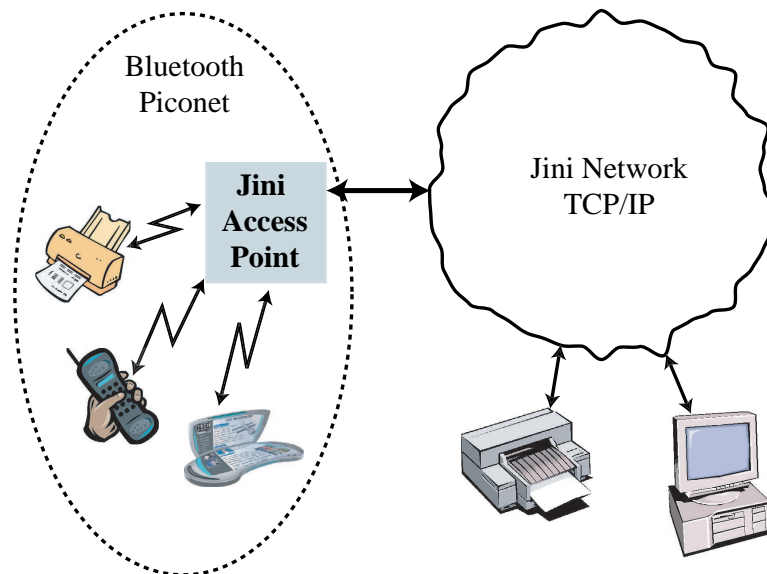


Samuel Kasper, D-ITET/8
Lorenz Bühner, D-ITET/8

Jini Discovers Bluetooth

Semester Thesis SA-2002.30
Summer 2002



Tutors:
Vincent Lenders
Jan Beutel

Supervisor:
Prof. Dr. B. Plattner

Tutors

Vincent Lenders, lenders@tik.ee.ethz.ch

Jan Beutel, beutel@tik.ee.ethz.ch

Supervisor

Prof. Dr. Bernhard Plattner, plattner@tik.ee.ethz.ch

Students

Samuel Kasper, skasper@ee.ethz.ch

Lorenz Bühner, lbuehner@ee.ethz.ch

Acknowledgments

This documentation was written at the *Computer Engineering and Networks Laboratory (TIK)* [1] of the *Swiss Federal Institute of Technology (ETH Zurich)* [2] during the summer semester 2002.

The authors would like to thank their tutors Vincent Lenders and Jan Beutel, PhD students at TIK, for their dedicated and competent help in various aspects. Vincent Lenders is an expert concerning Jini whereas Jan Beutel is a specialist regarding Bluetooth. Moreover, we thank Prof. Dr. Plattner for his assistant inputs.

We could benefit from an excellent infrastructure provided by TIK. As a consequence, it was possible to concentrate from the beginning on the important tasks.

Zurich, 5th July 2002

Samuel Kasper, Lorenz Bühner

Abstract

The goal of this thesis was to design an integration model between Jini and Bluetooth and to implement it on a working platform. The motivation of this work was:

- Extension of the limited discovery range (\sim ten meters) in Bluetooth. Discovery in Jini allows finding and using services over multiple hops and different physical networks. Thus, Bluetooth devices are able to communicate with Bluetooth devices located in piconets out of their range or even with devices without any Bluetooth connectivity.
- Bluetooth services can be used independently from the communication medium.
- Jini can use Bluetooth as wireless, robust communication medium.

Three approaches were investigated and designed. One uses Sun's surrogate architecture in order to connect a regular Bluetooth device to the Jini network. In addition, it was determined how Jini could be placed on top of the Bluetooth stack with the aid of Bluetooth' LAN and PAN profiles. Finally, an approach using a bridge between the two physical networks was chosen for further development.

A prototype of a Bluetooth-Jini bridge, enabling SDP requests from Bluetooth to Jini, was implemented. Moreover, the implementations were tested successfully on desktop PCs as well as on an iPAQ H3870.

To sum up, it is possible and worthwhile to combine Bluetooth and Jini in order to benefit from the respective strengths.

Kurzfassung

Das Ziel dieser Semesterarbeit war es, ein Integrationsmodell zwischen Jini und Bluetooth zu entwickeln und auf einer Plattform zu implementieren. Die Motivation dieser Arbeit war:

- Erweiterung des auf nur etwa zehn Meter limitierten Discovery Bereichs von Bluetooth. Discovery in Jini ermöglicht es, Dienste über mehrere Hops und in verschiedenen physikalischen Netzwerken zu finden und zu gebrauchen. Bluetooth Einheiten können dadurch mit anderen Geräten ausserhalb ihres Piconetzes kommunizieren, sogar wenn diese nicht mit Bluetooth Hardware ausgerüstet sind.
- Bluetooth Dienste können unabhängig vom Kommunikationsmedium benutzt werden.
- Jini kann Bluetooth als kabelloses, robustes Kommunikationsmedium nutzen.

Drei Ansätze wurden untersucht und entworfen. Einer davon benützt Sun's Surrogate Architektur, um ein normales Bluetooth Gerät mit dem Jini Netzwerk zu verbinden. Im weiteren wurde untersucht, wie Jini mit Hilfe von Bluetooth' LAN oder PAN Profilen auf den Bluetooth Stack gebracht werden könnte. Schliesslich wurde ein Ansatz, der eine Bridge zwischen den beiden physikalischen Netzen vorschlägt, für die Weiterentwicklung gewählt.

Ein Prototyp einer Bluetooth-Jini Bridge wurde implementiert. Die Bridge ermöglicht SDP Anfragen von Bluetooth nach Jini. Zudem wurden die Implementationen erfolgreich auf Desktop PCs und auf einem iPAQ H3870 getestet.

Zusammenfassend sei erwähnt, dass es möglich und lohnenswert ist, Bluetooth und Jini zu kombinieren, um von den entsprechenden Stärken zu profitieren.

Timeschedule

	Activity	Meetings	Milestones
Week 1	collecting information, frame report	administration	software installed
Week 2	studying specifications, learning by doing		
Week 3	studying specifications, learning by doing		tutorials and examples finished, report started
Week 4	looking for related stuff		search complete
Week 5	analysis of related work		analysis complete
Week 6	feasability study of different approaches		
Week 7	choosing appropriate approach for further development	Prof. Dr. Plattner	preliminary version report
Week 8	designing		design complete
Week 9	implementation, configuration		
Week 10	implementation, configuration		configuration ready
Week 11	implementation		software complete
Week 12	testing, evaluation		beta version report, testing finished
Week 13	working on presentation	presentation	
Week 14	working on report		delivering report and software

Contents

1	Introduction	1
2	What is Jini?	3
3	What is Bluetooth?	6
4	Motivation	9
5	Analysis of Related Work	12
5.1	Zucotto's Xpresso Java Native Processor	12
5.2	Jini in a Cellular Phone	14
5.3	Mapping Salutation Architecture to Bluetooth SDP Layer . .	15
5.4	Bluetooth ESDP for UPnP	17
5.5	IP Services over Bluetooth	19
6	Approaches to Combine Bluetooth and Jini	22
6.1	Surrogate Architecture and Bluetooth Interconnectivity . . .	22
6.2	Jini over Bluetooth	26
6.2.1	Solution with LAN Access Profile	27
6.2.2	Solution with PAN Profile	29
6.2.3	Solution with Surrogate Host	31
6.3	Bluetooth-Jini Bridge	33
7	Development of a Bluetooth-Jini Bridge	35
7.1	Preliminary Remarks	35
7.2	Requirements	35
7.3	Bluetooth Service Discovery Protocol (SDP)	36
7.3.1	Protocol Data Unit Types	37
7.3.2	Service Attributes	38
7.4	Jini Lookup Service and Discovery	40
7.4.1	Jini API Description	40
7.4.2	Service Attributes	41
7.5	Correlations Between Service Parameters	43
7.6	Design Model	44

7.7	Implementation	46
7.7.1	Jini Service	46
7.7.2	Bluetooth Devices and Tools	46
7.7.3	Bluetooth to Jini Translator (Bridge)	47
8	Testing	49
8.1	Configuration and Setup	49
8.1.1	iPAQ	49
8.1.2	Bluetooth on PC	49
8.1.3	Jini on iPAQ and PC	49
8.2	Experimental Environment	50
8.2.1	PC Based Experiment	50
8.2.2	iPAQ Based Experiment	51
9	Conclusions	52
9.1	Results	52
9.2	Outlook	53
A	Used Software Tools	54
B	Used Hardware	55
C	System Requirements	56
D	Howtos	57
D.1	Compile Sdptool and SDP Daemon	57
D.2	Compile Java/Jini Files	58
D.3	Usage of the Implemented Applications	58
E	Implemented and Modified Software	60
E.1	Jini Service	60
E.2	Jini Service Interface	61
E.3	sdptool	62
E.4	Daemon Request-Handling	70
E.5	Bridge	78
E.6	Client Request	80
E.7	Scripts	81

List of Figures

2.1	Typical Jini TCP/IP network with lookup service, clients and services	4
3.1	Typical Bluetooth piconet with one master controlling the communication of the slaves	6
4.1	The Jini architecture embedded in the Java environment . . .	9
4.2	The Bluetooth protocol stack	10
4.3	Position of Bluetooth in Jini in the OSI reference model . . .	11
5.1	The Xpresso architecture	13
5.2	Jini on a cell phone using Sun's surrogate architecture	14
5.3	Salutation API mapping to Bluetooth SDP	16
5.4	Salutation TM mapping to Bluetooth SDP	16
5.5	UPnPs L2CAP based solution	17
5.6	UPnPs IP based solution using LAN profile	18
5.7	UPnPs IP based solution using PAN profile	18
5.8	IP diagram flow between a MH and its peer, using mobile IP	19
5.9	Cellular IP access network	20
5.10	BLUEPAC reference network architecture	21
6.1	Surrogate architecture components	23
6.2	Case 1: Device enters a network where a surrogate host is already present. Case 2: A surrogate host is started in a network with devices present.	24
6.3	Device makes discovery of a surrogate host and uploads its surrogate	24
6.4	Surrogate makes service discovery and lookup	25
6.5	The device loading the required classes	25
6.6	The device using the service	25
6.7	A Bluetooth device communicating with a Jini device via lookup service	26
6.8	Bluetooth profile dependencies	27
6.9	Jini over Bluetooth using the LAN profile	28
6.10	Jini over Bluetooth using the PAN profile	30

6.11	BNEP with an Ethernet packet payload sent using L2CAP	31
6.12	Comparison of wireless networks	32
6.13	A surrogate host connecting a Bluetooth device with a Jini device via lookup service	33
6.14	A Bluetooth-Jini bridge connecting the two physical networks	33
7.1	Bluetooth stack with service record database	36
7.2	SDP client-server interaction	37
7.3	Bridge device embedded in a Jini-Bluetooth environment	44
7.4	The Jini-Bluetooth bridge in more details	48
8.1	Experiment with three Desktop PCs	50
8.2	Experiment with two iPAQs and one Desktop PC	51
B.1	ROK tester	55
B.2	iPAQ H3870	55

List of Tables

2.1	Relations between Jini and some alternatives	5
3.1	Comparison of wireless technologies	8
7.1	Matching Bluetooth and Jini service parameters	43
C.1	Minimal requirements to run the implemented code	56

Chapter 1

Introduction

A citation of two Java developers (Mark Sears and Gregory Lewis) should introduce and motivate our thesis: “One billion connected devices: This vision is quickly becoming a reality as more and more Internet-enabled devices are being deployed in environments as diverse as grocery stores, automobiles, headsets or even our shirt pockets. Today’s Internet provides a global, distributed network of millions of servers and PCs. Tomorrow’s dynamic, personal, and intermittent networking of one billion devices will make the wireless Internet a reality. With a plethora of wireless hardware and software services surrounding us, we are faced with the challenge of determining how best to take advantage of them. The ability to discover, use, and link these services in a beneficial way will make their deployment both meaningful and worthwhile.”

Communication is a keyword in today’s society. People communicate for social and business reasons and as technology evolves, the number of available communication channels increases. Today, we use cell phones, mail, fax and chat to communicate. The basic concept is still the same, but technology has largely extended our possibilities of communicating.

A way of making communication easier is the concept of dynamic service allocation. Instead of having device services installed the traditional way, they register themselves to a central lookup service and are automatically downloaded and configured if required. This technology is currently under development. One of the developers is Sun, working on the *Jini* technology (see Section 2).

On the other hand, the devices become always smaller and lighter. The generation of mobile devices was born. Using a wired system, the mobility is limited to the length of the cable. The wireless concept has lately got a lot of attention. One relatively new technology that operates in the wireless area is *Bluetooth* (see Section 3). Bluetooth is designed to automatically connect devices over short distances and as a cable replacement in networks.

To benefit from both network technologies, it is worthwhile to implement

an application that combines Bluetooth' wireless network transport with Jini's dynamic service allocation (see Section 4).

The analysis of related work can be found in Section 5. We introduce three models how to combine Bluetooth and Jini. Each of them uses a distinct approach. The design of these approaches is found in Section 6. While the development of a Bluetooth-Jini bridge is described in Section 7, the testing of the implementation follows in Section 8. Finally, Section 9 summarises the results and gives an outlook on further work.

Chapter 2

What is Jini?

The Jini networking system [3] is a distributed infrastructure built around the Java programming language. The basic communication model is based on the semantic model of the Java Remote Method Invocation (RMI) system. Objects in one Java Virtual Machine (JVM) communicate with objects in another JVM by receiving a proxy object, which implements the same interface as the remote object. This proxy object deals with all communication details between the two processes, and it may introduce new code into the process to which it is moved. This is possible because Java bytecode is portable. It is also safe because of the built-in verification and security of the Java environment.

To this underlying communication model, the Jini system adds some basic infrastructure and parts of a programming model. The infrastructure provides a mechanism by which clients and services can join into the Jini network, while the programming offers a mechanism to build reliable, distributed systems.

The infrastructure centers around the Jini lookup service (see Figure 2.1). This establishes a place for providers of a service to advertise their availability, and for clients desiring a service to find out what is available. Discovery is accomplished using the Jini discovery protocol, by which an entity (either hardware or software) can find a Jini lookup service when it first connects to the network. The discovery protocol is the specification of a multicast packet which, when received by a Jini lookup service, causes the lookup service to send back a Java object that implements the lookup service interface. This object is loaded into the sender's process. Then, the sender registers itself with the lookup service if it wishes to provide services, or queries the lookup service for other service providers. A Jini participant can be both a service and a client of other services.

Services describe themselves by the Java language interfaces that they implement. When a client wants to find a service, the client requests an object that implements a particular Java interface. Any object that is an

instance of the requested type can be returned to the requester. This includes objects that implement a subtype of the requested type, or a set of types that includes the requested type.

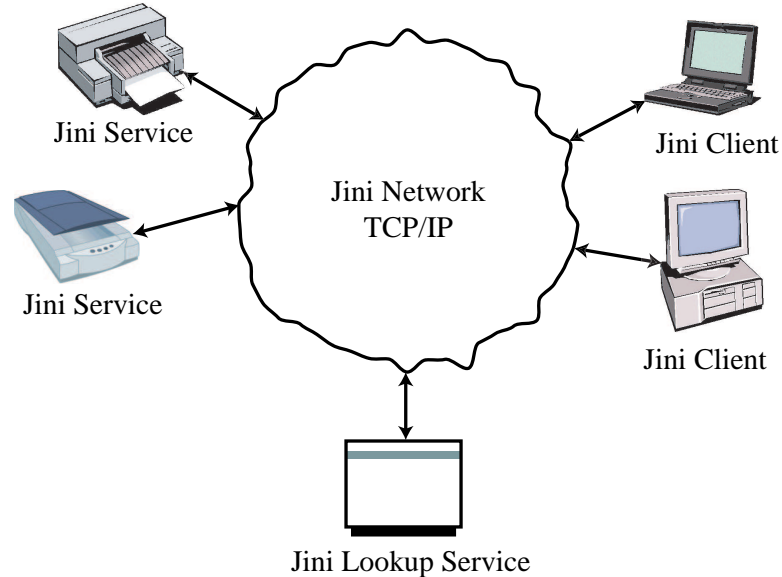


Figure 2.1: Typical Jini TCP/IP network with lookup service, clients and services

The interfaces that encapsulate the programming model center around three kinds of object interactions:

- A set of interfaces defines the notion of an event notification, allowing an object to give other objects the ability to register for notification when a conceptual event occurs in the first object.
- A set of transaction interfaces defines a two-phase commit protocol that can be used by objects to coordinate activities within the objects.
- A leasing interface defines a model of time-based resource allocation, in which a resource is granted for a renewable period of time, determined by a single-round negotiation between the resource requester and the resource granter.

Renewal of a lease can be requested by the holder of the lease, but if the lease is not renewed before it expires, both the grantor and the holder of the lease can assume that the resource is now available to others.

The infrastructure within the Jini system uses the Jini programming model. This enables the lookup service to register interest in various kinds

of events, allowing participants in a Jini network to track changes in the set of services available via the lookup service. Registration of a service within a lookup service is also leased, so if a service does not renew the lease, it is deleted from the set of available services. This ensures that the lookup service does not advertise services that no longer exist for any period of time longer than the maximum lease period, as such services will not renew their leases.

The Jini networking system is built around requirements that will be presented by the edge of the Internet in the near future. This is because Jini technology is designed to allow ad hoc networking, in which the network participants can join and leave the network with minimal human intervention. A Jini system allows participants to join the network by finding one or more lookup services, which can then be used to either register a service being offered or find a service to use. Once a participant receives a connection to a physical network (using, for example, a mechanism like DHCP), the finding of a lookup service can be entirely automated. Leaving the network is equally automated. Clients may leave the network at any time, because any references the client has to services are leased. When those leases expire, the referenced resources can be reclaimed. Services may leave in an equally abrupt manner, since their registrations with the lookup service are also leased. After a lease expires, no client will attempt to use that service, because it is no longer registered and therefore will not be considered part of the network federation [4].

To sum up, Jini helps to turn the network into a flexible, easily administered tool. In addition, Jini's strengths are its security model and the platform independency. However, there are also some weaknesses. Jini requires TCP/IP based network transport. It would be more powerful if any network protocol could be handled. Moreover, Jini needs Java 2 Standard or Enterprise Edition [5]. Therefore, limited devices cannot be directly attached to the Jini community.

Table 2.1 compares Jini technology with its hardest competitors.

Name	Originator	Platform	Language	Network	Security
Jini	Sun Microsystems	any	Java	TCP/IP	yes
Salutation	Consortium	any	any	any	no
UPnP	Microsoft	any	any	TCP/IP	no

Table 2.1: Relations between Jini and some alternatives

Chapter 3

What is Bluetooth?

The Bluetooth standard [6] is a relatively new technology using short-range radio links intended to replace the cable connections between nodes in a network. These links build an ad-hoc network, called piconet. Piconets can consist of up to eight units (see Figure 3.1) and each unit can communicate with other piconets as well. The piconets are established dynamically and automatically as Bluetooth devices enter and leave the radio proximity.

