Washington University in St. Louis

Washington University Open Scholarship

All Computer Science and Engineering Research

Computer Science and Engineering

Services Provision in Ad Hoc Networks

Radu Handorean and Gruia-Catalin Roman

The client-server model continues to dominate distributed computing with increasingly more flexible variants being deployed. Many are centered on the notion of discovering services at run time and on allowing any system component to act as a service provider. The result is a growing reliance on the service registration and discovery mechanisms. This paper addresses the issue of facilitating such service provision capabilities in the presence of (logical and physical) mobility exhibited by applications executing over ad hoc networks. The solution being discussed entailes a new kind of service model, which we were able to build as an adaption... Read complete abstract on page 2.

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

Services Provision in Ad Hoc Networks

Radu Handorean and Gruia-Catalin Roman

Complete Abstract:

The client-server model continues to dominate distributed computing with increasingly more flexible variants being deployed. Many are centered on the notion of discovering services at run time and on allowing any system component to act as a service provider. The result is a growing reliance on the service registration and discovery mechanisms. This paper addresses the issue of facilitating such service provision capabilities in the presence of (logical and physical) mobility exhibited by applications executing over ad hoc networks. The solution being discussed entailes a new kind of service model, which we were able to build as an adaption layer on top of an existing coordination middleware, LIME (Linda in a Mobile Environment).

Service Provision in Ad Hoc Networks

Radu Handorean and Gruia-Catalin Roman

WUCS-01-40

November 2001

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis MO 63130

Service Provision in Ad Hoc Networks

Radu Handorean and Gruia-Catalin Roman

Department of Computer Science
Washington University
Saint Louis, MO, 63130
{raduh, roman}@cs.wustl.edu

Abstract. The client-server model continues to dominate distributed computing with increasingly more flexible variants being deployed. Many are centered on the notion of discovering services at run time and on allowing any system component to act as a service provider. The result is a growing reliance on the service registration and discovery mechanisms. This paper addresses the issue of facilitating such service provision capabilities in the presence of (logical and physical) mobility exhibited by applications executing over ad hoc networks. The solution being discussed entails a new kind of service model, which we were able to build as an adaptation layer on top of an existing coordination middleware, LIME (Linda in a Mobile Environment).

1. Introduction

As the network infrastructure continues to grow, more and more devices are being attached directly to the network. The opportunity for applications to exploit an ever-increasing range of resources is expanding rapidly. All entities, which must be connected to the network, can provide services advertised over the network, making the network a service repository. The number of such services is expected to grow significantly in the coming years. Given the variety of devices accessing the network and the ever-growing number of services that become available, a high-level approach was needed to allow one to discover services dynamically, as the need arises.

In the client-server model, which continues to dominate distributed computing, the client knows the name 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. More recent strategies allow one to advertise services, to lookup services and to access them without explicit knowledge of the network structure and communication details. Services offered by servers may be discovered at runtime. They are being used through proxies they provide. A proxy abstracts the network from the client programmer by offering a higher-level interface, specialized only in service exploitation while the proxy's interface to the server remains unknown to the client. Services are advertised by publishing a profile containing attributes and capabilities useful when searching for a service and proper service invocation. Clients search for services using templates generated according to their momentary needs. These templates must be matched by the advertised profiles. A service profile can

1

include the service's location and a client can use this information in evaluating the suitability of that service. However, in contrast to the classic client-server model, publishing the location of a service it not a requirement for the service model to work, since it does not play a key role in the process of discovery. Services use a service registry to advertise their availability and clients use this registry to search the services they need. This approach enables a much greater degree of run time flexibility.

Our model completely eliminates network awareness from the process of discovery and utilization of a service. The client only has to ask for the service it needs and does not have to know how the service will be reached. Our model differs from others by providing a distributed service registry that is guaranteed to reflect the real availability of services at every moment in a mobile ad hoc environment and a communication technique that is not affected by physical or logical mobility. We achieve a consistent representation of the available services by atomically updating the view of the service repository as new connections are established or existing ones break down. A data repository similar to the service registry can be used for communication. This data repository offers both synchronous and asynchronous communication and hides physical host movement or logical agent migration.

