ROCHESTER INSTITUTE OF TECHNOLOGY

M2MI SERVICE DISCOVERY MIDDLEWARE FRAMEWORK

MS PROJECT TECHNICAL REPORT

(Version 3)

By JOEL VARELA DONADO

Committee:

Chairman: Prof. Hans-Peter Bischof

Reader: Prof. Alan Kaminsky

Observer: Prof. Warren Carithers

Observer: Prof. Sidney Marshall

ACKNOWLEDGEMENTS

I want to thank Prof. Hans-Peter Bischof for his continuous and valuable guidance, and Prof. Alan Kaminsky for his valuable comments and corrections on the Technical Report. I also want to thank the other members of my Project committee, Prof. Warren Carithers and Prof. Sidney Marshall for their assistance.

ABSTRACT

The 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 and Lime's Service Discovery, in the aspects of system requirements, design, architecture, and functionality.

TABLE OF CONTENTS

1. OBJECTIVES

M2MI 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). Below is the detailed description of the objectives of this project:

- M2MI Service Discovery API: provide an API written in Java which will enable the M2MI technology to incorporate a Service Discovery Protocol and Framework. This framework allows the proximal devices which are M2MI capable to use the service discovery as a tool to publish and retrieve services available in the proximal network they may be part of.
- Services Discovery Comparison: give an insight on how Jini and Lime behave for their service discovery middlewares, comparing their system requirements, architectures, and functionalities with their M2MI counterpart.
- Applications Examples: successfully test the API developed. Furthermore, these examples allow us to demonstrate the behavior of the M2MI service discovery middleware.

The following section will describe the architecture of the project, including the M2MI architecture and the Service Discovery Framework architecture. Section 3 describes in detail how the middleware works and how it should be used, including the M2MI Service Discovery Protocol and two applications examples. Section 4 gives a description of the internal design of the middleware. Sections 5 and 6 describe the comparisons with Jini and Lime, respectively. The final three sections are dedicated to the Future Work, Conclusions, and References. Appendix A gives the API Documentation.

2. ARCHITECTURE

2.1. M2MI ARCHITECTURE

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

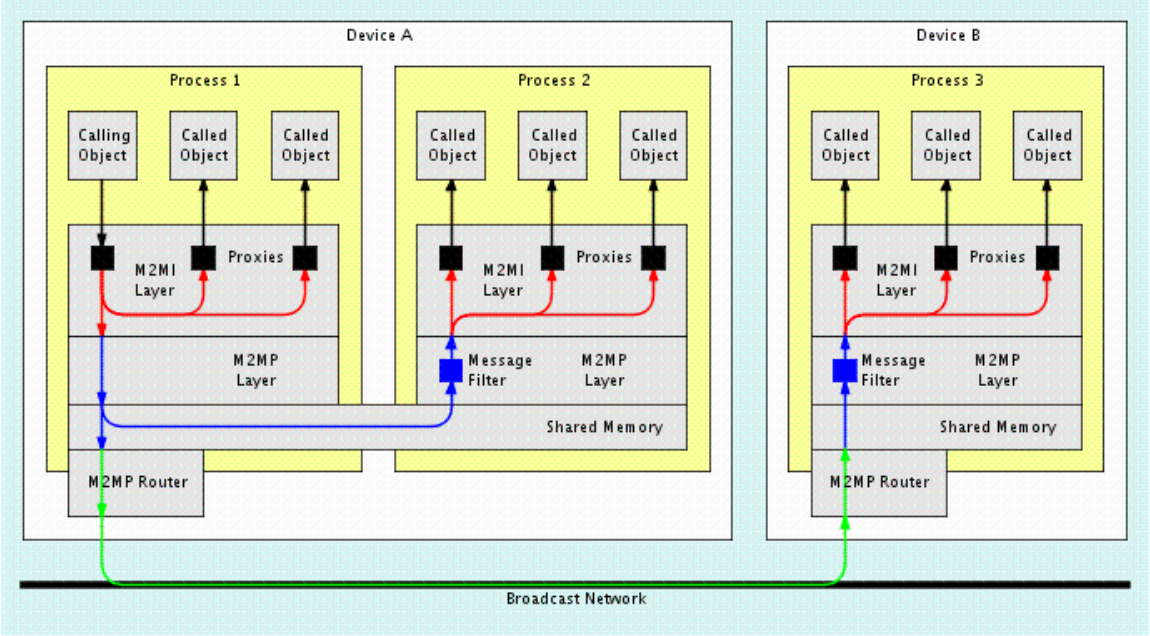

Figure 1: M2MI Architecture – [1]

Figure 1 shows the M2MI architecture. Here, there are two devices running the M2MI layer which needs to be initialized before running applications, and the scheme shows how method calls are transported through the M2MI layer in device A and through the broadcast network which could be wireless or wired, between devices A and B. All that is transported are method calls. The service discovery middleware sits on top of the M2MI layer and provides the user with the options or choices to look up, retrieve and start using

any service that may be available in the proximal network. Now follows the description of the M2MI Service Discovery Middleware architecture.

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.

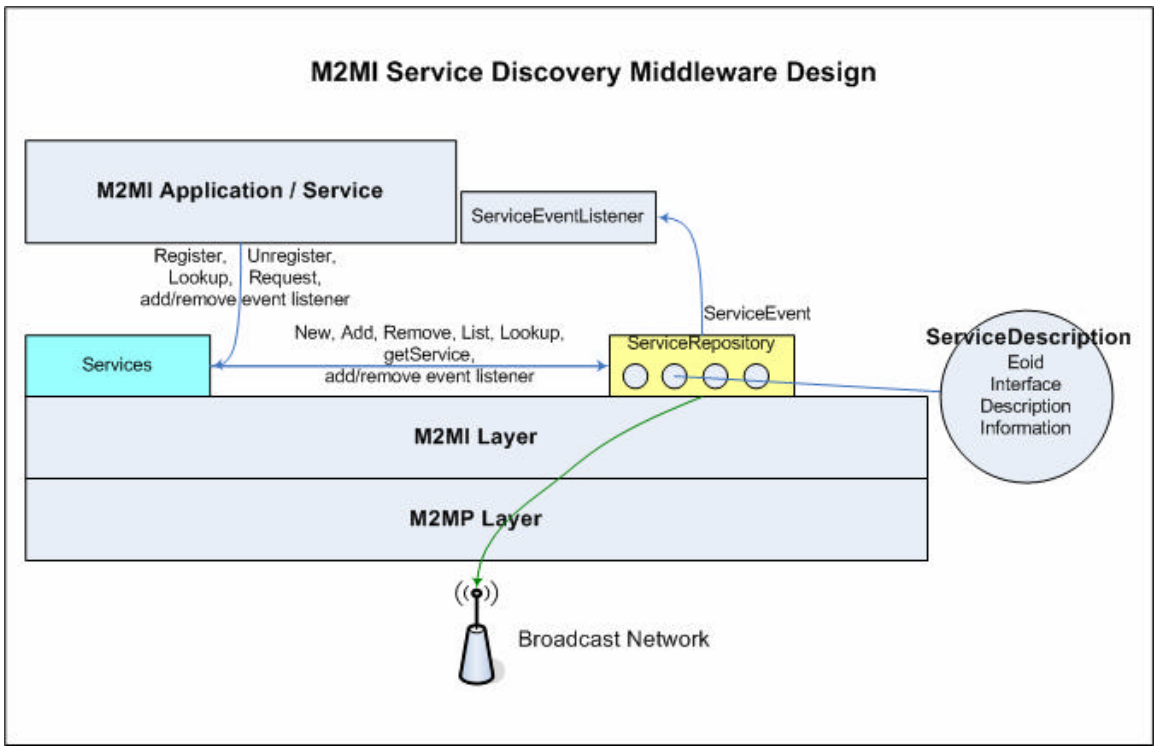

Figure 2: M2MI Service Discovery Middleware Architecture

Figure 2 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.

2.2.1. 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. These are the ServiceEvent class, and two interfaces: ServiceEventListener and ServiceEventFilter. ServiceEvents are passed by the ServiceRepository to the listeners (objects implementing the ServiceEventListener interface), which must have registered in advance with the Services class. When an object does not want to continue receiving these events, it can unregister. An object can also tell what events it wants to receive, passing another object implementing the ServiceEventFilter interface when registering. This interface gives a method which evaluates the ServiceEvent generated and determines if it should be passed to the ServiceEventListener object which originally provided this ServiceEventFilter.

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. The consistency of the ServiceRepository objects is discussed in detail in section 2.2.3.

After writing a service and having all the software ready to publish it on the network, it is desired to start it perhaps as the device starts up, or automatically with other services provided in the same device. The class ServicesLoader was developed to load all services wanted at once, through the reading of a properties file. This class can be run as a main program, and goes through the properties file in which it will find services to start. The ServicesLoader class merely will create an instance of the class found in the properties file, so the class developed to work with the ServicesLoader should take care of all procedures necessary to successfully load the service to the network. A full description of the use of the ServicesLoader is found on Section 3.

2.2.2. 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. The following sequence diagram illustrates the process of registering a service.

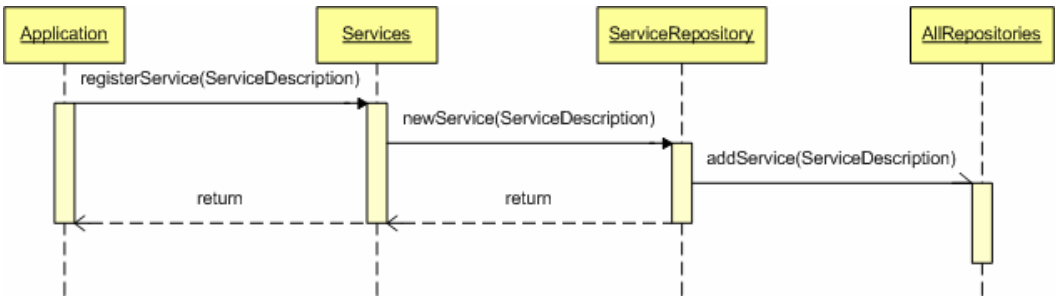

Figure 3: Sequence diagram for registering a service

The previous sequence is generated when an application calls the mentioned method in the Services class for some exported service, as such:

```
ServiceDescription sd = new ServiceDescription(serviceEoid,
                        serviceUnihandle, serviceInterface);
Services.registerService(sd);
```

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 following diagram illustrates this process.

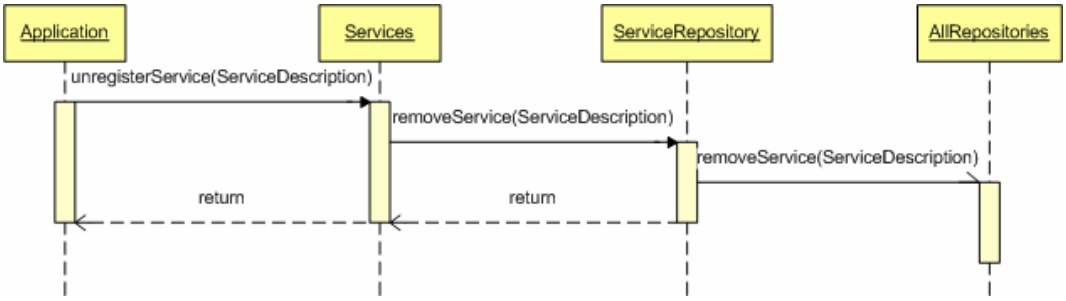

Figure 4: Sequence diagram for unregistering a service

The sequence presented above is generated with the following call in the Services class for some exported service:

```
Services.unregisterService(sd);
```

For the correct functioning of the network, it is important to keep all ServiceRepository objects updated. This is the topic of the following section.

2.2.3. 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 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:

1. **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.

2. **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. This time is hard coded in the Services class. In the future it could be changed to be

configurable from a properties file or a method and be set to be randomized to decrease the probability of collisions. An alternative to the polling can be that every ServiceRepository broadcasts its state periodically. This would reduce the amount of traffic in the network, and it could be implemented in future work.

The next section gives instructions on the usage of the middleware.

2.2.4. UML DIAGRAMS

The class diagrams in this section correspond to all elements developed. Detailed documentation on all elements of the API is found in Appendix A.

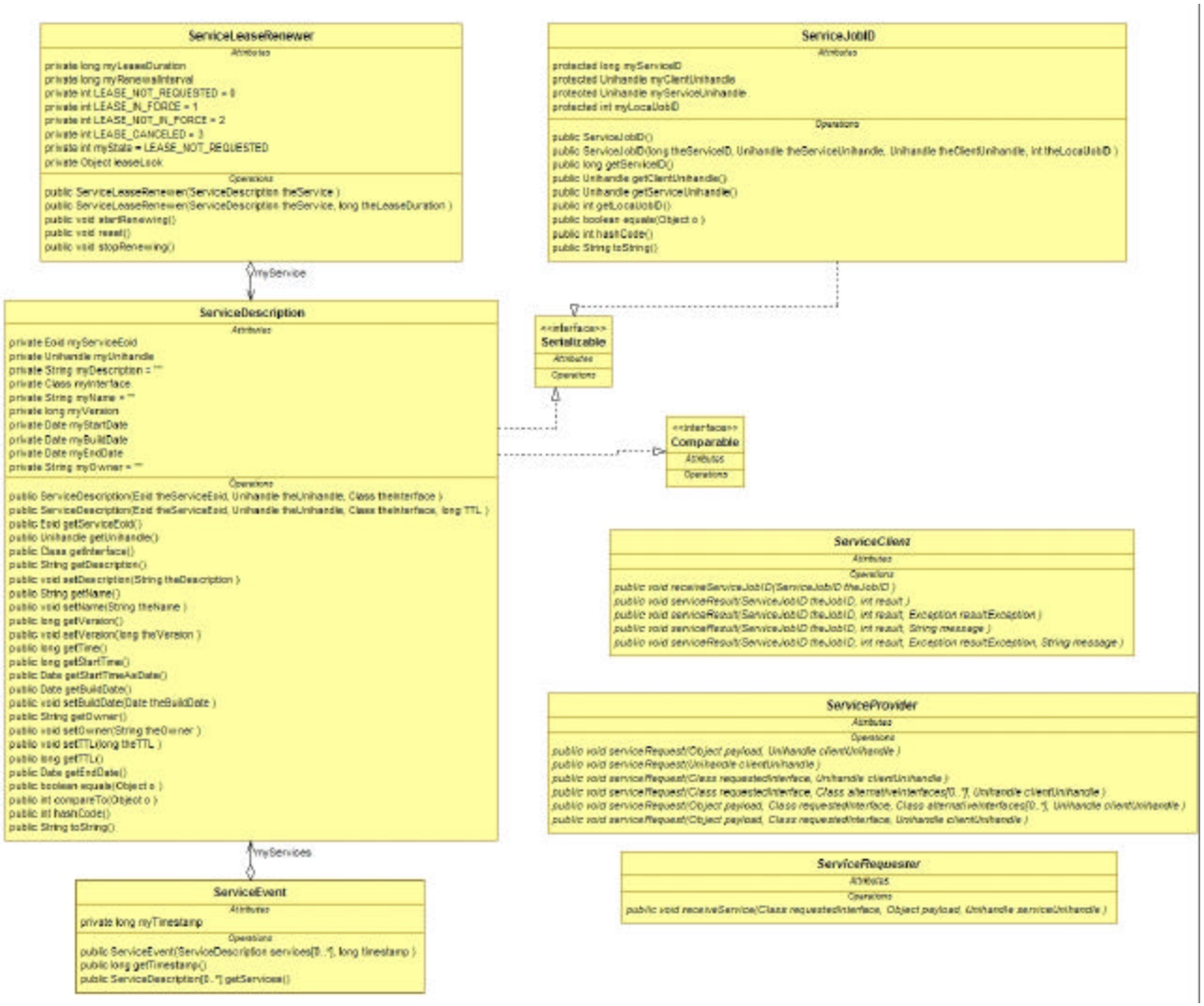

Figure 5: ServiceDescription, ServiceEvent, ServiceLeaseRenewer, Servi ceJobID, ServiceClient, ServiceProvider, ServiceRequester