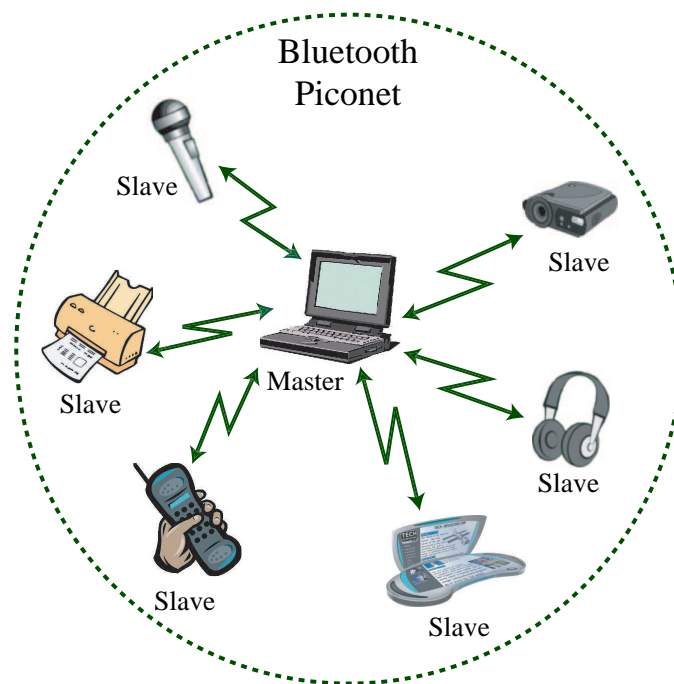


Figure 3.1: Typical Bluetooth piconet with one master controlling the communication of the slaves

Five promoter companies originally launched the Bluetooth standard: Ericsson, IBM Corporation, Intel, Nokia and Toshiba. Currently, over 1800 companies within the Bluetooth SIG (Special Interest Group) promote the Bluetooth standard. The specification is free and open to all, making it easier for developers and programmers to learn and use the technology and probably enlarging the market for Bluetooth. Key features that makes Bluetooth unique are its ability to operate on low power, relatively cheap devices, low complexity as well as robustness. Another key design issue was reuse of existing higher layer transport and application protocols already developed. The radio links replace the cables in networks on the lower levels in the protocol stack. The network is well suited for mobile devices which communicate over the air, requiring minimal user effort. The technology offers wireless access to LANs, the mobile phone network, the Internet and handheld devices. Bluetooth is divided into two parts, the Bluetooth protocols [7] and the Bluetooth profiles [8].

The features provided by Bluetooth go far beyond those offered by any other technology building a wireless standard. While conventional wireless standards define simple emulation of a wired LAN, Bluetooth offers a complete set of application profiles and protocols including:

- Service discovery - location of other users and services as diverse as data access within the office and electronic menu access within restaurants.
- Telephony - use as both an intercom service and to access existing telephone networks. Specifically Bluetooth defines the concept of a 3-in-1 phone with cellular, Bluetooth base station and Bluetooth intercom functions.
- Headsets - cable replacement for headset use.
- Synchronisation - synchronisation (such as address book and appointments) between PC, PDA and cellular phones.
- File transfer - operating system independent file transfer.
- Data transfer - full multi-protocol access to in-building data services.
- WAP - Bluetooth can be used to provide access to local WAP resources, rather than relying on the cellular network.

Bluetooth radios operate in the unlicensed ISM (Industrial Scientific and Medical) radio-band at 2.4 GHz. This radio band is available for worldwide use without license. Designed to operate in a noisy radio frequency environment, the Bluetooth radio uses a fast acknowledgement and frequency hopping scheme to make the link robust. Bluetooth radio modules avoid

interference from other signals by hopping to a new frequency after transmitting or receiving a packet. Compared with other systems operating in the same frequency band, the Bluetooth radio typically hops faster and uses shorter packets.

Future extensions to Bluetooth will add many new features including higher speed data links (at least eight times faster than current radio links), new applications such as enhanced data network integration or e-commerce [9].

Summing up, Bluetooth is inexpensive and designed for ad-hoc networking. It does not require line of sight and provides both data and voice channels. Bluetooth' most blazing strength is its low power consumption. Therefore, it can be used in very small devices with limited storage batteries. Moreover, Bluetooth offers well defined specifications and is supported by a large number of implementing companies. However, it also has some weaknesses. The maximal transfer rate (1 MB/s) is relatively low and the networking range is short. In addition, the number of network connections is limited.

Table 3.1 compares Bluetooth wireless technologies with some of its competitors.

Feature and Function	Infrared	HomeRF	Bluetooth
Connection Type	Infrared, Narrow Beam	Spread Spectrum	Spread Spectrum
Spectrum	Optical 850 nm	RF 2.4 GHz	RF 2.4 GHz
Transmission Power	100 mW	100 mW	1 mW
Data Rate	16 MB/s	1 MB/s	1 MB/s
Range	1 m	Typical Home	10 m
Devices	2	127	8
Voice Channels	1	6	3
Addressing	32 Bit Physical ID	48 Bit MAC	48 Bit MAC

Table 3.1: Comparison of wireless technologies

Chapter 4

Motivation

Jini and Bluetooth are technologies that allow spontaneous ad-hoc networking. Jini was developed by Sun Microsystems with the goal of providing a uniform platform for seamless interaction between heterogeneous devices and services in dynamic network environments. Any kind of device or service is able to automatically join a Jini community and offer services to other entities in a plug-and-play fashion without any human interaction. A possible application scenario could be a user equipped with his portable PC who wants to print at an unknown site. Traditionally, he first would have to look for a printer device, then to install the adequate drivers and finally to configure the address parameters before sending a print job. Jini provides mechanisms to automate these tiring processes.

Jini is completely based on Java. Hence, it is embedded between the application layer and Java as can be seen in Figure 4.1.

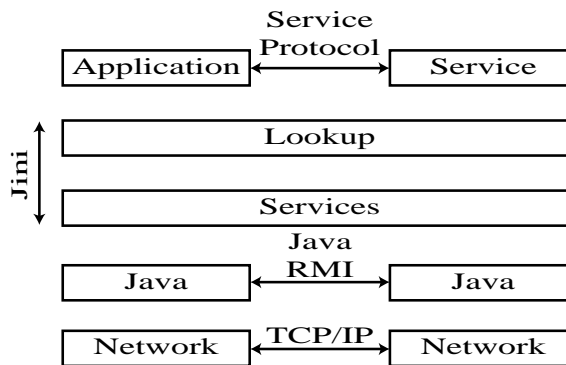


Figure 4.1: The Jini architecture embedded in the Java environment

On the other hand, the Bluetooth standard defines a set of communication protocols to form wireless ad-hoc networks with heterogeneous devices. Bluetooth aims at mobile devices with low power capabilities and restricted resources. Bluetooth is becoming more and more popular and many manufacturer equip their new devices with incorporated Bluetooth modules. Bluetooth addresses heterogeneity by defining a set of profiles which describe the different features that are device specific. If devices want to interoperate with others, they have to implement the same profile.

The Bluetooth stack consists of several protocols. Figure 4.2 shows an overview of the Bluetooth architecture.

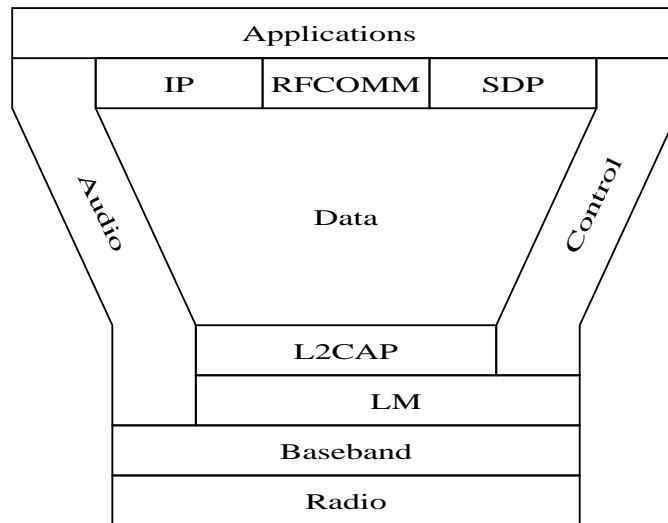


Figure 4.2: The Bluetooth protocol stack

Both Jini and Bluetooth address similar key issues of ad-hoc networking (discovery, interoperability, robustness, heterogeneity), but at a different scope. The Bluetooth standard addresses the communication issues from the physical to the application layer whereas Jini assumes an underlying TCP/IP connection infrastructure (see Figure 4.3). Jini and Bluetooth are not competing technologies. They can benefit from each other in ad-hoc networking environments.

The Bluetooth Service Discovery Protocol (SDP) [7] provides mechanisms to discover services offered by other Bluetooth enabled devices. SDP limits the discovery range to the active devices in piconets. It is desirable to enlarge this discovery range to devices in distant piconets or even to devices or services without any Bluetooth wireless communication capabilities. Jini can act as relay for multihop discovery.

On the other hand, Bluetooth is able to provide a robust TCP/IP ad-hoc

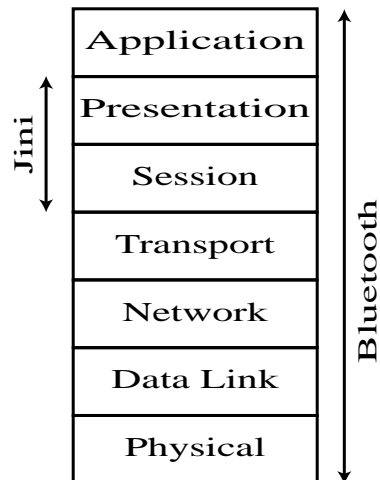


Figure 4.3: Position of Bluetooth in Jini in the OSI reference model

infrastructure for wireless communication in Jini communities. Furthermore, the Bluetooth stack can be implemented in a cheap module and does not require as much computational resources as required by Jini.

All in all, Jini and Bluetooth are strong technologies for ad-hoc networking. They perfectly complement each other. Bluetooth chips are already integrated in laptops, PDAs, cell phones, cameras and so on. On the other hand, many devices are equipped with a JVM and can join a Jini community. For all these reasons, it is straightforward and desirable to combine the two network technologies.

Chapter 5

Analysis of Related Work

After an intense investigation, we found some papers bothering the main idea of our work. Although there are some ideas how to build a seamless connection between Jini and Bluetooth, nobody did implement it so far. On Java's developer page, there are currently no projects concerning an extension of the Jini technology.

In the following subsections, some existing concepts as well as implementations are described and analysed. Studying related work makes it easier to choose an appropriate approach for our thesis. Additionally, some approaches may be used as originator for our work.

5.1 Zucotto's Xpresso Java Native Processor

Zucotto [10] is focusing on extending Jini technology to small, portable wireless devices. That allows these devices to plug into and participate in a robust distributed framework where they can dynamically communicate and cooperate.

In order to realise Zucotto's idea, the Jini surrogate architecture [11] is required. It takes the parts of the Jini framework that are difficult to embed and places them in a single, more capable device called a surrogate host. Under this model, a mobile device delivers code to the surrogate host, which will then perform Jini connection activities such as registering and using services on behalf of the device.

When a mobile device is added to a network containing a surrogate host, the interconnect adapter discovers the device and retrieves its surrogate. This activates the surrogate, which establishes a connection back to the device, then proceeds to interact with the Jini network on behalf of the device. When the device is removed from the network, its departure is detected by the interconnect adapter, which then deactivates the surrogate.

The Jini surrogate architecture is designed to work over any type of network, providing interconnect specifications that define how discovery, surro-

gate retrieval and reachability are carried out for a particular type of connection. Probably, there will be an interconnect specification for Bluetooth connectivity soon.

The Bluetooth wireless specification fully meets the requirement of the Jini surrogate architecture to provide a means of detecting when a device is added to or removed from the network.

Zucotto's Xpresso Java native processor (see Figure 5.1) is a platform for the Jini surrogate architecture, as it supports both Java and Bluetooth technologies. The Xpresso processor supports any Java 2 Micro Edition (J2ME) application and also directly executes Java bytecode in hardware. One version of the Xpresso processor family also contains on-chip Bluetooth baseband circuitry and Java APIs that enable wireless access to mobile devices.

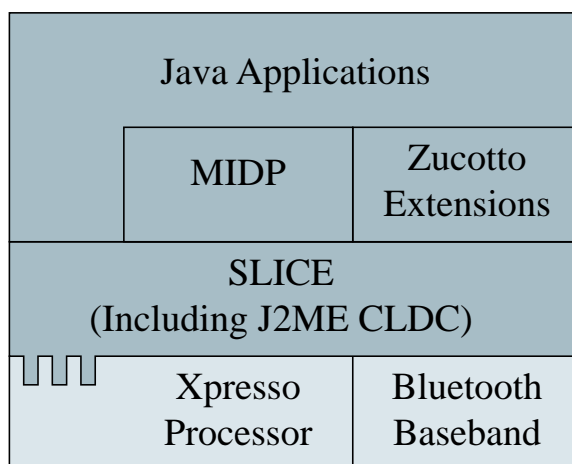


Figure 5.1: The Xpresso architecture

Conclusion: Zucotto's Xpresso processor allows small, wireless devices like cell phones or PDAs to run the Java 2 Micro Edition. Since Jini requires the Java Standard or Enterprise Edition, a surrogate host is needed. It acts as a proxy between the portable device and the Jini environment. As mentioned above, the surrogate architecture may shortly provide an interconnect specification for Bluetooth connectivity. Using this specification, it will be possible to use the surrogate architecture and Zucotto's processor in order to expand the range of small Bluetooth devices to the Jini network.

5.2 Jini in a Cellular Phone

In autumn 2000, a master thesis investigated how to run Jini in a cellular phone [12]. Jini needs resources such as an almost full strength JVM, which means a fast processor and a large memory. Both is not available on small devices like cellular phones. Sun developed a minimal JVM called KVM for the Java 2 Micro Edition. Unfortunately, the KVM lacks some basic features which are essential for Jini (e.g. RMI).

This approach uses Sun's surrogate architecture in order to run Jini on the cell phone. The surrogate solution makes it possible to run a collection of Jini enabled Java classes (the surrogate) on a surrogate host. This host can be seen as a Jini gateway.

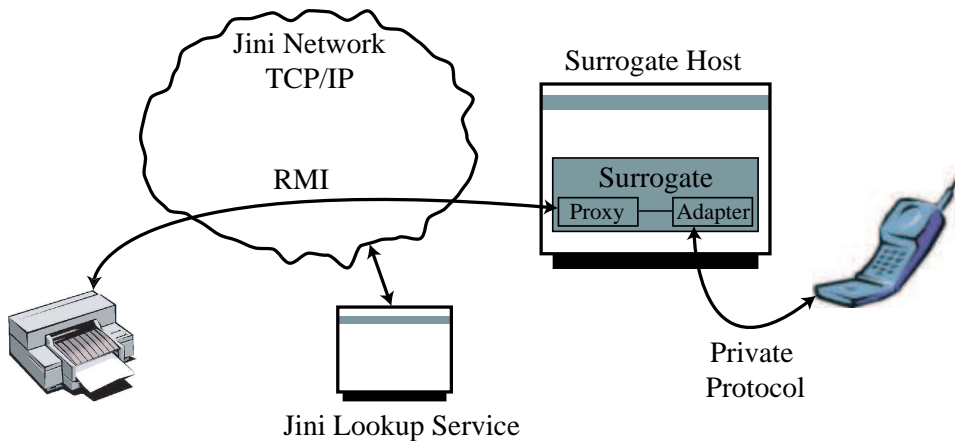


Figure 5.2: Jini on a cell phone using Sun's surrogate architecture

Currently, Sun provides two complementary specifications for the Surrogate architecture: the Jini surrogate architecture specification [11] and the Jini IP interconnect specification [13]. The surrogate host is located on the link between the two physical networks and acts as a bridge between the proxy and the device. It was proposed that the link between the telephone and the surrogate host could be Bluetooth with TCP/IP.

The Mobile Information Device Profile (MIDP) which is used from the cell phone is not designed to support Jini, not even the Jini surrogate architecture. As a consequence, an additional profile called MiniJini Profile (MJP) was designed on top of MIDP. With the aid of the MJP profile, it was possible to use an MIDP device as a Jini client.

Summing up, it is possible to connect a cellular telephone to a Jini network. Yet, Jini must be extended with the surrogate architecture and additional profiles. Moreover, it was found that Jini is the best, most robust

and securest technology compared to other similar technologies like UPnP [14] or Salutation [15].

Conclusion: The surrogate architecture, which in fact is a simple gateway solution, seems to be a very convenient way in order to link Bluetooth and Jini networks. Even small devices not being able to run Jini could be integrated with the aid of the surrogate host. However, the interconnect specification for wireless devices is not final yet.

5.3 Mapping Salutation Architecture to Bluetooth SDP Layer

The Bluetooth protocol stack contains the Service Discovery Protocol (SDP) [7]. It is used to configure the stack in order to support end-user applications. SDP can further be used to locate services that are available on devices in the vicinity of the user. Having located available services, a user may then select to utilise any of them.

The service discovery profile [8] describes both a generic syntax and semantics to be used by a service discovery application. The primitives are described in a generic way as they may be operating environment dependent.

The Salutation architecture [15] provides a standard method for applications, services and devices to describe and advertise their capabilities to other applications, services and devices and to find out their capabilities. The architecture also enables applications, services and devices to search other applications, services or devices for a particular capability, and to request and establish interoperable sessions with them to utilise their capabilities.

In the paper [16] from 1999, two approaches were considered in order to map Bluetooth SDP primitives to Salutation APIs. The first approach assumed that the Salutation APIs are implemented on top of the Bluetooth SDP (see Figure 5.3). In this case, the mapping showed how SDP attributes were passed to the Salutation APIs. The Salutation APIs were implemented as an interface to Bluetooth SDP.

The second approach assumed that the Salutation manager (SLM) can be mapped directly to the SDP protocol using a Bluetooth specific transport manager (TM). SDP was replaced by the Salutation manager mapping its functionality to the SDP protocol (see Figure 5.4). Only the transport manager had to be rewritten.

Since the Salutation manager is independent of underlying protocols and operating environments, a Salutation implementation can be a single application interface to numerous protocols. In addition to the Bluetooth mapping, Salutation has been specified for TCP/IP and IR.

