management. This allows for programming 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. We also presented a lightweight mechanism to upgrade services without completely shutting them down. 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 Limone coordination model, justified our design decisions, and presented future work plans.

References

[1] Palm Source Website, ," http://www.palmsource.com/palmos/, January 2005.

[2] H. Kuno V. Machiraju G. Alonso, F. Casati, *Web Services: Concepts, Architectures and Applications*, Springer Verlag, 2004.

[3] Jim Waldo, “The Jini Architecture for Network-Centric Computing,” *Communications of the ACM*, vol. 42, no. 7, pp. 76–82, 1999.

[4] Greg Hackmann Chien Liang Fok, Gruia-Catalin Roman, “A lightweight coordination middleware for mobile computing,” in *Proceedings of the 6th International Conference on Coordination Models and Languages*. 2004, vol. 2949 of *LNCS*, pp. 135–151, Springer Verlag.

[5] David Gerlenter, “Generative communication in linda,” *ACM Computing Surveys*, vol. 7, pp. 80 – 112, January 1985.

[6] A.L. Murphy, G.P. Picco, and G.-C. Roman, “LIME: A middleware for physical and logical mobility,” in *Proc. of the 21st Int’l Conf. on Distributed Computing Systems*, April 2001, pp. 524–533.

[7] Sun-Microsystems, “Java remote method invocation page,” http://java.sun.com/ products/jdk/rmi/, October 2003.

[8] 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.

[9] 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.

[10] JDT-Core Project, “The jdt-core homepage,” http://dev.eclipse.org /viewcvs/index.cgi/%7Echeckout%7E/jdt-core-home, January 2005.

[11] Eclipse Platform Project, “Swt standard widget toolkit,” http://www.eclipse.org/swt, January 2005.

[12] UDDI-Organization, “Uddi technical white paper,” http://www.uddi.org /pubs/Iru/UDDI/Technical/White/Paper.pdf, 2000.

[13] E. Guttmann, C. Perkins, J. Veizades, and M. Day, “Service location protocol,” IETF Internet Draft, RFC 2608, 1999.

[14] Microsoft-Corporation, “Universal plug and play device architecture,” http://www.upnp.org/download/UPnPDA10_20000613.htm, June 2000.

[15] Y. Goland, T. Cai, P. Leach, Y. Gu, and S. Albright, “Simple service discovery protocol/1.0: Operating without an arbiter,” http://www.upnp.org/download/draft_cai_ssdp_v1_03.txt, 2001.

[16] XML-Protocol-Working-Group, “W3c recommendation: Soap version 1.2 parts 0-2,” http://www.w3.org/TR/SOAP/, June 2003.

[17] J. Cohen and S. Aggarwal, “General event notification architecture,” http://www.globecom.net/ietf/draft/draft-cohen-gena-p-base-01.html, July 1998.

[18] Salutation-Consortium, “The salutation consortium homepage,” http://www.salutation.org, October 2003.

[19] J. Srinivas, “Open network computing remote procedure call protocol specification,” http://www.ietf.org/rfc/rfc1831.txt, August 1995.

[20] Clemens Szyperski, *Component Software, Beyond Object-Oriented Programming*, ACM Press - Addison-Wesley, 1997.

[21] Clemens Szyperski, *Foundations of Component-based Systems*, chapter Component Software and the Way Ahead, pp. 1–20, Cambridge University Pres, 2000.

[22] L. G.DeMichiel, L. U. Yalcinalp, and S. Krishnan, “Enterprise java beans specification,” Sun Microsystems, 2000.

[23] Ralph Keller and Urs Hölzle, “Binary component adaptation,” *Lecture Notes in Computer Science*, vol. 1445, pp. 307–324, 1998.

[24] D. Johansen, R. van Renesse, and F. B. Schneider, “An introduction to the TACOMA distributed system—version 1.0,” Tech. Rep. 95-23, June 1995.

[25] Joachim Baumann, Fritz Hohl, and Kurt Rothermel, “Mole - concepts of a mobile agent system,” in *Proceedings of the 2nd ECOOP Workshop on Mobile Object Systems*, 1997.

[26] Holger Peine and Torsten Stolpmann, “The architecture of the Ara platform for mobile agents,” in *1st Int’l Workshop on Mobile Agents*. 1997, vol. 1219 of *LNCS*, p. 50, Springer Verlag.

[27] R. S. Gray, “Agent Tcl: A flexible and secure mobile-agent system,” in *Fourth Annual Tcl/Tk Workshop (TCL 96)*, 1996, pp. 9–23.

[28] Mark Claypool and David Finkel, “Transparent process migration for distributed applications in a Beowulf cluster,” in *Proc. of the Int’l Networking Conf.*, July 2002.

[29] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna, “Understanding Code Mobility,” *IEEE Transactions on Software Engineering*, vol. 24, no. 5, pp. 342–361, 1998.

[30] G. P. Picco, “μCode: A lightweight and flexible mobile code toolkit,” in *Proceedings of the 2nd International Workshop on Mobile Agents*, September 1998, vol. 1477 of *LNCS*, pp. 160–171.

[31] Anurag Acharya, M. Ranganathan, and Joel Saltz, “Sumatra: A Language for Resource-aware Mobile Programs,” in *Mobile Object Systems: Towards the Programmable Internet*, J. Vitek and C. Tschudin, Eds., vol. 1222, pp. 111–130. Springer-Verlag: Heidelberg, Germany, 1997.

