M2MI Service Discovery Middleware Framework

Hans-Peter Bischof, Joel Varela Donado

Rochester Institute of Technology
102 Lomb Memorial Dr., Rochester, NY 14623
hpb@cs.rit.edu, jxv0462@cs.rit.edu

Abstract. *The Many-to-Many-Invocation (M2MI) Service Discovery Middleware Framework provides an API for publishing, providing, and using services in a serverless ad hoc network of devices implementing the M2MI architecture. It is built on top of the M2MI layer, as a middleware to interact with user applications and facilitate the deployment, use, and providing of services. The middleware uses the M2MI architecture, having ServiceRepository objects exported to it on each participating device to keep information of the available services in the network. These objects broadcast invocations to each other to update all information. The information is kept in the form of ServiceDescription objects which are moved from Repository to Repository. A service provider publishes or unpublishes a ServiceDescription object locally; the Repositories automatically update in the network. A client looks for services based on the implemented Java interfaces, and then requests them through its local Repository. Unihandles of services and clients using the middleware are exchanged for allowing direct interaction between them. Two sample applications were developed to demonstrate the capabilities of this middleware. The M2MI Service Discovery Middleware is compared in this document to Jini's Service Discovery [1] and Lime's Service Discovery [3] [4], in the aspects of system requirements, design, architecture, and functionality.*

Key words:: Ad-hoc network, Service Discovery Framework

1 Introduction

M2MI [2] has been thought of as a paradigm for applications running on serverless ad hoc networks of fixed and mobile wireless proximal devices. Using the broadcast capabilities of the network, devices implementing the M2MI technology receive and send invocations on objects on the network. This scheme requires a configuration of the devices to use the available services in other devices. The general approach is to join a network and automatically discover any services available. Afterwards, choose the service to be used, with minimal configuration and human intervention both for using and providing services. For this, a service discovery protocol and middleware is demanded, applied in the same fashion by all devices implementing the M2MI technology. This project is aimed toward the development of the service discovery middleware for the M2MI architecture. It includes the comparison to similar services, such as Lime (Linda for Mobile Environment) and Jini Network Technology (from Sun Microsystems). The following section will describe the architecture of the project, including the M2MI architecture and the Service Discovery Framework architecture. Section 3 and 4 describe the comparisons with Jini and Lime, respectively. The final sections are dedicated to the Future Work, Conclusions, and References.

2 Architecture

2.1 M2MI Architecture

M2MI is a new paradigm for building distributed ad hoc network applications, in which method multiple remote objects implementing the same interface execute invocations.

2.2 M2MI Service Discovery Middleware Architecture

The framework was intended to be generic for all devices using M2MI. Its design allows for complete interaction between hosts providing, requesting, or exchanging services, regardless of the service being provided.

Figure 1 shows the architecture of the M2MI Service Discovery middleware. The Services and ServiceRepository classes provide the main components of the middleware, they sit on top of the M2MI layer. The Services class is for interfacing with the higher level applications and the ServiceRepository for the interaction with the rest of the devices in the ad hoc network. Detailed information on the elements and their behavior is described in the following sections.

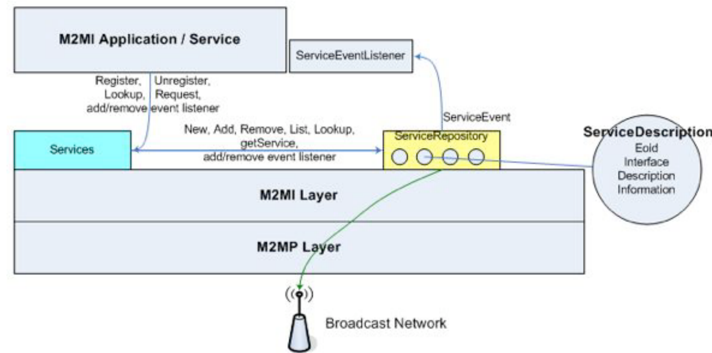

Fig. 1. M2MI Service Discovery Middleware Architecture

2.3 Elements

The components of the M2MI Service Discovery Middleware are described in this section. There are elements which are part of the core of the middleware, providing the main functionality. These are described in the next subsection. There are other elements which basically revolve around the core components helping with other important tasks as well. They will be described after the core elements.