Conclusion: Salutation's approaches show how to use the Salutation technology on Bluetooth enabled machines i.e. expanding Salutation to the

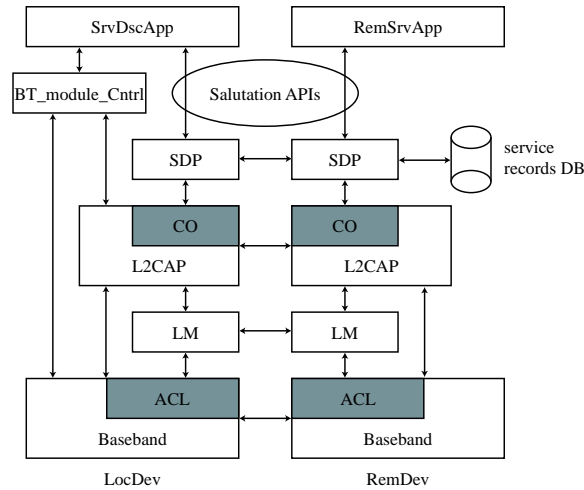


Figure 5.3: Salutation API mapping to Bluetooth SDP

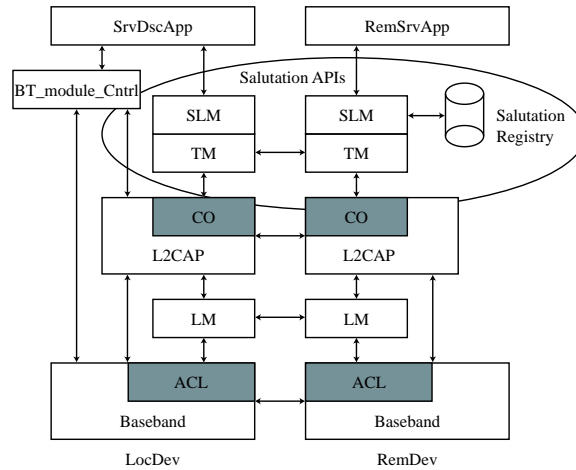


Figure 5.4: Salutation TM mapping to Bluetooth SDP

wireless range. Salutation does not require any specific network technology. Therefore, it is relatively easy mapped to Bluetooth. The second approach (Bluetooth transport manager) could be adapted for our intention. Instead of implementing a new transport manager, the appropriate network layer for Jini (TCP/IP) would have to be build on top of the Bluetooth stack.

5.4 Bluetooth ESDP for UPnP

In a Bluetooth networked computing environment, it is desirable for devices to be able to discover services beyond their current piconet. In addition, extended service discovery could enable devices with Bluetooth wireless technology to control remote resources on other devices within and outside their piconets.

Universal Plug and Play (UPnP) [14] is a distributed, open networking architecture that is designed to enable simple ad-hoc communication among distributed devices and services. UPnP leverages TCP/IP and the web model to provide seamless, media independent, peer-to-peer device connectivity and control. UPnP is a computing, electronics, telephony and networking industry initiative designed to enable connectivity among standalone devices and PCs. These UPnP characteristics make it suitable as a Bluetooth Enhanced Service Discovery Protocol (ESDP) that is intended to provide an enhanced mechanism for service discovery and control within Bluetooth environments.

The paper [17] was written in February 2001 and describes two different approaches to implement UPnP within a Bluetooth network:

- L2CAP (Logical Link Control and Adaptation Protocol) based solution: the UPnP service is layered over the L2CAP layer of the Bluetooth protocol stack for use with devices that lack IP support.

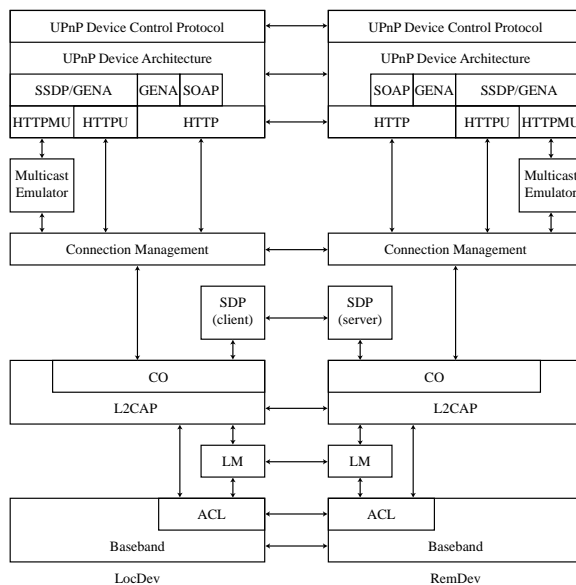


Figure 5.5: UPnP's L2CAP based solution

- IP based solution: the UPnP service is layered over the Bluetooth Personal Area Networking (PAN) profile [18] or the Local Area Network (LAN) profile [8]. These profiles define IP support for Bluetooth.

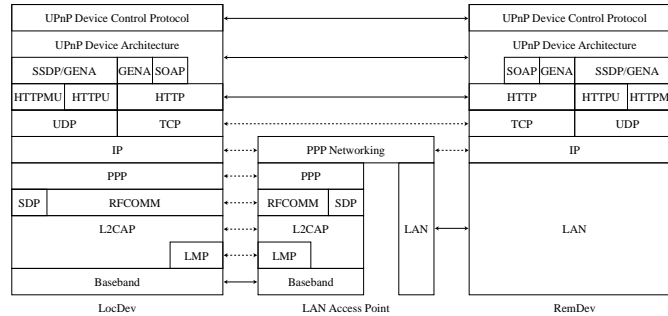


Figure 5.6: UPnP's IP based solution using LAN profile

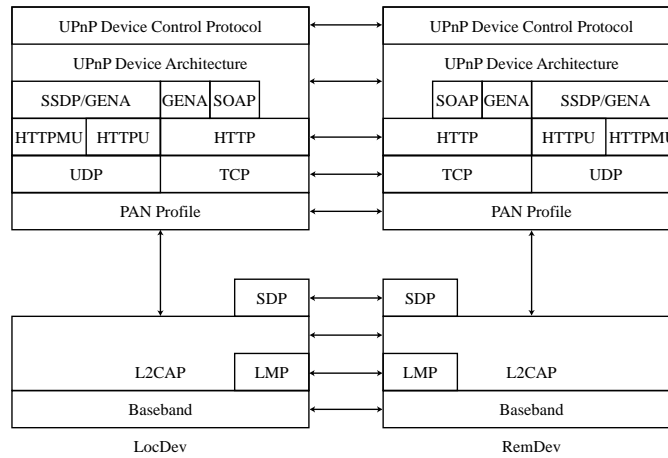


Figure 5.7: UPnP's IP based solution using PAN profile

Conclusion: Compared to Jini, UPnP does not require an underlying IP stack. Therefore, it is possible to map UPnP directly over the L2CAP layer of the Bluetooth protocol stack. L2CAP does not support the IP protocol but provides a simple peer-to-peer connection between the UPnP device and the Bluetooth stack. On the other hand, both IP based solutions can be used as a reference for layering Jini on the Bluetooth stack because the PAN profile as well as the LAN profile provide IP based connections.

5.5 IP Services over Bluetooth

The paper [19] describes a protocol concept for an extension of IP for mobility issues in Bluetooth networks. The protocol is called BLUEPAC IP (**BLU**etooth **P**ublic **AC**cess). Public access means access to various kinds of information in public areas.

The standard IP, designed for stationary computers, does not suit the requirements of a Bluetooth network where devices enter an area, stay in one Bluetooth cell or move from one to another in less than two seconds. BLUEPAC IP takes IP as a basis and additionally adds functionalities of mobile IP and cellular IP.

- Mobile IP

IP is based on the assumption that each host has a hierarchical IP address: a net ID followed by a host ID. IP datagrams to a host are primarily routed to the network given by the net ID and then delivered to the actual host by the host ID. Since many services require a static IP address but want to be accessed from different network places and devices want to be reachable at foreign networks by their home IP address, a entity called *home agent* and a entity called *foreign agent* have to be established (see Figure 5.8). On arrival at a new network, the Mobile Host (MH) has to contact the local foreign agent, which supplies a *care-of-address*, after that the MH's home agent is informed that all IP datagrams destined for the MH are forwarded (tunneled) to this new care-of-address.

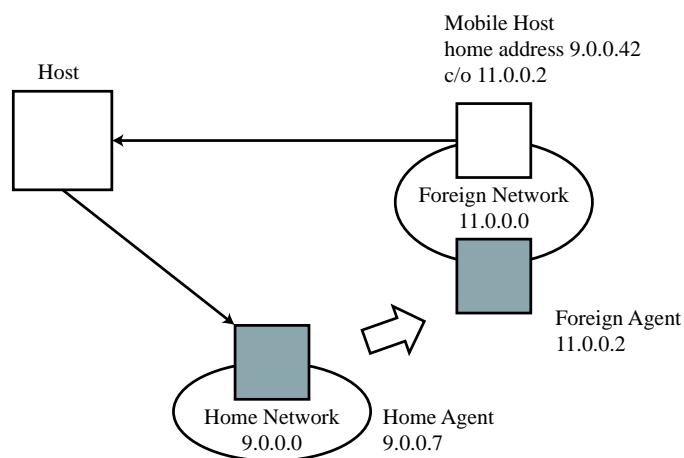


Figure 5.8: IP diagram flow between a MH and its peer, using mobile IP

- Cellular IP

Cellular IP represents a combination of the strengths of mobile IP and third generation cellular systems without inheriting their weaknesses. When entering a foreign network, the MH retains its home IP address. Cellular IP makes it possible to route IP packets regardless of the location in the cellular IP network. Thus, hosts are identified by their home address, but these IP addresses have no location significance. The cellular IP network consists of Gateways (GW), Base Stations (BS) and MH (see Figure 5.9). A GW connects the cellular IP network to a regular IP network. BS connect the wired network to the wireless network. They periodically send beacon signals which are used by the MH to locate the nearest BS. Each node has a routing cache which maps the MH IP address to the interface leading to the MH. Each packet from the MH going to the GW updates this cache. This mapping is only valid for a specific time. A handoff occurs when the MH leaves the range of a BS and approaches a new BS. The MH sends a new packet and routing cache is updated. IP packets may be sent to the old route as well, until the old routing cache times out.

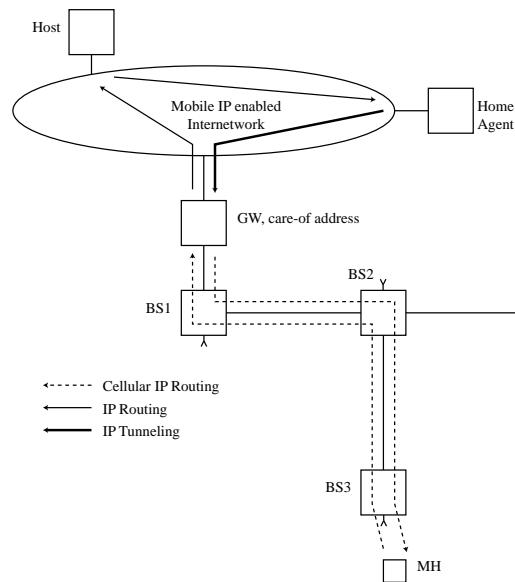


Figure 5.9: Cellular IP access network

Every device in a BLUEPAC network has to use BLUEPAC IP. Communication with devices outside the network is possible without any modifications. The network has to support moving as well as stationary devices

and has to handle devices that enter the network with or without an own IP address.

The concept splits the world into two parts, the BLUEPAC area and the rest of the world (see Figure 5.10). These two parts are connected by a GW. The main element of the BLUEPAC area is a wired BLUEPAC LAN to which the devices should gain access. This access is granted by the BLUEPAC Base Stations (BBS) which connects the wireless piconets to the wired BLUEPAC LAN. Possible scenario: several BBS cover a large area in a building. With features of cellular IP, a client who uses a service can jump from one piconet to another without loss of service.

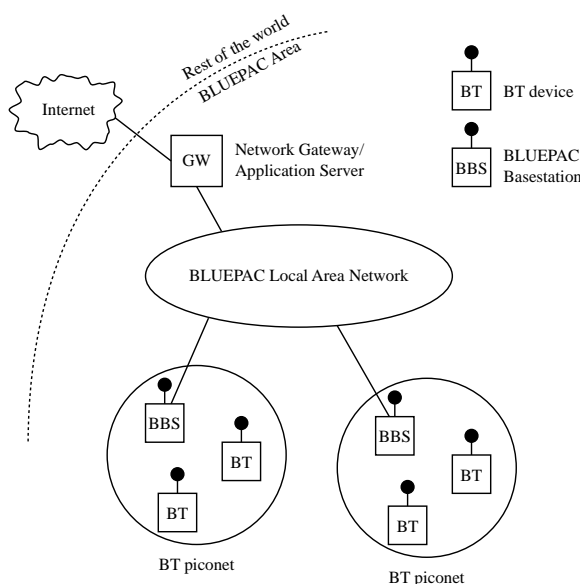


Figure 5.10: BLUEPAC reference network architecture

Conclusion: The paper presents a possible solution of the problem with moving devices in a Bluetooth area. The standard IP protocol does not cope this problem. It is assumed that Bluetooth devices can connect to a BBS and participate in the BLUEPAC LAN with BLUEPAC IP. Since Jini devices without Bluetooth features remain usually stationary, the standard IP suffice the requirements. However, if Jini devices are combined with Bluetooth, the standard IP does not satisfy all demands any more. BLUEPAC IP might be a suitable solution.

Chapter 6

Approaches to Combine Bluetooth and Jini

In this chapter, we describe our approaches how Jini and Bluetooth could be combined. Advantages as well as weaknesses of the relative solution are discussed.

6.1 Surrogate Architecture and Bluetooth Interconnectivity

- Introduction: In order that a hardware or software component can join a Jini network, it has to satisfy several critical requirements: it must be able to participate in the Jini discovery and join protocols, and it should be able to download and execute classes written in the Java programming language. In addition, it may need the ability to export classes written in the Java programming language so that they are available for download by a remote entity. For many hardware or software components, these requirements are not difficult to meet. However, there is a category of components that for one reason cannot satisfy one or more of the requirements and therefore cannot participate directly in a Jini network. The Jini surrogate architecture specification [11] addresses this problem by defining a means by which these components can participate in a Jini network while still maintaining the plug-and-work model of Jini technology. The common attribute of the hardware or software components targeted by the surrogate architecture is the inability to download code, because of either computational resource or network connectivity limitations.
- Idea: The Jini surrogate architecture (see Figure 6.1) enables to connect a Bluetooth device to a Jini network. Yet, one thing is missing: an interconnection interface between Bluetooth devices and the surro-

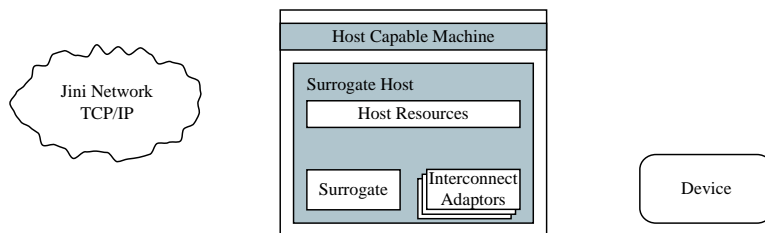


Figure 6.1: Surrogate architecture components

gate host. According to some Jini developers, a Bluetooth interconnect specification for the surrogate architecture is built (in June 2001, there was a conference in San Francisco that treated this topic). A device or surrogate host that uses the surrogate architecture for Bluetooth networks must comply with this specification, as well as with the Jini surrogate architecture specification.

An interconnect or surrogate is defined as *the logical and physical connection between the surrogate host and a device*. It is possible for the interconnect to be on the same physical connection that forms the Jini network. The Bluetooth interconnect specification will provide an interconnect-specific protocol programming model. The interconnect protocol includes an interconnect-specific mechanisms for discovery and retrieval of the surrogate.

In the following, the mechanism for running a surrogate on a Bluetooth supporting environment is described. Initially, a surrogate host is present on a Jini capable machine and there is an interconnect adapter monitoring the device interconnect. The surrogate host could be located at a Bluetooth control point and could be integrated with the lookup service.

The device and the surrogate host find each other by a discovery protocol, similar to the Jini discovery. When the discovery has been performed, the surrogate of the device must be retrieved (see Figure 6.2). This is done by either uploading the bytecode of the surrogate implementation or by uploading the URL to the bytecode which the surrogate host can then download. The bytecode and other relevant data are stored in a jar file. The registration can also include some initialising data for the surrogate.

The surrogate host then creates an instance of the surrogate and activates it. The surrogate is executing in the context of the surrogate host. It may use the Jini resources of the host.

Once the surrogate is executing, it may perform any task necessary for the device, especially communicate with the Jini network. The surro-

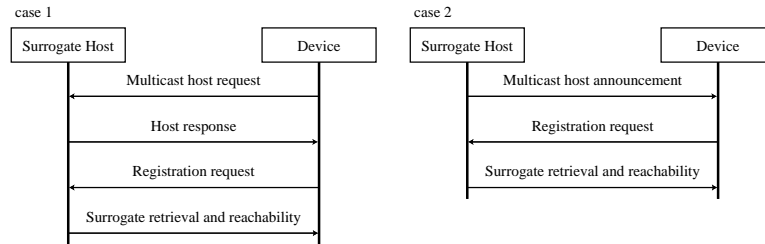


Figure 6.2: Case 1: Device enters a network where a surrogate host is already present. Case 2: A surrogate host is started in a network with devices present.

gate could communicate back to the device using a private protocol suited for the device.

When the surrogate is loaded and activated, it is its responsibility to determine the existence of the device. If the device cannot be reached, the surrogate is required to deactivate it by telling the surrogate host to dispose it. The device's resources in the surrogate host will then be reclaimed. Any remote resources, such as lookup registration, must be released as well.

The surrogate mechanism is visualised in four steps. The example includes a printer service and a cellular telephone. The link between the telephone and the surrogate host is Bluetooth.

Step 1: The Bluetooth device discovers a surrogate host and uploads a surrogate. The Jini service joins the Jini network. These two actions are totally asynchronous.

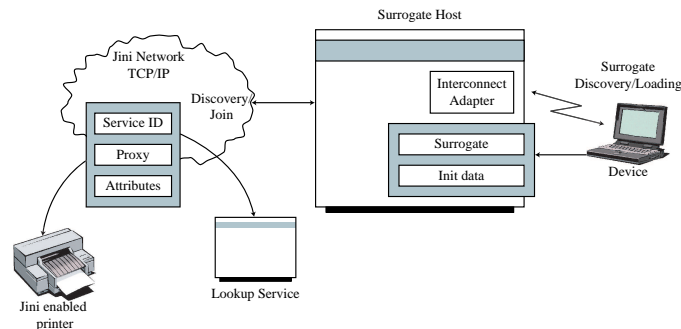


Figure 6.3: Device makes discovery of a surrogate host and uploads its surrogate

Step 2: The surrogate is activated in the context of the surrogate host. The surrogate makes a Jini discovery and lookup.

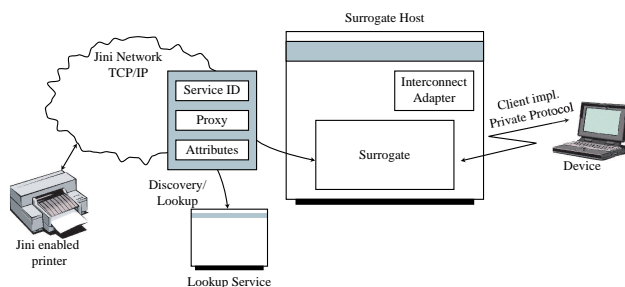


Figure 6.4: Surrogate makes service discovery and lookup

Step 3: The surrogate notifies the device about the discovered service through a private protocol implemented by the device. The device selects the service and gets the jad file for the required classes. It then dynamically downloads the Java bytecode for these classes.

