Secure Service Provision in Ad Hoc Networks

Radu Handorean and Gruia-Catalin Roman

Department of Computer Science and Engineering
Washington University in St. Louis
Campus Box 1045, One Brookings Drive
St. Louis, MO 63130-4899, USA
{radu.handorean, roman}@wustl.edu

Abstract. Ad hoc networks are formed opportunistically as mobile devices come within wireless communication range of each other. Since individual devices are typically subject to severe resource limitations, it is both possible and desirable for a device to enhance its functionality by taking advantage (in a cooperative manner) of capabilities available on other devices. Service provision refers to the process by which devices advertise their willingness to offer specific services and discover other services. This paper describes a service provision model designed specifically for use in ad hoc settings. Security policies governing service accessibility can be specified at the application level while secure communication among devices is ensured by the implementation.

1 Introduction

Advances in portable computing and wireless technology are opening up exciting possibilities for the future of mobile networking. The opportunity for applications to exploit an ever-increasing range of resources is expanding rapidly. Any application or device can advertise and provide services, turning the network into a global service repository. As portable devices become cheaper and more powerful, their number is expected to grow significantly in the coming years. At the same time, the ability to offer highly specialized capabilities by means of devices directly connected to the network turns such devices into service providers. In such an environment the interest in and reliance upon particular services changes over time. In part, this is because devices having limited capabilities exhibit a growing dependence on services provided by others. Expressive and dependable means are required to support service discovery. Such environments demand the ability to obtain support for the task at hand only when needed.

Traditionally, a host sought help from another by means of client-server interactions. It is customary to assume that the client knows the address of the server that supports the service it needs, has the code necessary to access the server, and knows the communication protocol the server expects. While this type of interaction between remote components still dominates distributed computing, new strategies have emerged to allow servers to advertise services, and clients to lookup and access them without explicit knowledge of the network structure and

M.E. Orlowska et al. (Eds.): ICSOC 2003, LNCS 2910, pp. 367–383, 2003.

communication details. These discovery-based techniques are growing in importance, as services become ubiquitous. The high level of abstraction characterizing these techniques frees the client from handling the communication protocol.

To accomplish this, services may be discovered at runtime and may be accessed through customized proxies. A proxy abstracts away the network from the client by offering a high-level interface specialized for service exploitation while hiding the proxy's interface to the server. Services are advertised by publishing a profile containing attributes and capabilities useful when searching for a service and for proper service invocation. Clients search for services using templates generated according to their momentary needs. These templates must be matched by the advertised profiles. Services use a service registry to advertise their availability and clients use the same registry to search for services they need. This approach offers an unprecedented degree of run-time flexibility.

Mobile ad hoc networks are opportunistically formed structures that change in response to the movement of physically mobile hosts running potentially mobile code. New wireless technologies allow devices to freely join and leave networks, form communities, and exchange data and services at will, without the need for any infrastructure setup and system administration.

A major challenge in ad hoc networking is frequent disconnection among hosts which often leads to data inconsistency in centralized service registries. Since this may affect the structure of the network and the availability of services, this paper seeks to provide service registries that guarantee the consistency of their content, i.e., information about the service availability is updated atomically with respect to configuration changes.

Architectures based on centralized lookup directories are no longer suitable. Mobile ad hoc computing cannot rely on any fixed infrastructure. The interactions among devices are peer-to-peer and entail no external infrastructure support. Since nodes are mobile, the network topology may change rapidly and unpredictably over time. The network is decentralized and all activities, including topology discovery and message delivery must be carried out by the nodes themselves. The network structure at any moment depends on the available direct connectivity between mobile devices (unless ad hoc routing is used, i.e., nodes route packets among nodes that cannot reach each other directly).

As flexibility and ease of access to resources increases, so do the possibilities to tamper with various parts of a distributed application. This represents yet another major challenge to service provision in ad hoc settings. In open systems, where the network access is not controlled at the physical level, there is a special need for security mechanisms that facilitate safe interactions among remote components of a distributed application and guarantee the identity of processes that offer or use resources over the network. The advertisement of a service must be protected from unauthorized removal, replacement, or use.

The goal of this paper is to extend the applicability of these kinds of techniques to ad hoc mobility. The reminder of the paper is organized as follows. Section 2 introduces the service model, security mechanisms, and the challenges of the ad hoc networking environment. Section 3 presents our model for secure service provision. Section 4 describes the implementation of the model. Section 5 presents a test application developed using our model. Section 6 discusses related work. Section 7 concludes the paper.

2 Secure Service Provision

The service model is composed of three components: services, clients and a discovery technology. Services provide functionality to clients. Clients use services. The registration feature enables services to publish their capabilities and clients to find and use needed services. As a result of a successful lookup, a client may receive a piece of code that actually implements the service or facilitates the communication to the server offering the service. Part of the quality of a service being offered by a server is the security guarantees the server offers to potential clients. Among them is the guarantee that the service cannot be faked by an intruder and that the usage of the service is safe for the client (e.g., it does not reveal personal information about the client to the external world).