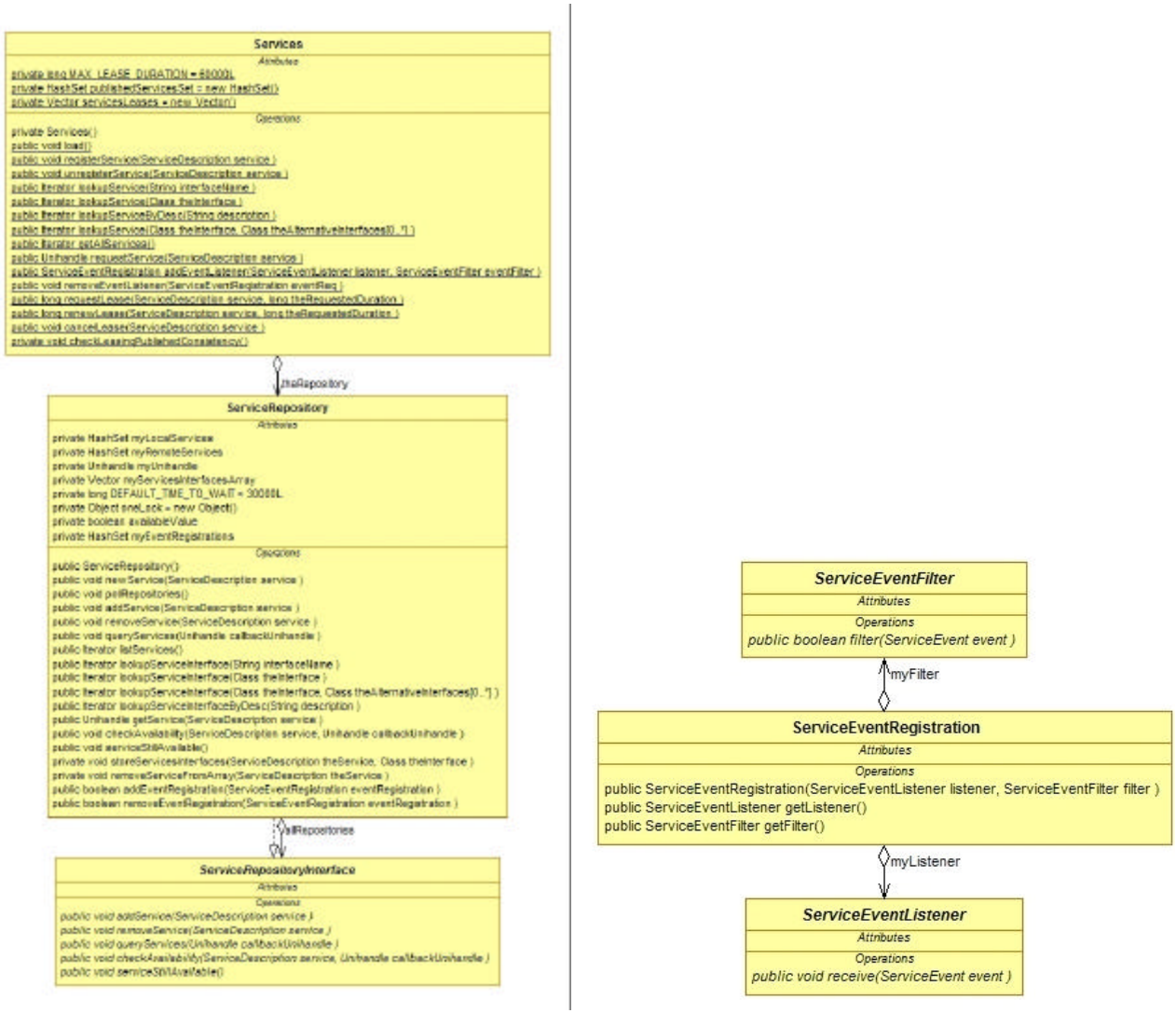

Figure 6: Services, ServiceRepository, ServiceRepositoryInterface, ServiceEventRegistration, ServiceEventFilter, ServiceEventListener

Figure 7: ServicesLoader, ServicesLoaderException, LeaseRenewerException, ServicesRuntimeException, ServiceNotPublishedException, ServiceAlreadyLeasingException, ServiceNotLeasingException.

3. M2MI SD FUNCTIONALITY

This chapter gives instructions on how to use the middleware with developed applications. The Service Discovery Protocol is described and two sample applications will close this chapter.

For the middleware, what will be provided is irrelevant. It is designed to work with a generic class encapsulating the advertised service information. This class is the `ServiceDescription`.

The main functionality for the service discovery framework is provided through the `Services` class, which provides static methods to publish, unpublish, lookup, and request a service in the network. These four actions are the basis for working with the services. The framework is based on the implemented interfaces by the services and their clients. This means, clients will mainly look for a service implementing a given interface, and the main identification pattern between services is the advertised interface they implement. In case there is more than one service object implementing the same interface in the network, since they are distinct, we will be able to distinguish them by their `Unihandle`. The final task of this middleware, at a low level, is to exchange unihandles between two hosts. In some cases, only the client will need the server's unihandle, but both possibilities can be accomplished with the middleware. The framework was developed to be thread safe. All methods on classes can be called concurrently, but watch carefully for restrictions, as only operations on one service can be performed at a time. Refer to the documentation of each class for details.

It is important to follow the **M2MI Service Discovery Protocol**, which is the main guideline for developing applications and synchronizing communication between a server and a client, and that is described below.

3.1. M2MI SERVICE DISCOVERY PROTOCOL

The M2MI Service Discovery Protocol defines the guidelines to start a communication process between two M2MI hosts, being those a *server* and a *client*, implementing the `ServiceProvider` and the `ServiceRequester` interfaces respectively. The *server* must be advertised on the network as explained in the following sections, using the middleware API to accomplish the initial communication (exchanging unihandles) which will allow the start of the protocol procedures.

There are two steps to follow in this process:

1. **Request:**

   After receiving the unihandle using the API, a client calls one of the `serviceRequest` methods on the server (which implements `ServiceProvider`) in which the client specifies the interface he is looking to be provided with. The client can specify one main interface, an array of alternative interfaces, or no interface in order to take a default interface from the server.

2. **Response:**

   The server calls the `receiveService` method on the client, in which it suggests a service to be provided implementing a default or one of the interfaces requested by the client on the first step. At this moment, the client receives the Unihandle for the service it is looking for, and then it can start calling methods inherent to the application.

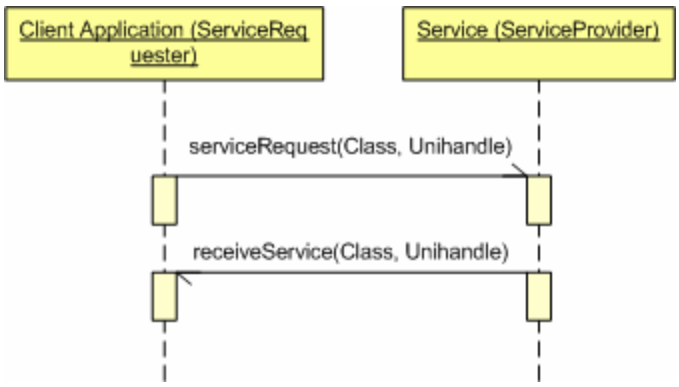

Figure 8: Sequence diagram for the M2MI SD Protocol

The interfaces mentioned above, `ServiceRequester` and `ServiceProvider`, were defined to have a standard use of the API following the protocol. However, and as it will be seen later in one of the examples on this chapter, it is not necessary to adopt this protocol to develop applications, but it is the recommendation to use it for standardization. The following three sections will give detailed instructions for using the middleware, divided by stages of publishing, using, and unpublishing services.

3.2. PUBLISHING AND PROVIDING A SERVICE

After a service application has been developed and is ready to go on the network to provide its service, then a choice must be made on how this application will be deployed. There are two ways to do this, one being to let the application handle the communication with the clients, following the protocol; or having another object or application, a third party, handle the communication with the client following the protocol to finally hand out to the client the unihandle for the actual service. The key point here is building the `ServiceDescription` object which will advertise the service. The unihandle to be put into the `ServiceDescription` object published must be chosen according to what is needed. In the first case, for example, since the service will be self-advertising, then it is a good idea to include the same service's unihandle in the `ServiceDescription` object. To do so, the following code can be applied:

```
Unihandle oneUnihandle;
SomeService theService = new SomeService(args);
M2MI.export(theService, SomeServiceInterface.class);
oneUnihandle = M2MI.getUnihandle(theService,
                  SomeServiceInterface.class);
oneServiceDescription = new ServiceDescription(
                  oneUnihandle.getEoid(),
                  oneUnihandle, SomeServiceInterface.class,
                  30000L);
```

Note here that `theService` is being exported, and this code could be put inside the service program itself, or in another program to load `theService`. The ServiceDescription object, called `oneServiceDescription`, is constructed with four arguments. The first argument (`oneUnihandle.getEoid()`) corresponds to the Exported

Object ID (Eoid) of the unihandle of the service, the second argument (`oneUnihandle`) is the unihandle of the service, the third argument (`SomeServiceInterface.class`) is the interface implemented by the service unihandle which will be advertised on the network, and the last argument (`30000L`) is the time to live for the service in milliseconds, or the total time this service will be advertised on the network. There are no automatic mechanisms implemented in the middleware to unload a service using its own time to live. This argument is passed to fill the corresponding field of the ServiceDescription object and is informational only.

For the second case described above, in which there is a third object performing the communication for the service, deploying will be made with the same simple steps as above. Only now, we must consider the fact we are passing a unihandle not for the service, but for this third party. The scheme is then have the third party handle the communication, implementing the Service Discovery Protocol.

After the service has been defined, with either of the schemes shown above, and the `ServiceDescription` object constructed, then it is time to publish it on the network. Before calling any of the methods on the Services class, the M2MI layer must have been initialized. To publish, call the static method `registerService`. After this, the `ServiceDescription` object must start leasing with the Services class. This is to keep the service published and take measures in case it is unexpectedly shutdown, that is, let the rest of the network know this service is no longer available. In order to lease, create a new instance of the `ServiceLeaseRenewer` class passing the same `ServiceDescription` object as a constructor argument. Use the `startRenewing` method to start the leasing process and continue with any other operations. The `ServiceLeaseRenewer` will start a thread to continue with the leasing process as long as necessary, until it is explicitly told to stop, or a failure occurs. See the `ServiceLeaseRenewer` class documentation for more information on this class. Now follows an example which shows how to code this second step. In the example, extra code is included at the beginning to populate the `ServiceDescription` object with relevant information regarding the advertised service.

```
oneServiceDescription.setDescription("This Service does
                                     something");
oneServiceDescription.setName("Some Service");
oneServiceDescription.setVersion(1);
```

```
oneServiceDescription.setBuildDate(new Date());
oneServiceDescription.setTTL(3600000L);
oneServiceDescription.setOwner("I am the owner");
Services.registerService(oneServiceDescription);
oneLeaseRenewer = new ServiceLeaseRenewer(oneServiceDescription);
oneLeaseRenewer.startRenewing();
```

The information added at the beginning of the example can be accessed later by means of using the methods in the `ServiceDescription` class. The last three lines show the only steps needed to actually publish the service and leave it leasing with the `Services` class as long as necessary. Right now, the service is on the network, shared in the `ServiceRepository` and available to receive requests of communication from any clients. It is important that the published service, being this one a third party or the service itself, implements the `ServiceProvider` interface to follow the SD Protocol. The usage is described in the following section, 3.3.

To facilitate the loading of services, a utility class `ServicesLoader` was developed. It will load any number of services according to a properties file specification. For more information please visit the mentioned class' documentation.

The following file is an example which shows how a class is used to load a service, this class can be used independently as a main program or in conjunction with the ServicesLoader class. Filename: ServicesLoaderExample.java

```
import edu.rit.m2mi.M2MI;
import edu.rit.m2mi.Unihandle;
import edu.rit.services.Services;
import edu.rit.services.ServiceDescription;
import edu.rit.services.ServiceLeaseRenewer;
import java.util.Date;
import java.util.Iterator;

/**
 * This is an example for loading a service. This class works as
 * a main program
 * or as a class to be loaded by the ServicesLoader class. In the
 * servicesloader.properties
 * file, add the following line (the arguments are optional and
 * do not add anything to the
 * program):
 * <br><code>ServicesLoaderExample arg1 arg2</code><br>
 *
 * As a main program, run it with or without arguments.
 * NOTE: As it is, this class is not expected to compile because
 * of the class
 * <code>SomeService</code> which does not exist. Replace this
 * class with a service
```

```
 * you may have developed.
 */
public class ServicesLoaderExample {
      //this is the unihandle of the service which will be
      //exported to M2MI.
      Unihandle oneUnihandle;

      //this is the object which will be passed to the Services
      //class for publishing
      //using the Service Discovery Framework
      ServiceDescription oneServiceDescription;

      //this is the object used to keep it renewing.
      ServiceLeaseRenewer oneLeaseRenewer;

      /**
       * Public constructor with the given arguments.
       */
      public ServicesLoaderExample(String[] args){
            /*
            * we do not initialize the M2MI layer here because if
            * this constructor is
            * called from the ServicesLoader class, then the M2MI
            * layer is already initialized.
            */
            SomeService theService = new SomeService(args);
            M2MI.export(theService, SomeServiceInterface.class);
            oneUnihandle = M2MI.getUnihandle(theService,
                        SomeServiceInterface.class);
            oneServiceDescription = new ServiceDescription(
                        oneUnihandle.getEoid(), oneUnihandle,
                        SomeServiceInterface.class, 30000L);
            oneServiceDescription.setDescription(
                        "This Service does something");
            oneServiceDescription.setName("Some Service");
            oneServiceDescription.setVersion(1);
            oneServiceDescription.setBuildDate(new Date());
            oneServiceDescription.setTTL(3600000L);
            oneServiceDescription.setOwner("I am the owner");
            Services.registerService(oneServiceDescription);
            oneLeaseRenewer = new ServiceLeaseRenewer(
                        oneServiceDescription);
            oneLeaseRenewer.startRenewing();
            //after this call, the program waits indefinitely,
            //renewing the lease and keeping
            //the reference to theService which prevents it from
            //being garbage-collected.
            //The following lines are for stopping a service.
            //They are commented, otherwise the program
            //will close.

            //oneLeaseRenewer.stopRenewing();
            //Services.unregisterService(oneServiceDescription);
      }

      /**
       * Main program.
```

```
         */
        public static void main(String args[]){
              //in this case we try to initialize the M2MI layer
              //because we don't know if it has
              //been initialized previously
              try{
                    M2MI.initialize();
              }
              catch (IllegalStateException ise){
                    System.err.println(ise.getMessage());
              }
              try{
                    //creating a new instance the service is
                    //loaded.
                    new ServicesLoaderExample(args);
                    //block forever.
                    Thread.currentThread().join();
              }
              catch (Exception e){
                    System.err.println("ServicesLoaderExample: “+
                                       ”Uncaught Exception");
                    System.err.println(e.getMessage());
              }
        }
}
```

The following section will explain how to use the services published.

3.3. REQUESTING AND USING A SERVICE

The Services class provides different methods to look for services, which are to be used by the clients. These methods return iterators to the `ServiceDescription` objects which match. Please see the methods documentation for specific functionality. The following example performs a lookup for a service which implements `SomeServiceInterface`:

```
Iterator it = Services.lookupService(SomeServiceInterface.class);
ServiceDescription sd=null;
if (it.hasNext()){
      sd = (ServiceDescription)it.next();
}
Unihandle uni=null;
if (sd!=null){
      uni = Services.requestService(sd);
}
if (uni == null){
      System.err.println(“The service is no longer available”);
}
```

The iterator `it` contains `ServiceDescription` objects. The previous example just takes the first of the objects returned, but after examining the `ServiceDescription` objects obtained, a client must decide which service to use and must request it by calling the `requestService` static method in the Services class. This will return the `Unihandle` to the requested service, specified in the `ServiceDescription` object that was passed as an argument in the call to `requestService`. It is recommended to get the `Unihandle` in this fashion rather than getting it directly from the `ServiceDescription` object. This is because the Services class will make sure this service (which could be remote) is still available before returning the `Unihandle` to the client. If the service is not available, the call will return `null`. After the `Unihandle` is returned the Services class use is finalized. With the `Unihandle` each client should start using the service as specified in its particular implementation. This procedure is to be followed regardless of the scheme used to publish a service, either directly or through the intermediate class following the M2MI Service Discovery protocol. This last approach is the most recommended for standardization.

