An Architecture for Migrating User Interfaces

David Svensson, Boris Magnusson
Dept. of Computer Science, Lund University
Ole Römers väg 3, Box 118, 221 00 Lund, Sweden
{david|boris}@cs.lth.se

Abstract

The MUI project looks at flexible ways of creating user-initiated connections between services in wireless networks. A central idea is to migrate user interfaces from controlled devices to devices with better input/output capabilities. The paper shows the different parts of the MUI architecture, and motivates design choices. An initial implementation and a framework for building MUI services are described.

1 Introduction

MUI (Migrating User Interfaces) is an architecture for services in wireless networks, where the user can connect and combine services in a simple, yet flexible, way. User interfaces can be migrated between devices, so the user can control several services from one device. MUI is developed as an ongoing research project at the department of computer science, Lund university. This paper presents the MUI architecture and the thoughts behind it. It focuses on the current state of the architecture, and on the implementation of the system.

MUI is initially designed to operate in networks of limited physical range, typically within a room, or even in networks formed between devices carried by a single person. This can be a very dynamic environment, where services enter and leave networks quite frequently, as people move around. This, in turn, puts requirements on the architecture: it must be smooth and simple to discover new services and connect to them. The diversity of equipment also requires the devices to interact with minimal or no preparation in advance. We think that the need for non-pre-planned communication is best fulfilled by defining interfaces at a very general level. Attempting to standardize the protocol for each interesting combination of services is, as we see it, not feasible—that process would be too complicated and time-consuming, as the number of services in the networks continues to grow. Instead, in the MUI architecture, a service should just present general-level information about the data it can provide and consume. It is up to the user to connect it to suitable services, with matching data types.

User interfaces are important in this context. To allow for the flexible connection of services, a service should be able to show an interface to the user,

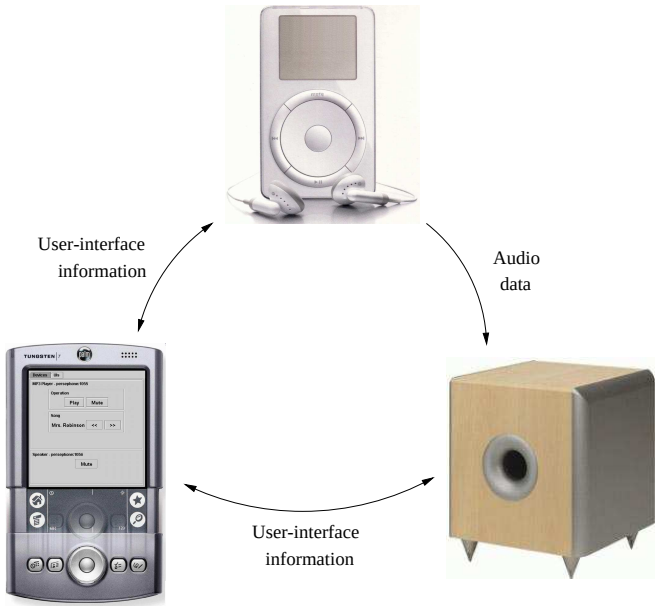
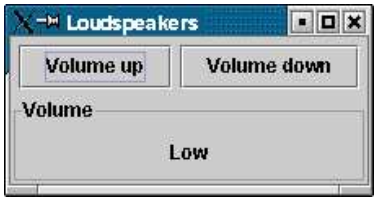


Figure 1: An example scenario with a handheld computer, an MP3 player, and loudspeakers.

where the communication with another service can be controlled in more detail. It should be possible to show the interface on a device with suitable input/output resources, such as display and keyboard. Therefore, migrating user interfaces between services is a central concept in the architecture, and user interfaces have got special attention during our initial work. We view a user interface as a service among other services, but it has a special protocol associated with it, which will be presented in section 8.