The reminder of the paper is organized as follows: Section 2 presents the motivation for this research. Section 3 introduces a variation on the basic service model, adapted for use in ad hoc mobility. Section 4 describes the implementation of the service model in terms of an existing coordination middleware. Section 5 discusses lessons learned and future work.

2. Motivation

The *service model* is composed of three components: services, clients and a discovery technology. Services provide useful functionality to clients. Clients use services. The discovery technique enables services to publish their capabilities and clients to find and use a needed service. 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.

The discovery technique constitutes the main difference among various existing implementations of the model. Sun Microsystems developed Jini [1, 2, 3, 4, 5] that 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 *even if services and potential users are co-located* (co-located refers to services and potential clients residing on the same physical host). IETF offers the Service Location Protocol [6, 7, 8, 9] where directory agents implement the service registry. They store service profiles and the location of the service but no executable 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

2

Authority (IANA). Microsoft proposed Universal Plug'n'Play [10], which uses the Simple Service Discovery Protocol [11]. This protocol also uses centralized directory services, called proxies, for registration and lookup of services. 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 eventually 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 [12] also 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 all 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.

All these models assume (more or less) a fairly stable network. Nomadic computing takes one step in breaking the wired network rigidity by allowing mobile users to connect to the network via wireless means. It is important to note that the universe in nomadic computing also includes a fixed component. The infrastructure composed of base stations ensures communication in a communication cell. The implementations of the service model may have some limitations in terms of functionality because of the particularities of the new settings. The idea of using the base stations as hosts for service registries is very appealing. However, overloading the base station of a communication cell may lead to defensive behavior of its software, e.g., terminating advertisement broadcasts or completely ignoring client communication. The failure of a base station leads to a complete lack of communication between the agents in its cell and between clients and services whose communication is routed via this base station, even if they could communicate directly. The similarity to classic networks with centralized service registries is strong. In nomadic networking, reusability of communication frequencies is very important, but introduces a new limitation. The larger the cell, the more devices inside a cell, and the more frequencies needed. Since the range of available frequencies is usually limited, this leads to a limitation of the devices in a cell, and implicitly to a limitation of the service availability. Broadcast discovery is also limited to the current cell, thereby limiting the scope of service advertisement. We can can still leverage off much of the infrastructure used in classic networking, since many of the new problems are solved at a lower level in the stack of communication protocols. In conclusion, even if the nomadic networks bring a certain degree of mobility, as long as there is an infrastructure that *must* be used by the agents to communicate, the differences from classic networks are little.

A higher degree of freedom and a fully decentralized architecture can be obtained in mobile ad hoc networks, at the expense of new challenges. Mobile ad hoc networks are opportunistically formed structures that change in response to the movement of physically mobile hosts running potentially mobile code. The 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. Application level communication has to keep up with the physical layer communication progress. Frequent disconnections inherent in ad hoc networks lead to inconsistency of data in centralized service directories. Architectures based on centralized lookup directories are no longer suitable. The broadcast implementations need a high frequency of messages in order to preserve this consistency. This leads to an increased consumption of bandwidth. The service model used needs to adapt to the new conditions.

As an example, a routing protocol that tries to adapt quickly to connections that come up and break down may lead to an overhead that will eventually make it inefficient in such an environment. In another example, if the node hosting the service registry suddenly becomes unavailable, the entire advertising and lookup of services becomes paralyzed even if the pair of nodes representing a service and a potential client remains connected. This scenario is depicted in Figure 1(left).
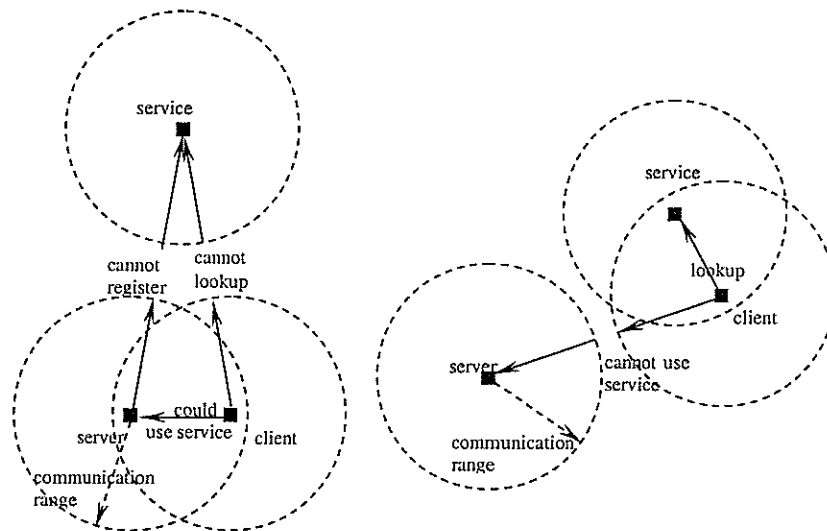