When following the Protocol, the returned unihandle to the client must be implementing the `ServiceProvider` interface, and in the same way, the `ServiceRequester` interface must be implemented by the client. As described in section 3.1., the client will request a determined service to the service provider, calling one of the `serviceRequest` methods on the `ServiceProvider` interface. Before calling any methods on the interface the unihandle reference must be casted to the `ServiceProvider` interface, as it is usually done when using M2MI. This can be done as:

```
((ServiceProvider) uni).serviceRequest(ServiceInterface.class,
                                       myOwnUnihandle)
```

In the `serviceRequest` method, the client can specify one or more interfaces it would like to use from the server (with other overloads), it will also pass its own unihandle so the `ServiceProvider` can respond directly to him, and a payload object which may be optional to the specific implementation of the service. The `ServiceProvider` will respond consequently with a service interface, a unihandle of the actual service implementing the interface requested, and an optional payload object according to the

specific implementation. This will be done by calling the `receiveService` method like such:

```
Object payload = new Object();
((ServiceRequester)clientUnihandle).receiveService(
                                ServiceInterface.class, payload,
                                theServiceUnihandle);
```

The payload object provided is just for the example, of course, for the implementation it will vary accordingly. A `null` reference can also be passed if that is the case.

3.4. UNREGISTERING A SERVICE

This section describes how to take a service out of the network, or unpublish it. Like stated for publishing it, also two steps are required. First, cancel the lease using the `ServiceLeaseRenewer` object constructed previously, calling the `stopRenewing` method. Second, unpublish the service calling the `unregisterService` static method on the Services class, providing the same `ServiceDescription` object used previously.
Note that nothing is exported or unexported to or from the M2MI layer in this procedure, the actual service being advertised must be independently exported and unexported, the `ServiceDescription` object is never exported. The following code can be used to unpublish the service related by the referenced `ServiceDescription`.

```
oneLeaseRenewer.stopRenewing();
Services.unregisterService(oneServiceDescription);
```

When the `ServiceLeaseRenewer` object is called for stopping the renewal of a lease, the Services class will unpublish the corresponding `ServiceDescription` after its next check of its leases times out, which could happen at any instant within one minute after the call to `stopRenewing`. The call to unregisterService on the second line of the above example makes the `Services` class unpublish the service immediately.
After the call to `stopRenewing` is made, there is still a chance to continue renewing with the same `ServiceLeaseRenewer`. To do this, call `reset` and follow with a call to

`startRenewing`. This last call must be made before the `Services` class removes this service for being published without a lease.

The next two sections, the last ones on chapter 3, describe a printing service and a multiple gaming service.

3.5. PRINTING SERVICE EXAMPLE

This example is provided in package `edu.rit.services.printing` to show the use of the M2MI SD Middleware API. This example implements the scheme of services being advertised themselves in the network, but they are loaded by a third party object. Hence, a class which is not performing the service itself, named `PrintServicesLoader` will upload the printing services and leave them ready for the clients to use it.

The printing clients when started will look for the main printing interface, `BasicPrinter`, and will display the available matching services. The user can then select any of the printers and enter text or load a text file that will be printed. The client is encapsulated in class `PrintingJFrame`.

The printing services are organized in the following hierarchy:

```
BasicPrinter
├── ColorPrinter
│   ├── PaperSelectionsColorPrinter
│   └── WideColorPrinter
├── PaperSelectionsPrinter
└── WidePrinter
```

Both interfaces and classes with the above names follow the hierarchy in the way described. The interfaces have the additional word “Interface” on their names to differentiate.

This example does not follow the M2MI Service Discovery Protocol. When a service is requested, the unihandle is returned immediately for the client to use it in case the service is still available. This means there is no other choice at requesting a service other than taking what is being advertised, which is the service itself.

On the other scheme, used in the other example in section 3.6. corresponding to package `edu.rit.services.game`, the Protocol is implemented and a third object is advertised on

the network as a service provider. Following the protocol, this service will negotiate a service to be provided to a client based on a certain implemented interface specified by the client or by default. To learn more see the mentioned package documentation and the M2MI Service Discovery Protocol in section 3.1.

Usage

To run this example, perform these steps, ensuring the M2MP Daemon has already been started, and the java `classpath` is set correctly:

1. Start the client with the following command:

   ```
   java edu.rit.services.printing.PrintingJFrame name
   ```

   where *name* is a name for the user of this service and it is optional. There can be as many clients as wanted in the system.
2. Start the printing services with the following command:

   ```
   java edu.rit.services.printing.PrintServicesLoader
   ```

   This command will start two services, one being `WideColorPrinterService`, and the second service is a `PaperSelectionsColorPrinterService`.

On the second step described above, the same class can be used in conjunction with the `ServicesLoader` class. In case two or more unrelated services need to be loaded, the `ServicesLoader` class will do this one by one. For using the `ServicesLoader` class in this case, add the following line to the `servicesloader.properties` file and according to the description on the mentioned class documentation:

```
edu.rit.services.printing.PrintServicesLoader
```

After this, run the `ServicesLoader` program. Only one of the two classes need to be

executed, either `ServicesLoader` or `PrintServicesLoader` to run this example. Refer to the `ServicesLoader` class documentation for more information.

Both services are used in the same fashion, the object of having two different services is to demonstrate how different services can be identified by their unihandles returned by the Framework, even though the search was performed by the interface `BasicPrinter`, implemented by both services. The lowest-level interfaces implemented by each service are `WideColorPrinter` and `PaperSelectionsColorPrinter`.

The printing services do not really print to any regular printer. They are an example, and what they do is take text from the client and print it on a TextArea in their GUI.

The client's GUI is displayed by the `PrintingJFrame` class and is shown below:

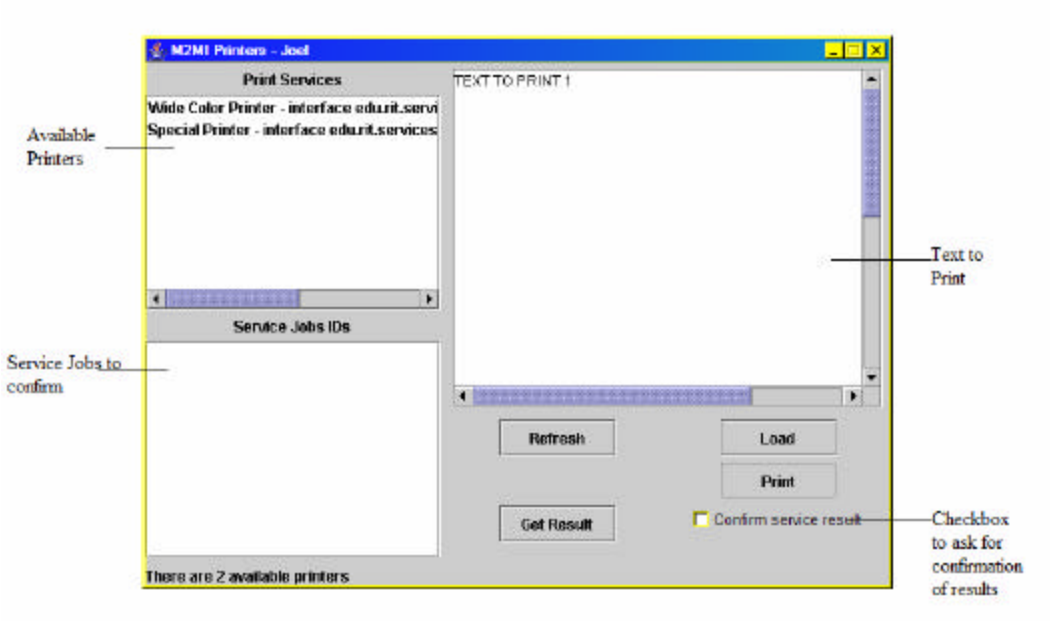

Figure 9: GUI layout for a printing service client.

On the left side, there are two scroll lists, one for holding the available printers and the lower one for storing the ServiceJobIDs returned from the printer. The ServiceJobIDs identify one printing job on a specific printer, and with this object the client can ask the printer about the status of the corresponding job. In this case, the client is implementing the `ServiceClient` interface and this is used by the printing service to notify of the job status. This will only be used if the client requests so, by selecting the checkbox on the right side of the frame marked as such for requesting a confirmation of the service. The TextArea on the right side is for typing any text to send to the printer, or to load a

file into it from the local file system using the "Load" button and then print it. To print, select one of the available services on the Print Services List and then click on the "Print" button. To clear the TextArea click on the "Clear" button.

When a confirmation of a job sent to the printer is asked for, this job will show in the lower scroll list. The printer automatically sends the result after it has completed the service, and alternatively, the client can ask for a confirmation if it selects the service job and clicks on the "Get Result" button. The following illustration shows how it is displayed on the client.

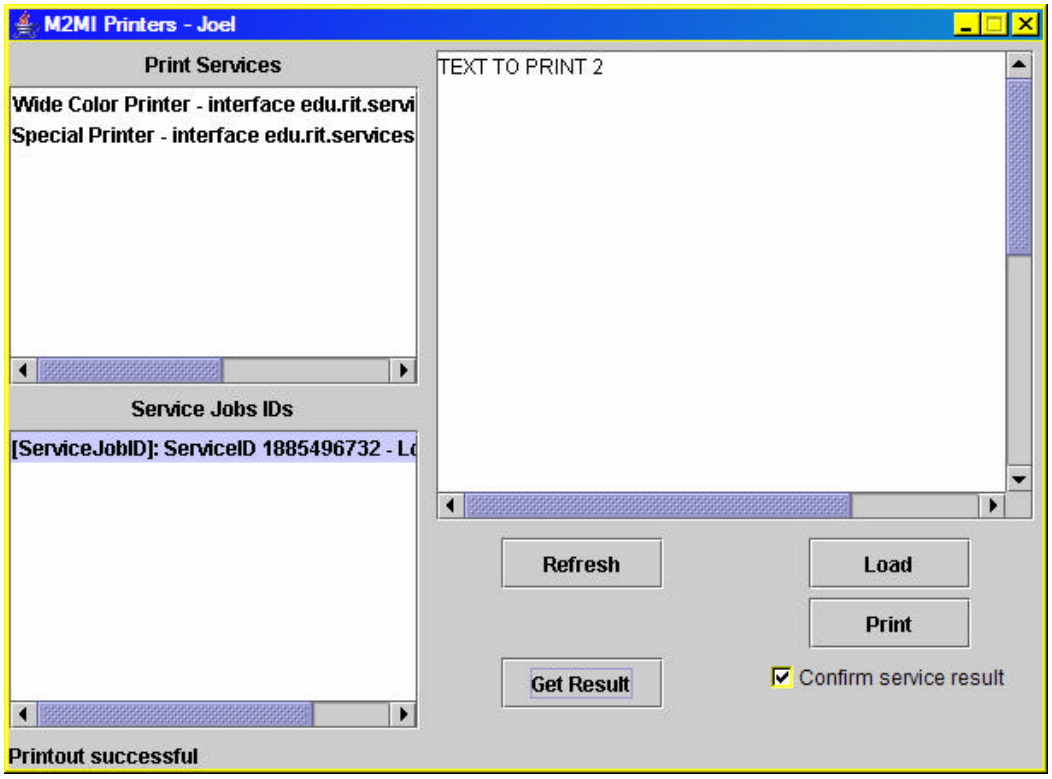

Figure 10: Result information from printer.

Above, the message from the printer "Printout Sucessful" is displayed on the bottom part of the frame.

The GUI displayed by both printing services is built by the class `PrintingOutputJFrame`. This GUI will show a queue with its pending jobs in the scroll list on the left side; on the right side there is the TextArea used to show the text from the print jobs. Below the TextArea there is a button tagged "Clear" which will clear the

TextArea when clicked on. The following two images illustrate a printer with a job on the queue, and then the same job after it has been printed.

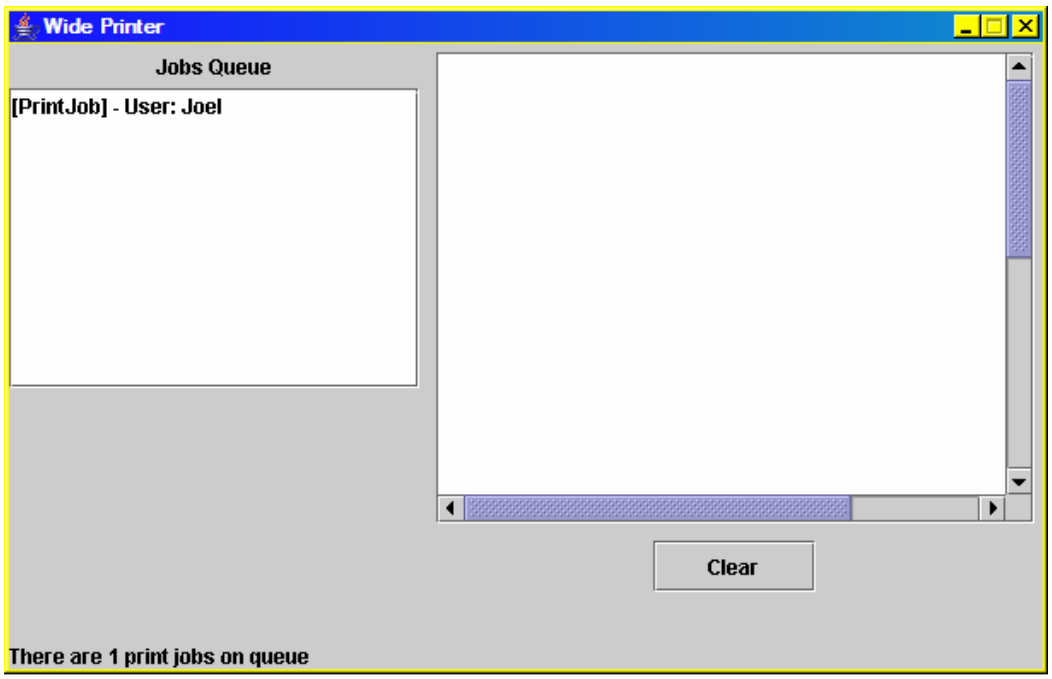

Figure 11: Job queued on the printer

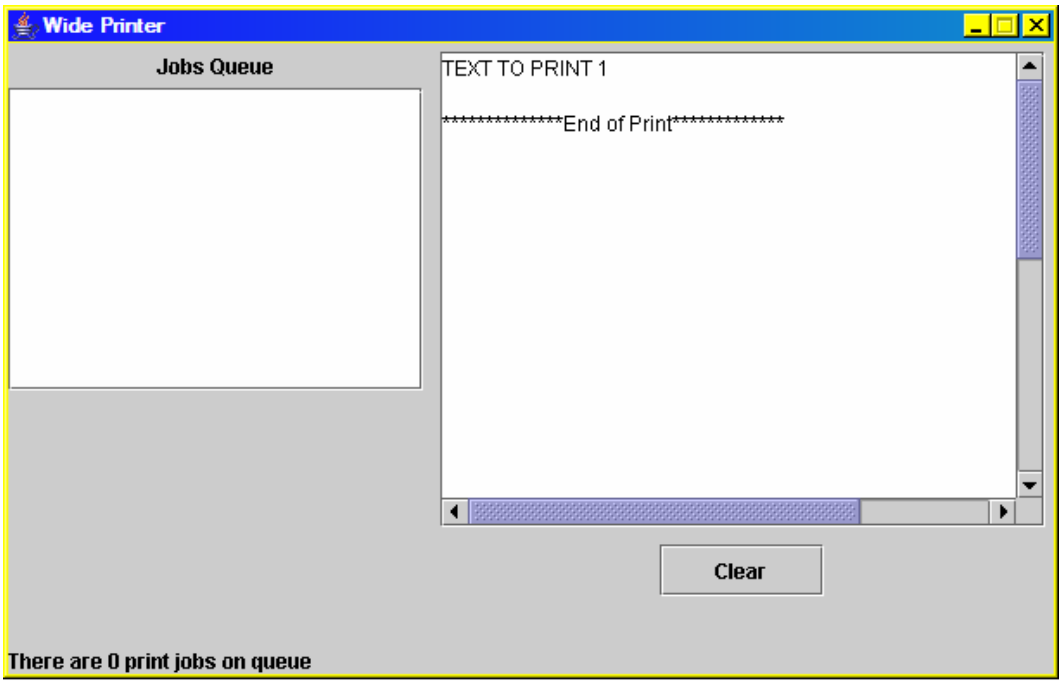

Figure 12: Job printed.