In a stationary setting, where all hosts access a wired (and therefore reliable) network, a server can be set up to run the service repository. All service providers can advertise their offers on this server and clients can connect to this server to search for the services they need. Security issues are easy 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) service ready to authenticate offers and requests.

In the ad hoc network case, both the discovery techniques and security considerations face new challenges. A simple transition of old software to new settings does not suffice. Service discovery protocols that rely on centralized service registries are prone to malfunctions, as described in Figure 1a: if the node hosting the service registry suddenly becomes unavailable, the entire advertising and lookup of services becomes paralyzed even if the nodes representing a service and a potential client remain connected. The goal is to make these two nodes communicate in a peer to peer fashion. Furthermore, because of frequent disconnections, the service registry should reflect changes affecting service availability in a timely fashion. Services that are no longer reachable should not be available for discovery. In Figure 1b we present a scenario that can happen in Jini [1,2], where the advertisement of a service is still available in the lookup table until its lease expires, even though the provider of the service is no longer reachable.

From the security point of view, in the ad hoc network case, 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. The two nodes will need to take steps to protect themselves. This leads to special safety measures that have to be taken both by clients and servers to ensure proper secure interaction in the ad hoc setting.

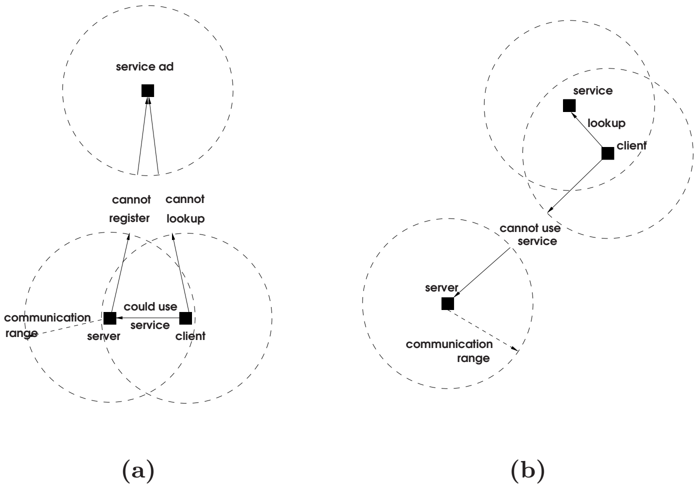


Fig. 1. Ad hoc environment challenges. **(a)** The client could use the service but cannot discover it. **(b)** The client discovers a service that is no longer reachable.

A new design solution is needed, one suited specifically for ad hoc networks. In a model addressing these issues, all nodes should be simple users or providers of services. The system should not depend on the behavior of a single node for service discovery or secure interactions. The new challenge is 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. The advertising and lookup of services in this setting need a lightweight model that supports direct communication and offers a higher degree of decoupling. A consistent, distributed, and simple implementation of the service registry is the key to the solution, along with a simple approach to trust management. Security enforcement is needed to protect the easily accessible service registries from tampering or unauthorized usage and the new model should address this issue to the maximum extent possible under the additional constraints imposed by ad hoc networking.

3 Secure Service Provision in Ad Hoc Settings

The new model, inspired by the service provision ideas described above, seeks to address challenges raised by the ad hoc environment while offering a reasonable level of security. Active entities are logically separated into service providers (servers) and service users (clients). Both clients and servers may be mobile agents able to migrate among reachable hosts. Hosts themselves are physically mobile and form ad hoc networks as connections are established and break down.

A service is the functionality offered by a program or a hardware device to other programs or hardware devices that need it at run time. In our model, the interface to the service is provided by a software object (proxy), that can be transferred over a network to the user of the service. The proxy will represent the service locally to the client. This distinguishes the approach from techniques where a client is expected to know a priori how to contact a specific server (i.e., its IP address and port number) and was expected also to handle all the communication details (i.e., the client was responsible for managing the communication protocol used by each server). The communication details between the proxy and the implementation of the service (i.e., the protocol being used, the handling of disconnections while work is in progress) are outside the client's area of concern. In some cases, the proxy can implement the entire service itself.

This approach is similar to Jini [1,2], which uses proxy objects and the RMI protocol to hide the communication details from the user of the service, and UPnP [3], which uses the SOAP [4] protocol to access objects remotely (the implementation of the service which runs on a different host). Models that do not aim to hide these details from the user of a service include SLP [5] and Bluetooth's SDP (Service Discovery Protocol) [6].

A service provider is any host that offers services over the network. Such a server registers (advertises) a service by publishing a profile that contains the capabilities of the service and attributes describing these capabilities, so as to allow clients to discover and use the services properly. Along with the profile, the server provides the service proxy. The server has the possibility to deregister a service previously advertised. This does not necessarily involve the termination of the service. Clients that are in process of using the service may be allowed to terminate their jobs in progress, depending on the implementation of the service.