The devices in the networks will often be very small, with limited memory and processing power. This gives further requirements: the architecture should not rely on facilities such as IP network connectivity or graphical displays on all devices, but allow a light-weight implementation. Our envisioned underlying technology for network communication is Bluetooth [2], even if the initial implementation with simulated devices uses IP, as will be discussed in section 9.

The paper continues with an example scenario, and a discussion about some previous work within this problem domain. Sections 4 through 8 look at different parts of the MUI architecture. In section 4, the fundamental structure of services and connections is presented. Section 5 deals with the discovery protocol, and section 6 with MUITP, the binary transport protocol for connections. Two protocols with XML messages, for connecting services with RemoteConnect and for representing user interfaces, are explained in sections 7 and 8. A presentation of the implementation, conclusions, and future work round off the paper.

(a) 

(b) 

Figure 2: A migrated user interface for loudspeakers, before (a) and after (b) adjusting the volume.

2 A Scenario

Figure 1 illustrates an example scenario where the architecture is at work: with a handheld computer in his hand, and a portable MP3 player in his pocket, a user enters a room where a set of loudspeakers are in the corner. The MP3 player and the loudspeakers show up as services in a browser application on the handheld. The user can see that they match, and connects them by joining them in the browser. Now, the MP3 player sends its music to the loudspeaker. The volume is a little low, though, so the user chooses to control the loudspeakers by clicking their service in the browser. A user interface is moved to the handheld and shown. It may look as in figure 2 (a). The user presses “Volume up”, the volume is adjusted, and he can enjoy the music. At the same time, the user interface on the handheld is updated to that of figure 2 (b). If he wants to control also the MP3 player from the handheld, a user interface can be obtained for it in the same way.

Tables 1 and 2 show XML documents that are transferred in this scenario. The document in table 1, which describes the user interface, is what the loudspeakers send when connected to from the handheld. Table 2 shows the document that is sent when the volume has been changed, so the user interface can be updated on the handheld.

3 Previous Work

Jini from Sun [1] has targeted many of the same requirements as MUI, and could be used in the scenario above. It is an architecture where services register their presence at a lookup service, so they can be found and used by clients. The lookup service distributes proxy objects, which are downloaded to clients and used for communicating with the service. The client knows about the programmatic interface of the proxy, but does not need to know about its implementation in order to use it.

One problem we see in Jini is that the proxy interfaces are quite specific. Sun and partners are standardizing interfaces for printers, scanners, storage devices, etc. This means that clients have to be written for using a specific kind of service, and as new kinds services are invented, new clients have to be written. We hope to relax this in MUI, with general-level interfaces.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE UI SYSTEM "mui-ui.dtd">
<UI text="Loudspeakers">
  <Button text="Volume up" command="volumeUp"/>
  <Button text="Volume down" command="volumeDown"/>
  <Panel text="Volume">
    <Label text="Low"/>
  </Panel>
</UI>
```

Table 1: An XML document for the loudspeaker user interface. There are two buttons with commands attached, and a panel with an inner label.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE UIUpdate SYSTEM "mui-ui.dtd">
<UIUpdate element="/1/3/1" text="Normal"/>
```

Table 2: A UI update for the user interface of table 1. The syntax for the `element` attribute is a restricted form of the XPointer child sequence syntax (see [16]). In this example, the attribute refers to the label of the document.

Another aspect to Jini is that it is heavily tied to Java. The proxy objects are Java objects, typically communicating with the service using RMI (Remote Method Invocation, see [14]). When considering small, embedded devices, this makes Jini services bulky. We see the need for allowing implementations in other languages, such as Smalltalk or C.

There is a specification for user interfaces in Jini, in the ServiceUI API [7]. The user interfaces are associated with a service, and are written to use the proxy object interface of that service. A problem, as we see it, is that the user interfaces themselves have specific programmatic interfaces and different semantics. As new services are standardized, they are expected to come with new user-interface types, so clients will still have to be written against a specific service.