To exit the client application, just close the window. The printers will not close by clicking on the close button on the window. The process that opened the printers must be killed.

3.6. MULTIPLE GAMING SERVICE EXAMPLE

This example is provided in package `edu.rit.services.game` to show the use of the M2MI SD Middleware API. This example implements the scheme of having a third party object being advertised on the network, which will handle the communication with the client and then handout another service which is not advertised itself through the service repositories. This third object can be loaded on the network by itself or by another program.

The scheme then presents a class named `GameServer` which will not perform the game service itself, but implements the M2MI SD Protocol and handles the communication with the client. After following the protocol the `GameServer` gives the client a unihandle for the service implementing an interface requested by the client. The game service class, named `GamePlayerServer`, implements interface `GamePlayerServerInterface` and this is the only service interface used for this example.

The clients in turn also follow the M2MI SD Protocol, they communicate with the `GameServer` using the M2MI SD Framework and after acquiring the unihandle for the `GameServer`, the clients invoke the request for a service that implements interface `GamePlayerServerInterface` to start playing.

In order to follow the M2MI SD Protocol, it is necessary for the communicating server and clients to implement the `ServiceRequester` and `ServiceProvider` interfaces, respectively. Two interfaces were written to identify this specific application, which extend each of the server and client interfaces in the M2MI SD Protocol. These are the `GameServiceRequester` and the `GameServiceProvider` interfaces.

Once the communication between the game service and the client is established, after following the M2MI SD Protocol, the game starts automatically. The purpose of this example is to demonstrate the implementation of the protocol and its functionality;

therefore, the game is very simple, it exchanges numbers between the client and the server and ends after 10 moves from each side.

The client is encapsulated in class `GamePlayer`. This class can be run as a console program. This does not provide good demonstrating behavior, as everything will load automatically as soon as the program is started and if a service is not found the game will close. Using the frame view of the client, running class `GameJFrame`, the example is more demonstrative and useful.

Usage

To run this example, perform these steps, ensuring the M2MP Daemon has already been started, and the java `classpath` is correctly set:

1. Start the gaming service with the following command:

   ```
   java edu.rit.services.game.GameServer
   ```

   This command will start the `GameServer` on console, which will wait for clients to connect to it.
2. Start the client with the following command:

   ```
   java edu.rit.services.game.GameJFrame name
   ```

   where *name* is a name for the user of this game and it is optional. There can be as many clients as wanted in the system.

The client's GUI is displayed by the `GameJFrame` class and is shown below:

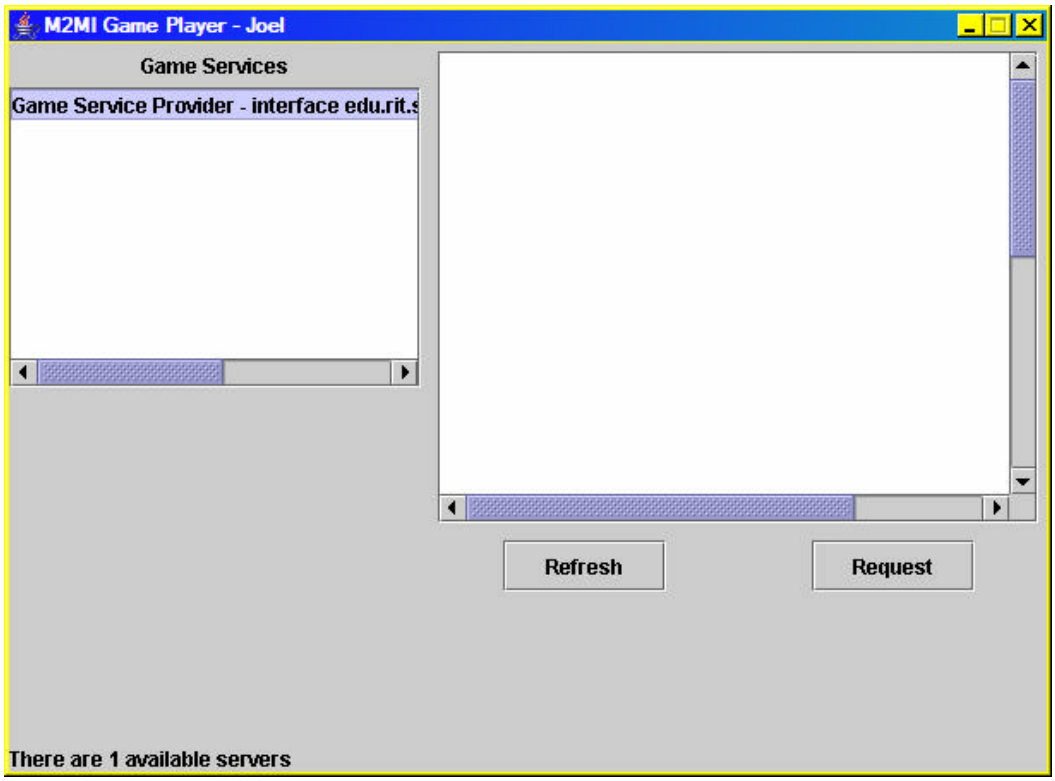

Figure 13: GUI layout for game client

On the left side, there is one scroll list for holding the available `GameServiceProvider` services. This is the interface the client is searching for. In this example, a confirmation scheme for the client using the `ServiceClient` interface was not adapted, the service is provided and then communication ends. The TextArea on the right side is for displaying the messages exchanged between the client application and the game server. The messages start when the client requests the service, they continue when the service has started to be provided and throughout the session until it ends. Below the TextArea there are two buttons tagged "Refresh" and "Request". The Refresh button fetches the network for the currently available services.

To start the communication process, select one of the service providers on the scroll list and click on the Request button. This will negotiate with the `GameServiceProvider` the use of a service implementing interface `GamePlayerServerInterface`. The provider will give the client a unihandle for the game object (class `GamePlayerServer`) implementing the requested interface and from that point the client will start playing automatically. The following two images show the interaction between the two services.

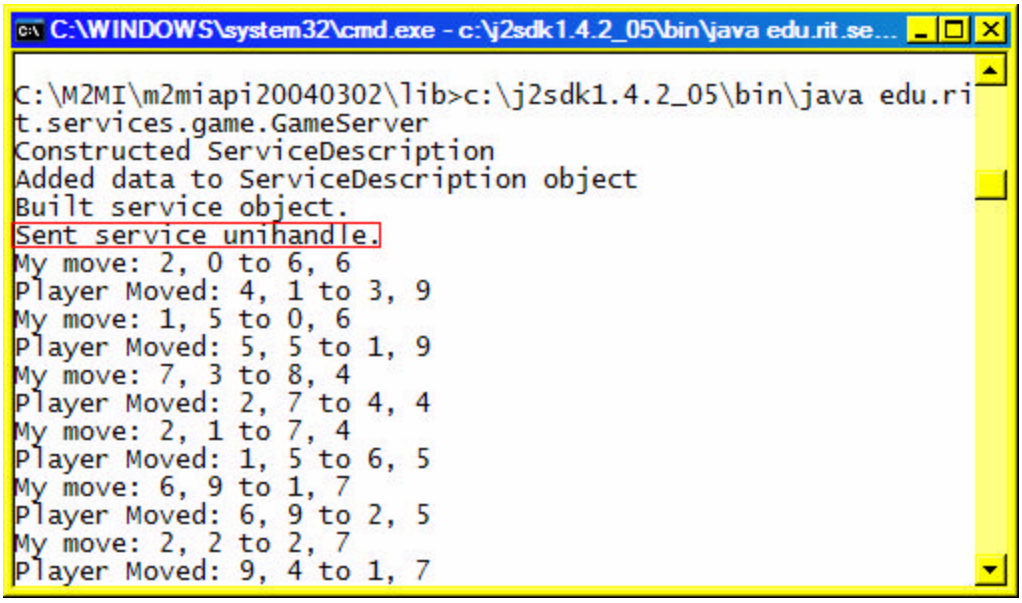

Figure 14: Game server output

The above image is from the server. The line squared in red corresponds to the moment the server receives the request and replies to the client. After this, the game is started by the client player and later finalized.

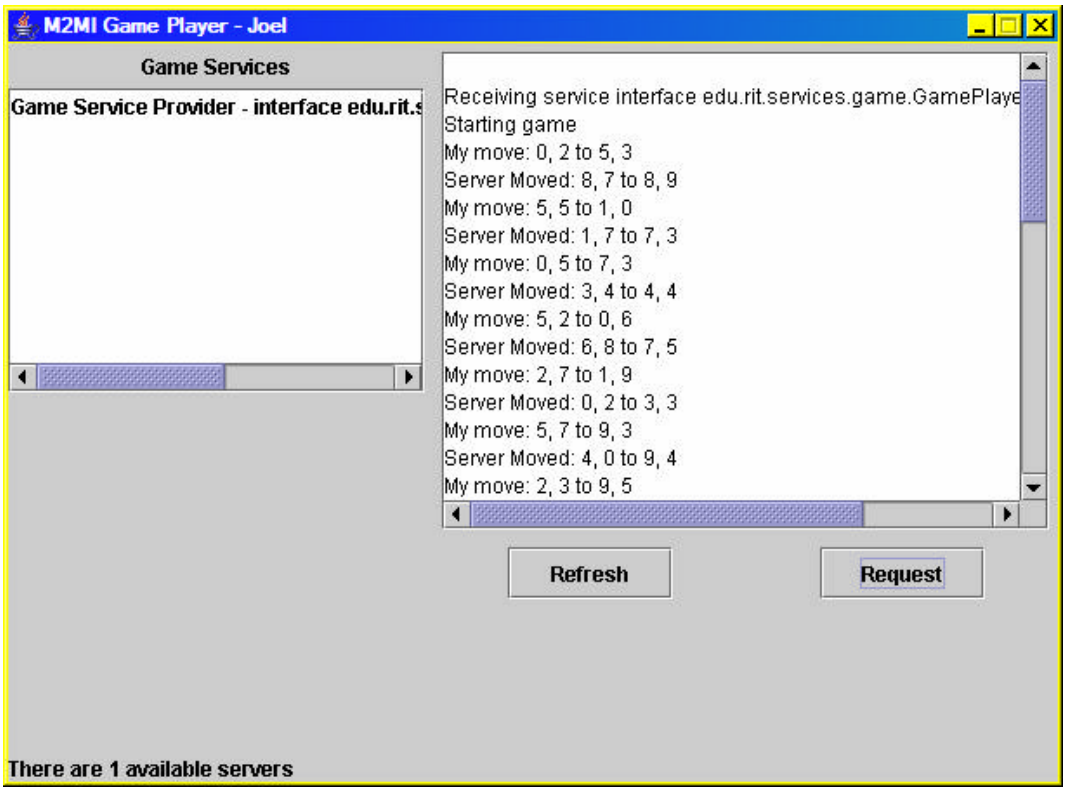

Figure 15: Game client output

This second image corresponds to the client side. The first line on the TextArea corresponds to the response from the provider, when it has sent the unihandle for the requested interface and the requester has received it. After this follows the development of the game, with subsequent moves from side to side until 10 moves are completed.

To close the client program, close the frame. To close the server program, kill the process.

To learn more about the M2MI SD Protocol, refer to section 3.1.

4. M2MI SD INTERNAL DESIGN

This section describes at a low level how the components of the middleware were developed and how they interact with each other. This section gives details on the implementation for facilitating the future work and development of the middleware. First, diagrams showing the internal interaction of the middleware components are shown and afterwards a description of the main three processing classes is given. These are the Services, ServiceRepository, and ServiceLeaseRenewer classes.

For information on the usage of the classes and the development of applications with this middleware please see section 3, M2MI SD Functionality, and Appendix A, API Documentation.

Process Diagram

The following diagram shows how the different components interact in one process.

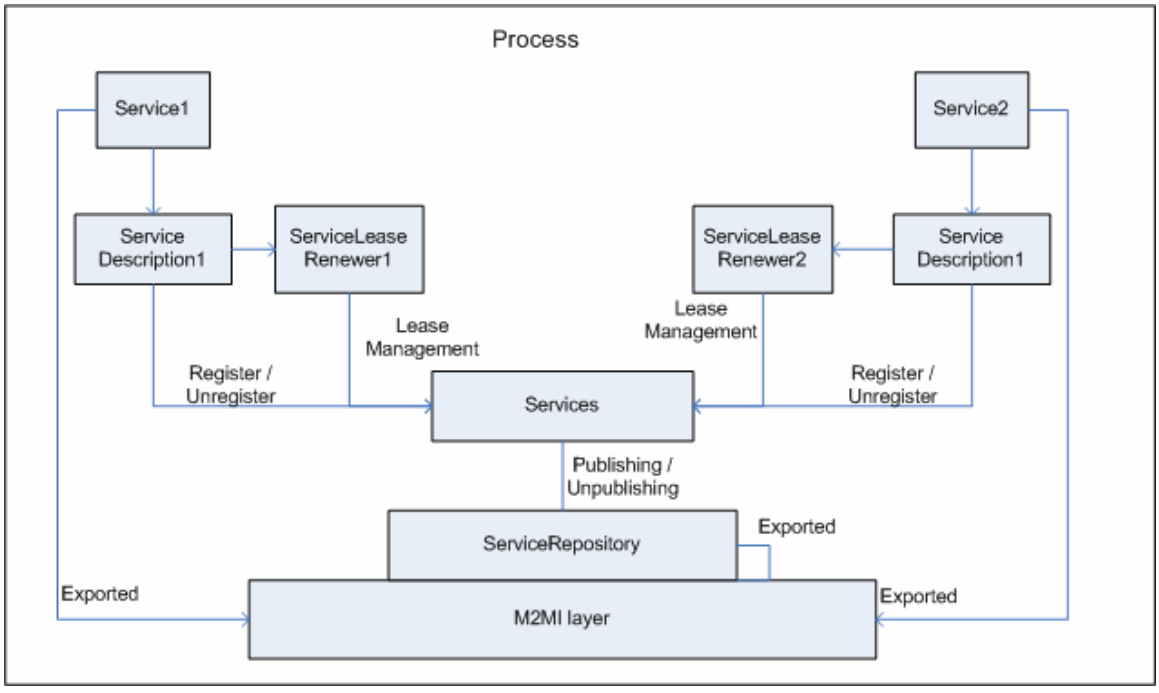

Figure 16: Process Diagram

The diagram shows that a service (Service1 or Service2 objects in the diagram) exported needs one ServiceDescription object. This ServiceDescription object is passed to the Services class for registration and unregistration, and the Services class handles the publishing and unpublishing with the ServiceRepository. The Services class also creates the ServiceRepository and exports it to the M2MI layer. One ServiceLeaseRenewer object is created for each published service and this object manages the leasing process with the Services class. The ServiceLeaseRenewer object has a thread which takes care of this automatically. The Services class also keeps two other threads, for handling the leasing services and for ordering the ServiceRepository to poll the rest of the repositories. Below there is a detailed description of the internals of the three main classes, Services, ServiceRepository, and ServiceLeaseRenewer.