A client searches for services using a template that defines what the needed service should do. The template contains attributes to request a certain level of quality. If a service profile satisfying all client's requirements is available (i.e., the advertised profile of the service is a superset of the client's template and the attributes advertised by the server meet the client's criteria), the service proxy, which is part of the service's profile, is returned to the client. This proxy handles the communication with the server offering the service. Location information can be considered a special attribute of the service's advertisement (if specified by server) but it is not required in order to discover the service requested by the client, i.e., the client does not *need* to know where the implementation of a service is running in order to discover the service, but can use the information as part of the quality of service evaluation. This information can map to a physical location (e.g., for a printer it is more important where it is physically located than what IP address it has) or to a logical location (e.g., for encryption services the client *may* care if the server is within its domain).

The registration, deregistration, and lookup processes take place in service registries. Service registries are identified by name. They are local to each agent (client or server) and are shared among agents that created service registries with the same name when the hosts they are running on are connected or when they run on the same host. Service registries shared by different agents under the same name form a single, federated service registry. After sharing its local service registry, each agent accesses the federated service registry as if it were local. The transient sharing of the service registries makes the update of their content transparent to disconnection and always consistent with respect to connectivity (Figure 2). Service availability is always consistent with advertisement availability and is based on (possibly multi-hop) connectivity among hosts.

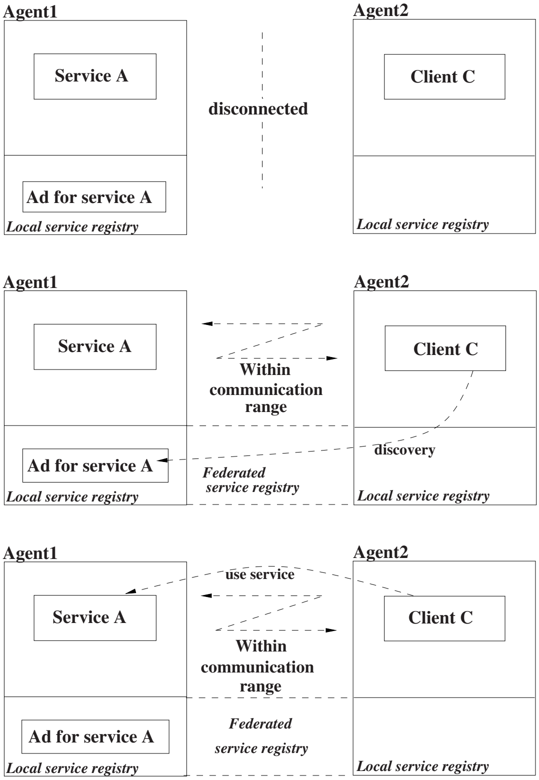


Fig. 2. Federated service registry access.

The model supports multiple servers (each server can offer multiple services), multiple clients, and multiple service registries. Clients and servers can interact via a service registry if they both have a local service registry with the same name. The model supports code and host mobility. This means that hosts can move in physical space and clients and servers can represent mobile code able to migrate anywhere in the network. Upon migration of a client or server the local service registries will migrate with them and will be re-shared from the new location. The availability of services is consistently taken care of by the nature of the service registries. Any connection and disconnection is atomically reflected in the visible content of the federated service registry. While the migration is in progress, the migrating agent is temporarily unavailable to the community. Once at destination, the agent restarts execution. Migration may not always be possible (e.g., if the service needs a piece of hardware on the host where it is currently running), may be restricted to a subset of the connected hosts (those that have the hardware needed by a service), or may not be restricted at all. A client's migration may be restricted by the type of services it is using at the moment of migration. Those services may support resuming service utilization at the new location or may require the service to be restarted. The programmer of a service has to take all these details into consideration. The designer of the proxy-server communication protocol may have to take into account possible disconnections (due to physical movement, logical migration, or communication failures).

The system provides a default public service registry, that can be used to advertise and lookup general services. Any server can create a new service registry. Any service registry, except for the default, can be password protected. The password is used to encrypt the clear name so that it won't be accessible except to those who can generate the same encrypted name from the correct clear name and the correct password. If a password is associated with a service registry, that password will secure all the transfers between remote hosts, associated with that service registry (i.e., any message carrying a query or the result of a query will be encrypted with the password that protects the registry involving that query). The use of different names and passwords enables creation of multiple (protected and/or unprotected) administrative domains where servers can publish services and clients can look them up, grouped by common interest or access rights. The same applies to clients, i.e., the clients also create a service registry locally for search purposes; a search operation will span all service registries with the same name, local to servers within communication range.

This default registry can be used by agents to advertise or verify the existence of a particular service registry. For example, all sorting services can be grouped in a public registry called "Sorting Services". Each agent looking for a painting service can look in the default, public registry and see if there's any note of a registry grouping painting services. Similarly a server trying to advertise a service will check for the note in the public registry. If such a pointer is found, the server can go use the same registry. If the pointer is not there, the server can assume it is the first one to offer such a service and create the note itself.