The Speakeasy project at the Palo Alto Research Center [4, 5] introduces the term *recombinant computing* for an architecture where the user can combine functionality from several services into one. Like MUI, they have an approach of generic interfaces for letting the user combine services in new ways. Other key concepts in their framework are mobile code and *user-in-the-loop interaction*. Mobile code means proxy objects, like in Jini, that are downloaded to clients and executed there. This requires some platform-independent code, and they have used Java in their implementation. For discovery and user-interface they seem to have used Jini. It is unclear to us how light-weight Speakeasy implementations can be. User-in-the-loop interaction means that the user should always be in control when connecting services, leaving it up to him to make sure the connection

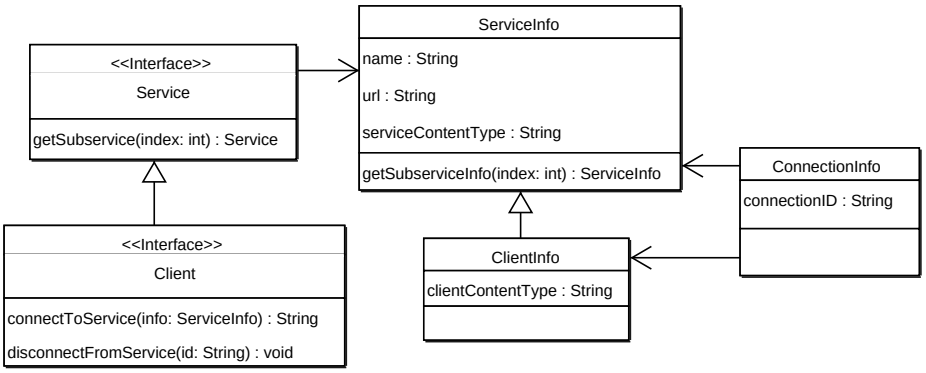


Figure 3: Classes and interfaces for services, clients, and connections. Not all aspects of the types are shown, only the most central.

makes sense. The latter point, the authors claim, helps keeping the interfaces small and generic.

In a previous project [6], we implemented a remote-control scenario for two devices with real hardware, where Bluetooth was used for the wireless communication. There was a VCR, which could be discovered from a Palm handheld. A remote-control user interface was downloaded to the handheld, and the VCR could be controlled by clicking buttons in the interface. With the experiences from that project, we are now working with simulated devices communicating over IP—this makes it easier to shape the architecture for multi-device scenarios.

One thing we found in the previous project considered the representation of the user interfaces. There, they were contained in small Java applications, J2ME MIDlets [12], that were downloaded and installed on the client device. We did not use Jini, but our own protocol for discovering and migrating the interfaces. We felt that using Java applications for the interfaces was a bit heavy-weight, so we are now working with an XML representation instead. The experiences from the previous project will be further discussed in section 8.

4 Services and Connections

The architecture of MUI is based on connections between services. A service runs on some device in the network. The service may represent the whole device, such as an MP3-player service representing an MP3 player, but there may also be several services on a device. The latter would typically be the case for larger devices, such as laptops. This *device agnosticism* is also present in Jini: everything is services, both hardware and software [3].

MUI services implement the interface `Service`, shown in figure 3. Services can have subservices in a tree structure, which is useful for grouping services into

logical units. When establishing a connection, one of the two parties must be a client. `Client` is a subinterface of `Service`, which means that clients are services themselves. The rationale behind this is the following: all clients offer a particular kind of service, namely the ability to connect to other services. The methods `connectToService` and `disconnectFromService` reflect this. Clients should also support initiation of connections over the network, using the RemoteConnect protocol (see section 7).

A service holds information about itself in a `ServiceInfo`. The information consists of the name of the service, a URL for connecting to the service, and the content type offered by the service. The content type is the MIME type of the data transferred over connections to the service. `ServiceInfos` can be nested, reflecting subservice trees. Information about clients, in `ClientInfos`, hold an additional item: the content type accepted from services. This is used for determining whether a client can connect to a certain service.