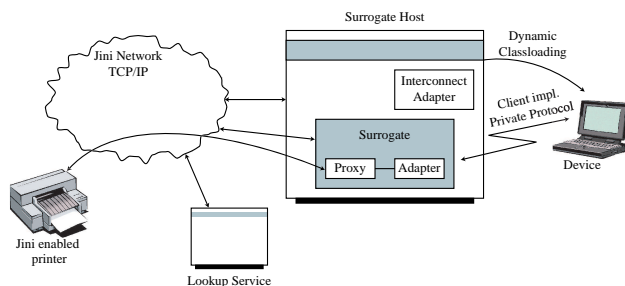


Figure 6.5: The device loading the required classes

Step 4: The device communicates with the adapter in the surrogate using a private protocol implemented by the service provider. The proxy is communicating with the service through RMI calls.

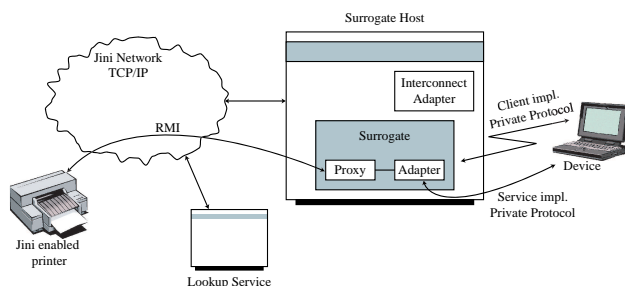


Figure 6.6: The device using the service

- Benefits: Jini's surrogate architecture was designed to connect devices not being able to run Jini to the Jini network. The devices can either be connected to the same TCP/IP based network as the target Jini network or to another non TCP/IP based network like Bluetooth.

Small Bluetooth devices (2nd generation devices) are not able to run Jini because of their limited JVM does not support RMI and serialisation. However, the surrogate host can solve the problem by bridging the two physical networks.

- Disadvantages: As mentioned, the Bluetooth interconnect specification is not available yet. The relative project for wireless protocols could not be found at the Jini project website [20]. It seems as if there is not to much effort going on. However, one version of Zucotto's Xpresso processors is equipped with Bluetooth chips. This processor does not make sense unless the interconnectivity to the surrogate host is available. Therefore, Zucotto is probably interested in pushing the specification.

6.2 Jini over Bluetooth

If Jini could be placed on top of the Bluetooth architecture, Jini devices would be able to communicate via the Bluetooth network. In addition, the Bluetooth range could be widened because Jini devices located outside the piconet could be connected via TCP/IP. Figure 6.7 shows an overview for this approach.

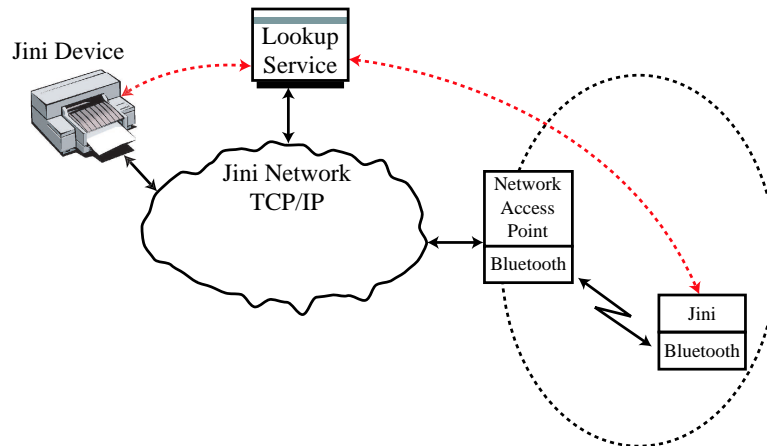


Figure 6.7: A Bluetooth device communicating with a Jini device via lookup service

However, Jini requires an underlying TCP/IP stack and at least the Java 2 Standard Edition. There are two possibilities to enable TCP/IP transport over Bluetooth. Bluetooth 1.0 provides only a LAN access profile [8]. With the aid of this profile, it is possible to layer IP over PPP (Point to Point Protocol). PPP is a packet-oriented protocol whereas RFCOMM (Serial Cable Emulation Protocol) expects serial data streams. Therefore, the PPP layer must use the serialisation mechanisms. Since June 2001, there is a new protocol called BNEP (Bluetooth Network Encapsulation Protocol) [21]. BNEP is defined by the PAN profile [18] (see Figure 6.8). It enables to layer IP directly over L2CAP. In the following subsections, solutions with both profiles are investigated.

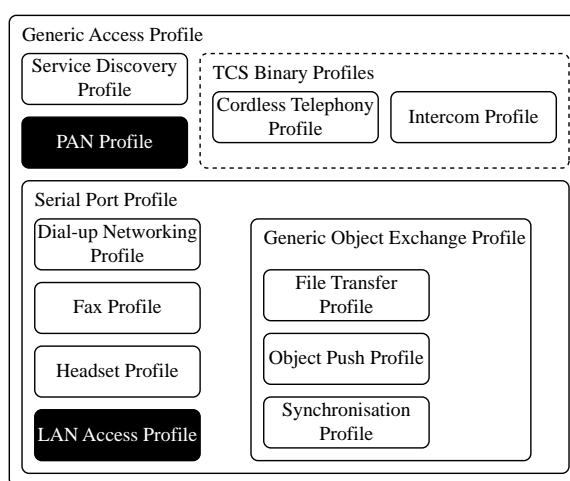


Figure 6.8: Bluetooth profile dependencies

6.2.1 Solution with LAN Access Profile

- Idea: Consider a Bluetooth piconet. Within radio range, there are at least one Bluetooth client and a Bluetooth LAN Access Point (LAP). The LAP is providing a PPP/RFCOMM/L2CAP service (see Figure 6.9) and is connected directly to at least one wired IP backbone (Jini network). It provides access to up to seven active client nodes per piconet. Once connected, the clients will operate as if they were connected to the LAN via dial-up networking. Note that the LAP can be located at a point where several piconets overlap. As a member of a scatternet, it has access to all the devices within range. However, since the piconets are not synchronised, the LAP cannot be active in more than one piconet simultaneously.

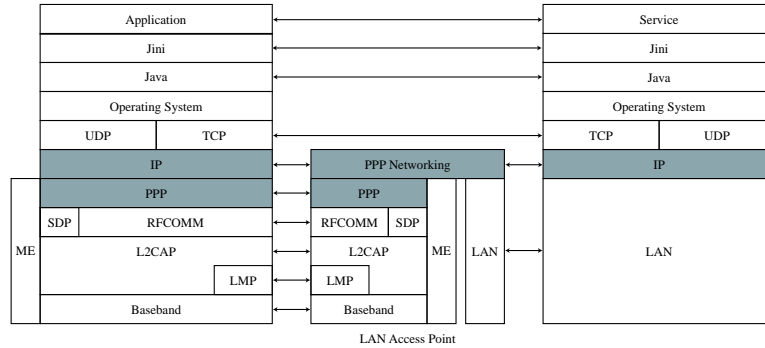


Figure 6.9: Jini over Bluetooth using the LAN profile

The subsequent steps describe the necessary interactions between a client and the LAP:

Step 1: The client must find and select an appropriate LAP.

Step 2: If there is no existing baseband link, the client requests a baseband physical link with the selected LAP.

Step 3: The client activates a PPP/RFCOMM/L2CAP connection. A PPP application is started to attempt to establish a connection to the selected LAP.

Step 4: A suitable IP address is negotiated between the LAP and the client as described below.

Step 5: IP traffic can now flow across the PPP connection.

Step 6: At any time, the client or the LAP may terminate the PPP connection.

PPP in this profile is similar to PPP operation in normal dial-up networking. It starts as soon as the RFCOMM link is established. The LAP exports a PPP Server interface. One implementation of a LAP could contain a PPP Server. Alternatively, the LAP could be some kind of proxy where PPP packets are transferred to or from a PPP server somewhere else on the network.

The PPP layer in the device will enable an IP interface when the Internet Protocol Control Protocol (IPCP) link has been established and a suitable IP address has been negotiated. Typically, the device will only have one PPP session in operation and only need one IP interface.

The device will require an IP address in order to operate on the LAN. Current PPP implementations allow three possibilities:

1. The IPCP option is used to specify a preconfigured IP address. If this address is not suitable on the LAN, the IPCP link will not be established.
 2. The IPCP option is used to request a suitable IP configuration from a PPP Server.
 3. The IPCP mobile IP options (see Section 5) are used to request a specified IP configuration. When moving between access points within the same LAN, it may be advantageous for the device to continue using the same IP configuration.
- Benefits: PPP provides user-based authentication, encryption, data compression and multiprotocol facilities. PPP over RFCOMM has been chosen as a means of providing LAN access for Bluetooth devices because of the large installed base of devices equipped with PPP software. Moreover, PPP carries IPv4, IPv6 and other protocols. It solves (to a reasonable degree) the IP address assignment problem in ad-hoc contexts.
 - Disadvantages: PPP on an emulated serial channel is not the most efficient and flexible solution. However, alternatives like IP directly on Bluetooth link layer or Ethernet encapsulation were discarded in Bluetooth version 1.0.

The LAN access profile does not deal with LAN emulation or ad-hoc networking. This fails to exploit the point-to-multipoint capabilities of the Bluetooth core and in particular the point-to-multipoint channels that L2CAP can provide. It converts a broadcast medium into a non-broadcast medium and leads to inefficiencies in IP multicasting and broadcasting. This introduces complications into IP networking functions that expect LANs to have broadcast media.

The PPP protocol can only be used to transport IP traffic to and from the clients to the LAP and out to the wired network. Therefore, clients can only communicate with each other via the LAP using IP forwarding.

Finally, it takes a long time for a PPP connection to be set up due to the authentication and authorisation functions.

6.2.2 Solution with PAN Profile

- Idea: Consider a Bluetooth piconet. Within radio range, there is at least one Bluetooth client and a Bluetooth Network Access Point (NAP). A Bluetooth device that supports the NAP service is a device that provides some of the features of an Ethernet bridge to support network services (see Figure 6.10). The device with the NAP service

forwards Ethernet packets between each of the connected clients, referred to as Personal Area Networking (PAN) users. The NAP and the PAN user exchange data using the Bluetooth Network Encapsulation Protocol (BNEP). The NAP has an additional network connection to a Jini network.

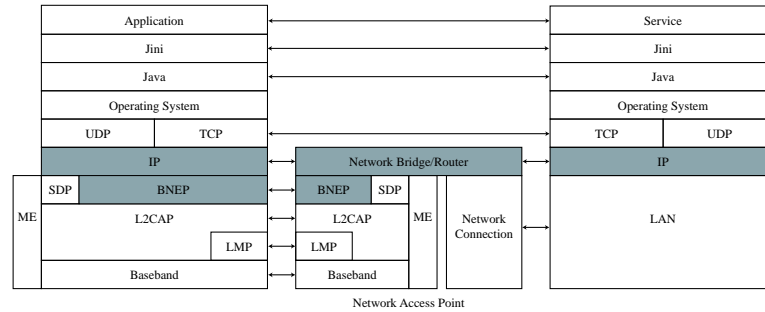


Figure 6.10: Jini over Bluetooth using the PAN profile

The subsequent steps describe the necessary interactions between a PAN user and the NAP:

Step 1: In order to find a NAP within radio range, the PAN user uses an application to inquire for nearby devices and then uses SDP to retrieve records that support the NAP service.

Step 2: If there is no existing Bluetooth connection, the PAN user requests a Bluetooth connection with the selected NAP.

Step 3: Once a connection is established, the PAN user creates a L2CAP channel for BNEP and uses the BNEP control commands to initialise the BNEP connection and setup filtering of different network packet types. The NAP stores all network packet type filters in a packet filter database, which maintains a set of packet filters for each connection.

Step 4: Ethernet traffic can now flow across the link. The PAN uses the services provided by the remote network such as obtaining an IP address by using DHCP. The NAP forwards the appropriate Ethernet packets to each of the connected PAN users and over the NAP network connection. This is a similar behaviour as a network hub.

The PAN profile describes how two or more Bluetooth devices can form an ad-hoc network and how the same mechanism can be used to access a remote network (e.g. a Jini network) through a NAP. A NAP can be a traditional LAN data access point. The PAN profile is dependent on the Generic Access Profile (GAP) (see Figure 6.8).

Support for the Internet Protocol (IP) is the major focus on the PAN profile. IP is described by a set of RFC documents defining its usage. The required RFCs, address assignment and name resolution techniques required to enable IP over Bluetooth are specified in the PAN profile specification [18].

The BNEP protocol is a new extension of the Bluetooth stack. The specifications [21] are available since June 2001. BNEP encapsulates packets from various networking protocols, which are transported directly over the Bluetooth L2CAP protocol. L2CAP provides a data link layer for Bluetooth.

BNEP removes and replaces the Ethernet header of a packet with the BNEP header. The Ethernet payload remains unchanged (see Figure 6.11). Finally, both the BNEP header and the Ethernet payload are encapsulated by L2CAP and are sent over the Bluetooth media.

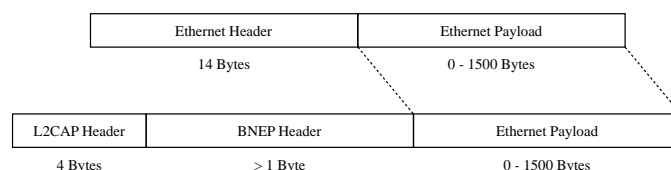


Figure 6.11: BNEP with an Ethernet packet payload sent using L2CAP

- **Benefits:** The PAN profile was primary written for IP support in Bluetooth devices. PAN networking (see Figure 6.12) enables multiuser collaboration by creating ad-hoc IP based personal area networks for data, voice, video and other forms of communications. Using PAN for ad-hoc network does not require the existence of any special purpose products or infrastructure, but it will interoperate with such devices if they exist. The network solution uses L2CAP to provide enhanced features and performance.
- **Disadvantage:** Only the newest Bluetooth devices will provide the BNEP protocol.

6.2.3 Solution with Surrogate Host

- **Idea:** If a Bluetooth device is not able to run at least Java 2 Standard Edition, Jini cannot be executed on the device. Especially small Bluetooth clients like cell phones or PDAs have only a limited Java Edition and therefore a simple JVM. It is desirable that such devices can also join a Jini community.

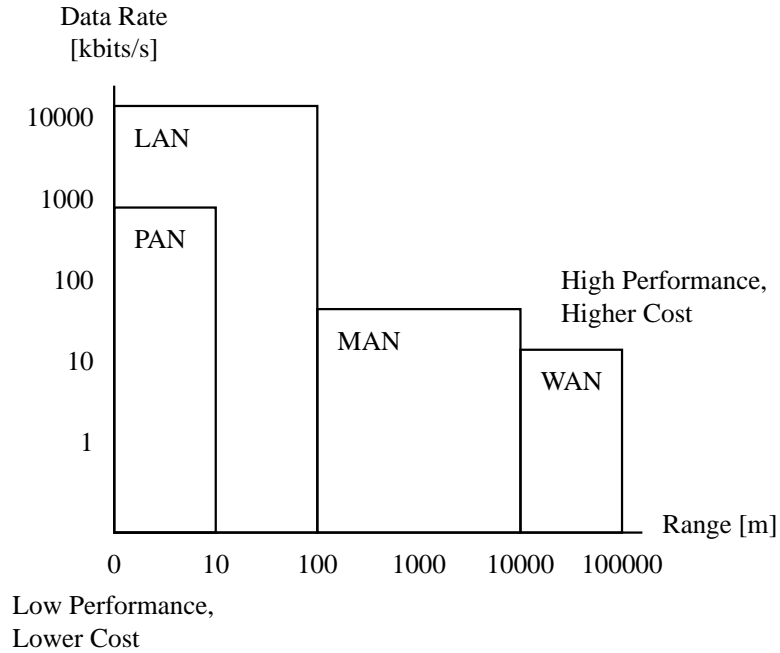


Figure 6.12: Comparison of wireless networks

In order to achieve this aim, a surrogate host [11] is required. The surrogate host must be a participant of the Jini network. Moreover, it should be connected (via TCP/IP) with a LAP/NAP within a Bluetooth piconet (see Figure 6.13). Since the IP interconnect specification [13] is already available, it is possible to link the LAP/NAP with the surrogate. As soon as a Bluetooth connection between the LAP/NAP and a limited device has been established, the client can access Jini's services via LAP/NAP and the surrogate host.

- **Benefits:** The above mentioned solutions (IP over PAN or LAN profiles) can also be used for limited devices with the aid of a TCP/IP connected surrogate host. In a typical Bluetooth environment, the majority of devices does not provide a Java Edition enabling Jini to run on top of the Bluetooth stack. Therefore, the surrogate solution makes IP based solutions even more attractive.
- **Disadvantages:** Because two devices (LAP/NAP and surrogate host) have to be located between the Bluetooth and the Jini networks, more effort has to be made in order to physically build up the environment. Besides, this approach has advantages and weaknesses corresponding to the chosen underlying profile (PAN or LAN).

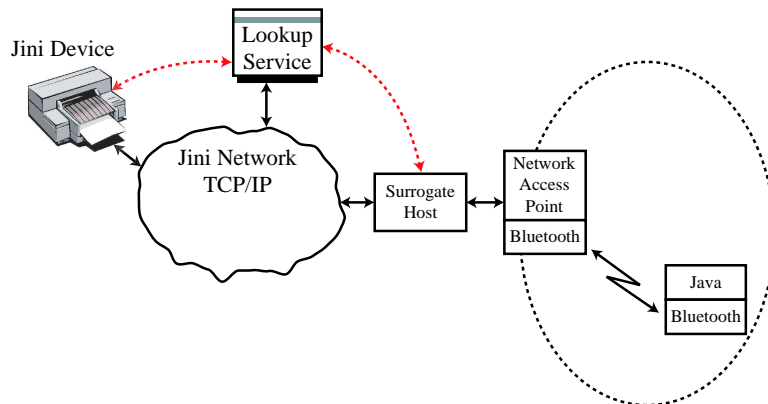


Figure 6.13: A surrogate host connecting a Bluetooth device with a Jini device via lookup service

6.3 Bluetooth-Jini Bridge

- Idea: This approach assumes a bridge in order to interconnect a Jini community to a Bluetooth network. A bridge device translates Jini into the context of Bluetooth and vice versa. The binding between Jini services and Bluetooth devices is local to the bridge. Therefore, the end nodes do not realise this association. The bridge has to be located within the Bluetooth piconet and must be connected to the Jini TCP/IP network.