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 registry. The ad for a service can be marked read-only by its provider protecting it with a remove-password. This will prevent unauthorized (accidental or fraudulent) removal of the service ad as well as the replacement of the ad with a fake one. Additionally, each ad can be protected by a read-password, different from the password that protects the entire registry. This will enable a service provider to protect services its in a highly individualized manner.

The transient sharing of service repositories supported by the system eliminates single points of failure scenarios. By simply disconnecting from a party (e.g., a conference room printer), the stability of a system (e.g., a PDA) is not affected and interactions with other devices continue to function normally (e.g., PDAs from the same conference room having a wider communication range than the printer). Services are discovered solely on the basis of connectivity among devices (producers and consumers).

4 Implementation

The implementation of the new model is based on Lime [7], an adaptation of the Linda [8] coordination model to the ad hoc networking setting. Lime's transient tuple space sharing together with the security features newly added to the original model provide a natural starting point for our implementation. Lime's implementation does not depend of whether ad hoc routing is used or not, only the notion of host connectivity is affected. The model we described works in the same way whether ad hoc routing is used or not.

4.1 Implementation Infrastructure: Lime

The Lime middleware supports the development of applications exhibiting physical mobility of hosts and logical mobility of agents. An agent is the basic building block for the mobile application, a software component that may reside permanently on a host or may move from one host to another connected host, hence the name *agent*. Hosts serve as containers for agents and run local versions of Lime. As suggested earlier, Lime extends the coordination model of Linda in significant ways. First, the globally accessed persistent tuple space of Linda is replaced in Lime by transient sharing of identically named tuple spaces belonging to agents that are part of the same group, i.e., reside on hosts that are mutually accessible over the ad hoc network. Other Lime extensions to Linda include location specific operations, transparent tuple migration, and the ability to react to the presence of tuples in tuple spaces.

Transparent Context Maintenance. The model underlying Lime accomplishes the shift from a fixed global context to a dynamically changing one by distributing the single Linda tuple space across multiple tuple spaces, each local to an agent, and by introducing rules for transient sharing of the individual tuple spaces based on naming and connectivity; Lime allows an agent to structure its holdings across multiple tuple spaces each being shared only with other identically named tuple spaces local to other agents within the group. Group membership is controlled by connectivity among hosts. Sharing of multiple tuple spaces results in the formation of a virtual global data structure called a federated tuple space. The content of a federated tuple space is the union of the contents of the contributing tuple spaces. Access to the federated tuple space is accomplished by simply accessing the API for the local tuple space.

Basic access to the tuple space takes place using traditional Linda primitives: **out** takes a tuple t and places it into a tuple space; **in** takes as parameter a template p and blocks until a tuple matching the template is written to the tuple space at which point **in** removes the tuple and returns its contents; **rd** is similar to **in** but does not remove the tuple. Details of the matching mechanism will be explained later. Lime offers also non-blocking versions of **in** and **rd** in the form of probe variants of the same operations (e.g., **inp**, **rdp**). In general, non-blocking operations return a matching tuple (if one is available) and null otherwise. Both blocking and non-blocking extensions designed to handle entire groups of tuples matching the same template are also included in Lime.

A tuple is an ordered list of fields. Each field has a type and a value. A template is an ordered list of fields that can contain type designators (*formals*) or explicit values (*actual*). A tuple and a template match if both contain the same number of fields and each corresponding pair of fields matches. Initially, two field-level matching policies were available (1) Exact type matching: the field in the template is a formal and its type is the same as the type of the object in the corresponding tuple field and (2) exact value matching, the template field provides an actual that will match exactly the type and the value of the corresponding field in the tuple. We've added a third one: (3) polymorphic matching: the field in the template can be a formal whose type is a supertype of the corresponding object in the tuple. This yields the highest degree of flexibility, since the Java Object class (or the Java Serializable interface for objects that travel across the network) works as a wildcard. We also added the possibility for a tuple to specify what type of matching must be used for each field (e.g., a tuple may require the template to provide the exact value for the first field).

Controlling Context Awareness. A read-only tuple space called the `LimeSystemTupleSpace` provides an agent with a view of the overall system configuration. Its tuples contain information about the mobile agents present in the community, physical hosts they execute on, and tuple spaces created for coordination. Standard tuple space operations on `LimeSystemTupleSpace` allow an agent to respond to the arrival and departure of other agents and hosts. Furthermore, Lime provides fine-grained control over the context on which an agent chooses to operate by extending its operations with tuple location parameters that define projections of the federated tuple space.

Reacting to Changes in Context. Lime extends the basic Linda tuple space with the notion of *reaction.* A reaction $\mathcal{R}(s, p)$ is defined by a code fragment s that specifies the actions to be executed when a tuple matching the template p is found in the tuple space. Blocking operations are not allowed in s, as they could prevent the program from ever terminating.