When a connection is established, a `ConnectionInfo` is put together. It contains information about the two parties, and an ID identifying the connection. The ID can be used for disconnecting later.

MUI connections are like socket connections: bidirectional connections that stay up until either party closes them or until a network error occurs. As mentioned above, the data is of a certain MIME type. It is transferred in chunks, called *documents*, and it is normal that several, or many, documents are transferred over the same connection. The MUITP protocol has been defined for the binary transport, as will be discussed in section 6.

5 Discovery

For clients to be able to use services, information about the services must reach the clients in some way. This is handled by a discovery protocol. The information transferred by the MUI discovery protocol is in `ServiceInfos` and `ConnectionInfos`. The `ServiceInfos` can be displayed to the user of an application on a device, letting him browse available services and establish connections between matching client-service pairs. `ConnectionInfos` are for managing established connections—disconnecting them, e.g. The messages are in an XML format, for which DTDs have been defined (see appendices A and B).

The MUI discovery protocol relies on having some broadcast mechanism. The existing implementation uses IP multicast, as discussed in section 9. The protocol has not yet been optimized to minimize the network traffic.

Figure 4 illustrates the process of discovering and connecting services. There are three devices: a service device, a client device, and a handheld device with a browser application. At first, the handheld broadcasts an inquiry (a). The service and client devices respond by broadcasting their information (b). In the browser application, the user can now see that the service and the client match, meaning that they can handle the same type of data. He chooses to connect them. A

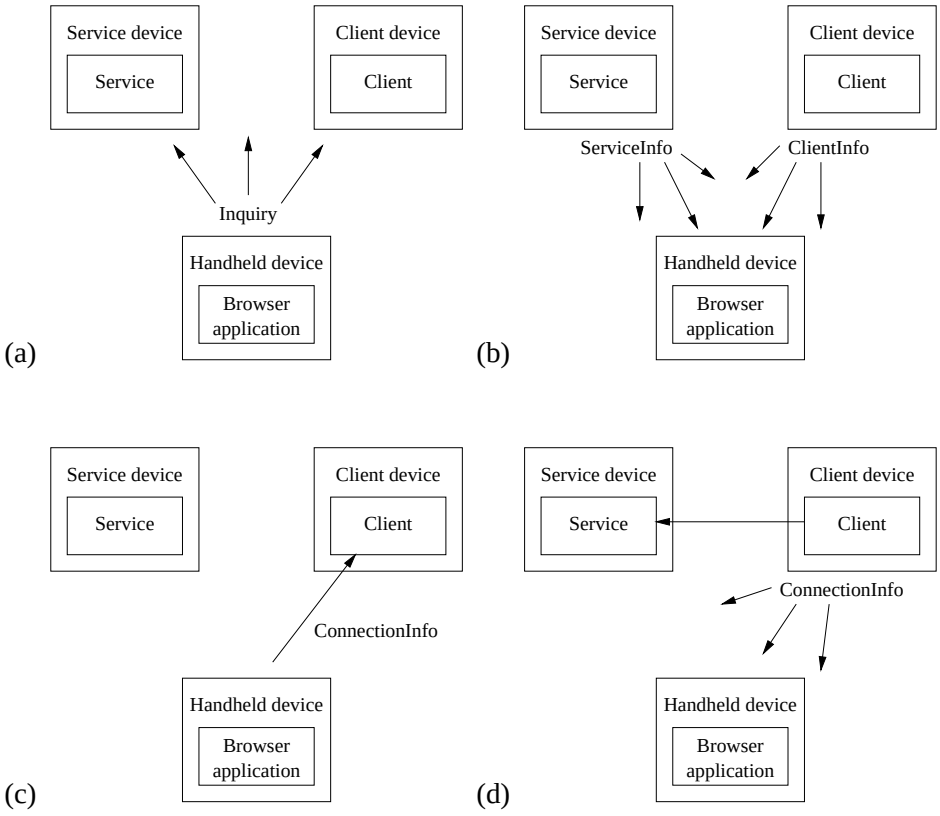


Figure 4: Discovering and connecting services.