Services Class

Internal Data

`private static ServiceRepository theRepository`: single static instance of a ServiceRepository object which will keep the services published. This instance is constructed when the `load()` method is first called or when a first service is registered with the method `registerService()`.

`private static HashSet publishedServicesSet`: stores the local services currently published with the ServiceRepository. The services are stored as ServiceDescription objects. This set is kept consistent in the Services class. If there is a new service or a service is removed from the network, using the `unregisterService()` method or because the lease of a service expired, then this hashset is updated accordingly, adding or removing the corresponding service. Only local services published using this class directly are stored in this set.

`private static Vector servicesLeases: java.util.Vector` which stores the information on the published services and their leases. The objects stored are from the nested class `ServiceLeaseInfo`. This nested class implements the `Comparable`

interface, and the vector is always sorted so the first element is the lease with the next timeout whenever the information is updated.

`private static LeasesThread theLeasesThread:` instance of nested class that takes care of the leasing processing, checking to see if there have been any expired leases and performing the required functions in case they have.

`private static PollerThread thePollerThread:` instance of nested class that takes care of making `theRepository` poll every minute or according to the setup of the `POLL_FREQUENCY` constant.

`private static final long POLL_FREQUENCY:` constant determining how the frequency (in milliseconds) the local ServiceRepository will poll the rest of the Repositories in the network. It is set to be one minute.

`private static final long MAX_LEASE_DURATION:` constant determining the maximum duration (in milliseconds) of a lease given to a service. It is set to be one minute.

Nested Classes

`private static class PollerThread extends Thread:` thread taking care of the polling. With the frequency established by the `POLL_FREQUENCY` constant, it will call the `pollRepositories()` method on the local ServiceRepository to trigger the polling process. The details on what happens when this method is called can be found ahead on this section on the part corresponding to the ServiceRepository internal class description.

`private static class LeasesThread extends Thread:` thread controlling the leasing of the services published. It checks the `servicesLeases` vector for expired leases and for services that have no lease. Afterwards, it removes those services from the local ServiceRepository and from the hashset and/or vector.

`private static class ServiceLeaseInfo implements Comparable:` used to store the information of the services leases, with their timeouts. The objects are stored in the `servicesLeases` vector and the vector is sorted to keep the most recent timeout first. This object's fields consist of a ServiceDescription for the service and a timeout in milliseconds corresponding to the time elapsed from January 1st 1970, according to the description of class `java.util.Date`. This time is compared to the current time to determine if the timeout has occurred.

Processing

Construction: the Services class cannot be instantiated. The constructor is private and all public methods were declared static.

Load: by calling the `load()` method it creates a new instance of the ServiceRepository and the PollerThread if they have not been created. To be called in case there needs to only listen and watch for services instead of publishing one.

Register a Service: calling the `registerService(ServiceDescription)` method will first perform a Load. The service will be passed to the local Repository calling the `newService(ServiceDescription)` method and then the service will be added to the `publishedServicesSet`. If the `LeasesThread` has not been created it will be created and started.

Unregister a Service: calling the `unregisterService(ServiceDescription)` method will make the local Repository remove the service with the `removeService (ServiceDescription)` method and then the service will be removed from the `publishedServicesSet`. Before the service is removed, a explicit call to its ServiceLeaseRenewer object must have been made to stop renewing the lease.

Lookup a Service: by calling any of the `lookupService` methods then the Services class makes one call to one of the methods in the ServiceRepository for getting an iterator to a set of services that match the lookup criteria.

Request a Service: by calling any the `requestService(ServiceDescription)` method a Unihandle to the service will be returned. This is done having the ServiceRepository return it with a call to its `getService(ServiceDescription)` method.

Add an event listener: a call to the `addEventListener(ServiceEventListener, ServiceEventFilter)` method returns a ServiceEventRegistration object containing the given filter and listener. They will be added to the ServiceRepository's set of listeners so when events are generated this listener is notified of the events passed by the given filter. The returned ServiceEventRegistration object is used later to remove the given listener and filter when event notifications are not needed anymore.

Remove an event listener: with the ServiceEventRegistration object obtained when a listener is added, the method `removeEventListener(ServiceEventRegistration)` is called to remove the listener. This will generate a call to `removeEventRegistration(ServiceEventRegistration)` in the ServiceRepository.

Request a lease: this action does not need to be performed by the user. The `requestLease` method is called by the ServiceLeaseRenewer. It will check that the given service has been published and that it does not have a lease in force or it will throw a ServiceNotPublishedException or a ServiceAlreadyLeasingException accordingly. After the check succeeds, the lease is added to the `servicesLeases` vector in a ServiceLeaseInfo object. The length of the lease is stored in milliseconds equivalent to the time in which it will expire, relative to January 1st 1970 (according to the description in `java.util.Date`). The method returns the duration of the lease, taken to be the least between the duration requested and `MAX_LEASE_DURATION`. The leases vector is sorted accordingly in ascending order for the expirations (next lease to expire in first element).

Renew a Lease: this action does not need to be performed by the user. The `renewLease` method is called by the ServiceLeaseRenewer. It will check that the given service is published and that it has a lease in force or it will throw a ServiceNotPublishedException

or a ServiceNotLeasingException accordingly. It will renew the lease updating the duration to the least value between the new requested duration and the `MAX_LEASE_DURATION`. The leases vector is sorted accordingly in ascending order for the expirations (next lease to expire in first element).

Cancel a Lease: this action does not need to be performed by the user. The `renewLease` method is called by the ServiceLeaseRenewer. It will check that the given service is published and that it has a lease in force or it will throw a ServiceNotPublishedException or a ServiceNotLeasingException accordingly. It removes the leasing information from the leases vector.

ServiceRepository Class

Internal Data

`private ServiceRepositoryInterface allRepositories`: reference to all the repositories in the network.

`private HashSet myLocalServices`: set storing the services published locally.

`private HashSet myRemoteServices`: set storing the services published remotely in other ServiceRepositories.

`private Unihandle myUnihandle`: unihandle to this repository.

`private Vector myServicesInterfacesArray`: array keeping information objects about the services and the names of the interfaces they implement, to facilitate searches. It stores objects from the nested class ServiceInterfaceCouple.

`private final long DEFAULT_TIME_TO_WAIT`: constant determining the time in milliseconds that the repository will wait for a response before timing out. This is used when a service is requested. The repository will check that the service is still available and wait for a response for 30 seconds.

`private Object oneLock`: object to lock on for synchronization. It is used when waiting for a response while checking that a service is still available.

`private boolean availableValue`: gives the value if a service is still available when requested. Used during a request synchronizing with the oneLock object. It is set to true when the service is still available.

`private HashSet myEventRegistrations`: stores the ServiceEventRegistration objects keeping record of the listeners and filters.

Nested Classes

`private class ServiceInterfaceCouple implements Comparable`: the objects of this type are used for storing a service with one name of one of the interfaces it implements. For every interface the service implements and that it provides a service for, one of these `ServiceInterfaceCouple` objects will be created and stored in the array `myServicesInterfacesArray`. This is made so the searching for a service based on the string name of the interface it implements is facilitated.

`private class EventThread extends Thread`: thread which reports events to the listeners. This is to avoid locking while other events and processing is needed. These threads are created just as each event occurs and needs to be reported, one thread per each event.

Processing

Construction: when the ServiceRepository is constructed, exports itself to the M2MI layer, gets an Omnihandle for all the repositories with interface ServiceRepositoryInterface and then gets a Unihandle for itself. Finally, the rest of the repositories are polled to obtain the services.

Add a local service: the Services class calls the `newService` method on the Repository to add a local service. If the service is already in it, nothing happens. Otherwise, all the names of the interfaces published are stored in the `myServicesInterfacesArray` vector. The service is reported to the rest of the repositories with a call to `addService` on `allRepositories`. The event is then reported to all listeners by a new `EventThread`.

Add a remote service: after a new local service is added in a repository, this in turn calls all other repositories with the omnihandle for adding the new remote service with the `addService` method. Because the repository performing the call will also receive the omnihandle's invocation, the method checks that the service is not in the local services set. After this, it checks that the service is not in the remote services set either and follows the same procedure as for local services. The names of all published interfaces are stored in `myServicesInterfacesArray`, and an event is notified to local listeners with a new `EventThread`.

Polling: in this process there are two methods involved, being those the `pollRepositories` and the `queryServices` methods. The `pollRepositories` method is called by the Services class. It cleans the remote services set and calls the `queryServices` method passing the local unihandle as an argument. The `queryServices` method, called on the omnihandle to all repositories, makes each repository call the `addService` method on the given unihandle with their local service.

Remove a service: the method in charge of this is the `removeService` method. It will check if the service is either local or remote, if it is not in either of the local or remote services sets then this method does nothing. After successfully identifying the service, it is removed from the corresponding set, then the `ServiceInterfaceCouple` objects stored for this service are removed from `myServicesInterfacesArray`. If the service is local then the rest of the repositories are notified of the removal and an event is generated for the local listeners.

Lookup services: the methods for looking up services are called by the Services class. The method checking for an interface by name will check on `myServicesInterfacesArray` for those elements matching the name of the interface (case insensitive). The methods performing searches by the interface type use the method `isAssignableFrom()` in a `java.lang.Class` object to determine if the interface or superinterface matches for the search being performed. All lookup methods return iterators to a set of ServiceDescription objects. The `listServices()` method returns an iterator to all services available, both local and remote.

Requesting a service: The Services class calls the `getService(ServiceDescription)` on the ServiceRepository to request a service. The ServiceRepository returns the Unihandle for the given service. This method will make sure the service is still available, locally or remotely, and will return null if this is not the case. A remote invocation from the local holder of this service is waited on for `DEFAULT_TIME_TO_WAIT` seconds; if a timeout occurs, it is understood this service is no longer available. The request is made by calling the `checkAvailability` method on all repositories, then the ServiceRepository waits for the invocation of method `serviceStillAvailable` called from a remote repository.

Events: when the available services set is changed, an instance of EventThread is created with the new list of services. This thread will make a ServiceEvent and scan through the set of registered listeners and filter this event. Only if the event passes the filter, then the listener will be notified of the event.

ServiceLeaseRenewer Class

Internal Data

`private WorkerThread myWorkerThread`: nested class instance which will do all the processing for managing a lease.

`private ServiceDescription myService`: service this ServiceLeaseRenewer instance is managing the lease for.

`private long myLeaseDuration`: duration of the current lease in force, in milliseconds. The default in construction is one minute.

`private long myRenewalInterval`: interval to renew for the lease. It is set to be half the time of `myLeaseDuration`, also in milliseconds.

`private final int LEASE_NOT_REQUESTED, LEASE_IN_FORCE, LEASE_NOT_IN_FORCE, LEASE_CANCELED`: constants determining the different states of the lease.

`private final int myState`: current state of the lease. It starts as `LEASE_NOT_REQUESTED`. After a lease has been requested and then granted then it is set to `LEASE_IN_FORCE`. If the lease is not granted when requested the state is set to `LEASE_NOT_IN_FORCE`. When a lease is in force and the call is made to be canceled, the state is set to `LEASE_CANCELED`. Before requesting a new lease, a call must be made to reset() to go back to the `LEASE_NOT_REQUESTED` state.

`private Object leaseLock`: object used for synchronizing `theWorkerThread` performing the renewals with the external calls canceling or stopping a renewal of a lease.

Nested Classes

`private class WorkerThread extends Thread`: thread which takes care of the lease renewing process. Once the lease is requested, this thread will continue to renew it automatically until the lease is canceled or denied. After the lease is canceled or denied, the ServiceLeaseRenewer object must be told explicitly to request a lease, the `WorkerThread` will not do so automatically.

Processes

Construction: in the construction only the ServiceDescription object it will renew for will be passed. The worker thread is created but not started.

Start renewing: with the call to the `startRenewing` method. The status of the lease must be `LEASE_NOT_REQUESTED` or an IllegalStateException will be thrown. The lease is requested with the Services class using the method `requestLease`. If the lease is granted, the worker thread will be started to start renewing and the state of the object will be set to `LEASE_IN_FORCE`.

Stop renewing: method `stopRenewing` takes care of this action. There must be a lease in force or an IllegalStateException will be thrown. It calls the `cancelLease` method on the Services class. The state is changed to `LEASE_CANCELED`.

Restarting the renewal: after a lease has been canceled, the worker thread will not attempt to request a lease again or restart the renewal. Call the `reset` method and then request the lease again. The `reset` method will clear the status and set it to `LEASE_NOT_REQUESTED` so the request is possible again. If the `reset` method is called when a lease is in force, an IllegalStateException will be thrown.

The following two sections compare the M2MI SD middleware to those Service Discovery technologies implemented in Jini and Lime.

5. 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].

5.1. SYSTEM REQUIREMENTS

Hosts that wish to participate in a distributed Jini network must have:

- A functioning JVM with access to all packages needed to run Jini software.
- A properly configured network protocol stack. [6]

These requirements are similar to those for M2MI, which also needs devices to have access to a JVM and a network. However, M2MI is targeted towards taking advantage of the network's broadcast capabilities; it will not use any routing protocols. The M2MI devices may not have addresses, in fact, the implementation does not use addresses of any kind; method invocations are broadcasted on the network [5]. Jini, meanwhile, must rely on the correct implementation of the network, making use of its addresses and routing protocols. Based on the currently used network technologies, taking wireless Ethernet for the example, this translates into being able to have a larger network with Jini, but a potentially faster communication between M2MI devices due to the one-hop distances of the hosts in it.

The software requirements for both are basically to have the APIs of the respective services. For Jini, all its packages will take approximately 5MB without the source files,

which take an additional 10MB. M2MI's API including the SD packages and the source files, will take 4.7MB approximately. M2MI is more lightweight than Jini.

5.2. 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.

The next step is to *Join* the Lookup Service. This consists of forwarding the proxy class developed to the Lookup Service in order to make the service available for the rest of the network. In case the host needs to get a service, it will discover the Lookup Service and

then perform a *Lookup*. This lookup is based on the Java type, this is, the implemented interface [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. M2MI is also a technology for developing distributed applications in serverless ad hoc networks. Specifically into the Service Discovery Framework design and architecture, M2MI SD's analogue to Jini's Lookup Service is the ServiceRepository because it also holds the network's services information. The difference is that every device or host must have a ServiceRepository to participate in the Service Discovery mechanisms. In addition, M2MI SD requires less manual configuration, just by loading the ServiceRepository locally a host will have information on all available services. The local ServiceRepository gathers information from the other repositories in the network, receiving information on the remote services. Local services are published using the local ServiceRepository. This procedure is repeated throughout the network, having in the end all repositories with the same information about the available services.

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.

M2MI SD specifies as the SD Protocol, a procedure to take place after the acquisition of the Unihandle. The services returned are identified as service providers, which will carry out the protocol in communication with service requesters. The difference with Jini is that Jini's protocols define all the communication needed to start providing services, whereas M2MI negotiates only after the Unihandle is returned. Nevertheless, it can be the case that the service itself is returned in the unihandle and the client program will only have to start calling methods on it, similar to what happens in Jini.

Jini services have the ability to define attributes, allowing clients to perform specific lookups based on interfaces and attributes. M2MI SD's most similar feature is the *description* field in the ServiceDescription object. In this case, Jini provides an advantage because many attributes can be set and lookups can be performed based on many attributes. Lookups on M2MI's ServiceDescription *description* field are based on a substring search which may not be as specific as that of Jini's.

An important aspect of the Lookup Service and the ServiceRepository is the consistency throughout the network. This is achieved in both cases using the leasing scheme. Services not leasing will be withdrawn from the information sets kept by the Lookup Service and ServiceRepository objects. ServiceRepository objects will only control leases for the local services, and this is achieved through the Services class.

5.3. 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. The client can start calling methods here.

6. 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 6.2. and 6.3.

6.1. SYSTEM REQUIREMENTS

Hosts that wish to implement the Lime API must have:

- A functioning JVM with access to all packages in the Lime API.