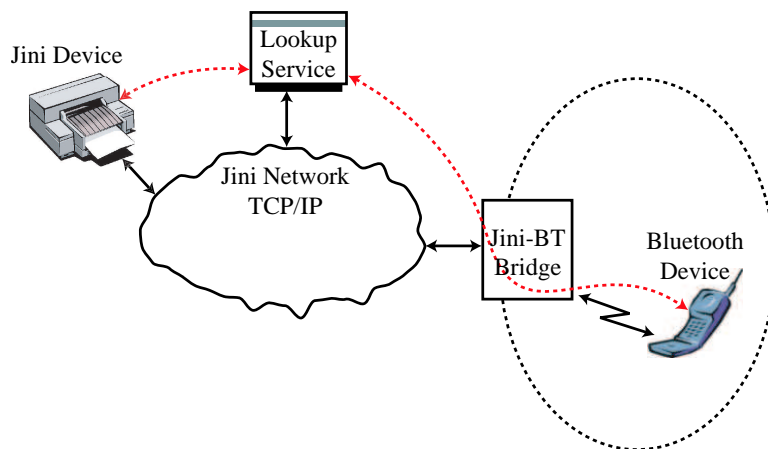


Figure 6.14: A Bluetooth-Jini bridge connecting the two physical networks

- **Benefits:** By using a Bluetooth-Jini bridge, regular Bluetooth devices are able to communicate with Jini devices and vice versa. There is no need for an IP stack, surrogate host or a Bluetooth interconnectivity. Moreover, the bridge is transparent. As a consequence, it allows an integration of Bluetooth to Jini as well as an integration of Jini to Bluetooth.
- **Disadvantages:** To build a bridge that translates all mechanisms from Jini to Bluetooth, both systems should provide similar functionalities. Jini needs leasing, event and transaction interfaces. Leasing helps Jini communities to ensure stable, self healing and resilient networks in spite of inevitable network failures. Events allow Jini communities to asynchronously communicate. Transactions help to successfully complete complex operations and provide resolutions for incomplete operations. These important mechanisms are not available in a Bluetooth environment. It might be possible to introduce Bluetooth profiles for leasing, events and transactions in order to bridge all the Jini features to Bluetooth. However, discovery, join and lookup interactions can be realised with the standard Bluetooth profiles.

Chapter 7

Development of a Bluetooth-Jini Bridge

7.1 Preliminary Remarks

Our approaches regarding the surrogate host or the IP based network transport over Bluetooth can run without additional implementation. On the other hand, no bridge implementation from Bluetooth to a Jini could be found. However, Salutation's approach to combine their architecture with Bluetooth may help to develop a prototype of a Bluetooth-Jini bridge. We will concentrate on basic SDP functions.

7.2 Requirements

The Bluetooth bridge device should interact with other devices with Bluetooth wireless communications as yet another piconet member. The bridge device is intended to enhance the functionality of devices within a piconet by extending their range in service discovery and device control beyond their piconet. Otherwise, Jini devices that are not equipped with Bluetooth connectivity must be able to access Bluetooth services via the bridge. Thus, the bridge should also be a member of the Jini TCP/IP network.

Jini allows interaction between lookup services of different subnets. As a consequence, a device can also discover services that are not located in the same subnet. On the other hand, no routing mechanism is provided for Bluetooth scatternets. A Bluetooth device can only discover devices within the same piconet. Thus, every piconet that wants to be connected to a Jini network needs its own bridge. It is important to implement the bridge in a small portable device, for example in a iPAQ. Otherwise, Bluetooth' ad-hoc network principle would be violated.

7.3 Bluetooth Service Discovery Protocol (SDP)

SDP [7] provides a means for client applications to discover the existence of services provided by server applications as well as the attributes of those services. The attributes of a service include the type or class of service offered and the mechanism or protocol information needed to utilise the service. SDP involves communication between a SDP server located on a remote device and a SDP client located on a local device. The server maintains a list of service records (see Figure 7.1) that describe the characteristics of services associated with the server. Each service record contains information about a single service. A client may retrieve information from a service record maintained by the SDP server by issuing an SDP request. All of the information about a service is contained within a single service record. The service record consists of a list of service attributes. Each service attribute describes a single characteristic of a service.

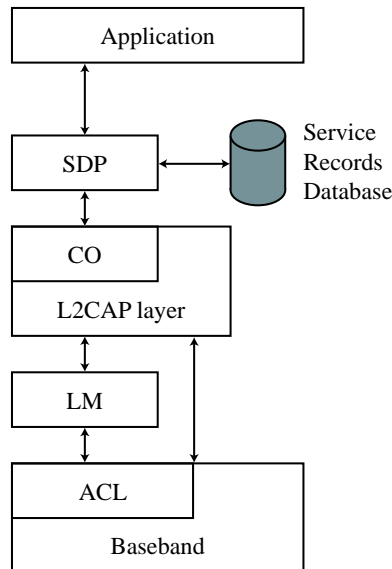


Figure 7.1: Bluetooth stack with service record database

SDP uses a request/response model where each transaction consists of one request Protocol Data Unit (PDU) and one response PDU (see Figure 7.2). Generally, each type of request PDU has a corresponding type of response PDU. However, if the server determines that a request is improperly formatted or for any reason the server cannot respond with the appropriate PDU type, it will respond with an `SDP_ErrorResponse` PDU.

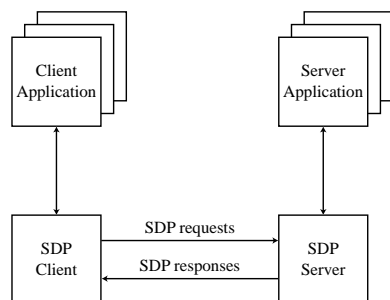


Figure 7.2: SDP client-server interaction

The SDP uses the connection-oriented transport service in L2CAP, which in turn uses the baseband asynchronous connectionless (ACL) link to transmit the SDP PDUs.

The SDP specification does not define an application programming interface (API) for SDP. Therefore, a standard method to access the functions of the SDP is missing. There is a Java protocol stack for Bluetooth called BlueJ [22] available. However, since BlueJ is not documented (yet), it is not useful.

7.3.1 Protocol Data Unit Types

- **ServiceSearch Transaction:**
The SDP client generates a `SDP_ServiceSearchRequest` to locate service records that match the service search pattern composed of the UUID associated with the desired profile. After receiving this request, the SDP server examines its service record database and returns a `SDP_ServiceSearchResponse` containing one or more 32-bit service record handles. The corresponding service records contain the UUID of the desired profile.
- **ServiceAttribute Transaction:**
The SDP client generates a `SDP_ServiceAttributeRequest` to retrieve specified attribute values from a specific service record. The service record handle of the desired service record and a list of desired attribute IDs to be retrieved from that service record is supplied as parameters. Usually, a `ServiceAttributeRequest` is inquired after a `ServiceSearch Transaction`.
- **ServiceSearchAttribute Transaction:**
The `SDP_ServiceSearchAttributeRequest` transaction combines both the `SDP_ServiceSearchRequest` and the `SDP_ServiceAttributeRequest` into a single request. As parameters, it contains both a service search pattern and a list of attributes to be retrieved from service records that

match the service search pattern. Using `SDP_ServiceSearchAttributeRequest` may reduce the total number of SDP transactions, particularly when retrieving multiple service records.

- **Browsing for Services:**
Normally, a client searches for services based on some desired characteristics of the services. However, sometimes it is desirable to discover which types of services are described by an SDP server's service records without any a priori information about the services. This process of looking for any offered services is termed browsing. In SDP, the mechanism for browsing for services is based on an attribute shared by all service classes. This attribute is called the `BrowseGroupList` attribute. Each attribute represents a browse group with which a service may be associated for the purpose of browsing. When a client desires to browse an SDP server's services, it creates a service search pattern containing the attribute that represents the root browse group.

7.3.2 Service Attributes

A service is any entity that can provide information, perform an action, or control a resource on behalf of another entity. A service may be implemented as software, hardware or a combination of software and hardware.

All the information about a service maintained by a SDP server is contained within a single service record. The service record consists of a list of service attributes.

A service attribute has two components: an attribute ID and an attribute value. An attribute ID is a 16-bit integer that distinguishes each service attribute from other attributes within a service record. The attribute value is a variable length field whose meaning is determined by the attribute ID associated with it and by the service class of the service record in which the attribute is contained.

Only two attributes are required to exist in every service record instance. They are the `ServiceRecordHandle` and the `ServiceClassIDList`. All other attributes are optional within a service. Below, important attributes supported by SDP are listed.

Attribute Name	Attribute ID	Attribute Value Type
<code>ServiceRecordHandle</code>	0x0000	32-bit unsigned integer

Description: A service record handle is a 32-bit number that uniquely identifies each service record within a SDP server. Each handle is unique only within each SDP server.

Attribute Name	Attribute ID	Attribute Value Type
ServiceClassIDList	0x0001	Data Element Sequence

Description: The ServiceClassIDList consists of a data element sequence in which each data element is a UUID representing the service classes that a given service record conforms to.

Attribute Name	Attribute ID	Attribute Value Type
ServiceID	0x0003	UUID

Description: The ServiceID is a UUID that universally and uniquely identifies the service instance described by the service record.

Attribute Name	Attribute ID	Attribute Value Type
ProtocolDescriptorList	0x0004	Data Element Sequence

Description: The ProtocolDescriptorList describes one or more protocol stacks that may be used to gain access to the service described by the service record.

Attribute Name	Attribute ID	Attribute Value Type
BrowseGroupList	0x0005	Data Element Sequence

Description: The BrowseGroupList consists of a data element sequence in which each element is a UUID that represents a browse group to which the service record belongs.

Attribute Name	Attribute ID Offset	Attribute Value Type
ServiceName	0x0000	String

Description: The ServiceName is a string containing the name of the service represented by a service record.

Attribute Name	Attribute ID Offset	Attribute Value Type
ServiceDescription	0x0001	String

Description: The ServiceDescription is a string containing a brief description of the service. It should be less than 200 characters in length.

7.4 Jini Lookup Service and Discovery

The Jini lookup service [23] is an important component of Jini. It provides an easy way for services to register (discovery and join), and it allows clients to locate their services within the Jini network. Before communicating with a service, a client has to download a *ServiceRegistrar* via the discovery protocols. These protocols enable services and clients to locate lookup services. The *ServiceRegistrar* object, which implements the interface, enables the client to interact with the lookup service. By building a *ServiceTemplate* and passing it with one of the two *lookup()* methods to the lookup service, the client can find a desired service.

7.4.1 Jini API Description

- *lookup()*:
There are two different *lookup()* methods. The one-parameter method returns a single object item, whereas the two-parameter form returns multiple matches.
 - One-parameter *lookup()*:
The one-parameter method takes a *ServiceTemplate* and returns one matching service object chosen randomly from all the matches or *null* if no matching service was found. Because this method returns just one object, the clients can minimize the amount of objects that are downloaded. The returned object is selected arbitrarily and is not identified by a service ID or described by associated attribute sets, so the client must be confident that *any* matching service object will suffice. If a received object file cannot be deserialised, an *UnmarshallException* is thrown.
 - Two-parameter *lookup()*:
The two-parameter method takes a *ServiceTemplate* and an integer *maxMatches* and returns at most *maxMatches* items matching the template. The returned items array is *null* if *maxMatches* is zero or if no matching services were found.
- browsing methods:
Jini provides three functions in order to browse the lookup server. The return value of the browsing is dependent on the parameters used by the *ServiceTemplate*.
 - *getServiceTypes()*:
This method takes a *ServiceTemplate* and a *string* prefix. It returns an array of *class* instances representing the most specific types (class or interface). These service objects are neither equal nor a superclass of any of the types specified in the template

and have names that start with the specified prefix. The service objects for whom *class* instances are returned are instances of the types passed in the template.

- `getEntryClasses()`:
This method takes a *ServiceTemplate* and returns an array of *class* instances that represent the most specific classes of entries for the service items that match the template.
- `getFieldValues()`:
This method takes a *ServiceTemplate*, an integer index and a *string* field name and returns an array of *objects* for the named field of all instances of the entry that appears in the *ServiceTemplate*'s *Entry []* array at any matching service item's passed index.
- `notify()`:
In addition to the *lookup()* and the browsing methods, the *ServiceRegistrar* interface also contains a *notify()* method that informs clients when a new service registers or when a registered service changes or unregisters. The client must invoke *notify()* with the same *ServiceTemplate* as for the *lookup()* method to register itself. It receives a distributed event whenever a service that matched the passed template changed its state or a new service that matches the template has registered. These transitions are described by the three following transition parameters.
 - `TRANSITION_MATCH_MATCH`:
Indicates that at least one service item matched the template before and after an operation (state change of a service).
 - `TRANSITION_MATCH_NOMATCH`:
Indicates that at least one particular service matching the template before the operation, does no longer match the template.
 - `TRANSITION_NOMATCH_MATCH`:
Indicates that a service has changed its state or a new service has registered and matches the template. To be notified as soon as a new service has been added, a template that matches any service has to be passed to the *notify()* method.

7.4.2 Service Attributes

As mentioned above, Bluetooth provides a wide range of well-defined attributes in order to specify a particular service type. Jini does have similar attributes. Each attribute is an own Java class and consists of a fix serial version UID and at least one value. The most important attributes are listed below.

Attribute Name	Attribute Value Type
ServiceName	string

Description: The ServiceName is a string containing the name of the service.

Attribute Name	Attribute Value Type
Comment	string

Description: Additional information or descriptions can be found in the Comment attribute.

Attribute Name	Attribute Value Type
ServiceInfo	string list

Description: The ServiceInfo contains a list of entries like manufacturer, model, serial number, vendor or version.

Attribute Name	Attribute Value Type
ServiceType	long

Description: The ServiceType attribute is a UID referring to a particular service.

Attribute Name	Attribute Value Type
StatusType	string list

Description: The StatusType reflects warnings, error messages or other notices.

In addition to the mentioned attributes, there is a ServiceTemplate in order to define search criteria for a Jini lookup service. This template consists of three public fields:

- Entry[] attributeSetTemplates:
Describes attributes of a desired service like locations, names or descriptions.
- ServiceID serviceID:
Uniquely identifies a service and attributes which must exactly match the attributes uploaded in the service item by the service provider.

- `Class[] serviceTypes`:
Describes the type of the desired service such as a printer in a Jini network.

The template can contain wildcards for any of these fields. Such a wildcard will match any service. The template class has no methods, it is basically a struct-like container for lookup service queries.

7.5 Correlations Between Service Parameters

As described in the previous subsections, Bluetooth and Jini use similar mechanisms in order to handle and represent their services. This is a good basis for further considerations. To enable seamless communication between Bluetooth and Jini, translations are required. Table 7.1 summarises corresponding service parameters of the two technologies.

	Bluetooth	Jini
<code>Service_Description</code>	Profile	Java Interface
<code>Service_Registry</code>	Service Record Database	Lookup Service
<code>Service_Discovery</code>	SDP	Unicast/Multicast Discovery
<code>Service_Search_Pattern</code>	<code>ServiceID</code> , <code>Service_Record_Handle</code> , <code>AttributeID_List</code>	<code>Service_Template</code> (<code>ServiceEntry</code> , <code>ServiceID</code> , <code>ServiceTypes</code>)
<code>Service_Response</code>	<code>Service_Record_Handle_List</code> , <code>AttributeList</code>	<code>ServiceProxy</code>
<code>Service_Limitation</code>	<code>Maximum_Service_Record_Count</code>	<code>maxMatches</code>
<code>Found_Services</code>	<code>Total_Service_Record_Count</code>	<code>totalMatches</code>
<code>Service_Invocation</code>	L2CAP	RMI

Table 7.1: Matching Bluetooth and Jini service parameters

7.6 Design Model

In order to explain the chosen design model, a simple Jini-Bluetooth scenario is illustrated. A Jini TCP/IP based network is connected with a Bluetooth piconet with the aid of a Jini-Bluetooth bridge. The bridge device supports both Jini and Bluetooth. Therefore, it must be connected to the TCP/IP network and simultaneously be located within range of the Bluetooth piconet (see Figure 7.3). The Jini service discovers and joins the lookup server whereas the Bluetooth device provides its service(s) in a own service record database.

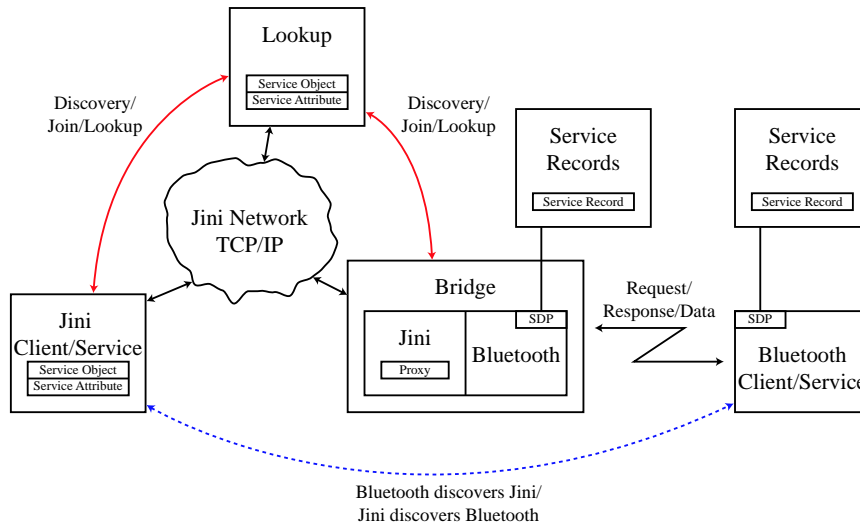


Figure 7.3: Bridge device embedded in a Jini-Bluetooth environment

Two procedures have to be considered. A Bluetooth client discovers a Jini service and uses this service or a Jini client discovers and uses a Bluetooth service. Both directions are equally important. However, since the described system is not symmetric (no central lookup service for the Bluetooth piconet), it is much easier to realise the former process. Thus, both directions are specified, but the implementation is only unidirectional.

Bluetooth client discovers a Jini service:

Step 0:

The Jini device discovers and joins the lookup service.

Step 1:

The Bluetooth client makes a ServiceSearch request. This request is sent to all Bluetooth devices within range. In particular, the bridge device is inquired as well.

Step 2:

The bridge device translates the SDP request into a Jini lookup request.

Step 3:

The registered Jini device is found in the lookup server.

Step 4:

The proxy of the Jini service is downloaded to the bridge.

Step 5:

The Jini service is recorded in the Bluetooth record database.

Step 6:

The bridge responds with a ServiceSearch response. The Bluetooth client receives the service handle of the desired service.

Step 7:

The proxy object on the bridge interacts with its linked Jini service as well as with the Bluetooth client. Finally, the Jini service can be accessed.

Jini client discovers a Bluetooth service:

Step 0:

For every Bluetooth service type, there must be a suitable Jini service object stored on the bridge.

Step 1:

The Jini client makes a lookup request. This request is sent to all lookup servers within its discovery range.

Step 2:

The bridge device translates the lookup request into a Bluetooth ServiceSearch request.

Step 3:

The recorded Bluetooth service is found in the record database of a suitable Bluetooth device.

Step 4:

The returned service classifications are translated into the context of Jini.

Step 5:

The Bluetooth service is registered to the Jini lookup server by uploading the service object from the bridge.

Step 6:

The proxy object is downloaded to the Jini client.

Step 7:

The Jini proxy object starts to communicate with its service backend through the bridge device.