In Lime, reactions come in two forms: *strong reactions* and *weak reactions.* Strong reactions execute atomically with the writing of the tuple that enables them. They must be restricted to a host or agent because the requirements of atomicity would entail a distributed transaction encompassing multiple hosts for every tuple space operation. Lime also provides the notion of *weak reaction.* The difference is that the execution of s does not happen atomically with the detection of a tuple matching p, but it is guaranteed to take place eventually (if connectivity is preserved). This eliminates the need for a distributed transaction and allows this type of reaction to execute over the federated tuple space.

4.2 Security Support Implementation

The security extensions we had to introduce to Lime in order to support secure service provision and other secure interactions were designed so as to have minimal impact over the programming interface offered to the developer. The extensions take password(s) as extra parameter(s) in calls that handle protected targets (i.e., tuples and tuple spaces). Secure inter-host communication is automatically turned on by the usage of secure tuple spaces, therefore it has no impact on the programmer interface.

For encryption we use the 3DES private key encryption algorithm (the keys are generated internally in a deterministic fashion from the provided passwords). The data being encrypted represents messages passed between hosts and not data that has to be stored safely. We assume that Java language protection mechanisms are robust enough (e.g., a *private* member of an object cannot be accessed by any other object). We do not address physical level attacks like wireless signal jamming. The architecture overview is shown in Figure 3.

Password Protected Tuple Spaces. The name of the tuple space is the key to gaining access to the information in that tuple space. To protect the information means to protect the name of the tuple space. Changes are required to ensure that extracting the name of a protected tuple space from `LimeSystemTupleSpace` will no longer provide enough information for an agent to create a tuple space with the same name and share it with other agents thus gaining unauthorized access to information.

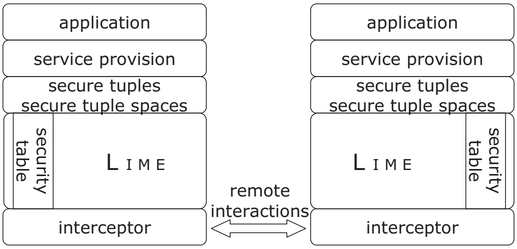


Fig. 3. Architecture overview.

The tuple space name (clear or encrypted) will be prefixed by a differentiator: letter "U" for unencrypted or "S" for secure tuple space. The tuple space called "blue" is different from the tuple space called "blue" and protected with password "pwd" (the latter will actually have the internal name $K_{pwd}(blue)$). They can coexist but no sharing takes place. The prefixes ensure that a tuple space cannot be created incorrectly. Since they are internally added, they cannot be manipulated by agents. This will prevent an attacker from successfully creating a protected tuple space after inspecting the content of `LimeSystemTupleSpace`.

The constructor call (Figure 4) is the only place where the agent explicitly uses the password. Once the agent has the handle to the tuple space, it does not need the password anymore. A tuple space operation can only be called by the Lime agent that created it. Even if the handle of a tuple space is obtained correctly by an agent, it cannot be transferred and used by another agent. Thus, a tuple space password is not needed for tuple space access operations.

The encrypted name of a protected tuple space and the password that protects it are also important for inter-host communication. This is why the redesigned Lime server has a **SecurityTable** that stores entries of the form [encrypted name, password]. This **SecurityTable** is a very important target that has to be protected. Since this paper does not address the Java security model, we assume this model is secure enough for our research.

SecureLimeTupleSpace(java.lang.String name, java.lang.String password)

— creates a new secure tuple space using the public tuple space name and the password. This call places an entry in the **SecurityTable** mapping the encrypted name to the password.

Fig. 4. The call that creates a secure tuple space.

Tuple Level Protection. To implement password-protected tuples, every tuple space operation will add to the end of the user specified fields (if any) three fields: the read-password, the remove-password and the name of the operation that uses that tuple or template (e.g., ”rd” for any type of read operation or reaction, ”in” for any type of remove operation—these two apply to templates—and ”out” for any type of write operation—this will differentiate the tuple from templates, since they both are instances of the same Tuple class). The fields containing passwords are subject to the exact value matching policy. These fields are not explicitly available to the programmer for direct manipulation but are used by methods calls used to manipulate tuples.

The tuple content is not encrypted. The access control is realized by filtering tuples based on the three security fields. The filtering happens during the matching process between the template used by an operation that tries to access a tuple in the tuple space and the tuple(s) already in the tuple space. It is important to note that the matching of the three fields added to the end of the tuples/templates is slightly altered to accommodate common sense scenarios like: the tuple has both a read-password and a remove-password and the template provided by a read operation has the remove-password and not the read-password. We consider normal for this operation to succeed because if someone has the password to remove a tuple, he/she should also be able to read the tuple (note that removing a tuple returns the tuple to the process and doesn't just erase it from the tuple space, therefore this will not produce a security breach). Another example is the case when a tuple has a read-password but no remove-password. A remove kind of tuple space operation will need to provide the tuple's read-password as the template's remove-password for the operation to succeed. The reason is that if the user needs a password only to read the tuple, he/she will need a password to remove the tuple and therefore the read-password will protect the tuple from removal. If the tuple has a different, non-null remove-password, the mechanism described above does not work anymore.

lts.out(ITuple tuple, char[] readPwd, char[] removePwd)
— writes a tuple to the tuple space and protects it against reading and/or removing. Any combination of the two passwords is permitted.

lts.rd(ITuple template, char[] readPwd)
— reads a tuple from the tuple space if the tuple and the template match (and the correct password is provided).

lts.in(ITuple template, char[] removePwd)
— removes a tuple from the tuple space if the tuple and the template match (and the correct password is provided).