Lime is fully implemented in Java, with support for version 1.1 and higher. Communication is handled entirely at the socket level - no support for RMI or other additional communication mechanisms is needed or exploited in Lime [3].

As seen on the previous chapter, M2MI also needs devices to have access to a JVM and a network, taking advantage of the network's broadcasting capabilities and without the use of any routing. The M2MI devices may not have addresses, in fact, the implementation does not use addresses of any kind; method invocations are broadcasted on the network [5]. For Lime, a Lime server must be started, set to listen on a port, and messages will be unicasted between two hosts, distant from M2MI's scheme in which everything is broadcasted among hosts.

The software requirements for both are basically to have the APIs of the respective services. The lime package is about 5,000 non-commented source statements, for about 100 Kbyte of jar file. The companion lighTS package provides a lightweight tuple space implementation plus an adapter layer integrating other tuple space engines, for an additional 20 KB [3]. M2MI's API including the SD packages and the source files, will take 4.7MB approximately. Lime is more lightweight than M2MI.

6.2. 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 (`out`), one to get a copy of a tuple in the tuple space that matches a given template (`rd`), and a third that in addition to getting a copy of a matching tuple withdraws it from the tuple space (`in`). 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].

The data structures used in Lime differ intrinsically from the structures used in M2MI SD. However, the concept is very similar. A ServiceRepository keeps information on the available services on the network, all in the same form of ServiceDescription objects that could be analogized with tuples. Tuples in Lime are shared by all hosts, tuples are distributed in the network but used by any host or agent that needs it as if it was in one place.

6.3. 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.

7. 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.

The API has not been tested in high-performance, heavy-loads environments. Simulations for deployment in large networks with a large number of services and activities, which could be the case in a crowded conference room, were not performed. It will be helpful to find out about the behavior of the API's responsiveness to such environments.

The services cannot be federated into groups. The middleware will detect and interact with all services available, even though developed applications can choose to see only a subset of the services through interfaces lookups. This could prove useful in a network with a large number of devices and services.

The middleware can be further improved by making the service objects not needed to reside in the same process as the ServiceRepository. This will make the leasing functionality more useful and imperative to use. The leasing functionality must be used in this middleware, but it is not really needed under the architecture of the service and the ServiceRepository being in the same process.

The ServiceRepository objects can be developed to broadcast their status periodically, instead of having them poll the other repositories and then have the rest of the repositories report their services. This translates in less traffic on the network.

8. 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 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.

9. REFERENCES

[1] The Anhinga Project. http://www.cs.rit.edu/anhinga

[2] Alan Kaminsky. Computer Science Course Library. http://www.cs.rit.edu/~ark/cscl.shtml

[3] G.P. Picco, A.L. Murphy, and G.-C. Roman. Developing mobile computing applications with LIME. In *Proc. of the 22nd Int. Conf. on Software Engineering*, pages 766-769, 2000.

[4] G.P. Picco, A.L. Murphy, and G.-C. Roman. Lime: Linda Meets Mobility. In *Proc. of the 21st Int. Conf. on Software Engineering*, pages 368-377, May 1999.

[5] Hans-Peter Bischof and Alan Kaminsky. Many-to-Many Invocation: A new framework for building collaborative applications in ad hoc networks. *CSCW 2002 Workshop on Ad Hoc Communication and Collaboration in Ubiquitous Computing Environments*, New Orleans, Louisiana, USA, November 2002.

[6] Ken Arnold, Bryan O'Sullivan, Robert W. Scheifler, Jim Wlado, Ann Wollrath. *The Jini Specification.* AddisonWesley Longman, Inc. 1999.

APPENDIX A. API DOCUMENTATION

The M2MI Service Discovery middleware API is encapsulated in package edu.rit.services. This chapter gives a full documentation of all interfaces, classes, and exceptions. For interaction of these elements, their usage and how to develop applications, see chapter 5.

A.1. INTERFACES

ServiceProvider Interface

Specifies the interface for a service provider. This service provider may be the service itself, or an intermediate object which handles the communication between the client and the service. The `ServiceProvider` interface is one of two fundamental parts of the M2MI Service Discovery protocol. The other part is the `ServiceRequester` interface, which must be implemented by a client communicating with the provider. These two interfaces must interact as their methods must call each other during the execution of the M2MI SD protocol. The protocol is discussed in section 3.1.

Methods

- serviceRequest

public void **serviceRequest**(java.lang.Object payload,
edu.rit.m2mi.Unihandle clientUnihandle)

Requests a service on the server for a default interface.

Parameters:

payload - A payload object for the request.

clientUnihandle - The unihandle of the client requesting the service.

- serviceRequest

public void **serviceRequest**(edu.rit.m2mi.Unihandle clientUnihandle)

Requests a service on the server for a default interface.

Parameters:

clientUnihandle - The unihandle of the client requesting the service.

- serviceRequest

public void **serviceRequest**(java.lang.Class requestedInterface,
edu.rit.m2mi.Unihandle clientUnihandle)

Requests a service on the server for the given *requestedInterface*.

Parameters:

requestedInterface - The main requested interface for this service.

clientUnihandle - The unihandle of the client requesting the service.

- serviceRequest

public void **serviceRequest**(java.lang.Class requestedInterface,
java.lang.Class[] alternativeInterfaces, edu.rit.m2mi.Unihandle clientUnihandle)

Requests a service on the server for the given *requestedInterface* or for one of the *alternativeInterfaces*.

Parameters:

requestedInterface - The main requested interface for this service.

alternativeInterfaces - An array of interfaces which the client could also receive for the service.

clientUnihandle - The unihandle of the client requesting the service.

- serviceRequest

public void **serviceRequest**(java.lang.Object payload,
java.lang.Class requestedInterface, java.lang.Class[] alternativeInterfaces,
edu.rit.m2mi.Unihandle clientUnihandle)

Requests a service on the server for the given *requestedInterface* or for one of the *alternativeInterfaces*.

Parameters:

payload - A payload object for the request.

requestedInterface - The main requested interface for this service.

alternativeInterfaces - An array of interfaces which the client could also receive for the service.

clientUnihandle - The unihandle of the client requesting the service.

- serviceRequest

public void **serviceRequest**(java.lang.Object payload,
java.lang.Class requestedInterface, edu.rit.m2mi.Unihandle clientUnihandle)
Requests a service on the server for the given *requestedInterface*.

Parameters:

requestedInterface - The requested interface for this service.

payload - A payload object for the request.

clientUnihandle - The unihandle of the client requesting the service.

ServiceRequester Interface

public interface **ServiceRequester**

Specifies the interface for a service requester. The `ServiceRequester` interface is one of two fundamental parts of the M2MI Service Discovery protocol. The other part is the `ServiceProvider` interface, which must be implemented by a server receiving a request from the client. These two interfaces must interact as their methods must call each other during the execution of the M2MI SD protocol. The protocol is discussed in section 3.1.

Methods

- receiveService

public void **receiveService**(java.lang.Class requestedInterface,
java.lang.Object payload, edu.rit.m2mi.Unihandle serviceUnihandle)
Receives a service from a ServiceProvider for a given *requestedInterface*.

Parameters:

requestedInterface - The interface this service provider is offering to provide a service with.

payload - A payload object for receiving a service.

serviceUnihandle - The unihandle for the service implementing the given interface.

ServiceClient Interface

public interface **ServiceClient**

Interface ServiceClient defines an M2MI client which will connect to an M2MI service. This interface has methods which allow a client to receive results for its invocations and service requests. The implementation of the results information on the host providing a service is totally optional, it is not mandatory to implement this feature on servers using the Service Discovery Framework and it is also an optional interface to implement on client hosts. This interface is provided solely to facilitate and standardize this procedure. This interface does not play any role in the M2MI Service Discovery Protocol.

Methods

- receiveServiceJobID

public void **receiveServiceJobID**(ServiceJobID theJobID)

Gives the client implementing this interface a ServiceJobID a request made to a service.

Parameters:

theJobID - ServiceJobID of one request in a given service provider.

- serviceResult

public void **serviceResult**(ServiceJobID theJobID, int result)

Gives the client implementing this interface the result of the service request specified by the given ServiceJobID.

Parameters:

theJobID - ServiceJobID of one request in a given service provider.

result - The result of the invocation, which is dependent on the actual implementation of the interface.

- serviceResult

public void **serviceResult**(ServiceJobID theJobID, int result,
java.lang.Exception resultException)

Gives the client implementing this interface the result with the given Exception of the service request specified by the given ServiceJobID.

Parameters:

theJobID - ServiceJobID of one request in a given service provider.

result - The result of the invocation, which is dependent on the actual implementation of the interface.

resultException - An Exception which occurred during the execution of this request.

- serviceResult

public void **serviceResult**(ServiceJobID theJobID, int result,
java.lang.String message)

Gives the client implementing this interface the result with the given message from the service request specified by the given ServiceJobID.

Parameters:

theJobID - ServiceJobID of one request in a given service provider.

result - The result of the invocation, which is dependent on the actual implementation of the interface.

message - Message from the service provider about the result of this request.

- serviceResult

public void **serviceResult**(ServiceJobID theJobID, int result,
java.lang.Exception resultException, java.lang.String message)

Gives the client implementing this interface the result with the given Exception of the service request specified by the given ServiceJobID.

Parameters:

theJobID - ServiceJobID of one request in a given service provider.

result - The result of the invocation, which is dependent on the actual implementation of the interface.

resultException - An Exception which occurred during the execution of this request.

message - Message from the service provider about the result of this request.

ServiceRepositoryInterface Interface

public interface **ServiceRepositoryInterface**

The ServiceRepositoryInterface defines the interface for a `ServiceRepository` class. This is the interface a ServiceRepository will be exported with to the M2MI layer. To develop applications using the M2MI Service Discovery framework, use the static methods in the `Services` class. Do not use the `ServiceRepository` class directly.

Methods

- addService

public void **addService**(ServiceDescription service)

Adds a remote service to this Repository.

Parameters:

service - The service to be added.

- removeService

public void **removeService**(ServiceDescription service)

Removes a service from the repository.

Parameters:

service - The service to be removed.

- queryServices

public void **queryServices**(edu.rit.m2mi.Unihandle callbackUnihandle)

Gives the repository of the argument unihandle all the local services this repository has.

Parameters:

callbackUnihandle - The unihandle of the Repository which called this method.

- checkAvailability

public void **checkAvailability**(ServiceDescription service,
edu.rit.m2mi.Unihandle callbackUnihandle)

Asks this repository if the given service is still available. For the implementation: This repository will respond by calling the serviceStillAvailable method on the callbackUnihandle, which belongs to the ServiceRepository originating this call. If the service is not in this Repository, then the Repository will do nothing.

Parameters:

service - Service to check for availability on this repository.

callbackUnihandle - The Unihandle to the remote ServiceRepository performing this call.

- serviceStillAvailable

public void **serviceStillAvailable**()

Notifies this Repository the service it asked for is still available. For the implementation: A repository can only ask for one service to be available at a time and this is determined in its call to checkAvailability, which generates this call to serviceStillAvailable as a response.

ServiceEventListener Interface

public interface **ServiceEventListener**

Identifies an object that can receive ServiceEvent objects.

Methods

- receive

public void **receive**(ServiceEvent event)

Receives the given event.

Parameters:

event - The event sent from the local ServiceRepository.

ServiceEventFilter Interface

public interface **ServiceEventFilter**

Identifies an object which filters ServiceEvent objects to be reported to a ServiceEventListener.

Methods

- filter

public boolean **filter**(ServiceEvent event)

Filters the given event. If the filter is passed, then the event will be reported to the corresponding ServiceEventListener which this ServiceEventFilter belongs to.

Parameters:

event - The event to send to a listener object.

Returns:

true if the event passes the filter, meaning it must be reported.

A.2. CLASSES

Services Class

public class **Services**

extends java.lang.Object

The Services class provides the main functions for using the M2MI Service Discovery Framework API (the **API**). Using this class, the developer will only have to worry about

developing the service itself, and then it will be easy to use this API and mainly this class to deploy and publish such service on the network. The Services class is in charge of publishing, maintaining, and unpublishing developed services. The Services class works with `ServiceDescription` objects to perform these functions. Hence, when using the Services class only `ServiceDescription` objects must be provided. The `ServiceDescription` objects encapsulate all the information needed for publishing one service in the network, including the most important and fundamental ones: the service's `Unihandle` and the implemented interface. For each published service a `ServiceDescription` object must be created to deploy it on the network using this API.

Developed services can be published on the network on their own, advertising themselves in a `ServiceDescription` object, ready to give clients their `Unihandle` for use or an intermediate class handling the communication can be used instead. This intermediate class may be published on the network using the Services class, communicating with clients and later providing a `Unihandle` for the service itself. Section 3 gives complete descriptions on how to use this API, the M2MI Service Discovery Protocol, and sample applications.

Methods

- load

public static void **load**()

Initializes the ServiceRepository. This single call will make it load all services available on the network.

- registerService

public static void **registerService**(ServiceDescription service)

Registers a service locally and publishes it on the network. If the service has been registered already then this method will do nothing. If the ServiceRepository has not been initialized, this method will do this automatically. After calling this method, this service must be setup for leasing using the ServiceLeaseRenewer class. If the service does not start a lease, it will be removed automatically within one minute.

Parameters:

service - The service to be added.

Throws:

java.lang.NullPointerException - If the service is null.

- unregisterService

public static void **unregisterService**(ServiceDescription service)

Removes the given service from the network. If the service does not exist then this method does nothing. The lease for this service must be canceled prior to removing it.

Parameters:

service - The service to be removed.

Throws:

java.lang.NullPointerException - (unchecked) if the service is null.

java.lang.IllegalStateException - (unchecked) if the ServiceRepository has not been initialized.

- lookupService

public static java.util.Iterator **lookupService**(java.lang.String interfaceName)

Searches for a service implementing the given interface. It will return an iterator to ServiceDescription objects containing a service that implements the given interface. The search will return a ServiceDescription object even if the given interface is implemented as one of the superinterfaces of the original interface included in the ServiceDescription object.

Parameters:

interfaceName - Name of the interface to look for.

Returns:

An Iterator to ServiceDescription objects implementing the given interface.

Throws:

java.lang.NullPointerException - (unchecked) if the interface name is null.

java.lang.IllegalStateException - (unchecked) if the ServiceRepository has not been initialized.

- lookupService

public static java.util.Iterator **lookupService**(java.lang.Class theInterface)

Searches for a service implementing the given interface. It will return an iterator to ServiceDescription objects containing a service that implements the given interface. The search will return a ServiceDescription object even if the given interface is implemented as one of the superinterfaces of the original interface included in the ServiceDescription object.

Parameters:

theInterface - Interface to look for.

Returns:

An Iterator to ServiceDescription objects implementing the given interface.

Throws:

java.lang.NullPointerException - (unchecked) if the Interface is null.

java.lang.IllegalStateException - (unchecked) if the ServiceRepository has not been initialized.

- lookupServiceByDesc

public static java.util.Iterator **lookupServiceByDesc**(java.lang.String description)

Searches for a service whose description contains a substring matching the given search string. It will return an iterator to ServiceDescription objects containing a service with a description that contains the substring given in the passed argument. The search is case insensitive.

Parameters:

description - String to look for in the service description.

Returns:

An Iterator to ServiceDescription objects implementing the interface searched for.

Throws:

java.lang.NullPointerException - (unchecked) if the description is null.

java.lang.IllegalStateException - (unchecked) if the ServiceRepository has not been initialized.

- lookupService

public static java.util.Iterator **lookupService**(java.lang.Class theInterface, java.lang.Class[] theAlternativeInterfaces)

Searches for a service implementing the given interface or until it finds matches for any of the alternative interfaces. It will return an iterator to ServiceDescription objects containing a service that implements the given interface. If no matches are found, it will search for the alternative interfaces, until it finds the first interface that returns an iterator with results. The search will return a ServiceDescription object even if the given interface is implemented as one of the superinterfaces of the original interface included in the ServiceDescription object. The alternative interfaces array is allowed to be null, in this case, the search will only include the main interface supplied as first argument. This first argument cannot be null.