Fig. 1. Left: The client could use the service but it cannot discover it since the service registry is not accessible. Right: A client discovers a service that is no longer reachable

Our aim is to make these two nodes communicate. Furthermore, because of frequent disconnections, the service registry should immediately reflect changes affecting service availability. Services that are no longer reachable should not be available for discovery. In Figure 1 (right) we present a scenario that can happen in Jini, where the advertisement of a service is still available in the lookup table until its lease expires. In a model addressing these issues, all nodes should be simple users or providers of services. However, the system should not depend on the behavior of a

single node. Broadcast discovery reduces the timeline to discrete moments when clients can update their knowledge about the available services introducing intervals of inconsistency of duration up to the frequency of broadcasts. In ad hoc networking, frequent disconnections may prevent a client from ever discovering the service it needs, even if the service is present and it periodically announces its availability via broadcast messages. 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 interface). 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.

3. Service Model Revisited for Mobility

Each active entity, with the ability to perform some computation will be called an agent. Agents represent application components or devices. An agent can be a client, a server, or both. An agent that can migrate from one host to another will be called a mobile agent. Two agents that reside on the same host are co-located. Hosts are assumed to be mobile, can move freely through space, and form ad hoc networks when within communication range.

Services are advertised by publishing a profile that contains the capabilities of the service and attributes describing these capabilities, so the clients can discover and use the services properly. The interface used to access services refers only to capabilities advertised in the profile. Location information is not needed to discover the service requested by the client. The client can use the attributes to decide if the service meets its requirements in terms of quality of service parameters. Along with the profile, the server provides a service proxy. This proxy will represent the service locally to the client.

Clients search for services using a template that defines what the needed service profile must match. If a service profile satisfying all client's requirements is available, the service proxy, as part of the profile, is returned to the client. The client will use the proxy to interact with the service as if it were local.

The interface provides primitives for service advertisement and lookup and the proxy offers the service interface. Every agent has its own service registry where it advertises the services it provides. The registries of co-located agents are automatically shared. Thus, an agent requesting a service that is provided by a co-located agent can always access the service. If two hosts are within communication range they form a community and their service registries engage, forming a federated service registry. Upon engagement, the primitives operating on the local service registry are extended automatically to the entire set of service registries present in the ad hoc network. An agent in the community will access this federated registry via the same API, i.e., via its own local registry. The sharing of the service registries is thus completely transparent to agents.

Maintaining the consistency of data in the service registry is a very important issue in such a rapidly changing environment. In our model, an ad for a specific service can