Core Elements

The API classes are built on top of the M2MI layer. Communicating directly among the devices implementing the framework is the ServiceRepository class. An instance of this class is exported to the M2MI layer so it can send and receive invocations from other devices through M2MI. It holds ServiceDescription objects, which encapsulate the information of advertised services. These objects are obtained locally, belonging to an application in the device, or they are received from the other Repositories sharing the services in their own devices. The end result is that the ServiceRepository contains information of all services available in the network. Because this class has very specific and critical functionality to the middleware, the Services class was written to provide interaction with the developed applications at a higher level and to handle other functions that did not need to be included in the ServiceRepository. More information on these functions is provided in the following section. There are also three elements that allow applications to be updated with the new information that develops in the network, in terms of which services go in or out.

Complementary Elements

The ServiceRepository objects in the M2MI layer of different devices need to be consistent. That is, they must have the correct information about the services provided by themselves and by others. When a service is added to the ServiceRepository it must stay there, being advertised, for as long as the service is running. The middleware's architecture was designed so the developed services and the ServiceRepository belong in the same process. This means that the developed services and the ServiceRepository are brought down altogether in case the process in which they reside fails. However, if in the future the architecture is changed to have the services and the ServiceRepository in different processes, then it could be the case that a service is unexpectedly brought down. When this happens the ServiceRepository must tell this somehow. The solution to this problem is very common, and that is with a leasing scheme. This is handled by the Services class, each ServiceDescription object must have a lease to stay in the ServiceRepository. To facilitate the leasing the class ServiceLeaseRenewer was written. The leasing scheme was implemented and the developed applications are compelled to use it even though the architecture itself was not designed to require it. The reason it was included is so in the future it is easier to extend it to an architecture where the ServiceRepository containing the information on the service resides in a remote process.

2.4 Interaction

The Services class provides the main interaction with the API for the developer; it is in this class from where the developer will call methods to use the framework. The Services class is not instantiated, rather the methods and structures are static, and then it is this class the one that instantiates the ServiceRepository. The Services class gets invocations from the upper level application to register and unregister a service, this is, to publish and share them on the network and to take them off of it. When a service is registered, the Services class will pass the ServiceDescription object it received when the method for registration was invoked to the ServiceRepository in order to share it with the rest of the devices on the network. The ServiceDescription object is filled with information inherent to the service itself, therefore, it is the same service which will create and fill the ServiceDescription object, and then pass it to the Services class.

When a service is taken off the network through a call to the Services class, then the Services class tells the ServiceRepository to take this service out of the network, unpublishing it. The ServiceRepository then takes the corresponding measures, notifying listeners and other repositories about the changes. The diagram in Figure 3 illustrates this process.

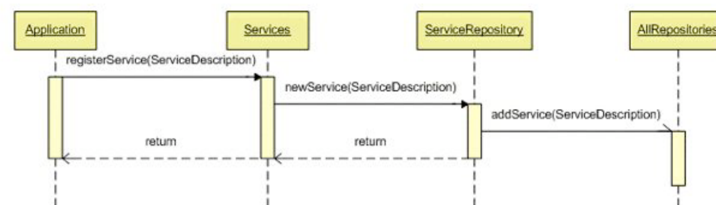

Fig. 2. Sequence diagram for registering a service

2.5 Consistency

Because of the nature of an ad hoc network, in which devices can join it and leave it frequently and unexpectedly, the consistency of the information contained in the ServiceRepository objects about the services available takes a special significance. It is necessary to keep the ServiceRepository objects with updated information as