`ConnectionInfo` is assembled, and sent to the client using the RemoteConnect protocol (c). The client uses the service URL in the `ConnectionInfo` for connecting to the service, and starts receiving data. It also broadcasts information about the new connection (d).

The risk of *partial failure* is an issue for discovery. This is one thing that makes distributed systems more complicated than non-distributed: there is always a risk that one node goes down in an unclean way, without having the time to inform the other nodes [3]. This could, e.g., be due to a software crash, power loss, or physical damage. From the other nodes, it can be hard to detect this, because it may look like a node is simply slow in response. There must be a way to handle devices that suddenly leave the network without sending out notifications, so that the lists on all devices can be updated. In Jini, *leasing* is used to remedy these problems: all resources that are used by other nodes are leased for a limited time. If the lease is not renewed regularly, the resource will be removed. This keeps old resources from filling up memory on devices, and gives a form of self-healing to the system. In MUI, we plan to implement some leasing mechanism, or to use a simpler scheme where devices periodically send out broadcasts, signalling that they are still alive.

6 MUITP

In order to transfer documents over a connection, a transport protocol is needed. We considered HTTP, but its request-response nature makes it inadequate for MUI, where any party should be able to initiate a transfer at any time[1]. Instead, MUITP was defined (MUI transport protocol).

The protocol works as follows. When a connection is established, both parties start reading from their input streams. At first, small headers are sent, where the non-client supplies the content type. The client leaves this header empty. After that follows a sequence of zero or more documents in both directions, with an arbitrary delay before and in between. The data of each document is preceded by a header, containing the length of the data. Any party may terminate the connection at any time, by simply closing it.

The definition of a new transport protocol leads to a new scheme for URLs. For simulated devices communcating over IP, a URL may look like

```
muitp://130.235.16.32:6427/
```

This URL can be used for connecting to a service listening on port 6427 of host 130.235.16.32. For a discussion about Bluetooth as the underlying protocol, and an example of a URL for Bluetooth, see section 9.

An important difference, compared to HTTP, is that in MUI the URL refers to the connection, not to a document. This is more appropriate in the MUI case, because the number of documents and identities of individual documents may be unknown to the client.

On top of MUITP, a protocol is needed for the flow of documents over the connection—which party should start to send, will there be responses, and so on. The current approach in MUI is to have an extremely simple protocol for general connections, with a little more sophisticated protocols for two special cases: RemoteConnect and user-interface connections. In the simple case, MUI connections are uni-directional; only the non-client sends documents. A uni-directional connection can be combined with a user-interface connection, letting the user control the data flow through the user interface. This is enough in many applications, where a service is the provider of some data, but perhaps the protocol will have to be changed, or split into several subprotocols, as more experience is gained from building and evaluating prototypes.

7 RemoteConnect

An important feature of MUI is the ability to establish a connection between two services, and to close it, from a third device on the network. This is handled by RemoteConnect, a protocol on top of MUITP. The messages of the protocol

[1] This requirement initially comes from UI connections, where commands and UI updates are sent asynchronously from both parties (see section 8).

are listed in the DTD shown in appendix C. For these XML messages, a special MIME type has been defined: `application/x-mui-remote-connect`.

The URLs of MUI clients are RemoteConnect URLs. When an application wants to establish a connection, it connects to this URL and sends a `ConnectRequest`. When the requested connection is up, the client sends back an `OKResponse` (or an `ErrorResponse` if something goes wrong). Disconnecting a connection with `DisconnectRequest` works similarly.