5

be discovered if and only if the service is available. The update of the content of the federated service registry occurs atomically, at the moment of engagement or disengagement of each host. Discovery and accessibility of remote servers is scoped by host connectivity. Thus, when the host of the service becomes unreachable (i.e., gets disconnected), the local repository *atomically* becomes unavailable as well and the service cannot be discovered anymore. This helps solve several important problems. First, it eliminates the need for a centralized directory for registration and lookup. Second, it guarantees that two hosts within communication range are able to exchange services. Third, it prevents a client from discovering a service that is not available anymore at the time of the lookup. Figure 2 presents the use of the distributed service registry.
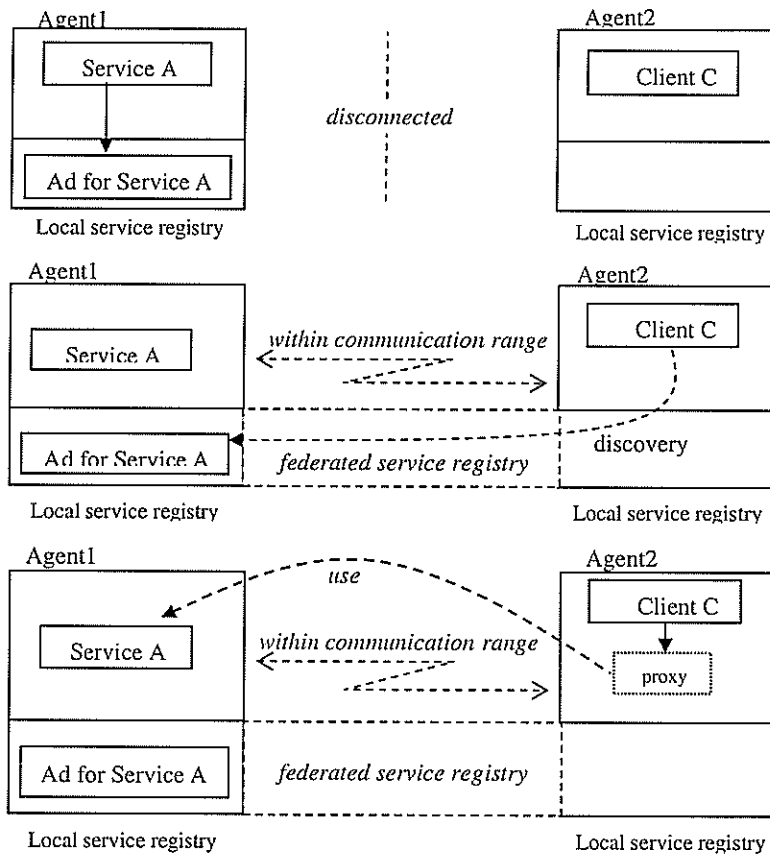

Fig. 2. Local tuple space sharing and service proxy utilization

4.Coordination-Based Design Solution

We implemented the service model as a veneer on top of LIME [13, 14], a middleware for mobility with strong support for coordination in ad hoc networks. The veneer, a thin adaptation layer, uses LIME tuple spaces to store service advertisements and pattern matching to find services of interest. More significantly, our veneer exploits the transient tuple space sharing feature of LIME to provide consistent views of the available services in the entire ad hoc network in a uniform manner i.e., as if all service advertisements were part of the local registry. This allows us to achieve the deployment of the new service infrastructure in an ad hoc setting with minimal programming effort. This section provides an overview of the implementation and discusses some of the technical difficulties we encountered during this on-going development effort.

4.1 LIME Overview

LIME is a middleware supporting applications that involve physical and logical mobility. LIME extends the Linda [15] coordination model by adding support for coordination in ad hoc networks. The global and persistent tuple space of Linda is replaced in LIME by tuple spaces carried by individual agents residing on hosts that can be mobile. These local tuple spaces are automatically shared among co-located agents and transparently shared among all hosts within communication range. The resulting federated tuple space acts as a distributed repository of elementary data units called tuples. Tuples are sequences of typed values and can contain data or code. These tuples can be added to a tuple space using an `out(tuple)` operation and can be removed by executing an `in(template)` operation. Tuples can also be read from the tuple space using the `rd(template)` operation. Tuples are selected using pattern matching between a template and the tuples available in the tuple space. The template can be composed of actuals and formals. Actuals are values, while formals are types that are used as wild cards. Both `in` and `rd` are blocking operations. LIME offers an extension to this synchronous communication by providing probe variants for the traditional blocking operations, e.g., `rdp(template)` and `inp(template)`. If no matching tuple is found, a NULL value is returned. For any of these operations, if several tuples match the pattern, one is selected non-deterministically. LIME also implements several other extensions of the basic Linda primitives designed to handle groups of tuples e.g., `outg`, `rdg` and `ing`.

Another extension to the basic Linda model are the location parameters LIME primitives can use. Hosts and agents store information about their location, information that can be used to define a projection of the transiently shared tuple space. For example, an `out` operation could specify a destination location for the tuple it writes. This tuple will be written to the local tuple space of the agent issuing

the out operation and then it will migrate to the specified destination, if available. The in and rd operations can also use location information to restrict the scope of their actions to a projection of the complete federated tuple space.