Fig. 5. The tuple space interaction operations.

Communication Level Protection. When an agent executes an operation that extends beyond the current host, an interceptor catches it, analyzes the tuple space that the message refers to (the name of the tuple space is always present in the message that travels across hosts) and takes the appropriate action. The use of the interceptor pattern [9] is natural for this case, when we add security to a system that in its initial design did not address this issue. It also offers great flexibility with respect to the choice of encryption protocol.

The interceptor checks whether the tuple space name in the outgoing message is in the `SecurityTable`. If the message refers to an unprotected tuple space, the interceptor lets it pass through unchanged. If the tuple space is a secure one, the interceptor will extract from the table the password that corresponds to that tuple space and will use it to encrypt the message. The interceptor creates a packet that contains the encrypted message and the encrypted name of the tuple space the message refers to and forwards this packet to the destinations. On the recipient's side, actions happen symmetrically. The returned results are handled similarly.

Note that using protected tuples in unprotected tuple spaces leads to these tuples/templates being shipped between hosts unencrypted and thus vulnerable to eavesdropping. Once an attacker captures a packet and extracts the passwords clearly written in the tuple, he will be able to remove these tuples. Unprotected tuple spaces do not provide any security guarantees for protected tuples when communication across hosts is involved.

4.3 Service Model Implementation

The implementation consists of a new LIME agent class, `ServiceAwareAgent`. This is because LIME restricts the tuple space access to a LIME agent, the one that created it. The programmer will need to extend the `ServiceAwareAgent` class and implement its `run()` method. The new agent will inherit from `ServiceAwareAgent` the functionality it needs to manipulate service ads (Figure 6).

A service repository is created around a LIME tuple space, with or without a password (which makes it public or protected). Each client/server can create multiple service registries and receives handles to each of them. The handles will be used when registering, deregistering of searching for services.

A service is advertised as a tuple in the specified service registry (or in the default public service registry) together with a profile (the **Profile** class is a container for pairs [interface, attributes], where the attributes describe the associated interface which the service implements). When the service is advertised, it will be assigned by the system a unique service identifier (`sid`), if not specified by the server (i.e., it is advertised for the first time). This `sid` will be used for removing the advertisement when no longer needed.

When searching for a service, a client can specify the `sid` if it had previously used the service. This is the easiest way to rediscover a service. If the client does not have the `sid` for the needed service or if the service identified by the sid is no longer available, the search has to be performed using the service description. The client will provide a profile pattern for the service it needs. The tuple matching mechanism will return tuples that represent services whose profiles match the request made by the client (i.e., implement all the interfaces the client requested and the values of the advertised attributes match the values requested by the client). The result is a wrapper (the **Service** class) which contains the proxy

public ServiceAwareAgent()
— creates a new ServiceAwareAgent.
public abstract void run()
— abstract method that defines the agent's behavior and has to be implemented by the programmer in the class that extends ServiceAwareAgent.
public String[] listPublicRegistries()
— returns the list of the names of all public registries currently available.
protected ServiceRegistry createServiceRegistry(String name, char[] pwd)
— creates a new service registry protected with password pwd, if specified. If pwd is null the service registry will be unprotected.
public ServiceID registerService(ServiceRegistry sr, ServiceID sid, Serializable proxy, Profile prof, char[] upwd, char[] rpwd)
— registers a new service in the specified service registry.
public void deregisterService(ServiceRegistry sr, ServiceID sid, char[] pwd)
— deregisters the service identified by 'sid', if all conditions are met.
public Service findService(ServiceRegistry sr, ServiceID sid, Profile prof, char[] pwd)
— searches for a service that corresponds to the description.

Fig. 6. ServiceAwareAgent's public interface.

object and the sid of the service. The security requirements must be met in both cases (i.e., when using a sid and when using the entire profile).

Service deregistration translates to removing the tuple carrying the service advertisement from the corresponding service registry, i.e., the corresponding tuple space. The agent trying to deregister the service must have the service `sid` and password (if applicable). The passwords that protect a service translate to read-password and remove-password in the Lime secure tuple space access.