8 User Interfaces

The second kind of service with a special protocol—besides the RemoteConnect service—is the *user-interface* service. This service lets user interfaces be migrated to clients, so that users can control services from a device with suitable input/output capabilitities. The interfaces are described in an XML format, of which the loudspeaker document in table 1 is an example and whose DTD is shown in appendix D. This format has been given the special MIME type `application/x-mui-ui+xml`. There are XML elements for different widgets, with attributes for specifying a certain command to be sent when the widget is chosen (clicked). The current widget types supported are buttons, labels, and panels. Natural widgets to add next are text input fields, and perhaps check boxes and radio buttons. Adding images would also be nice, but that requires a mechanism for referring to external image resources.

When a client connects to a user-interface service, the XML description will be sent back. It is interpreted by the client, and the user interface is shown with help from a user-interface library, such as MIDP [13]. Commands are sent to the service when the user performs an action in the user interface. The service reacts to these in a domain-specific way: the set of commands is specified entirely be the service. It does not need to be standardized, because the service provides the user-interface description itself.

Web forms is a natural comparison for MUI user interfaces. In contrast to these, we wanted to make the communication two-way. The service can send a UI update at any time, which will result in a change in the interface on the client, as was exemplified in section 2. The commands and the UI updates give both pull and push functionality, and more dynamic interfaces than for Web forms. Using HTTP to accomplish this would require polling, with repeated requests to see if something has changed. In relation to web forms, the set of supported widgets is also relevant. With the extensions discussed above, we will support roughly the same set of widgets as for web forms.

In the previous project [6], we implemented the user interfaces in Java, and moved a small Java application to the client instead of XML data. The application was run on the client, displaying the interface. That gave the full power of Java, and the ability to create very dynamic interfaces, but we also felt that it was a quite heavy process to transfer, install, and start an application for each interface.

We think XML will suffice for many interfaces, and it has the advantage of being independent of the client platform—it can be rendered differently depending on screen-size, e.g. Still, we consider adding an applet-like mechanism to the architecture, for situations where the power of Java is needed.

9 Implementation and Framework

An implementation of MUI has been written for the standard edition of Java (J2SE). The network communication is over an IP network, with IP multicast as broadcast mechanism in the discovery protocol. This implementation allows simulated devices, entirely in software, that can be used for testing the architecture.

As stated before, the intention is to use MUI in wireless networks. There are several options for the wireless communication, but Bluetooth [2] has been a target from the beginning. The standard for using Bluetooth from Java, JSR-82 [10], was constructed mainly for the small-device edition of Java (J2ME, [12]). In the future, J2ME is indeed a natural choice for MUI, as many devices in the networks will have limited memory and processing power. Consequently, the design of the network classes has been made with J2ME in mind.

One goal of the implementation is to provide a framework for construction of MUI services. The classes and interfaces presented in section 4 are included, but there are also abstract classes that implement more of the behaviour expected for typical services and clients. Creating a service should be simple, if it does not have very specific needs.

We have tried to build the framework with loose coupling between components, using events and listeners for their communication. The idea is that when implementing a new device, it should be possible to pick just the components needed. Functionality that has been implemented includes the following:

- There is support for discovery. These classes rely on IP multicast, but it should be possible to change their implementation to use Bluetooth discovery, without changing their interface too much.
- RemoteConnect is implemented in the abstract client class, so by default all clients speak RemoteConnect. A class `RemoteConnectClient` represents the client part of the RemoteConnect protocol, and can be used by applications for establishing remote connections.
- Migratable user interfaces have support in the form of a number of UI component classes, which make up the interfaces, a `MigratableUI` service, which provides interfaces to clients, and a `UIClient`, which can receive the interfaces and manage the client side of the communication. The UI components can be rendered as XML or as Swing components. On smaller devices, MIDP [13] would be a more suitable user-interface library than Swing.