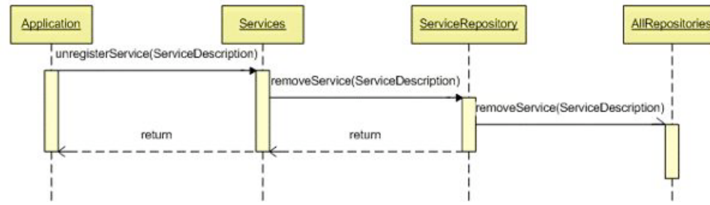

Fig. 3. Sequence diagram for unregistering a service

much as possible. Adding and removing services are the actions that change the contents of the repositories and they will be done manually and locally by the developed services; the responsibility of the updates of the rest of the repositories falls on different entities. Depending on the cause of the change, then the update will be performed in a different fashion. These are the two cases:

Normal Behavior: on the services, it is meant as when they are registered or unregistered by calling the corresponding methods from the Services class. After the local ServiceRepository is updated with the new or removed service (by the Services class), it will notify the other repositories broadcasting an M2MI invocation to all the repositories with an omnihandle. This takes care of updating the repositories. At a higher level, where the applications reside, the updating is important, too. The ServiceRepository sends events to the registered listeners, which are local to the device every time a change in the service occurs, that is, when a repository adds or removes a service locally or after receiving a remote invocation regarding a remote service. When an event is generated, a new thread is created to send this event. This is to allow the repository to take care of other actions and also to prevent from blocking any other events that may generate.

Failures: in this case, an individual service may go down unexpectedly, or even a whole device (leaving the network or also failing). To take care of an individual service going down, a leasing scheme was implemented. As soon as a service is registered, and for as long as the service runs and wishes to be advertised on the network, it must lease with the Services class. If the service fails unexpectedly, then the lease will fail to be renewed and it will expire on the Services class. The Services class will then take the needed measures, notifying the local ServiceRepository about the service to be removed and the ServiceRepository will go with the usual procedure invoking all the repositories and sending an event to local listeners. In the current architecture, it is not possible for a service to fail without having the ServiceRepository fail as well, because they are both in the same process. The leasing scheme was implemented to allow an easier extension for the future in which the ServiceRepository may reside in a remote process, hence, having a different architecture than the one present.

When the whole device goes down or leaves the network, then the ServiceRepository is not able to notify of any changes. To address this situation, every repository frequently polls the rest of the network asking the rest of the repositories to report their local services. After updating, the repository notifies the local listeners of any changes with the usual events. The polling occurs every minute. This will let the devices have only one minute of erroneous information before they realize that a remote service has gone off the network.

3 M2MI SD Comparison with Jini SD

This is the first of two chapters dedicated to the comparison of M2MI's SD framework to the SD frameworks of other technologies also for serverless distributed systems. Both chapters cover the system requirements, design and architecture, and functionality of the technologies. This chapter discusses the Jini Network Technology of Sun Microsystems. Jini was developed for providing services in a network, considering the reliability of the visible services and allowing the services to join and leave the network in a robust fashion, leaving them available for and ready to be used by clients in the network [6].

3.1 Design and Architecture

Proxy objects are central to the functionality of the Jini architecture. Proxy objects are Java objects which act as surrogates to a remote object. By calling methods on a local proxy object, the calling process can communicate with the remote object. The proxy object handles the remote call, transporting the request through a communications channel to the remote process and receiving the result of the remote invocation. In Jini's architecture, they define a service type with their particular Java type. The proxy objects implement the appropriate type (interface) for the service they will provide the client with. The proxy is the part of the service that will run on the client's virtual machine. A client downloads the proxy object for the specific service and then calls methods on this proxy object to use the remote service. The proxy object may be downloaded or obtained in a different fashion or the client may already have it, but it is important to have the right proxy object for each service.

Jini is based on the discovery of services in a network without central servers. As soon as a device joins a Jini network and needs or wants to deploy a service, it must start the service discovery and lookup mechanisms. Hosts entering the network must communicate with a Lookup Service in one of the hosts already in the network, which stores the information about the services available. Using their Discovery Protocol, the Lookup Service will be found and a proxy class for this Lookup Service will be returned. The Discovery can have three forms: a host broadcasting a request for Lookup Services to respond, a Lookup Service broadcasting a message identifying itself, or a direct message addressed to a known Lookup Service for requesting its proxy class. No matter the method adopted, the end result is the host having the proxy class to the Lookup Service[6].