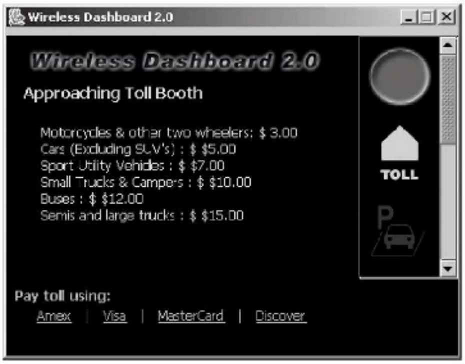
5 Sample Application

The framework presented in this paper was first evaluated in a test application (Figure 7) that allows a car driving down a highway to make an electronic payment to an approaching tollbooth. As the car approaches and discovers the tollbooth, it receives the proxy object for the payment service, the list of prices, pays by credit card, and continues its trip without stopping. The proxy completes the payment when arriving at the exit tollbooth. Meanwhile the proxy obtained from the car the distance it travelled (also computable from the two tollbooths positions), the type of car (sedans, SUVs and trucks have different prices), and the driver's credit card number from his/her PDA.

The car has an agent specialized in automatic payments (toll roads, parking, etc.). All these charge points are configured to establish contact with vehicle agents in a predefined, unprotected tuple space, called ”payments”. The agent in the car is continuously looking for payment services. Near the tollbooth, the car discovers the payment service, retrieves the proxy object and ”launches” the service, displaying the proxy's GUI on a screen inside the car.

The tollbooth advertises the service in a read-only tuple. However an attacker can be nearby and advertise using a read-only tuple, in the same "payments" tuple space, another proxy object that claims to handle the toll payment. The car agent has to decide now which of the two services to use. One possibility is for the car to specifically read the service from tollbooth's local tuple space. This implies that the car agent knows how to use location parameters to limit the service query to the tollbooth part of the tuple space. While this is doable for the daily drive from home to office and back (assuming the same route), on a long journey this can get complicated. The configuration time can be much longer than a regular cash payment. In this second case, once the proxy object is "launched" on the car's computer, it can connect to the tollbooth and light up a visual signal to indicate it can communicate with the tollbooth. A proxy retrieved from an attacker will not be able to interact with the tollbooth. Short range wireless communication is another trick that can help at the physical level.

Since the car can read the tuple **from** the tollbooth, the vulnerability is to read a tuple put there by an attacker. The car agent will have to verify that the tuple is read-only (i.e., by failing in an attempt to remove it). The reader is reminded that the read-only tuple could not have been placed there by anybody else since such tuples cannot migrate.



Wireless Dashboard 2.0

Approaching Toll Booth

Motorcycles & other two wheelers: $ 3.00
Cars (Excluding SUV's) : $ $5.00
Sport Utility Vehicles : $ $7.00
Small Trucks & Campers : $ $10.00
Buses : $ $12.00
Semis and large trucks : $ $15.00

Pay toll using:
Amex | Visa | MasterCard | Discover

Fig. 7. Automatic toll payment is one of the features offered by a simulated wireless dashboard application.

While on the highway on the way to the exit point, the service GUI displays the price for the current type of car and the list of credit cards the proxy retrieved from the driver's PDA (or from the software running on car's computer). The driver can choose which one to use and this completes the interaction with the toll proxy. Note that for the daily route from home to the office (for example), most of the actions can be automated. The driver could be prompted only if there is a change in price. The driver can also designate a default credit card for this type of payments, etc.

When leaving the highway, the proxy displays the distance driven and the price, and sends the payment information to the tollbooth, issues an electronic receipt and then terminates execution. The proxy-tollbooth interaction is carried out via a protected tuple space and then using a private, raw socket protocol.

The application shows how transient tuple space sharing between the car and the tollbooth makes the payment service available to the car and how security mechanisms employed by the infrastructure help establish a safe interaction between the two parties. The effort invested to deliver this application sums to less than 300 lines of Java code, not including the GUI.

6 Related Work

The discovery technique constitutes the main difference among various existing implementations of the service discovery model. Sun Microsystems developed Jini [1,2]. It uses as a service registry lookup tables managed by special services called lookup services. These tables may contain executable code in addition to information describing the service. A Jini community cannot work without at least one lookup service. IETF offers the Service Location Protocol [10] where directory agents implement the service registry. They store service profiles and the location of the service but no code. The discovery of services involves first locating these directory agents. If no directory agent is available, clients may multicast requests for services and servers may multicast advertisements of their services. The most common service types use, by default, the service templates standardized by Internet Assigned Numbering Authority (IANA). Microsoft proposed Universal Plug'n'Play [3], which uses the Simple Service Discovery Protocol [11]. This protocol relies upon centralized directory services for registration and lookup. If no such proxy is available, SSDP uses multicast to announce new services or to ask for services. The advertisement contains a Universal Resource Identifier (URI) that leads to an XML description of the service. This description is accessible only after the service has been already discovered through a lookup service. The novelty of this model is the auto configuration capability based on DHCP or AutoIP. The Salutation project [5] uses a relatively centralized service registry called Salutation Manager (SLM). There may be several such managers available, but the clients and servers can establish contact only via these SLMs. The advantage of this approach is the fact that these SLMs can have different transport protocols underneath, unlike the above-mentioned models that assume an IP transport layer. To realize this, Salutation uses transport-dependent modules, called Transport Managers that broadcast internally, helping SLMs from different transport media interact with each other. In [12], the authors address the nomadic computing case from the point of view of the flexibility of the service advertisement and lookup matching.