Step 8:

Finally, the Bluetooth service can be accessed.

7.7 Implementation

7.7.1 Jini Service

Since no Jini devices are available yet, we had to implement a simple Jini service. Advanced mechanisms like leases were omitted because they are not necessary to demonstrate a lookup scenario. However, the Jini application requires at least the subsequent classes:

- Service class interface:
The service class interface basically describes what a service can do. In this simple Jini application, the interface provides two methods, *send()* and *receive()*. These functions represent a data transfer service where data can be sent and received. Actually, just a string is returned by calling the method. The service proxy implements the interface and a client uses the interface whenever it searches for a service which also implements this interface and provides the desired methods.
- Jini service:
The service implements the methods from the service class interface. Additionally, it takes care of the discovery/join process, publishes the proxy object and handles the leasing for the proxy to assure that the lookup services holding the proxy do not discard it. The application is called the “wrapper” and has a *main()* method which manages all the mentioned tasks. The service itself is the proxy object. Such an approach can be used if the methods return a string and no communication between the client and any device or a backend process is needed.

Moreover, a HTTP server, Java Remote Method Invocation (RMI) and a Jini lookup service are necessary to build up a executable Jini environment.

The implemented code as well as the shell scripts to compile and start the services are available in Appendix E. For further information refer to [24], [25] and [26].

7.7.2 Bluetooth Devices and Tools

Two Bluetooth devices are required for our scenario. The bridge must be equipped with Bluetooth hardware as well as TCP/IP connection via wireless card or network cable. An additional PC is used as Bluetooth client. Its Bluetooth chip is connected via USB.

Both machines run Linux. There is a Bluetooth driver for Linux called *BlueZ* [30]. BlueZ provides support for core Bluetooth layers and protocols. Currently, BlueZ consists of BlueZ core, *hci_uart*, *hci_usb*, *hci_pemcia* and virtual hci drivers, L2CAP protocol module, SCO module and several

additional utilities. The BlueZ protocol stack is interfacing to the Linux socket layer, providing a new address family. An application can send any `hci` command to a device and receive events in return.

There is a tool called *hcitool* which allows to use the `hci` interface. It works properly and is helpful to verify applications in the user space.

The SDP package provides a tool called *sdptool*. With the aid of this tool, other devices can be found and browsing can be performed. The client uses this function in order to make a `ServiceSearch` request. On the other hand, the *SDP daemon* running on the bridge waits for requests, manages the service record database and responds the requests.

The SDP package is a new product. Updated versions are following quickly. The SDP code is rather erroneous. As a consequence, we had to make some modifications before we could use the provided tools. In addition, the daemon had to be extended in order to meet our demands.

The code of the modified tools is available in Appendix E.

7.7.3 Bluetooth to Jini Translator (Bridge)

This Java program is used as soon as a Bluetooth client interacts with a Jini service. It runs on the bridge device and must access the Bluetooth SDP layer as well as the Jini lookup service. In addition, Bluetooth attributes have to be translated to the Jini format. Finally, the respective Jini function has to be called.

Since both Bluetooth and Jini provide a wide range of possible services, a general implementation would be very elaborate and extensive. Thus, we focused on implementing functions for a lookup scenario.

When the bridge application (see Figure 7.4) is started, the SDP daemon is activated with the aid of a command-line call. As soon as a Bluetooth client looks for a particular service, the daemon returns the wanted servicetype. In case of a `JINIFTP` service request, the bridge starts a discovery/lookup process. If the suitable service is registered to one of the lookup servers, its service object is returned and translated. Finally, the bridge notifies the client about the existence of the requested service.

The implemented code is available in Appendix E.

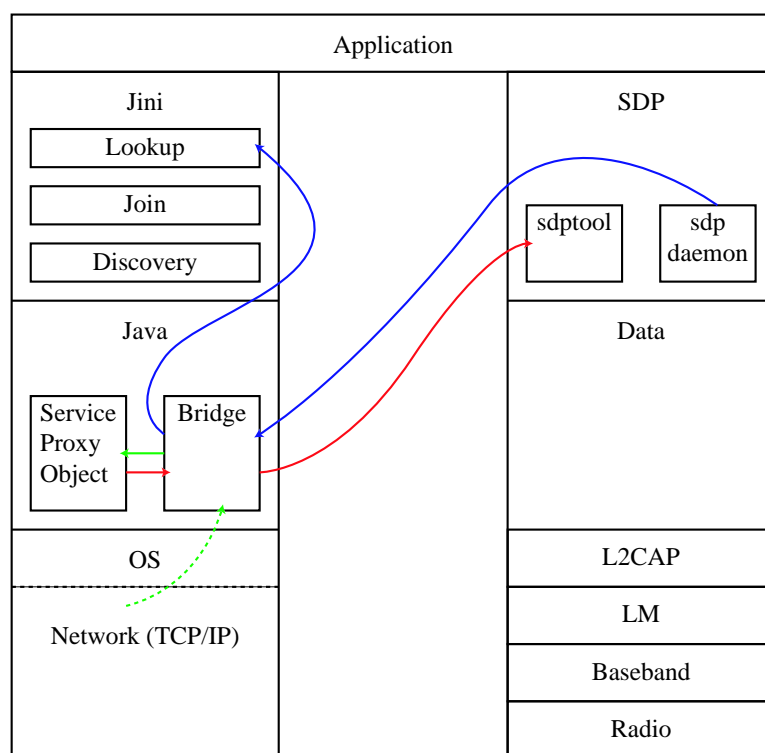


Figure 7.4: The Jini-Bluetooth bridge in more details

Chapter 8

Testing

8.1 Configuration and Setup

8.1.1 iPAQ

Compaq's handheld iPAQ runs Windows CE by default. However, in order to use the BlueZ Bluetooth stack, Linux is needed.

First, the Linux bootloader had to be installed on iPAQ. A helpful howto [27] can be found on the web. Then, the Linux root image was copied to the iPAQ. Next, the wireless LAN card had to be configured. It is very helpful to have this card working because all Linux packages can then be easily downloaded with the aid of `ipkg`, a tool to manage and install packages. Moreover, the iPAQ can be controlled from a PC via SSH instead of the tiring Hyperterminal and the COM interface. After having installed a graphical environment and some other tools, the iPAQ was properly running Linux.

In a further step, the drivers for Bluetooth (BlueZ) had to be installed. There are howtos available for the PC [28] as well as for the iPAQ [29].

8.1.2 Bluetooth on PC

As mentioned above, the BlueZ Bluetooth drivers [30] are available for Linux only. Therefore, the desktop PC had to be equipped with Linux as well. The following packages were configured and installed: `bluez-kernel`, `bluez-libraries`, `bluez-utilities` and `bluez-sdp`. Finally, the Bluetooth chip could be connected to the PC via USB. After having loaded the `hci_usb` module, the Bluetooth device was ready.

8.1.3 Jini on iPAQ and PC

Since Java and Jini programs are platform independent, any operating system offering a full JVM can be used in order to run the code. By adding the

appropriate Jini classpath and using a JVM that supports RMI, Jini based programs run properly.

8.2 Experimental Environment

8.2.1 PC Based Experiment

- Client:
The client makes a SDP search request. It must run Linux and the appropriate BlueZ packages need to be installed.
- Bridge:
The bridge translates the incoming SDP search request to a Jini lookup request. If the desired Jini service has already joined a lookup service within range of the bridge, the bridge sends a “found” response. It must run Linux. The BlueZ packages and Java/Jini have to be installed.
- Server:
The Jini service and the Jini lookup service are located on the same host. Since both the lookup service and the service need a HTTP server, two HTTP servers are running on different ports (8080, 8085). The server does not need the BlueZ packages. Java/Jini are platform independent. Therefore, any operating system can be used.

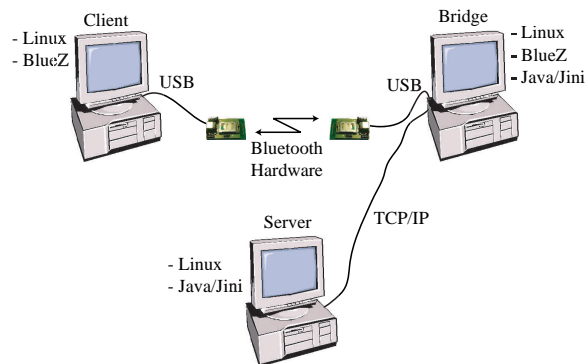


Figure 8.1: Experiment with three Desktop PCs

8.2.2 iPAQ Based Experiment

- Client:
The iPAQ client makes the same SDP search request as the PC client. While the PC needs an external Bluetooth device, the iPAQ is equipped with an internal Bluetooth chip.
- Bridge:
The bridge is equipped with a wireless LAN card. With the aid of this card, the bridge can access the TCP/IP network. The bridge is not necessarily located in the same subnet as the server.
- Server:
The lookup service on the server have to register to the lookup service on the bridge or vice versa unless they are located in the same subnet.

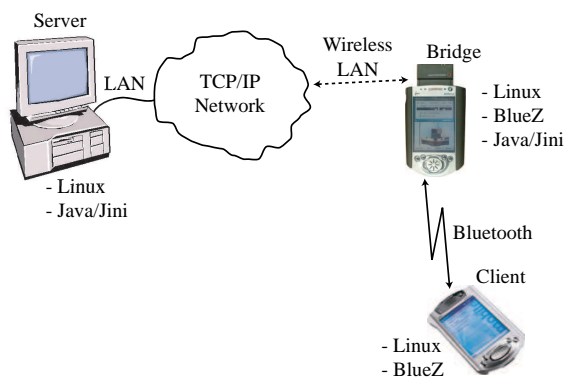


Figure 8.2: Experiment with two iPAQs and one Desktop PC

Chapter 9

Conclusions

9.1 Results

This thesis consists of four main parts. In the following, the result of every of these parts are summarised.

- **Analysis of related work:**
After a widespread investigation, it was found that there is neither an existing implementation nor a usable design of a Bluetooth-Jini bridge so far. However, some similar approaches exist (other technologies than Jini). By analysing them, some helpful conclusions could be drawn.
- **Concepts:**
Three approaches were investigated and designed. One uses Sun's surrogate architecture in order to connect a regular Bluetooth device to the Jini network. In addition, it was determined how Jini can be placed on top of the Bluetooth stack. Bluetooth' PAN and LAN profiles play an important role to realise this approach. Finally, the feasibility of a Bluetooth-Jini bridge was investigated. The bridge was chosen as initial point for the implementation.
- **Implementation of a bridge:**
After having compared the service description used in Bluetooth and Jini, a design model was developed providing discovery interaction from Bluetooth to Jini and vice versa. However, it was found that only requests from Bluetooth to Jini can be handled with reasonable effort. Therefore, the implemented bridge works only unidirectionally.
- **Testing:**
All the implementations were tested both on PCs equipped with Bluetooth hardware and on an iPAQ H3870. The tests were terminated successfully.

9.2 Outlook

This thesis can be used as a basis for further work. Some possible upgrading are outlined in the following.

- Bridging of service invocation:
Since it was focused on discovery interaction, no service invocations can be bridged so far. It would be worthwhile to add this feature in order to use real services.
- Implementation of Jini services:
Bluetooth provides several profiles (FAX, LAN, DUN, ..). To use them, appropriate Jini services have to be implemented.
- Bridge from Jini to Bluetooth:
So far, the bridge acts as translator from Bluetooth to Jini. It is desirable to implement the other direction as well. However, it will be difficult and elaborated to perform this task.

Appendix A

Used Software Tools

Java Software

Java 2 SDK 1.4.0 (see [5])
Blackdown JRE 1.2 (see [31])

Jini Software

Jini 1.2 on PC (see [3])
Jini 1.1 on iPAQ

Bluetooth Software

BlueZ-Kernel 2.1 (see [30])
BlueZ-Utills 2.0
BlueZ-Libs 2.0
BlueZ-SDP 0.4

Linux Software

Suse Linux 7.3, Kernel 2.4.10 on PC (see [32])
Debian Linux, Kernel 2.4.18 on PC (see [33])
Familiar Linux 0.5.2, Kernel 2.4.18 on iPAQ (see [34])

Applications

Forte 3.0 (see [35])
LaTeX (see [36])
GNU Emacs 21.1.2 (see [37])
Adobe Illustrator 9.0/Photoshop 6.0 (see [38])

Appendix B

Used Hardware

PC's

- Compaq iPAQ H3870 (see [39])
- 2 x PC Pentium P4 1000/866 MHz, 256 MB RAM

Bluetooth Chips

- Bluetooth ROK tester rev2, Ericsson
Testerboard developed at TIK [1]

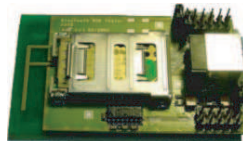


Figure B.1: ROK tester

- Compaq iPAQ H3870, internal CSR



Figure B.2: iPAQ H3870

Appendix C

System Requirements

Since two technologies (Jini and Bluetooth) are involved in our implementation, many tools are needed to run it. In addition, the iPAQ has a different architecture and lesser memory than a regular PC. Therefore, adapted software packages have to be used.

	PC	iPAQ
Operating System	Linux Kernel 2.4.x	Familiar Linux 0.5
Java	Java 2 Standard Edition	Blackdown 1.2 JRE
Jini	Version 1.2	Version 1.1
BlueZ	Kernel 2.1, SDP 0.4	Kernel 2.1, SDP 0.4
C Compiler	gcc	Cross Compiler/Skiff Cluster
Available Harddisk Space	~ 85 MB	~ 16 MB
Network	TCP/IP Network Card	Wireless LAN Card
Additional HW	Bluetooth Chip	Integrated Bluetooth HW

Table C.1: Minimal requirements to run the implemented code

Appendix D

Howtos

Be sure to adjust all Java/Jini classpaths, hostnames and IP numbers in the scripts before compiling or running any of the following applications. The folders client, service and service-dl must be generated before compiling the files.

D.1 Compile Sdptool and SDP Daemon

PC Version

- bash-2.05\$./configure
Run this script in the bluez-sdp-0.4 folder. It gathers all the information of the Linux system and generates the Makefile.
- bash-2.05\$ make
This command executes the Makefile located in the bluez-sdp-0.4 folder which compiles all the tools that come with the bluez-sdp package.

iPAQ Version

Because the iPAQ does not provide enough system resources to store a complete C/C++ compiler, the files from the bluez-sdp package have to be compiled differently. There are two methods to do that. Although the cross compiler is able to compile the code, the skiff cluster [40] is much simpler to utilise.

In order to start the skiff cluster, take a SSH shell and login as “guest” on ipaq[3-7].handhelds.org with password “<cr>”. Upload the bluez-sdp package to this host and compile it by executing the same two steps as on a normal PC. Finally, download sdptool, sdpd and libbluetooth.so to the home account.

D.2 Compile Java/Jini Files

JINIFTPService

```
bash-2.05$ ./compservice
```

Bridge

```
bash-2.05$ ./compbridge
```

JiniSDP

```
bash-2.05$ javac JiniSDP.class
```

D.3 Usage of the Implemented Applications

Sdptool

```
sdptool <command> [service] [bt_address]
```

- Search for a specific service without bt_address:
bash-2.05\$ sdptool search JINIFTP
- Search for a specific service with bt_address:
bash-2.05\$ sdptool search JINIFTP 00:02:5B:FF:06:67
- Browse without bt_address (no specific service):
bash-2.05\$ sdptool browse
- Browse with bt_address (no specific service):
bash-2.05\$ sdptool search NULL 00:02:5B:FF:06:67
- Add a specific service:
bash-2.05\$ sdptool add JINIFTP

SDP Daemon

```
sdpd [option]
```

- Start as daemon:
bash-2.05\$ sdpd
- Start in the foreground:
bash-2.05\$ sdpd -n

Bridge.class

runridge [service]

- Run the bridge without adding a Bluetooth service:
bash-2.05\$./runbridge
- Run the bridge and add a Bluetooth service:
bash-2.05\$./runbridge FAX

JiniSDP.class

runjinisdg <service>

- Make an SDP request for a specific service:
bash-2.05\$./runjinisdg JINIFTP

Appendix E

Implemented and Modified Software

E.1 Jini Service

```
/*
 * JINIFTPService.java
 *
 * Created on June, 2002
 */

/**
 * @author skasper, lbuehrer
 * @version 0.1
 * @comment JINIFTPService publishes a proxy that returns
 *          a string when asked by clients.
 */

package myService;

import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
```

```
import net.jini.discovery.LookupDiscovery;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceRegistration;
import java.util.Hashtable;
import java.io.IOException;
import java.io.Serializable;
import java.rmi.RemoteException;
import java.rmi.RMI SecurityManager;

// This is the proxy object that will be downloaded
// by clients. It's serializable and implements
// the JINIFTPServiceInterface.

class JINIFTPServiceProxy implements Serializable, JINIFTPServiceInterface {
    public JINIFTPServiceProxy() {
    }
    public String send(){
        return "send ...";
    }
    public String receive(){
        return "receive ...";
    }
}

// JINIFTPService is the "wrapper" class that
// handles publishing the service item.

public class JINIFTPService implements Runnable {
    protected final int LEASE_TIME = 10 * 60 * 1000; // 10 minute leases
    protected Hashtable registrations = new Hashtable();
    protected ServiceItem item;
    protected LookupDiscovery disco;

    // Inner class to listen for discovery events

    class Listener implements DiscoveryListener {
        // Called when we find a new lookup service.
        public void discovered(DiscoveryEvent ev) {
            System.out.println("discovered a lookup service!");
            ServiceRegistrar[] newregs = ev.getRegistrars();
            for (int i=0 ; i<newregs.length ; i++) {
                if (!registrations.containsKey(newregs[i])) {
                    registerWithLookup(newregs[i]);
                }
            }
        }
    }

    // Called ONLY when we explicitly discard a
```

```

// lookup service, not "automatically" when a
// lookup service goes down. Once discovered,
// there is NO ongoing communication with a
// lookup service.
public void discarded(DiscoveryEvent ev) {
    ServiceRegistrar[] deadregs = ev.getRegistrars();
    for (int i=0 ; i<deadregs.length ; i++) {
        registrations.remove(deadregs[i]);
    }
}
}

public JINIFTPService() throws IOException {
    item = new ServiceItem(null, createProxy(), null);

    // Set a security manager
    if (System.getSecurityManager() == null) {
        System.setSecurityManager(new RMISecurityManager());
    }

    // Search for the "public" group, which by
    // convention is named by the empty string
    disco = new LookupDiscovery(new String[] { "" });

    // Install a listener.
    disco.addDiscoveryListener(new Listener());
}

protected JINIFTPServiceInterface createProxy() {
    return new JINIFTPServiceProxy();
}

// This work involves remote calls, and may take a
// while to complete.

protected synchronized void registerWithLookup(ServiceRegistrar registrar) {
    ServiceRegistration registration = null;

    try {
        registration = registrar.register(item, LEASE_TIME);
    } catch (RemoteException ex) {
        System.out.println("Couldn't register: " + ex.getMessage());
        return;
    }

    // If this is our first registration, use the
    // service ID returned to us. Ideally, we should
    // save this ID so that it can be used after
    // restarts of the service