Finally, LIME offers mechanisms to react to changes in the contents of tuple spaces i.e. support for reactive programming. A reaction is specified by providing a template and a piece of code that must execute in response to the appearance of a matching tuple in a particular tuple space. Two types of reactions are available in LIME. Strong reactions are executed atomically with the context change that enables them. For practical reasons, these reactions can only be enabled by tuples residing in the local tuple space. Achieving the atomicity of execution in a distributed environment involves transactions across multiple hosts, which are expensive to implement in highly dynamic environments. Another reason for this restriction is the possibility of creating chains of reactions spanning the network, i.e., global transactions whose scope may expand out of control. LIME also offers weak reactions to detect changes in the federated tuple space. If a matching tuple is found, the execution of the code associated with a weak reaction is guaranteed to take place eventually if the connectivity is preserved, but not atomically with the detection of the tuple. For technical reasons, blocking operations are not allowed in the code associated with a reaction.

4.2 Service Representation

Each service is represented by a profile stored in a tuple. This tuple contains fields for a service id, a list of attributes, a list of capabilities, a proxy object, and some information about the communication between the proxy object and the server. When the service registers, the system assigns it a globally unique service id. This id will represent the service as long as it is available, and can be used for rediscovery of the same service. The attributes quantify the capabilities of the service (e.g., "color" and "laser" can be attributes for a service advertising the "print" capability). The client may use attributes when searching for services to filter the results. The proxy object is the front end of any service. The client uses the proxy object to access the service as if the service were implemented locally. The proxy object may have one of two behaviors: it may fully implement the service with no remote communication, or it may provide an interface to a remote service provider while hiding the details of the communication protocol from the client. The latter situation is encountered when the service needs a specific piece of hardware to execute the job (e.g., a printer), or other resource that cannot migrate to the client. The protocol used by the server and the proxy for their private communication is arbitrary. It can be a well-known protocol (e.g., Java RMI) or a proprietary protocol that is well suited for the application needs (e.g., LIME).

While the proxy hides the network from the client, the proxy must know where the server is located. In the presence of mobility, the location information may change upon migration of the service. For example, if the agent providing the service moves to another host, the IP address changes but the port number may not. Likewise, if the proxy and the server use RMI to communicate, it is very likely for the server to use the same registration string at the new location, even though its IP changed. This led

8

us to a design that splits the location information in two parts. One part represents the physical location of the agent running the server and the other part represents a logical address within the addressing space available at the physical location (e.g., the range of useable ports or the RMI registration strings). While mobility causes physical location to change, this logical address is not likely to change. Since the physical location is unique for all servers run by each mobile agent but the logical address is specific to each server and does not change, we chose to publish them separately. The physical location is published along with the agent's id in a special tuple space, called the location tuple space (ServiceLocationTupleSpace). This tuple space contains one location tuple for each agent, and the content is updated upon agent migration. This way, an agent needs to update only one tuple when it migrates, regardless of the number of services it provides. The logical address is part of the tuple that contains the advertisement of the service. Upon migration, these tuples will follow the agent automatically and unchanged.
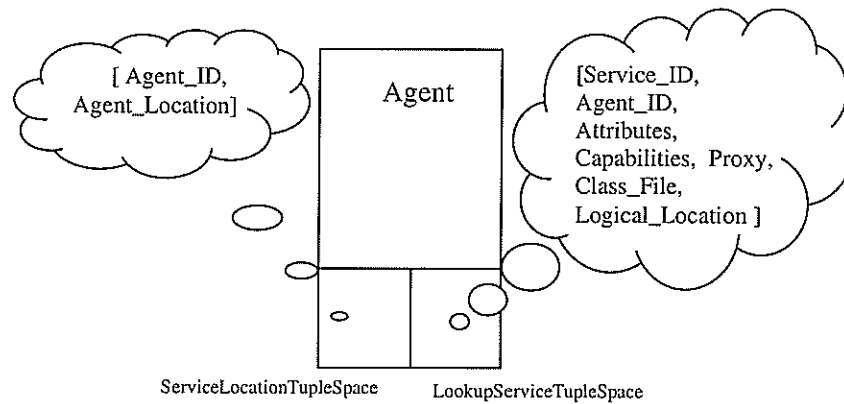