Security issues are addressed in [13] which discusses threats and protection mechanisms for distributed systems. A capability-based security system is presented in [14]. 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[15]). In[16] 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. [17] addresses proxy-based security protocols for mobile devices, but also relies on a relatively centralized architecture for accomplishing some key tasks. Security is accomplished by adapting SPKI/SDSI [18,19] for proxy-server and/or proxy-proxy interactions. In [20] the authors use public keys as authentication certificates for Jini services, manually managed in local databases as initial trust relationships. In 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 [21]. Decentralized trust management issues are also addressed in

KeyNote [22]. The authors describe an infrastructure that binds keys to the authorization to perform specific tasks rather than to names. KeyNote was inspired by PolicyMaker [23] and supports local control of trust relationships thus eliminating the need for certifying authorities.

7 Conclusions

In this paper we presented a technique that allows for consistent service advertisement and discovery in ad hoc networks. Key is the fact that service registry updates are atomic with respect to changes in service availability due to mobility and disconnections. We also described 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 the development of algorithms for establishing session keys or simply the distribution of secret keys.

Acknowledgements. 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. The authors would also like to thank Rohan Sen for his contribution in the implementation and testing.

References

1. Sun Microsystems: Jini technology core platform specification (2000)
2. Edwards, K.: Core JINI. Prentice Hall (1999)
3. Microsoft Corporation: Universal plug and play forum. (http://www.upnp.org)
4. W3C: Simple object access protocol (soap). (http://www.w3.org/TR/SOAP)
5. Salutation Consortium: Salutation specifications. (http://www.salutation.org)
6. SIG, B.: Bluetooth specification. (https://www.bluetooth.org/foundry/ specification/document/specification)
7. Murphy, A., Picco, G., Roman, G.C.: Lime: A middleware for physical and logical mobility. In: Proceedings of the 21^{st} International Conference on Distributed Computing Systems. (2001) 524–533
8. Gelernter, D.: Generative communication in Linda. ACM Transactions on Programming Languages and Systems **7** (1985) 80–112
9. Schmidt, D., Stal, M., Rohnert, H., Buschmann, F.: Pattern Oriented Software Architecture. Volume 2. John Wiley & Sons, Ltd. (1999)
10. Guttman, E.: Service location protocol: Automatic discovery of IP network services. IEEE Internet Computing **4** (1999) 71–80
11. Goland, Y., Cai, T., Leach, P., Gu, Y., Microsoft Corporation, Albright, S., Hewlett-Packard Company: Simple service discovery protocol/1.0: Operating without an arbiter. http://www.upnp.org/download/draft_cai_ssdp_v1_03.txt (2001)
12. Jacob, B.: Service discovery: Access to local resources in a nomadic environment. In: OOPSLA '96 Workshop on Object Replication and Mobile Computing. (1996)

13. Hubaux, J.P., Buttyan, L., Capkun, S.: The quest for security in mobile ad hoc networks. (In: ACM MobiHOC 2001 Symposium)
14. Gong, L.: A secure identity-based capability system. In: Proceedings of the IEEE Symposium on Security and Privacy. (1989) 56–63
15. Lampson, B.: Protection. In: 5th Princeton Conf. on Information Sciences and Systems. Volume ACM Operating Systems Rev. 8. (1971) 18–24
16. Czerwinski, S.E., Zhao, B.Y., Hodes, T.D., Joseph, A.D., Katz, R.H.: An architecture for a secure service discovery service. In: Mobile Computing and Networking. (1999) 24–35
17. Burnside, M., Clarke, D., Mills, T., Devadas, S., Rivest, R.: Proxy-based security protocols in networked mobile devices. In: Proceedings of Selected Areas in Cryptography. (2002)
18. Ellison, C., Frantz, B., Lampson, B., Rivest, R., Thomas, B., Ylonen, T.: Simple public key certificates. Internet Draft http://world.std.com/cme/spki.txt (1999)
19. Rivest, R.L., Lampson, B.: Sdsi – a simple distributed security infrastructure. (Presented at CRYPTO'96 Rumpsession (http://citeseer.nj.nec.com/ rivest96sdsi.html))
20. Eronen, P., Lehtinen, J., Zitting, J., Nikander, P.: Extending jini with decentralized trust management. In: The Third IEEE Conference on Open Architectures and Network Programming (OPENARCH). (2000)
21. Vettorello, M., Bettstetter, C., Schwingenschlgl, C.: 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)
22. Blaze, M., Feigenbaum, J., Keromytis, A.D.: Keynote: Trust management for public-key infrastructures. In LNCS, S., ed.: Security Protocols International Workshop. Volume 1550. (1998) 59–63
23. Blaze, M., Feigenbaum, J., Lacy, J.: Decentralized trust management. In Press, I.C.S., ed.: 17th Symposium on Security and Privacys. (1996) 164–173