Parameters:

theInterface - Interface to look for.

theAlternativeInterfaces - Array of interfaces to look for in case the first interface is not found.

Returns:

An Iterator to ServiceDescription objects implementing the interface searched for.

Throws:

java.lang.NullPointerException - (unchecked) if the main interface is null.

java.lang.IllegalStateException - (unchecked) if the ServiceRepository has not been initialized.

- getAllServices

public static java.util.Iterator **getAllServices**()

Lists all local and remote services available.

Returns:

An iterator to all ServiceDescription objects in the repository.

Throws:

java.lang.IllegalStateException - (unchecked) if the ServiceRepository has not been initialized.

- requestService

public static edu.rit.m2mi.Unihandle **requestService**(ServiceDescription service)

Returns the Unihandle for the given service. When requested, the Repository will make sure this service is still available, locally or remotely, and will return null if this is not the case. A remote invocation from the local holder of this service is waited on for 30 seconds, if a timeout occurs, it is understood this service is no longer available.

Parameters:

service - The service we need a unihandle for.

Returns:

The unihandle for the ServiceDescription object, or null if the service is no longer available.

Throws:

java.lang.NullPointerException - (unchecked) if the service is null.

java.lang.IllegalStateException - (unchecked) if the ServiceRepository has not been initialized.

- addEventListener

public static ServiceEventRegistration

addEventListener(ServiceEventListener listener, ServiceEventFilter eventFilter)

Adds the given listener with the given filter so it will receive the events originated by the ServiceRepository. When services are added or removed, the ServiceRepository will originate events and pass them to the registered listeners according to the filter they specify.

Parameters:

listener - The event listener.

eventFilter - The filter for passing the events. If it is null, this listener will receive all events originated.

Returns:

The ServiceEventRegistration object originated, containing the listener and the filter.

Throws:

java.lang.NullPointerException - (unchecked) if the listener is null.

- removeEventListener

public static void **removeEventListener**(ServiceEventRegistration eventReg)

Removes the given ServiceEventRegistration object from the ServiceRepository. After the removal, the listener contained in the given object will not receive any events from the ServiceRepository anymore.

Parameters:

eventReg - the ServiceEventRegistration object to be removed.

Throws:

java.lang.NullPointerException - (unchecked) if the object is null.

- requestLease

public static long **requestLease**(ServiceDescription service,

long theRequestedDuration)

Requests a lease for the given service and duration. The duration must be specified in milliseconds. The maximum lease duration is 60 seconds. If a lease is requested for more than the maximum lease, the maximum lease duration will be granted. After registering a service, it must start leasing. Do not call the leasing methods of this class directly. Instead, use class ServiceLeaseRenewer for this purpose.

Parameters:

service - The service requesting a lease.

theRequestedDuration - duration of lease requested.

Returns:

the duration granted for this lease.

Throws:

java.lang.NullPointerException - (unchecked) if the service is null.

java.lang.IllegalStateException - (unchecked) if the ServiceRepository has not been initialized.

ServiceNotPublishedException - (unchecked) if the lease is requested for a service which is not registered.

java.lang.IllegalArgumentException - (unchecked) if the requested duration is 0 or less than 0.

ServiceAlreadyLeasingException - (unchecked) if the service is already leasing.

- renewLease

public static long **renewLease**(ServiceDescription service,
long theRequestedDuration)

Renews a lease currently in force for the given service and duration.

Parameters:

service - The service renewing a lease.

theRequestedDuration - duration of lease requested.

Returns:

the duration granted for this lease.

Throws:

java.lang.NullPointerException - (unchecked) if the service is null.

java.lang.IllegalStateException - (unchecked) if the ServiceRepository has not been initialized.

ServiceNotPublishedException - (unchecked) if the service is not registered.

java.lang.IllegalArgumentException - (unchecked) if the requested duration is 0 or less than 0.

ServiceNotLeasingException - (unchecked) if the service does not have a lease in force.

- cancelLease

public static void **cancelLease**(ServiceDescription service)

Cancels a lease for the given service.

Parameters:

service - The service cancelling a lease.

Throws:

java.lang.NullPointerException - (unchecked) if the service is null.

java.lang.IllegalStateException - (unchecked) if the ServiceRepository has not been initialized.

ServiceNotPublishedException - (unchecked) if the service is not registered.

ServiceNotLeasingException - (unchecked) if the service does not have a lease in force.

ServiceRepository Class

public class **ServiceRepository**
extends java.lang.Object
implements ServiceRepositoryInterface

The ServiceRepository class is the main class in charge of the exchange of information for the Service Discovery framework. This class is not to be used directly by applications, instead use the `Services` class static methods to develop applications.

Constructor

- ServiceRepository

public **ServiceRepository**()

Constructs a new ServiceRepository. It will automatically export this Repository to the M2MI layer, and get information of available services from the rest of the Repositories.

Methods

- newService

public void **newService**(ServiceDescription service)

Adds a new local service to the repository for publishing it to the network. It will automatically retrieve all the interfaces for the given service, and add it to the repositories currently in the proximal network using the omnihandle. If the service has already been added, the Repository will do nothing.

Parameters:

service - The service to be added to the local repository.

Throws:

java.lang.NullPointerException - (unchecked) if the service is null.

- pollRepositories

public void **pollRepositories**()

This method will request all the repositories to report their services. To be used for updating the contents of the local repository with the remote services published in other remote repositories.

- addService

public void **addService**(ServiceDescription service)

Adds a remote service to this repository. If the service is identified to be local or already existing, then the repository does nothing.

Specified by:

addService in interface ServiceRepositoryInterface

Parameters:

service - The service to be added.

- removeService

public void **removeService**(ServiceDescription service)

Removes a service from the repository. If the service does not exist then the repository does nothing.

Specified by:

removeService in interface ServiceRepositoryInterface

Parameters:

service - Service to be removed.

- queryServices

public void **queryServices**(edu.rit.m2mi.Unihandle callbackUnihandle)

Gives the repository of the argument unihandle all the local services this repository has. This method is called on all repositories during the polling to update the remote services information in a repository.

Specified by:

queryServices in interface ServiceRepositoryInterface

Parameters:

callbackUnihandle - The unihandle of the Repository which called this method.

- listServices

public java.util.Iterator **listServices**()

Lists all local and remote services in this Repository.

Returns:

An iterator to all ServiceDescription objects in the repository.

- lookupServiceInterface

public java.util.Iterator **lookupServiceInterface**(java.lang.String interfaceName)

Searches for a service implementing the given interface. It will return an iterator to ServiceDescription objects containing a service that implements the given interface. The search will return a ServiceDescription object even if the given interface is implemented as one of the superinterfaces of the original interface included in the ServiceDescription object.

Parameters:

interfaceName - Name of the interface to look for.

Returns:

An Iterator to ServiceDescription objects implementing the given interface.

- lookupServiceInterface

public java.util.Iterator **lookupServiceInterface**(java.lang.Class theInterface)

Searches for a service implementing the given interface. It will return an iterator to ServiceDescription objects containing a service that implements the given interface. The search will return a ServiceDescription object even if the given interface is

implemented as one of the superinterfaces of the original interface included in the ServiceDescription object.

Parameters:

theInterface - Interface to look for.

Returns:

An Iterator to ServiceDescription objects implementing the given interface.

- lookupServiceInterface

public java.util.Iterator **lookupServiceInterface**(java.lang.Class theInterface, java.lang.Class[] theAlternativeInterfaces)

Searches for a service implementing the given interface or until it finds matches for any of the alternative interfaces. It will return an iterator to ServiceDescription objects containing a service that implements the given interface. If no matches are found, it will search for the alternative interfaces, until it finds the first interface that returns an iterator with results. The search will return a ServiceDescription object even if the given interface is implemented as one of the superinterfaces of the original interface included in the ServiceDescription object.

Parameters:

theInterface - Interface to look for.

theAlternativeInterfaces - Array of interfaces to look for in case the first interface is not found.

Returns:

An Iterator to ServiceDescription objects implementing the interface searched for.

- lookupServiceInterfaceByDesc

public java.util.Iterator **lookupServiceInterfaceByDesc**(java.lang.String description)

Searches for a service whose description contains a substring matching the given search string. It will return an iterator to ServiceDescription objects containing a service with a description that contains the substring given in the passed argument. The search is case insensitive.

Parameters:

description - String to look for in the service description.

Returns:

An Iterator to ServiceDescription objects implementing the interface searched for.

- getService

public edu.rit.m2mi.Unihandle **getService**(ServiceDescription service)

Returns the Unihandle for the given service. When requested, the Repository will make sure this service is still available, locally or remotely, and will return null if this is not the case. A remote invocation from the local holder of this service is waited on for 30 seconds, if a timeout occurs, it is understood this service is no longer available.

Parameters:

service - The service we need a unihandle for.

Returns:

The unihandle for the ServiceDescription object, or null if the service is no longer available.

- checkAvailability

public void **checkAvailability**(ServiceDescription service,
edu.rit.m2mi.Unihandle callbackUnihandle)

Asks this repository if the given service is still available.

Specified by:

checkAvailability in interface ServiceRepositoryInterface

Parameters:

service - Service to check for availability on this repository.

callbackUnihandle - The Unihandle to the remote ServiceRepository performing this call.

- serviceStillAvailable

public void **serviceStillAvailable**()

Notifies this Repository the service it asked for is still available. A repository can only ask for one service to be available at a time and this is determined in its call to checkAvailability, which generates this call to serviceStillAvailable as a response.

Specified by:

serviceStillAvailable in interface ServiceRepositoryInterface

- addEventRegistration

public boolean **addEventRegistration**(ServiceEventRegistration eventRegistration)

Adds the given event registration object to the repository. The repository will then report the corresponding events to the listener in this event registration object.

Parameters:

eventRegistration - the registration object to add

Returns:

True if the eventRegistration object is added. False if the object is already in the set of objects receiving events.

- removeEventRegistration

public boolean **removeEventRegistration**(
ServiceEventRegistration eventRegistration)

Removes the given event registration object from the repository. The corresponding listener in this event registration object will not receive any more events according to the filter also defined within it.

Parameters:

eventRegistration - the registration object to remove

Returns:

True if the eventRegistration object was removed. False if the object was not found in the registration objects set.

ServiceDescription Class

public class **ServiceDescription**
extends java.lang.Object

implements java.lang.Comparable, java.io.Serializable

Provides information on one service advertised in the network. A ServiceDescription object is created for each service advertised and it must be published in the local ServiceRepository so the rest of the network can see and request the corresponding service. The service actions for publishing and unpublishing a ServiceDescription object must be done using the static methods of the Services class. Do not use the ServiceRepository class directly.

When a ServiceDescription object is created, its fields must be populated with the desired information. If they are not, this will not prevent the ServiceDescription object from being published, but these fields will be empty. There are three mandatory fields which must be supplied at the moment of construction: the service object Eoid, the service object Unihandle, and the service object's implemented interface. This service object is the actual object in charge of providing the service or an intermediate "man in the middle" implementing the ServiceProvider interface. This interface must be the same interface used to export the service to the M2MI layer. If the service was exported more than one time using different interfaces, for each different interface we want to advertise this service with then a different ServiceDescription must be created and published to the ServiceRepository. This does not apply for superinterfaces, since they are all obtained from the subinterface being advertised.

In spite of having Start and End Date fields, these are merely informational, since these values are not taken into account automatically for publishing or unpublishing a service. If no values are specified for the Date fields, the defaults will be the system's current time at the moment of construction.

Constructors

- public **ServiceDescription**(edu.rit.m2mi.Eoid theServiceEoid, edu.rit.m2mi.Unihandle theUnihandle, java.lang.Class theInterface)

  Constructs a ServiceDescription object with the given Service Eoid, Unihandle, and implemented interface. This interface must be the same which was used to export this service to the M2MI layer. If there is more than one interface from the same exported

service which needs to be advertised, then a different ServiceDescription object must be created. This does not apply to the superinterfaces of the given interface, which are automatically obtained from this one at the moment of publishing.

Parameters:

theServiceEoid - The service's Exported Object ID (Eoid).

theUnihandle - The service's Unihandle.

theInterface - The service's exported interface for the given Unihandle.

- public **ServiceDescription**(edu.rit.m2mi.Eoid theServiceEoid, edu.rit.m2mi.Unihandle theUnihandle, java.lang.Class theInterface, long TTL)

Constructs a ServiceDescription object with the given Service Eoid, Unihandle, implemented interface, and Time To Live (TTL). This interface must be the same which was used to export this service to the M2MI layer. If there is more than one interface from the same exported service which needs to be advertised, then a different ServiceDescription object must be created. This does not apply to the superinterfaces of the given interface, which are automatically obtained from this one at the moment of publishing. The TTL is for calculating the EndDate field, which is merely informational and will not generate an automatic unpublishing of the ServiceDescription object

Parameters:

theServiceEoid - The service's Exported Object ID (Eoid).

theUnihandle - The service's Unihandle.

theInterface - The service's exported interface for the given Unihandle.

TTL - The time in milliseconds this service will be published for.

Methods

- getServiceEoid

public edu.rit.m2mi.Eoid **getServiceEoid**()

Gets the Eoid of the service advertised by this ServiceDescription object.

Returns:

The service Eoid.

- getUnihandle

public edu.rit.m2mi.Unihandle **getUnihandle**()

Gets the Unihandle of the service advertised by this ServiceDescription object. The returned Unihandle implements the interface specified in this object.

Returns:

The service's Unihandle.

- getInterface

public java.lang.Class **getInterface**()

Gets the interface implemented by the service being advertised by this ServiceDescription object. This is the interface implemented by the Unihandle returned by this object.

Returns:

The interface implemented by this service.

- getDescription

public java.lang.String **getDescription**()

Gets the description for the service advertised by this ServiceDescription object.

Returns:

The description of the service.

- setDescription

public void **setDescription**(java.lang.String theDescription)

Sets the string description of the service being advertised by this ServiceDescription object.

Parameters:

theDescription - description of the service advertised by this object.

- getName

public java.lang.String **getName**()

Gets the name of the service advertised by this ServiceDescription object.

Returns:

The service name.

- setName

public void **setName**(java.lang.String theName)

Sets the name of the service advertised by this ServiceDescription object.

Parameters:

theName - name to be given to the service.

- getVersion

public long **getVersion**()

Gets the version number of the service advertised by this ServiceDescription object.

Returns:

the version number of the service

- setVersion

public void **setVersion**(long theVersion)

Sets the version number of the service advertised by this ServiceDescription object.

Parameters:

theVersion - version number of the service

Returns:

The version number of the service.

- getTime

public long **getTime**()

Gets the time (in milliseconds) this service has been advertised.

Returns:

the time (in milliseconds) this service has been advertised.

- getStartTime

public long **getStartTime**()

Gets the time (in milliseconds) at which this service was first started. The format in milliseconds is that as specified in Date.

Returns:

The start time.

- getStartTimeAsDate

public java.util.Date **getStartTimeAsDate**()

Gets the time (as a Date) at which this service was first started.

Returns:

The start time.

- getBuildDate

public java.util.Date **getBuildDate**()

Gets the time (as a Date) at which the service was built.

Returns:

The build date.

- setBuildDate

public void **setBuildDate**(java.util.Date theBuildDate)

Sets the time at which the service advertised by this ServiceDescription object was built.

Parameters:

theBuildDate - date at which this service was built for its current version.

- getOwner

public java.lang.String **getOwner**()

Gets the owner of this advertised service.

Returns:

The name of the owner.

- setOwner

public void **setOwner**(java.lang.String theOwner)

Sets the owner of this advertised service.

Parameters:

theOwner - name of the owner of this service.

- setTTL

public void **setTTL**(long theTTL)

Sets the time to live for this advertised service. This is informational, no automatic actions will be taken by the API to unpublish this service after this time expires.

Parameters:

theTTL - time to live (in milliseconds) for this service.

- getTTL

public long **getTTL**()

Gets the time to live (measured from this instant) in milliseconds for the advertised service.

Returns:

The time to live for this service.