[32] Joachim Baumann, Fritz Hohl, Nikolaos Radouniklis, Kurt Rothermel, and Markus Stra&#223;er, “Communication concepts for mobile agent systems,” in *MA ’97: Proceedings of the First International Workshop on Mobile Agents*. 1997, pp. 123–135, Springer-Verlag.

[33] Michael Hicks, Jonathan Moore, and Scott Nettles, "Dynamic software updating," in *Proceedings of the ACM SIGPLAN Workshop on Types in Compilation*, September 2000.

[34] 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.

[35] Jonathan Cook and Jeffrey Dage, "Highly reliable upgrading of components," in *Proceedings of the 1999 International Conference on Software Engineering*, 1999, pp. 203–212.

[36] 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.

[37] Premysl Brada, "Component change and verification in sofa," in *Proceedings of SOFSEM 1999*. 1999, number 1725 in LNCS, Springer.

[38] Vladimir Mencl, Zuzana Petrova, and Frantisek Platil, "Update description language," June 1999.

[39] Jean-Pierre Hubaux, Levente Buttyan, and Srdan Capkun, "The quest for security in mobile ad hoc networks," in *ACM MobiHOC 2001 Symposium*.

[40] Li Gong, "A secure identity-based capability system," in *Proceedings of the IEEE Symposium on Security and Privacy*, 1989, pp. 56–63.

[41] Butler Lampson, "Protection," in *5th Princeton Conf. on Information Sciences and Systems*, 1971, vol. ACM Operating Systems Rev. 8, pp. 18–24.

[42] Steven E. Czerwinski, Ben Y. Zhao, Todd D. Hodes, Anthony D. Joseph, and Randy H. Katz, "An architecture for a secure service discovery service," in *Mobile Computing and Networking*, 1999, pp. 24–35.

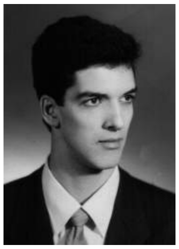
[43] Matthew Burnside, Dwaine Clarke, Todd Mills, Srinivas Devadas, and Ronald Rivest, "Proxy-based security protocols in networked mobile devices," in *Proceedings of Selected Areas in Cryptography*, 2002.

[44] Pasi Eronen, Johannes Lehtinen, Jukka Zitting, and Pekka Nikander, "Extending jini with decentralized trust management," in *The Third IEEE Conference on Open Architectures and Network Programming (OPENARCH)*, 2000.

[45] Marco Vettorello, Christian Bettstetter, and Christian Schwingenschlgl, "Some notes on security in the service location protocol version 2 (slpv2)," in *Proc. Workshop on Ad hoc Communications, in conjunction with 7th European Conference on Computer Supported Cooperative Work (ECSCW'01)*, 2001.

[46] Tomas Sander and Christian F. Tschudin, "Protecting mobile agents against malicious hosts," *Lecture Notes in Computer Science*, vol. 1419, pp. 44–60, 1998.

[47] G. Karjoth, D.B. Lange, and M. Oshima, "Mobile agents and security," *Lecture Notes in Computer Science*, vol. 1419, pp. 188–205, 1998.



Radu Handorean received the B.S. degree in computer science from Politehnica University in Bucharest, Romania, in 1999, and a M.S. degree in computer science from Washington University in Saint Louis in 2003. He is a PhD candidate in the Department of Computer Science at Washington University in Saint Louis. He is a member of the Mobile Computing Laboratory. His research focuses on service provision, mobile computing, and ad hoc networks.

Gruia-Catalin Roman was a Fulbright Scholar at the University of Pennsylvania, in Philadelphia, where he received a B.S. degree (1973), an M.S. degree (1974), and a Ph.D. degree (1976), all in computer science. He has been on the faculty of the Department of Computer Science at Washington University in Saint Louis since 1976. Roman is a professor and chairman of the department.

Roman has an established research career with numerous published papers in a multiplicity of computer science areas including mobile computing, formal design methods, visualization, requirements and design methodologies for distributed systems, interactive high speed computer vision algorithms, formal languages, biomedical simulation, computer graphics, and distributed database. His current research involves the study of formal models, algorithms, design methods, and middleware for mobile computing.

Roman served as an associate editor for ACM TOSEM and will serve as general chair for ICSE 2005.

Roman is also a software engineering consultant. His consulting work involves development of custom software engineering methodologies and training programs.

Roman is a member of Tau Beta Pi, ACM, and IEEE Computer Society.

Rohan Sen received his B.S. Degree in Computer Science from Washington University in St. Louis. He is currently in his second year of study, working towards a doctoral degree in Computer Science at the same university. His research interests lie in software engineering and service-oriented computing for mobile ad-hoc networks.

Greg Hackmann received the B.S. degree in computer science from Washington University in St. Louis in 2004. He is currently pursuing a doctoral degree in computer science in the Computer Science and Engineering Department at Washington University. His current research interests include mobile computing and service deployment in mobile ad-hoc networks.

Christopher Gill was a national merit scholar at Washington University in Saint Louis, where he received a B.A. degree in English and Biology. He received a M.S. degree from the University of Missouri at Rolla (1997) and a D.Sc. degree from Washington University in Saint Louis (2002), both in Computer Science. He is an Assistant Professor in the Department of Computer Science and Engineering at Washington University in Saint Louis. His research, published in over 50 refereed and invited papers, focuses on middleware for distributed real-time, embedded and mobile systems. He is a member of ACM and the IEEE Computer Society.