Fig. 3. The agent and the two local tuple spaces storing the service profiles and the agent's physical location

4.3 Service Access

The tuples describing the services an agent wants to publish are written (using the out operation) into a tuple space local to the agent, called lookup tuple space. By using the same name (LookupTupleSpace) for all local lookup tuple spaces, we are able to take advantage of LIME's transparent sharing of tuple spaces with the same name. Thus, each agent's lookup tuple space is automatically shared with any co-located agents. Upon engagement with a new host or group of hosts, this tuple space is shared with all the agents in the community, forming a federated lookup tuple

space. Since engagement and disengagement are atomic operations, each agent sees a consistent and up to date set of services available across the ad hoc network.

A client searches for services by querying the federated lookup tuple space using a template that describes the desired service. In this template, the client can request a service with a specific id, services that have certain attributes, services that implement certain interfaces, or a combination of the above. A tuple is considered to match the client's requirements if the service it advertises has *all* the properties the client demands. For example, if a client wants a color (attribute) printer (interface), a service that only specifies printing as a capability without giving details about the quality of printing will not be returned as a possible match for the query.

Once the client has obtained a copy of the matching tuple, it has access to the service proxy, and can call methods using the interface that the proxy object implements. There is no standardization for service interfaces. If the client does not know anything about the proxy interface, it could use the Java reflection mechanism to discover what a proxy can do, but the semantics of the method and argument names are difficult to correctly interpret automatically. This led us to favor an approach that assumes a common Java interface known to both the programmer of the client and the programmer of the service. This way, the client can correctly prepare the arguments and call the methods on the proxy object.

4.4 Service Continuity upon Migration – Mobility Implications

Mobile agents run the clients and the services. At some point, an agent running a client or an agent providing a service may decide to migrate to a new host. With tuple space based communication no special measures are required to resume collaboration between the client and the server when migration occurs and if client and server remain within communication range. The tuple spaces (associated with the application and used for client/server coordination) are automatically transferred to the new location, and the respective tuple spaces continue to be uniformly accessed since the location does not influence the process of tuple retrieval.

If the agent running the client decides to move, a private socket protocol between the proxy and its server must reopen the communication channel with the server using the same location information. If RMI is being used for communication, the client will reuse the location information to contact the remote RMI registry and obtain a new proxy object. This is necessary because the RMI implementation embeds the location of both communication ends in the proxy object generated (i.e., an RMI stub object cannot be transferred and reused on a different host from the one where it was - deployed by the RMI infrastructure – the stub is tied to the host where is was first deployed).

If the agent running server code migrates, its physical location tuple must be updated. The clients will need to reconnect to the server using the new location information. In the case of RMI communication, the server also needs to re-register with the new RMI registry at the new location. The client will need to download a new proxy object (RMI stub file) from the RMI registry using the new physical location information.

Agent migration in LIME is supported via µCode. The implementation preserves the memory state, but not the execution state. This means that at its destination, the agent restarts execution with the memory initialized to the content present when the migration was triggered. This initialization includes the re-registration of the services. Having the memory content preserved helps implementing a *resume* behavior. That is, it can only perform those actions from the registration that are absolutely needed (e.g., it can only update the location tuple). This also allows the client and the server to resume the communication from a certain point without restarting the entire task.

4.5 Discussion

LIME offers support for implementing the service model in the mobile ad hoc networking environment. The transient sharing of the tuple spaces used in LIME 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 LIME were required. The initial public release of LIME has typical Linda-style pattern-matching capabilities. The actuals in the template fields must 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. Given the particularities of the Java programming language, in order to use an object, the JVM must have access to the class file of which the object is an instance. To use a service that is discovered for the first time, the client needs to obtain the class file of the proxy object, not only an instance of it. In Jini or in simple RMI based applications, the class file is obtained via HTTP download from the host that offers the service. We provide this class file via the registration tuple. This code on demand approach helps keep a small footprint for services on the client side. The code can de downloaded if needed and discarded when no longer useful. The approach also enables dynamic updates to the latest version of the service advertised by a server.