- getEndDate

public java.util.Date **getEndDate**()

Gets the Date at which this service will be unpublished.

Returns:

The date this service will be unpublished.

- equals

public boolean **equals**(java.lang.Object o)

Determines if two ServiceDescription objects are equal.

Returns:

True if both ServiceDescription objects have the same service Eoid and implemented interface.

- compareTo

public int **compareTo**(java.lang.Object o)

Compares two ServiceDescription objects.

Specified by:

compareTo in interface java.lang.Comparable

Parameters:

o - object to compare.

Returns:

A number less than 0 if this ServiceDescription is less than o; 0 if this ServiceDescription is equal to o; a number greater than 0 if this ServiceDescription is greater than o.

- hashCode

public int **hashCode**()

Gives a hashCode for this object.

Returns:

A hashcode for this object.

- toString

public java.lang.String **toString**()

Gives a string version for this ServiceDescription object.

Returns:

A string version for this object.

ServiceLeaseRenewer Class

public class **ServiceLeaseRenewer**

extends java.lang.Object

This class manages the renewal of a lease for a service that has been published. It will take a `ServiceDescription` and automatically renew its lease with the `Services` class as long as this service is needed to be kept available.

After a service has been published it must lease and renew its lease before it expires. To start this process, create a new `ServiceLeaseRenewer` with the published `ServiceDescription` and call `startRenewing`. This will automatically request and renew the given lease between the Services class and the published service, for as long as the service shall be published. If the lease is not renewed, it will expire and the Services class will remove this published service from its published services set. The actual service will not be unexported from the M2MI layer, only its `ServiceDescription` will not be advertised as a service. After this happens, the service must explicitly be registered again with the Services class.

When the `ServiceLeaseRenewer` object is called for stopping the renewal of a lease, the Services class will unpublish the corresponding `ServiceDescription` after its next check of its leases times out, which could happen at any instant within one minute after the call to `stopRenewing`. The service can also be unpublished immediately by calling the `unregister` method on the Services class.

After a call to `stopRenewing` is made, there is still a chance to continue renewing with the same `ServiceLeaseRenewer`. To do this, call `reset` and follow with a call to `startRenewing`. This last call must be made before the `Services` class removes this service for being published without a lease.

All the renewal processing is made by a worker thread in this class. If during this processing an Exception takes place, an unchecked `LeaseRenewerException` is thrown.

Constructors

- public **ServiceLeaseRenewer**(ServiceDescription theService)

  Constructs a new ServiceLeaseRenewer for the given Service. It will request a lease duration of one minute to the Services class. It will try to renew the lease every 30 seconds.

  Parameters:

theService - ServiceDescription object for which this ServiceLeaseRenewer will manage a lease.

- public **ServiceLeaseRenewer**(ServiceDescription theService, long theLeaseDuration)

Constructs a new ServiceLeaseRenewer for the given Service and with the given lease duration. It will request a lease for the given duration to the Services class, which may grant a lease for the requested time or less. The interval for renewing the lease will be set at half the time the lease was granted for.

Parameters:

theService - ServiceDescription object for which this ServiceLeaseRenewer will manage a lease.

theLeaseDuration - Duration of lease to be requested for this service.

Methods

- startRenewing

public void **startRenewing**()

Starts renewing the lease for this ServiceLeaseRenewer.

Throws:

java.lang.IllegalStateException - (unchecked) if this lease has already been requested or not reset

- reset

public void **reset**()

Resets the state of the ServiceLeaseRenewer. This is to be called only if a lease is not being renewed (in force). After the call to this method, a call to startRenewing must follow to resume renewal.

Throws:

java.lang.IllegalStateException - (unchecked) if the lease is in force.

- stopRenewing

public void **stopRenewing**()

Stops the renewal of the lease held by this ServiceLeaseRenewer. If the lease needs to be restarted after stopping it, call reset and follow with a call to startRenewing.

Throws:

java.lang.IllegalStateException - if a lease is not in force

ServiceEvent Class

public class **ServiceEvent**

extends java.lang.Object

Encapsulates an event generated by a `ServiceRepository` object. This event reports the new set of services available in the network, as an array of `ServiceDescription` objects. It also contains a timestamp of the moment the event was generated. Since they are all reported to local objects, the timestamp in milliseconds is absolute for all running processes on the system, guaranteeing correct sequencing when examining timestamps of different events.

Constructor

- public **ServiceEvent**(ServiceDescription[] services, long timestamp)

Creates a new ServiceEvent object with the given array of services and timestamp.

Parameters:

services - The list of all services to report in this event

timestamp - Time in milliseconds at which this event was generated. The time is referred to as it is described in Date.

Throws:

java.lang.NullPointerException - (unchecked) if the service is null.

Methods

- getTimestamp

public long **getTimestamp**()

Gets the timestamp in milliseconds at which this event occurred.

Returns:

the timestamp in milliseconds.

- getServices

public ServiceDescription[] **getServices**()

Gets the list of services in this event.

Returns:

An array with the services in the repository.

ServiceEventRegistration Class

public class **ServiceEventRegistration**

extends java.lang.Object

Encapsulates a listener with a filter which have been registered to receive ServiceEvent objects.

Constructor

- public **ServiceEventRegistration**(ServiceEventListener listener, ServiceEventFilter filter)

Creates a new ServiceEventRegistration object with the given listener and filter.

Parameters:

listener - The event listener which will receive the events filtered by the given filter.

filter - The event filter which will decide if the events should be sent to the given listener.

Throws:

java.lang.NullPointerException - (unchecked) if the listener is null. If the filter is null, all events will be passed.

Methods

- getListener

public ServiceEventListener **getListener**()

Gets the listener of this object.

Returns:

the listener in this object.

- getFilter

public ServiceEventFilter **getFilter**()

Gets the filter in this object.

Returns:

the filter in this event.

ServiceJobID Class

public class **ServiceJobID**

extends java.lang.Object

implements java.io.Serializable

Base class for identifying one specific service request at a service provider. It can be used for confirming results to a `ServiceClient` either directly or extending it for specific implementation functionality. This class stores a `serviceID` long integer (unique in the network for a published service), two `Unihandles` for the client and the service, and a local job ID integer corresponding to this job in the given service.

Fields

- myServiceID

protected long **myServiceID**

Identification of the Service provider in the network

- myClientUnihandle

protected edu.rit.m2mi.Unihandle **myClientUnihandle**

Client's Unihandle

- myServiceUnihandle

protected edu.rit.m2mi.Unihandle **myServiceUnihandle**

Service's Unihandle

- myLocalJobID

protected int **myLocalJobID**

Internal identification of this job or request in the Service Provider

Constructor

- public **ServiceJobID**()

Constructs a new ServiceJobID. To be used internally for serialization; use the other constructor for developing applications since the values cannot be modified after construction.

- public **ServiceJobID**(long theServiceID,

edu.rit.m2mi.Unihandle theServiceUnihandle,

edu.rit.m2mi.Unihandle theClientUnihandle, int theLocalJobID)

Constructs a new ServiceJobID with the given service ID, service Unihandle, client Unihandle, and Local Job ID.

Parameters:

theServiceID - service identification in the network

theServiceUnihandle - Unihandle for the providing service

theClientUnihandle - Unihandle for the requesting client

theLocalJobID - identification of this request for the service provider

Methods

- getServiceID

public long **getServiceID**()

Returns the service provider ID for this ServiceJobID.

Returns:

the service provider ID

- getClientUnihandle

public edu.rit.m2mi.Unihandle **getClientUnihandle**()

Returns the client's Unihandle for this ServiceJobID

Returns:

the client's Unihandle

- getServiceUnihandle

public edu.rit.m2mi.Unihandle **getServiceUnihandle**()

Returns the service's Unihandle for this ServiceJobID

Returns:

the service's Unihandle

- getLocalJobID

public int **getLocalJobID**()

Returns the job ID in the service provider

Returns:

the local job ID

- equals

public boolean **equals**(java.lang.Object o)

Compares two ServiceJobID objects. They are the same if all their fields are equal.

Returns:

true if all fields in both ServiceJobID are equal.

- hashCode

public int **hashCode**()

Returns a hash code for this ServiceJobID

Returns:

hash code for this ServiceJobID

- toString

public java.lang.String **toString**()

Returns a string representation of this ServiceJobID

Returns:

string representation of this ServiceJobID

ServicesLoader Class

public class **ServicesLoader**

extends java.lang.Object

The ServicesLoader class is a main program which facilitates the loading of services in a device. The ServicesLoader class basically takes advantage of Java's Reflection capabilities to load the classes it is told to. The classes loaded are specified in file `servicesloader.properties`. The format and usage of this file is explained below.

Operation

The ServicesLoader will start the M2MI layer (M2MP layer must be initialized previously) and look for the `servicesloader.properties` file in four locations:

1. Current working directory: if the file is not found here, it will look for it in the second location.
2. User home directory: if the file is not found here, it will look for it in the third location.
3. "lib" directory inside the Java home directory: if the file is not found here, it will look for it in the fourth location.
4. Java home directory: if the file is not found here, the program will throw an exception and exit.

Whenever the file is found it will be read to start processing and loading the classes specified in it. This class was developed following the design of class Start in the RIT Computer Science Course Library [2].

Properties File Specification

The file is a text file containing in each line one class to be loaded along with its corresponding arguments. The file can have commented lines, these begin with the '#' character. The ServicesLoader program will ignore such lines. The structure for a line specifying a class to be loaded is the following:

```
ClassName arg1 arg2 ... argn
```

ClassName is the qualified classname of the class to be loaded. It is followed by any number of arguments needed to load this class using a constructor with the following signature:

```
String[]
```

This implies that every class written with the purpose of being loaded by this `ServicesLoader` MUST have a constructor with a single argument being that an array of strings. If the given class does not have a constructor with this signature then this class will not be loaded. The `ServicesLoader` will limit itself to create a new instance of the given class by invoking this constructor. The purpose of this class must be that of setting up the service on the network using the M2MI Service Discovery Framework. Since the `ServicesLoader` class initializes the M2MI layer, then the developed class could ignore the initialization of the M2MI layer. Here is an example of a line specifying the class PrintLoader to be loaded with the arguments "name1 size1":

```
PrintLoader name1 size1
```

After reading the previous line, the `ServicesLoader` will look for the PrintLoader class and will try to create a new instance of it. It will try to do that using a constructor which will receive the argument `{"name1","size1"}`. In case no arguments are needed to load a class, then a line with the ClassName alone should be provided in the properties file. The `ServicesLoader` will create an empty array of Strings, and will pass that as an

argument to the constructor with the signature as specified above. This means that even if no arguments are needed, this constructor with the single `String[]` argument must be coded in the class, and the `ServicesLoader` will look for that constructor.
An example on the use of this class can be found in section 3.5.

A.3. EXCEPTIONS

ServicesRuntimeException

public class **ServicesRuntimeException**
extends java.lang.RuntimeException
Signals that an exception occurred during execution in the `Services` class.

Constructors

- public **ServicesRuntimeException**()
  Constructs a new ServicesRuntimeException.

- public **ServicesRuntimeException**(java.lang.String message)

Constructs a new ServicesRuntimeException with the given message.

Parameters:

message - Detail message

- public **ServicesRuntimeException**(java.lang.String message, java.lang.Throwable cause)

Constructs a new ServicesRuntimeException with the given message and the given underlaying cause.

Parameters:

message - Detail message.
cause - Underlying exception which originated this new ServicesRuntimeException.

- public **ServicesRuntimeException**(java.lang.Throwable cause)

Constructs a new ServicesRuntimeException with the given underlying cause

Parameters:

cause - Underlying exception which originated this new ServicesRuntimeException.

ServiceNotLeasingException

public class **ServiceNotLeasingException**
extends java.lang.RuntimeException

Signals that a leasing operation (renew, cancel) was attempted on a service which does not have a lease in force.

Constructors

- public **ServiceNotLeasingException**()

Constructs a new ServiceNotLeasingException

- public **ServiceNotLeasingException**(java.lang.String message)

Constructs a new ServiceNotLeasingException with the given message

Parameters:

message - Detail message

- public **ServiceNotLeasingException**(java.lang.String message, java.lang.Throwable cause)

Constructs a new ServiceNotLeasingException with the given message and the given underlying cause

Parameters:

message - Detail message

cause - Underlying exception which originated this new ServiceNotLeasingException

- public **ServiceNotLeasingException**(java.lang.Throwable cause)

Constructs a new ServiceNotLeasingException with the given underlying cause

Parameters:

cause - Underlying exception which originated this new ServiceNotLeasingException

ServiceNotPublishedException

public class **ServiceNotPublishedException**
extends java.lang.RuntimeException

Signals that an operation was attempted on a service which is not published.

Constructors

- public **ServiceNotPublishedException**()

Constructs a new ServiceNotPublishedException

- public **ServiceNotPublishedException**(java.lang.String message)

Constructs a new ServiceNotPublishedException with the given message

Parameters:

message - Detail message

- public **ServiceNotPublishedException**(java.lang.String message, java.lang.Throwable cause)

Constructs a new ServiceNotPublishedException with the given message and the given underlying cause

Parameters:

message - Detail message

cause - Underlying exception which originated this new ServiceNotPublishedException

- public **ServiceNotPublishedException**(java.lang.Throwable cause)

Constructs a new ServiceNotPublishedException with the given underlying cause

Parameters:

cause - Underlying exception which originated this new ServiceNotPublishedException

ServiceAlreadyLeasingException

public class **ServiceAlreadyLeasingException**
extends java.lang.RuntimeException

Signals that a service is leasing in case this same service requests another lease. Each service must have only one lease at a time.

Constructors

- public **ServiceAlreadyLeasingException**()

Constructs a new ServiceAlreadyLeasingException.

- public **ServiceAlreadyLeasingException**(java.lang.String message)

Constructs a new ServiceAlreadyLeasingException with the given message.

Parameters:

message - Detail message

- public **ServiceAlreadyLeasingException**(java.lang.String message, java.lang.Throwable cause)

Constructs a new ServiceAlreadyLeasingException with the given detail message and the underlying cause.

Parameters:

message - Detail message
cause - Underlying exception which originated this new ServiceAlreadyLeasingException

- public **ServiceAlreadyLeasingException**(java.lang.Throwable cause)

Constructs a new ServiceAlreadyLeasingException with the given underlying cause.

Parameters:

cause - Underlying exception which originated this new ServiceAlreadyLeasingException

LeaseRenewerException

public class **LeaseRenewerException**
extends java.lang.RuntimeException

Signals that an exception has occurred in `ServiceLeaseRenewer`.

Constructors

- public **LeaseRenewerException**()

Constructs a new LeaseRenewerException.

- public **LeaseRenewerException**(java.lang.String message)

Constructs a new LeaseRenewerException with the given message

Parameters:

message - Detail message

- public **LeaseRenewerException**(java.lang.String message, java.lang.Throwable cause)

Constructs a new LeaseRenewerException with the given message and the given underlying cause

Parameters:

message - Detail message

cause - Underlying exception which originated this new LeaseRenewerException

- public **LeaseRenewerException**(java.lang.Throwable cause)

Constructs a new LeaseRenewerException with the given underlying cause

Parameters:

cause - Underlying exception which originated this new LeaseRenewerException

ServicesLoaderException

public class **ServicesLoaderException**
extends java.lang.RuntimeException

Signals that an exception occurred during processing of the ServicesLoader class.

Constructors

- public **ServicesLoaderException**()

Constructs a new ServicesLoaderException.

- public **ServicesLoaderException**(java.lang.String message)

Constructs a new ServicesLoaderException with the given message

Parameters:

message - Detail message.

- public **ServicesLoaderException**(java.lang.String message, java.lang.Throwable cause)

Constructs a new ServicesLoaderException with the given message and the given underlying cause.

Parameters:

message - Detail message.

cause - Underlying exception which originated this new ServicesLoaderException.

- public **ServicesLoaderException**(java.lang.Throwable cause)

Constructs a new ServicesLoaderException with the given underlying cause

Parameters:

cause - Underlying exception which originated this new ServicesLoaderException.