Lookup Services can be located in many places in the same network, any host can hold a Lookup Service. This is because of the architecture being designed to work without central servers. However, a host does not need to be a Lookup Service. Thus, in case there is only one Lookup Service on the network, if it goes down then it is a single point of failure and the network may become disconnected, hosts may go in and out of the network without having a way to let the rest of the network know. It may be solved having a Lookup Service locally, then a service will not have to depend on another host's Lookup Service. Different Lookup Services within the network carry the same information, allowing duplication and failure tolerance. Looking into M2MI, there are similarities and differences.

Another similarity found is that an M2MI service will also have to Join (in Jini's terms) the network by registering the service with the Services class, and perform lookups based on Java interfaces using the static methods of the Services class. The services are requested to the Services class, which returns a Unihandle to the requested service. Any Handle in M2MI (unihandle, omnihandle, or multihandle) is a proxy for the object associated with that handle, thus being similar with Jini also in this aspect. One palpable difference in the procedures for acquiring services is that of the usage of a local class in the case of M2MI; Jini uses a proxy class received from the Lookup Service which the remote service will use to perform certain operations. In M2MI, the lookups are performed locally on the same JVM with the Services class, a class developed specifically to provide interaction with the user and isolate the ServiceRepository from possible direct user modifications.

3.2 Functionality

Services developed for Jini must have a proxy class to be distributed on the network. This proxy class is the one passed along to the rest of the network clients who want to use it. To publish or obtain these proxy classes, hosts entering the network must communicate with a Lookup Service, which stores the information about the services on the network with its proxy classes. When a client performs a lookup using the Lookup Service proxy object and then chooses a service to use, the Lookup Service will return the requested service's proxy class for the client to use. At this point, the Service Discovery procedure is completed in Jini, any other operations are inherent to the service itself, like any application protocol it may implement for itself with its clients. M2MI provides the same lookup functionality through the Services class. These are performed and then a group of matching services are returned, similar to Jini. The Services class checks with the ServiceRepository the set of available services to provide the matching later returned to the caller. The client will choose what it needs

and request the service to the Services class. The Services class will check with the ServiceRepository that the requested service is indeed still available in case it is remote and then it will proceed to return a Unihandle to the client, which is the remote reference implementing the service's interface.

4 M2MI SD Comparison with Lime SD

This chapter covers the comparison of M2MI's SD framework to Lime: Linda in a Mobile Environment. Lime is a system designed to assist in the rapid development of dependable mobile applications over both wired and ad hoc networks. Mobile agents reside on mobile hosts and all communication takes place via transiently shared tuple spaces distributed across the mobile hosts. The decoupled style of computing characterizing the Linda model is extended to the mobile environment [4]. There was no clear service discovery scheme found in Lime documentation. However, its architecture and design provide for capabilities of joining a network and using available services, as it will be seen in the following chapters.

4.1 Design and Architecture

Lime is based on transient shared tuple spaces, which are themselves based on Linda's shared tuple spaces. In Linda, a tuple space is a global and persistent repository of tuples, essential data structures constituted by an ordered sequence of typed fields. Every concurrent process in the system is supposed to have access to the tuple space (hence the globality property), which exists independently from the existence of the processes (hence the persistency property). Linda provides a minimal interface to the tuple space, with only three operations: one to write a tuple to the tuple space, one to get a copy of a tuple in the tuple space that matches a given template, and a third that in addition to getting a copy of a matching tuple withdraws it from the tuple space. In case of multiple matching tuples, one is returned non- deterministically. Lime adapts this fundamental model of coordination and communication to encompass both physical and logical mobility. In Lime, agents (the active components in the system) can roam across mobile hosts (which act as mere containers for the agents), which can roam across the physical space. The presence of mobility prevents the existence of a global and persistent tuple space. Therefore, in Lime each mobile agent owns at least one Lime tuple space (LTS) that follows the agent during migration. The notion of a global and persistent tuple space is dynamically recreated on a host by merging the LTSs of all the mobile agents there co-located, thus creating a host-level transiently shared tuple space. Similarly, it is recreated across hosts by merging the host-level tuple spaces into a federated transiently shared tuple space. The contents of these tuple spaces are dynamically reconfigured by the system when an agent arrives or connection is established (tuple space engagement) and when an agent leaves or some host gets disconnected (tuple space disengagement). This way, Lime provides the programmer with the illusion of a global and persistent tuple space, which can still be accessed using the conventional Linda operations. It is this tuple space we refer to as the federated tuple space [4].