The pattern matching policy used has a big impact on the representation of a service in the tuple space. Because of the Linda-style pattern matching, a tuple can match only templates of the same arity. A service can advertise itself by publishing multiple capabilities and attributes. A client may be interested in only part of service's portfolio. This eventually leads to a situation where the template generated by the client needs to match only part of the service's advertisement. Because of the strict pattern matching implementation currently available, the representation of a service in the tuple space is a group of tuples that covers the cartesian product of the set of attributes and the set of capabilities advertised by the service. We can reduce the number of tuples and, implicitly, the memory usage and the time for service lookup

11

by changing again the pattern-matching algorithm. We plan to adapt the pattern matching policy so that each field contains an element that is a set and a field in the template will match a field in a tuple if the template field represents a subset of the corresponding field in the tuple. The set inclusion should be understood as using the polymorphism mentioned above. This change will allow us to the use only one tuple for each advertised service, where the fields representing the attributes and capabilities are sets both in the tuple and in the template. The basic element in pattern matching algorithms is the evaluation of a predicate over the elements being checked. In our case these elements are corresponding fields from the tuple and the template. In the first step we relaxed the strict Linda-like pattern matching by enabling the object oriented polymorphism in predicate evaluation. Next step is to relax even more the algorithm by providing a way to overcome the limitation introduced by the arity of tuples or templates. A template with n fields could match at most the tuples of arity n. While this condition is still preserved, introducing set inclusion predicates, will allow us to relax further the comparison of individual fields. Other policies may allow templates to specify a range of values for the data in the tuples or any other type of predicate to be used. Our current plan is to include application-supplied field matching conditions

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 for 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 look up primitive that handles groups of tuples. Other situations may permit a client to use a service whenever the latter becomes available. All these kinds for implementation of the lookup primitive are eas ily constructed on top of LIME.

Finally, another interesting observation is that the clients and the servers could use the entire LIME API as an extension to the standard Java API. This API can be explicitly used for communication and coordination.

5. Conclusions

This paper examined the service model (widely used in client-server systems) with respect to its ability to support flexible application development in an ad hoc networking environment. We have been able to show that the complexity of the ad hoc networking can be hidden behind a simple service registry interface that can offer transparent access to both local and remote services in a uniform way. The effects of mobility are completely masked and expressed simply as changes in the contents of a local service registry. This study serves also as a demonstration of expressive power of coordination models, and of the LIME middleware, in particular.

References

1. W. K. Edwards: Core Jini. The Sun Microsystems Press. Java Series. 1999
2. J. Newmarch, "Guide to JINI Technologies", APress, November 2000
3. Jini home page http://www.sun.com/jini/
4. Jini specifications http://www.sun.com/jini/specs/
5. The community resource for Jini technology http://www.jini.org/
6. E. Guttman, Service Location Protocol: Automatic Discovery of IP Network Services, *IEEE Internet Computing*, 3(4): 45-53, July 1999
7. C. Renner, Introduction to the SLP Implementation, 2000, http://www.lkn.e-technik.tu-muenchen.de/~chris/slp/IntroSLP.html
8. C. Perkins, White Paper on SLP, 1997. http://playground.sun.com/srvloc/slp_white_paper.html
9. E. Guttman, C. Perkins, RFC 2608: Service Location Protocol, Sun Microsystems, June 1999.
10. Universal Plug and Play Forum. Universal Plug and Play home page. http://www.upnp.org/, 2001
11. Y. Goland, T. Cai, P. Leach, Y. Gu: Simple Service Discovery Protocol http://www.upnp.org/download/draft_cai_ssdp_v1_03.txt
    Microsoft Corporation, Shivaun Albright, Hewlett-Packard Company, October 1999
12. Salutation Specifications, http://www.salutation.org/, 2001
13. Murphy, A. L., Picco, G. P., and Roman, G.-C., "LIME: A Middleware for Physical and Logical Mobility," *Proceedings of the 21st International Conference on Distributed Computing Systems*, April 2001, pp. 524-533.
14. Picco, G.P., Murphy, A.L., and Roman, G.-C., "Lime: Linda meets Mobility," In *Proceedings of the 21st International Conference on Software Engineering*, May 1999, pp. 368-377.
15. D. Gelernter, N. Carriero, "Coordination Languages and Their Significance", *Communications of the ACM*, vol. 35, no. 2 feb 1992, pp. 96-107