- The network classes have been designed to fit into the Generic Connection Framework, the more light-weight network API that is used in J2ME [11]. A protocol handler has been written for MUITP. When establishing a connection to a URL, the scheme part of the URL will be used by Java library classes to select the right protocol handler (both for J2SE and J2ME). Adjusting the network connections to Bluetooth will mainly mean to rewrite the protocol handler. The contents of a MUI URL will also have to change, from IP address and port to a Bluetooth 48-bit device address and a server channel identifier[2], something like

  ```
  muitp://0050C000321B:5
  ```

- There is support for parsing and generating XML messages (used for discovery, RemoteConnect and user interfaces). The DOM API in the Java standard classes is used. In J2ME, there is no built-in support for XML, so we will have to use a third-party parser, or write our own. See [9] for a discussion about parsers and performance considerations with XML in J2ME.

The MUI implementation is provided as a single JAR file, which can be run on any platform with J2SE and an IP network. It includes some sample simulated devices and services: there is a handheld device with a browser application for discovering and connecting services, a slideshow service, which can be connected to a screen, and a poetry service, which produces text that can be shown on a poetry client device. The implementation of the sample services has helped form the framework.

10 Conclusions

This paper has presented the current state of the MUI architecture and implementation, and the ideas behind it. We believe that the basic architecture is simple and flexible enough to be useful in the context of a wireless network where many devices provide services that can be used by clients over user-initiated connections.

Even if the implementation is in Java, the architecture is not Java-specific. The protocols and XML formats from sections 4 to 8 could be implemented by devices with runtime environments for programs written in other languages, such as Smalltalk or C. This could be an advantage where the memory and processing-power resources are scarce.

The use of XML as data format seems reasonable. The documents following our DTDs are quite compact, and we get XML's advantages of a human-readable

[2] Perhaps, it would be the best to make both IP and Bluetooth possible. Then, the URL would have to be extended with some protocol specifier after `muitp`.

format with many available parser implementations. There are XML parsers for J2ME that should be small enough (see [9]). XML is well established as a data format in many domains, and standardized as a W3C recommendation [15].

Thanks to the framework, the sample services were quite simple to write. It seems to be a good idea to provide the basis for a standard service, that can be used when developing custom services.

11 Future Work

The current work on MUI involves fixing smaller things in the implementation, and extending the testing of the classes, using the JUnit framework [8]. We will add a few more widget types, as discussed in section 8. Then, we plan to add support in the architecture for user-controlled composition of *virtual services*: several connected services combined into one, for smoother repeated use. It will be interesting to see if the user interface for the virtual service can be generated from user interfaces of individual services.

We will investigate further the options for migrating to real wireless hardware, such as Bluetooth. We also want to make it possible for services to provide applet-style executable code to be run on clients, as a complement to the XML UI descriptions. Prototypes for more different kinds of services will be built and investigated. Finally, we will work more on the error handling, especially that concerning the partial failure issues discussed in section 5.

References

[1] Arnold, Ken, et al. *The Jini Specification*. Addisson-Wesley. 1999.

[2] Bluetooth.org. `https://www.bluetooth.org/`

[3] Edwards, W. Keith. *Core Jini*. Prentice Hall, Inc. 1999.

[4] Edwards, W. Keith, et al. Challenge: Recombinant Computing and the Speakeasy Approach. In *Proceedings of Mobicom '02*. September 2002.

[5] Edwards, W. Keith, et al. The Case for Recombinant Computing. Xerox Palo Alto Research Center Technical Report CSL-01-1. April 20, 2001.

[6] Eklund, Torbj¨orn and David Svensson. Mui: Controlling Equipment via Migrating User Interfaces. Master Thesis. Department of Computer Science. Lund Institute of Technology. 2003.

[7] Jini.org. The ServiceUI API specification. `http://www.jini.org/standards/ServiceUI/ServiceUISpec.html`

[8] JUnit. Testing Resources for Extreme Programming. `http://www.junit.org/`

[9] Knudsen, Jonathan. Parsing XML in J2ME. http://developers.sun.com/techtopics/mobility/midp/articles/parsingxml/. March 7, 2002.

[10] Motorola. Java API for Bluetooth Wireless Technology (JSR-82). Specification version 1.0a. April 5, 2002.

[11] Ortiz, C. Enrique. The Generic Connection Framework. http://developers.sun.com/techtopics/mobility/midp/articles/genericframework/. August, 2003.