4.2 Functionality

Because of the mobility nature of hosts in the ad hoc network, the situations of joining and leaving the network must be handled. Lime introduces the concepts of engagement and disengagement. On engagement, a host joins Lime and updates the shared space. In this case, it could receive tuples meant for providing services, and also publish its own. Since the space is shared, all agents in the Lime group are able to see these changes and use or provide services they may have been waiting for. Disengagement could happen by accident or on purpose. Failure tolerance mechanisms must be applied. There are two schemes in disengagement, one is called beacon, in which an agent will beacon periodically, and when it is about to leave it will stop beaconing. If there is a failure, the beaconing will stop and it will be noticed. This is similar to leasing in M2MI SD. The second scheme is called safe-distance. Since hosts are moving, using distance measuring mechanisms it can be established how far from the Lime group in the network the agent is, when it has reached a certain distance

then it will be disengaged [4]. This scheme has no similar implementation in M2MI, other than the actual network broadcast range. A host in M2MI will stop receiving invocations if it gets out of the broadcast range of the network, but a key difference is M2MI devices will not be able to tell and will not take any measures when a host is about to leave the proximal network. Since the shared tuple space is basically a field with data structures, Lime introduces reactive programming. It is a paradigm in which code or a program is executed in reaction to changes in the tuple space when new tuples become available. Using a new command called reactsTo (not available in regular Linda), a tuple can act as an event with information valuable to the code or program executed when this tuple becomes available on the network [4]. Similarly, M2MI handles changes on the services available using events and its own remote method calling capabilities to update the information.

5 Future Work

More applications can be developed using the Service Discovery API for the enhancement and greater proof of robustness of the middleware, as well as the usefulness of M2MI. Authentication, confidentiality, and access control to services was not implemented in this API. It is an important issue for providing services that are not open to all devices or users in a network, or that require some sort of control and that would be standardized in all the M2MI architecture. These controls cannot be carried out in a central server, since these are not available in the network. Security for services can be implemented by the developer at the user application level.

6 Conclusion

The M2MI Service Discovery API provides a middleware and framework which allows devices M2MI capable in a proximal network to publish, provide, and use services. M2MI's architecture and paradigm of broadcasted invocations gives an advantage at developing serverless distributed systems for ad hoc networks. The Java programming language provides an advantage at developing this type of systems due to its reusability; the JVM sits on top of the different platforms allowing portability. The API was successfully tested on Solaris and Windows systems, both independently and simultaneously. The tests were successful and functional, meaning the API serves its purpose. After comparing it with Jini and Lime, it is found that capabilities between the systems are much alike; the functions they provide and their procedures, even though implemented differently, carry similar purposes, specifically those of publishing, lookup, and retrieval. The concept of having shared data structures, such as the ServiceRepository in M2MI and tuples in Lime, proved helpful in the implementation of the Service Discovery middleware.

References

1. Ken Arnold, Bryan O'Sullivan, Robert W. Scheifler, Jim Wlado, and Ann Wollrath. The jini specification. Addison-Wesley, 1999.
2. Alan Kaminsky and Hans-Peter Bischof. Many-to-many invocation: A new object oriented paradigm for ad hoc collaborative systems". In *7th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications, Onward Track*, 2002.
3. A.L. Murphy Picco, , and G.-C. Roman. Developing mobile computing applications with lime, 2000.
4. A.L. Murphy Picco and G.-C. Roman. Linda meets mobility. In *Proc. of the 22nd Int. Conf. on Software Engineering*, pages 368–377, New York, NY, USA, 2000. ACM Press.