```

```

if (item.serviceID == null) {
    item.serviceID = registration.getServiceID();
    System.out.println("Set serviceID to " + item.serviceID);
}

registrations.put(registrar, registration);
}

// This thread prevents the VM from exiting

public void run() {
    while (true) {
        try {
            Thread.sleep(1000000);
        } catch (InterruptedException ex) {}
    }
}

// Create a new JINIFTPService and start
// its thread.

public static void main(String args[]) {
    try {
        JINIFTPService hws = new JINIFTPService();
        new Thread(hws).start();
    } catch (IOException ex) {
        System.out.println("Couldn't create service: " +
            ex.getMessage());
    }
}
}

```

E.2 Jini Service Interface

```

/*
 * JINIFTPServiceInterface.java
 *
 * Created on June, 2002
 */

/**
 * @author skasper, lbuehrer
 * @version 0.1
 * @comment The Interface implemented by the Service's
 *         Proxy
 */

package myService;

```

```

public interface JINIFTPServiceInterface {
    public String send();
    public String receive();
}

```

E.3 sdpool

```

/*
 * @ authors   skasper, lbuehrer
 * @ version   0.4_corr (modified and corrected version of 0.4)
 *             Note: sdp version 0.6 (june 20th 2002) does provide more
 *             services but the bugs of version 0.4 were not fixed. Therefore,
 *             it does not matter if version 0.4 or 0.6 are taken as template.
 * @ comment   The input arguments had to be modified. Some loops were wrong
 *             or not implemented. A JINIFTP service was introduced by changing
 *             the existing FTRN service.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <getopt.h>
#include "sdp.h"
#include "sdp_lib.h"

char UUID_str[MAX_LEN_UUID_STR];

typedef int (*handler_t) (bdaddr_t *bdaddr, void *arg);

extern int optind,opterr,optopt;
extern char *optarg;

#define for_each_opt(opt, long, short) while
((opt=getopt_long(argc, argv, short ? short:"+", long, NULL)) != -1)

static void inquiry(handler_t handler, void *arg)
{
    inquiry_info ii[20];
    uint8_t count = 0;
    int i;

    printf("Inquiring ... \n");
    if (sdp_general_inquiry(ii, 20, 8, &count) < 0) {
        printf("Inquiry failed\n");
        return;
    }
}

```

```

for (i=0; i<count; i++)
    handler(&ii[i].bdaddr, arg);
}

static void print_service_class(gpointer value, gpointer userData)
{
    uuid_t *uuid = NULL;
    char ServiceClassUUID_str[MAX_LEN_SERVICECLASS_UUID_STR];

    uuid = (uuid_t *) value;
    sdp_uuid2strn(uuid, UUID_str, MAX_LEN_UUID_STR);
    sdp_svcclass_uuid2strn(uuid, ServiceClassUUID_str, MAX_LEN_SERVICECLASS_UUID_STR);
    printf("  \"%s\" (0x%s)\n", ServiceClassUUID_str, UUID_str);
}

static void print_service_desc(gpointer value, gpointer userData)
{
    sdp_proto_desc_t *desc = NULL;
    char str[MAX_LEN_PROTOCOL_UUID_STR];

    desc = (sdp_proto_desc_t *) value;
    sdp_uuid2strn(&desc->uuid, UUID_str, MAX_LEN_UUID_STR);
    sdp_proto_uuid2strn(&desc->uuid, str, MAX_LEN_PROTOCOL_UUID_STR);

    printf("  \"%s\" (0x%s)\n", str, UUID_str);
    if (desc->port)
        printf("    Channel/Port: %d\n", desc->port);
    if (desc->version)
        printf("    Version: %d\n", desc->version);
}

void print_access_proto(gpointer value, gpointer userData)
{
    GSList *protDescSeq = (GSList *) value;
    g_slist_foreach(protDescSeq, print_service_desc, NULL);
}

void print_profile_desc(gpointer value, gpointer userData)
{
    sdp_profile_desc_t *desc = (sdp_profile_desc_t *) value;
    char str[MAX_LEN_PROFILEDESCRIPTOR_UUID_STR];

    sdp_uuid2strn(&desc->uuid, UUID_str, MAX_LEN_UUID_STR);
    sdp_profile_uuid2strn(&desc->uuid, str,
        MAX_LEN_PROFILEDESCRIPTOR_UUID_STR);

    printf("  \"%s\" (0x%s)\n", str, UUID_str);
    if (desc->version)
        printf("    Version: %d\n", desc->version);
}

```

```

/*
 * Support for Service (de)registration
 */

typedef struct {
char *name;
char *provider;
char *desc;

unsigned int class;
unsigned int profile;
unsigned int channel;
} svc_info_t;

static int add_dun(sdp_svcrec_t *rec, void *arg)
{
GSList *svclass_id = NULL;
GSList *pfseq = NULL;
GSList *apseq = NULL;
GSList *root = NULL;
uuid_t root_uuid, dun_uuid, gn_uuid;
sdp_profile_desc_t profile;
sdp_access_proto_t aproto;
sdp_proto_desc_t proto[2];
sdp_svcrec_t *svcrec;
int status;

svcrec = sdp_svcrec_alloc();
if (!svcrec) {
printf("Failed to create service record\n");
return -1;
}

sdp_create_uuid16(&root_uuid, PUBLIC_BROWSE_GROUP);
root = g_slist_append(root, &root_uuid);
sdp_set_browse_grp_list(svcrec, root);

sdp_create_uuid16(&dun_uuid, DIALUP_NET_SVCLASS_ID);
sdp_create_uuid16(&gn_uuid, GENERIC_NETWORKING_SVCLASS_ID);
svclass_id = g_slist_append(svclass_id, &dun_uuid);
svclass_id = g_slist_append(svclass_id, &gn_uuid);
sdp_set_svclass_id(svcrec, svclass_id);

sdp_create_uuid16(&profile.uuid, DIALUP_NET_PROFILE_ID);
profile.version = 0x0100;
pfseq = g_slist_append(pfseq, &profile);
sdp_set_profile_desc(svcrec, pfseq);

sdp_create_uuid16(&proto[0].uuid, L2CAP_UUID);

```

```

proto[0].port = 0xffff;
proto[0].version = 0xffff;

sdp_create_uuid16(&proto[1].uuid, RFCOMM_UUID);
proto[1].port = 1;
proto[1].port_size = 2;
proto[1].version = 0xffff;

aproto.seq = NULL;
apseq = g_slist_append(apseq, &proto[0]);
apseq = g_slist_append(apseq, &proto[1]);
aproto.seq = g_slist_append(aproto.seq, apseq);
sdp_set_access_proto(svcrec, &aproto);

sdp_set_info_attr(svcrec, "Dial-Up Networking", NULL, NULL);

status = sdp_svcrec_register(svcrec);
if (status) {
printf("Service Record registration failed.\n");
return -1;
}

printf("Dial-Up Networking service registered\n");
return 0;
}

static int add_lan(sdp_svcrec_t *rec, void *arg)
{
GSList *svclass_id = NULL;
GSList *pfseq = NULL;
GSList *apseq = NULL;
GSList *root = NULL;
uuid_t root_uuid, svclass_uuid;
sdp_profile_desc_t profile;
sdp_access_proto_t aproto;
sdp_proto_desc_t proto[2];
sdp_svcrec_t *svcrec;
int status;

svcrec = sdp_svcrec_alloc();
if (!svcrec) {
printf("Failed to create service record\n");
return -1;
}

sdp_create_uuid16(&root_uuid, PUBLIC_BROWSE_GROUP);
root = g_slist_append(root, &root_uuid);
sdp_set_browse_grp_list(svcrec, root);

```

```

sdp_create_uid16(&svclass_uid, LAN_ACCESS_SVCLASS_ID);
svclass_id = g_slist_append(svclass_id, &svclass_uid);
sdp_set_svclass_id(svcrec, svclass_id);

sdp_create_uid16(&profile.uid, LAN_ACCESS_PROFILE_ID);
profile.version = 0x0100;
pfseq = g_slist_append(pfseq, &profile);
sdp_set_profile_desc(svcrec, pfseq);

sdp_create_uid16(&proto[0].uid, L2CAP_UUID);
proto[0].port = 0xffff;
proto[0].version = 0xffff;

sdp_create_uid16(&proto[1].uid, RFCOMM_UUID);
proto[1].port = 1;
proto[1].port_size = 2;
proto[1].version = 0xffff;

aproto.seq = NULL;
apseq = g_slist_append(apseq, &proto[0]);
apseq = g_slist_append(apseq, &proto[1]);
aproto.seq = g_slist_append(aproto.seq, apseq);
sdp_set_access_proto(svcrec, &aproto);

sdp_set_info_attr(svcrec, "LAN Access over PPP", NULL, NULL);

status = sdp_svcrec_register(svcrec);
if (status) {
    printf("Service Record registration failed.\n");
    return -1;
}

printf("LAN Access service registered\n");
return 0;
}

static int add_fax(sdp_svcrec_t *rec, void *arg)
{
    GSList *svclass_id = NULL;
    GSList *pfseq = NULL;
    GSList *apseq = NULL;
    GSList *root = NULL;
    uid_t root_uid, fax_uid, tel_uid;
    sdp_profile_desc_t profile;
    sdp_access_proto_t aproto;
    sdp_proto_desc_t proto[2];
    sdp_svcrec_t *svcrec;
    int status;

    svcrec = sdp_svcrec_alloc();

```

```

if (!svcrec) {
    printf("Failed to create service record\n");
    return -1;
}

sdp_create_uid16(&root_uid, PUBLIC_BROWSE_GROUP);
root = g_slist_append(root, &root_uid);
sdp_set_browse_grp_list(svcrec, root);

sdp_create_uid16(&fax_uid, FAX_SVCLASS_ID);
svclass_id = g_slist_append(svclass_id, &fax_uid);
sdp_create_uid16(&tel_uid, GENERIC_TELEPHONY_SVCLASS_ID);
svclass_id = g_slist_append(svclass_id, &tel_uid);
sdp_set_svclass_id(svcrec, svclass_id);

sdp_create_uid16(&profile.uid, FAX_PROFILE_ID);
profile.version = 0x0100;
pfseq = g_slist_append(pfseq, &profile);
sdp_set_profile_desc(svcrec, pfseq);

sdp_create_uid16(&proto[0].uid, L2CAP_UUID);
proto[0].port = 0xffff;
proto[0].version = 0xffff;

sdp_create_uid16(&proto[1].uid, RFCOMM_UUID);
proto[1].port = 2;
proto[1].port_size = 2;
proto[1].version = 0xffff;

aproto.seq = NULL;
apseq = g_slist_append(apseq, &proto[0]);
apseq = g_slist_append(apseq, &proto[1]);
aproto.seq = g_slist_append(aproto.seq, apseq);
sdp_set_access_proto(svcrec, &aproto);

sdp_set_info_attr(svcrec, "Fax", NULL, NULL);

status = sdp_svcrec_register(svcrec);
if (status) {
    printf("Service Record registration failed.\n");
    return -1;
}

printf("Fax service registered\n");
return 0;
}

static int add_opush(sdp_svcrec_t *rec, void *arg)
{
    GSList *svclass_id = NULL;

```

```

GSList *pfseq = NULL;
GSList *apseq = NULL;
GSList *root = NULL;
uuid_t root_uuid, opush;
sdp_profile_desc_t profile[1];
sdp_access_proto_t aproto;
sdp_proto_desc_t proto[3];
sdp_svcrec_t *svcrec;
int status;

svcrec = sdp_svcrec_alloc();
if (!svcrec) {
    printf("Failed to create service record\n");
    return -1;
}

sdp_create_uuid16(&root_uuid, PUBLIC_BROWSE_GROUP);
root = g_slist_append(root, &root_uuid);
sdp_set_browse_grp_list(svcrec, root);

sdp_create_uuid16(&opush, OBEX_OBJPUSH_SVCLASS_ID);
svclass_id = g_slist_append(svclass_id, &opush);
sdp_set_svclass_id(svcrec, svclass_id);

sdp_create_uuid16(&profile[0].uuid, OBEX_OBJPUSH_PROFILE_ID);
profile[0].version = 0x0100;
pfseq = g_slist_append(pfseq, &profile[0]);
sdp_set_profile_desc(svcrec, pfseq);

aproto.seq = NULL;

sdp_create_uuid16(&proto[0].uuid, L2CAP_UUID);
proto[0].port = 0xffff;
proto[0].version = 0xffff;
apseq = g_slist_append(apseq, &proto[0]);

sdp_create_uuid16(&proto[1].uuid, RFCOMM_UUID);
proto[1].port = 3;
proto[1].port_size = 2;
proto[1].version = 0xffff;
apseq = g_slist_append(apseq, &proto[1]);

sdp_create_uuid16(&proto[2].uuid, OBEX_UUID);
proto[2].port = 1;
proto[2].port_size = 2;
proto[2].version = 0xffff;
apseq = g_slist_append(apseq, &proto[2]);

aproto.seq = g_slist_append(aproto.seq, apseq);
sdp_set_access_proto(svcrec, &aproto);

```

```

sdp_set_info_attr(svcrec, "OBEX Object Push", NULL, NULL);

status = sdp_svcrec_register(svcrec);
if (status) {
    printf("Service Record registration failed.\n");
    return -1;
}

printf("OBEX Object Push service registered\n");
return 0;
}

/-->This function was modified for our purpose

static int add_jini(sdp_svcrec_t *rec, void *arg)
{
    GSList *svclass_id = NULL;
    GSList *pfseq = NULL;
    GSList *apseq = NULL;
    GSList *root = NULL;
    uuid_t root_uuid, ftrn;
    sdp_profile_desc_t profile[1];
    sdp_access_proto_t aproto;
    sdp_proto_desc_t proto[3];
    sdp_svcrec_t *svcrec;
    int status;

    svcrec = sdp_svcrec_alloc();
    if (!svcrec) {
        printf("Failed to create service record\n");
        return -1;
    }

    sdp_create_uuid16(&root_uuid, PUBLIC_BROWSE_GROUP);
    root = g_slist_append(root, &root_uuid);
    sdp_set_browse_grp_list(svcrec, root);

    sdp_create_uuid16(&ftrn, OBEX_FILETRANS_SVCLASS_ID);
    svclass_id = g_slist_append(svclass_id, &ftrn);
    sdp_set_svclass_id(svcrec, svclass_id);

    sdp_create_uuid16(&profile[0].uuid, OBEX_FILETRANS_PROFILE_ID);
    profile[0].version = 0x0100;
    pfseq = g_slist_append(pfseq, &profile[0]);
    sdp_set_profile_desc(svcrec, pfseq);

    sdp_create_uuid16(&proto[0].uuid, L2CAP_UUID);
    proto[0].port = 0xffff;

```

```

proto[0].version = 0xffff;
apseq = g_slist_append(apseq, &proto[0]);

sdp_create_uuid16(&proto[1].uuid, RFCOMM_UUID);
proto[1].port = 3;
proto[1].port_size = 2;
proto[1].version = 0xffff;
apseq = g_slist_append(apseq, &proto[1]);

sdp_create_uuid16(&proto[2].uuid, OBEX_UUID);
proto[2].port = 2;
proto[2].port_size = 2;
proto[2].version = 0xffff;
apseq = g_slist_append(apseq, &proto[2]);

aproto.seq = g_slist_append(NULL, apseq);
sdp_set_access_proto(svcrec, &aproto);

sdp_set_info_attr(svcrec, "JINI File Transfer", NULL, NULL);

status = sdp_svcrec_register(svcrec);
if (status) {
printf("Service Record registration failed.\n");
return -1;
}

printf("JINIFTP service registered\n");
return 0;
}

struct {
char *name;
uint16_t class;
int (*add)(sdp_svcrec_t *rec, void *arg);
} service[] = {
{ "DUN", DIALUP_NET_SVCLASS_ID, add_dun },
{ "LAN", LAN_ACCESS_SVCLASS_ID, add_lan },
{ "FAX", FAX_SVCLASS_ID, add_fax },
{ "OPUSH", OBEX_OBJPUSH_SVCLASS_ID, add_opush },
{ "JINIFTP", OBEX_FILETRANS_SVCLASS_ID, add_jini },
{ NULL }
};

/-->The following functions had to be modified and completed

/* Add local service */

int add_service(bdaddr_t *bdaddr, void *arg)
{

```

```

char *name = arg;
int i;

for (i=0; i<5; i++) {
if (!strcmp(service[i].name, name))
{
return service[i].add(NULL, NULL);
}
}

printf("Unknown service name: %s\n", arg);
return -1;
}

static struct option add_options[] = {
{"help", 0, 0, 'h'},
{"channel", 1, 0, 'c'},
{0, 0, 0, 0}
};

static char *add_help =
"Usage:\n"
"\tadd service\n";

void cmd_add(int argc, char **argv)
{
int opt, chan = 0;

for_each_opt(opt, add_options, NULL) {
switch(opt) {
case 'c':
chan = atoi(optarg);
break;
default:
printf(add_help);
return;
}
}

if (argc < 2) {
printf(add_help);
return;
}

add_service(NULL, argv[1]);
}

/* Delete local service */

```

```

int del_service(bdaddr_t *bdaddr, void *arg)
{
    char *rec = arg;

    if (rec) {
        uint32_t handle = strtoul(rec, NULL, 16);
        sdp_svcrec_t *svcRec;
        int status = 0;

        GSList *search, *attr, *seq = NULL;
        uint32_t range = 0x0000ffff;
        uint16_t count = 0;
        uuid_t root;

        sdp_create_uuid16(&root, PUBLIC_BROWSE_GROUP);
        attr = g_slist_append(NULL, &range);
        search = g_slist_append(NULL, &root);
        status = sdp_service_search_attr_req(*BDADDR_LOCAL, search,
            SDP_ATTR_REQ_RANGE, attr, 65535, &seq, &count);

        if (status) {
            printf("Service Search request failed.\n");
            return -1;
        }

        svcRec = sdp_svcrec_find(*BDADDR_LOCAL, handle);
        if (svcRec) {
            if (sdp_svcrec_delete(svcRec))
                printf("Service Record delete failed.\n");
            else
                printf("Service Record deleted.\n");
        } else {
            printf("Service Record not found.\n");
        }
        else {
            printf("Record handle was not specified.\n");
            return -1;
        }
        return 0;
    }

    static struct option del_options[] = {
        {"help", 0, 0, 'h'},
        {0, 0, 0, 0}
    };

    static char *del_help =
        "Usage:\n"
        "\tdel record_handle\n";

```

```

void cmd_del(int argc, char **argv)
{
    int opt;

    for_each_opt(opt, del_options, NULL) {
        switch(opt) {
            default:
                printf(del_help);
                return;
        }
    }

    if (argc < 2) {
        printf(del_help);
        return;
    }
    printf("delarguemtn %s\n", argv[1]);
    del_service(NULL, argv[1]);
}

/*
 * Browse SDP database
 */
int do_browse(bdaddr_t *bdaddr, void *arg)
{
    GSList *attrid, *search;
    GSList *pSeq = NULL;
    GSList *pGSList = NULL;
    uint32_t range = 0x0000ffff;
    uint16_t count = 0;
    int status = -1, i;
    sdp_access_proto_t *access_proto = NULL;
    uuid_t group;
    char str[20];

    if (!bdaddr) {
        inquiry(do_browse, NULL);
        return 0;
    }

    ba2str(bdaddr, str);
    printf("Browsing %s ... \n", str);

    if (!arg)
        sdp_create_uuid16(&group, PUBLIC_BROWSE_GROUP);
    else
        group = *(uuid_t *) arg;