[12] Sun. Java 2 Platform, Micro Edition (J2ME). http://java.sun.com/j2me/

[13] Sun. Mobile Information Device Profile. http://java.sun.com/products/midp/

[14] Sun. Java Remote Method Invocation (Java RMI) http://java.sun.com/products/jdk/rmi/

[15] W3C. Extensible Markup Language (XML) 1.0 (Third Edition). W3C Recommendation. 4 February 2004. http://www.w3.org/TR/2004/REC-xml-20040204/

[16] W3C. XPointer xpointer() Scheme. Working Draft. 19 December 2002. http://www.w3.org/TR/xptr-xpointer/

A mui-info.dtd

This DTD defines the format for information about MUI services, clients, and connections:

```
<!ENTITY % serviceinfoelement
         "(ServiceInfo | ClientInfo)">
<!ENTITY % boolean
         "(true | false)">

<!ELEMENT ServiceInfo ((%serviceinfoelement;)*)>
<!ATTLIST ServiceInfo
  name CDATA #REQUIRED
  serviceContentType CDATA #REQUIRED
  url CDATA #REQUIRED>

<!ELEMENT ClientInfo ((%serviceinfoelement;)*)>
<!ATTLIST ClientInfo
  name CDATA #REQUIRED
```

```
  clientContentType CDATA #REQUIRED
  remoteConnectURL CDATA #REQUIRED>

<!ELEMENT ConnectionInfo (ServiceInfo, ClientInfo)>
<!ATTLIST ConnectionInfo
  connectionID CDATA #IMPLIED>

<!ELEMENT ServiceInfoEvent (%serviceinfoelement;)>
<!ATTLIST ServiceInfoEvent
  active %boolean; #REQUIRED>

<!ELEMENT ConnectionInfoEvent (ConnectionInfo)>
<!ATTLIST ConnectionInfoEvent
  active %boolean; #REQUIRED>
```

B mui-discovery.dtd

The DTD for discovery contains a single element, except those included from `mui-info.dtd`:

```
<!-- Include declarations from info DTD -->
<!ENTITY % infodecl SYSTEM "mui-info.dtd">
%infodecl;

<!ELEMENT Inquiry EMPTY>
```

C mui-remote-connect.dtd

The messages of the RemoteConnect protocol are defined in `mui-remote-connect.dtd`:

```
<!-- Include declarations from info DTD -->
<!ENTITY % infodecl SYSTEM "mui-info.dtd">
%infodecl;

<!ELEMENT ConnectRequest (ConnectionInfo)>

<!ELEMENT DisconnectRequest (ConnectionInfo)>

<!ELEMENT OKResponse EMPTY>
<!ATTLIST OKResponse
  connectionID CDATA #REQUIRED>
```

```
<!ELEMENT ErrorResponse EMPTY>
<!ATTLIST ErrorResponse
  message CDATA #REQUIRED>
```

D mui-ui.dtd

The UI DTD defines elements for widgets, commands and UI updates:

```
<!ENTITY % uielements  "(Panel | Button | Label)*">

<!-- UI elements -->
<!ELEMENT UI %uielements;>
<!ATTLIST UI
  text CDATA #REQUIRED>

<!ELEMENT Panel %uielements;>
<!ATTLIST Panel
  text CDATA #REQUIRED>

<!ELEMENT Button EMPTY>
<!ATTLIST Button
  text CDATA #REQUIRED
  command CDATA #REQUIRED>

<!ELEMENT Label EMPTY>
<!ATTLIST Label
  text CDATA #REQUIRED>

<!-- Elements for communication with the user -->
<!ELEMENT Command EMPTY>
<!ATTLIST Command
  name CDATA #REQUIRED>

<!ELEMENT UIUpdate EMPTY>
<!ATTLIST UIUpdate
  element CDATA #REQUIRED>
  text CDATA #REQUIRED>
```