```

```

attrid = g_slist_append(NULL, &range);
search = g_slist_append(NULL, &group);
status = sdp_service_search_attr_req(&bdaddr, search, SDP_ATTR_REQ_RANGE,
    attrid, 65535, &pSeq, &count);

if (status) {
    printf("Service Search failed\n");
    return -1;
}

for (i = 0; i < (int) g_slist_length(pSeq); i++) {
    uint32_t svcrec;
    svcrec = *(uint32_t *) g_slist_nth_data(pSeq, i);

    sdp_svcrec_print(&bdaddr, svcrec);

    printf("Service RecHandle: 0x%x\n", svcrec);

    if (sdp_get_svclass_id_list(&bdaddr, svcrec, &pGSList) == E_OK) {
        printf("Service Class ID List:\n");
        g_slist_foreach(pGSList, print_service_class, NULL);
    }

    if (sdp_get_access_proto(&bdaddr, svcrec, &access_proto) == E_OK) {
        printf("Protocol Descriptor List:\n");
        g_slist_foreach(access_proto->seq,
            print_access_proto, NULL);
    }

    if (sdp_get_profile_desc(&bdaddr, svcrec, &pGSList) == E_OK) {
        printf("Profile Descriptor List: \n");
        g_slist_foreach(pGSList, print_profile_desc, NULL);
    }

    printf("\n");

    if (sdp_is_group(&bdaddr, svcrec)) {
        uuid_t grp;
        sdp_get_group_id(&bdaddr, svcrec, &grp);
        if (grp.value.uuid16 != group.value.uuid16)
            do_browse(&bdaddr, &grp);
    }
}
return status;
}

static struct option browse_options[] = {
    {"help", 0, 0, 'h'},
    {0, 0, 0, 0}
};

```

```

static char *browse_help =
    "Usage:\n"
    "\tbrowse [bdaddr]\n";

void cmd_browse(int argc, char **argv)
{
    int opt;

    for_each_opt(opt, browse_options, NULL) {
        switch(opt) {
            default:
                printf(browse_help);
                return;
        }
    }

    if (argc == 3) {
        bdaddr_t bdaddr;
        str2ba(argv[2], &bdaddr);
        do_browse(&bdaddr, NULL);
    } else
        do_browse(NULL, NULL);
}

/*
 * Search for a service
 */
int do_search(bdaddr_t *bdaddr, void *arg)
{
    GSLList *attr, *srch, *rsp = NULL, *list;
    uint32_t range = 0x0000ffff;
    uint16_t count = 0;
    int status = -1, i;
    sdp_access_proto_t *access_proto = NULL;
    uint16_t class = 0;
    uuid_t svclass;
    char str[20], *svc = arg;

    if (!bdaddr) {
        inquiry(do_search, arg);
        return 0;
    }

    for (i=0; service[i].name; i++)
        if (!strcasecmp(svc, service[i].name)) {
            class = service[i].class;
            break;
        }
}

```

```

if (!class) {
printf("Unknown service %s\n", svc);
return -1;
}

ba2str(bdaddr, str);
printf("Searching for %s on %s ...\n", svc, str);

sdp_create_uuid16(&svclass, class);
srch = g_slist_append(NULL, &svclass);

//attr = g_slist_append(NULL, &range);
// status = sdp_service_search_attr_req(*bdaddr, srch,
// SDP_ATTR_REQ_RANGE, attr, 65535, &rsp, &count);
status = sdp_service_search_req(*bdaddr, srch,
65535, &rsp, &count);

if (status) {
printf("Not found!\n");
return -1;
}

for (i = 0; i < g_slist_length(rsp); i++) {
uint32_t svcrec;
svcrec = *(uint32_t *) g_slist_nth_data(rsp, i);

sdp_svcrec_print(*bdaddr, svcrec);

printf("Service ReHandle: 0x%x\n", svcrec);

if (sdp_get_svclass_id_list(*bdaddr, svcrec, &list) == E_OK) {
printf("Service Class ID List:\n");
g_slist_foreach(list, print_service_class, NULL);
}

if (sdp_get_access_proto(*bdaddr, svcrec, &access_proto) == E_OK) {
printf("Protocol Descriptor List:\n");
g_slist_foreach(access_proto->seq,
print_access_proto, NULL);
}

if (sdp_get_profile_desc(*bdaddr, svcrec, &list) == E_OK) {
printf("Profile Descriptor List: \n");
g_slist_foreach(list, print_profile_desc, NULL);
}

printf("\n");

if (sdp_is_group(*bdaddr, svcrec)) {

```

```

uuid_t group;
sdp_get_group_id(*bdaddr, svcrec, &group);
sdp_uuid2strn(&group, UUID_str, MAX_LEN_UUID_STR);
if (group.value.uuid16 != svclass.value.uuid16)
do_search(bdaddr, &group);
}
}
return status;
}

static struct option search_options[] = {
{"help", 0, 0, 'h'},
{"service", 1, 0, 's'},
{0, 0, 0, 0}
};

static char *search_help =
"Usage:\n"
"\tsearch [--service= SVC] [bdaddr]\n";

void cmd_search(int argc, char **argv)
{
char *svc = argv[1];
int opt;

for_each_opt(opt, search_options, NULL) {
switch(opt) {
case 's':
svc = strdup(optarg);
break;

default:
printf(search_help);
return;
}
}

if (argc == 3) {
bdaddr_t bdaddr;
str2ba(argv[2], &bdaddr);
do_search(&bdaddr, svc);
} else
do_search(NULL, svc);
}

struct {
char *cmd;
void (*func)(int argc, char **argv);
char *doc;
} command[] = {

```

```

{ "search", cmd_search, "Search for a service" },
{ "browse", cmd_browse, "Browse all available services" },
{ "add", cmd_add, "Add local service" },
{ "del", cmd_del, "Delete local service" },
{ NULL, NULL, 0 }
};

static void usage(void)
{
int i;

printf("sdptool - SDP Tool ver %s\n", VERSION);
printf("Usage:\n"
"\tsdptool [options] <command> [command parameters]\n");
printf("Options:\n"
"\t--help\tDisplay help\n");

printf("Commands:\n");
for (i=0; command[i].cmd; i++)
printf("\t%-4s\t\t%s\n", command[i].cmd, command[i].doc);

printf("\nServices:\n\t");
for (i=0; service[i].name; i++)
printf("%s ", service[i].name);
printf("\n");
}

static struct option main_options[] = {
{"help", 0, 0, 'h'},
{0, 0, 0, 0}
};

int main(int argc, char **argv)
{
int opt, i;

while ((opt=getopt_long(argc, argv, "+h", main_options, NULL)) != -1) {
switch(opt) {
case 'h':
default:
usage();
exit(0);
}
}

argc -= optind;
argv += optind;
optind = 0;

if (argc < 1) {

```

```

usage();
exit(0);
}

if (sdp_init() < 0) {
printf("SDP library initialization failed\n");
return 1;
}

for (i=0; command[i].cmd; i++) {
if (strncmp(command[i].cmd, argv[0], 3))
continue;
command[i].func(argc, argv);
break;
}

return 0;
}

```

E.4 Daemon Request-Handling

```

/*
 * @ authors skasper, lbuehrer
 * @ version 1.4_corr (modified version of 1.4)
 *
 * @ comment We decided to work with service_search requests. However,
 * it is also possible to use service_search_attribute
 * requests. To make this change, our modifications have to be
 * copied to the respective function. Moreover, the
 * service_search_attribute_request function has to be
 * called instead of the service_search_request (sdptool.c).
 */

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>

#include "sdp.h"
#include "sdp_internal.h"
#include "sdp_lib.h"

#include "sdpd.h"

//->Modifications had to be made here!
static int service_search_req(sdp_req_t *req, sdp_pduform_t * pduform)

```

```

{
int status = 0, i;
char *pdata;
int scanned = 0;
GSList *searchPattern;
int exp_svcrec_num, act_svcrec_num;
uint8_t dataType;
sdp_cont_state_t *cstate = NULL;
char *pCacheBuffer = NULL;
int handleSize = 0;
long cStateId = -1;
short rsp_count = 0;
short *pTotalRecordCount;
short *pCurrentRecordCount;
int mtu;
uuid_t *serviceClassID;

SDPDBG("");

pdata = req->buf + sizeof(sdp_pdu_hdr_t);

searchPattern = NULL;
scanned = sdp_extract_des(pdata, &searchPattern, &dataType, SDP_UUID128);

serviceClassID = (uuid_t *) g_slist_nth_data(searchPattern, 0);
if(serviceClassID->value.uuid16 == 0x1106) //JINIFTP_SERVICE_ID
    printf("JINIFTP Service requested!\n");
else printf("NO JINI Service requested!\n");

pdata += scanned;
exp_svcrec_num = ntohs(*(uint16_t *) pdata);

SDPDBG("Expected count: %d\n", exp_svcrec_num);
SDPDBG("Bytes scanned : %d\n", scanned);

pdata += sizeof(uint16_t);

cstate = sdp_cstate_get(pdata);

mtu = req->mtu - (sizeof(sdp_pdu_hdr_t) + sizeof(uint16_t) +
                sizeof(uint16_t) + SDP_CONT_STATE_SIZE);
act_svcrec_num = MIN(exp_svcrec_num, (mtu >> 2));

pdata = pduform->data;
pTotalRecordCount = (short *) pdata;
*(uint16_t *) pdata = 0;

```

```

pdata += sizeof(uint16_t);
pduform->data_size += sizeof(uint16_t);
pCurrentRecordCount = (short *) pdata;
*(uint16_t *) pdata = 0;
pdata += sizeof(uint16_t);
pduform->data_size += sizeof(uint16_t);

if (cstate == NULL) {
/*
 * For every element in the sdp_svcrec_t DB,
 * do a pattern search
 */
int index = 0;
GSList *svcList;

handleSize = 0;
svcList = sdp_get_svcrec_list(*BDADDR_LOCAL);

for (;;) {
gpointer data;
sdp_svcrec_t *svcRec;

data = g_slist_nth_data(svcList, index);
if (data != NULL) {
svcRec = (sdp_svcrec_t *) data;

SDPDBG("Checking svcRec : 0x%x\n", svcRec->handle);

if (sdp_match_uuid(searchPattern, svcRec->pattern)) {
//desired class is found in the database...
rsp_count++;
*(uint32_t *) pdata = htonl(svcRec->handle);
pdata += sizeof(uint32_t);
handleSize += sizeof(uint32_t);
}
index++;
} else
break;
}

SDPDBG("Match count: %d\n", rsp_count);

/*
 * Add "handleSize" to pduform buffer size
 */
pduform->data_size += handleSize;
*pTotalRecordCount = htons(rsp_count);
*pCurrentRecordCount = htons(rsp_count);

```

```

if (rsp_count > act_svcrec_num) {
/*
 * We need to cache the rsp and generate a
 * continuation state
 */
cStateId = sdp_cstate_alloc_buf(req->bdaddr, pduform);
/*
 * Subtract the handleSize since we now send only
 * a subset of handles
 */
pduform->data_size -= handleSize;
} else {
/*
 * NULL continuation state
 */
sdp_set_cstate_pdu(pduform, NULL);
}
}

/*
 * Under both the conditions below, the rsp buffer is not
 * built up yet
 */
if (cstate || (cStateId != -1)) {
short lastIndex = 0;

if (cstate) {
sdp_pduform_t *pCache;
/*
 * Get the previous sdp_cont_state_t and obtain
 * the cached rsp
 */
pCache = sdp_get_cached_rsp(req->bdaddr, cstate);
if (pCache != NULL) {
pCacheBuffer = pCache->data;
/*
 * Get the rsp_count from the cached buffer
 */
rsp_count = ntohs(*(uint16_t *) pCacheBuffer);
}

/*
 * Get the index of the last sdp_svcrec_t sent
 */
lastIndex = cstate->cStateValue.lastIndexSent;
} else {
status = SDP_INVALID_CSTATE;
goto done;
}
} else {
pCacheBuffer = pduform->data;

```

```

lastIndex = 0;
}

/*
 * Set the local buffer pointer to beyond the
 * current record count. And increment the cached
 * buffer pointer to beyond the counters ..
 */
pdata = (char *) pCurrentRecordCount + sizeof(uint16_t);

/*
 * There is the toatalCount and the currentCount
 * fields to increment beyond
 */
pCacheBuffer += (2 * sizeof(uint16_t));

if (cstate == NULL) {
handleSize = act_svcrec_num << 2; //sizeof(uint32_t);
i = act_svcrec_num;
} else {
handleSize = 0;
for (i = lastIndex; (i - lastIndex) < act_svcrec_num && i < rsp_count; i++) {
*(uint32_t *) pdata = *((uint32_t *) pCacheBuffer) + i;
pdata += sizeof(uint32_t);
handleSize += sizeof(uint32_t);
}
}

/*
 * Add "handleSize" to pduform buffer size
 */
pduform->data_size += handleSize;
*pTotalRecordCount = htons(rsp_count);
*pCurrentRecordCount = htons(i - lastIndex);

if (i == rsp_count) {
/*
 * Set "null" continuationState
 */
sdp_set_cstate_pdu(pduform, NULL);
} else {
/*
 * There is more to it .. set the lastIndexSent to
 * the new value and move on ..
 */
sdp_cont_state_t newState;

SDPDBG("Setting non-NULL sdp_cstate_t\n");

memset((char *) &newState, 0, sizeof(sdp_cont_state_t));

```

```

if (cstate != NULL) {
memcpy((char *) &newState, cstate, sizeof(sdp_cont_state_t));
newState.cStateValue.lastIndexSent = i;
} else {
newState.timestamp = cStateId;
newState.cStateValue.lastIndexSent = i;
}
sdp_set_cstate_pdu(pduform, &newState);
}

done:
if (searchPattern != NULL)
sdp_free_slist(searchPattern);

return status;
}

/*
 * Extract attribute identifiers from the request PDU.
 * Clients could request a subset of attributes (by id)
 * from a service record, instead of the whole set. The
 * requested identifiers are present in the PDU form of
 * the request
 */
static int extract_attrs(char *buf, sdp_svcrec_t *svcRec, GSList *attrSeq,
uint8_t attrDataType, sdp_pduform_t *pduform, int maxExpectedAttrSize)
{
int status = 0;

if (!svcRec)
return SDP_INVALID_SVCREC_HANDLE;

/*
 * Set the maximum expected size before hand
 */
pduform->buf_size = maxExpectedAttrSize;

/*
 * This is a request for the entire attribute range
 */
if (svcRec->pduform.data == NULL) {
SDPDBG("Generating svc rec pdu form\n");
sdp_gen_svcrec_pduform(svcRec);
}

#ifdef SDP_DEBUG
if (attrSeq != NULL) {
SDPDBG("Entries in attr seq : %d\n", g_slist_length(attrSeq));
} else {

```

```

SDPDBG("NULL attribute descriptor\n");
}
SDPDBG("AttrDataType : %d\n", attrDataType);
#endif

if (attrSeq != NULL) {
int attrLength;
int index = 0;
attrLength = 0;

if (attrDataType == SDP_UINT16) {
index = 0;
while (1) {
gpointer pAttr;
uint16_t attrId = 0;

/*
 * Now we have a sequence of attribute identifiers for
 * which we need to generate PDU form
 */
pAttr = g_slist_nth_data(attrSeq, index);
if (pAttr == NULL) {
SDPDBG("Break out at index : %d\n", index);
break;
}
index++;
attrId = *(uint16_t *) pAttr;

SDPDBG("Attr id requested : 0x%x\n", attrId);

sdp_gen_svcrec_pduform_by_id(svcRec, attrId, pduform);
} else if (attrDataType == SDP_UINT32) {
uint8_t completeAttrSetBuilt = 0;
index = 0;

for (;;) {
gpointer pAttr = NULL;
uint32_t attrRange = 0;
uint16_t lowIdLim = 0;
uint16_t highIdLim = 0;

pAttr = g_slist_nth_data(attrSeq, index);
if (pAttr == NULL) {
SDPDBG("Break out at index : %d\n", index);
break;
}
index++;
attrRange = *(uint32_t *) pAttr;
lowIdLim = ((0xffff0000 & attrRange) >> 16);

```

```

highIdLim = 0x0000ffff & attrRange;

SDPDBG("attr range : 0x%x\n", attrRange);
SDPDBG("Low id : 0x%x\n", lowIdLim);
SDPDBG("High id : 0x%x\n", highIdLim);

if ((lowIdLim == 0x0000) && (highIdLim == 0xffff) && (completeAttrSetBuilt == 0)
    && (svcRec->pduform.data_size <= pduform->buf_size)) {
    completeAttrSetBuilt = 1;
    /*
    * Create a copy of it
    */
    SDPDBG("Copy attr pdu form : %d\n", svcRec->pduform.data_size);
    memcpy(pduform->data, svcRec->pduform.data, svcRec->pduform.data_size);
    pduform->data_size = svcRec->pduform.data_size;
} else {
    /*
    * Not the entire range of attributes, pick
    * and choose attributes requested
    */
    int attrId = 0;
    for (attrId = lowIdLim; attrId <= highIdLim; attrId++) {
        sdp_gen_svcrec_pduform_by_id(svcRec, attrId, pduform);
    }
} else {
    status = SDP_INVALID_REQ_SYNTAX;
    SDPERR("Unexpected data type : 0x%x\n", attrDataType);
    SDPERR("Expect uint16_t or uint32_t\n");
} else {
    SDPDBG("Attribute sequence is NULL\n");
}

return status;
}

/*
* A request for the attributes of a service record.
* First check if the service record (specified by
* service record handle) exists, then call the attribute
* streaming function
*/
static int service_attr_req(sdp_req_t * req, sdp_pduform_t * pduform)
{
    char *pdata;
    sdp_svcrec_t *svcRec;
    uint32_t svcrec;
    sdp_cont_state_t *cstate = NULL;

```

```

char *pResponse = NULL;
short cstate_size = 0;
GSList *attrSeq = NULL;
uint8_t attrDataType = 0;
int scanned = 0;
int max_rsp_size;
int status = 0;

SDPDBG("");

pdata = req->buf + sizeof(sdp_pdu_hdr_t);
svcrec = ntohl(*(uint32_t *) pdata);
pdata += sizeof(uint32_t);
max_rsp_size = ntohs(*(uint16_t *) pdata);
pdata += sizeof(uint16_t);

/*
* Extract the attribute list
*/
scanned = sdp_extract_des(pdata, &attrSeq, &attrDataType, SDP_UINT16);
pdata += scanned;

/*
* Check if continuation state exists, if yes attempt
* to get rsp remainder from cache, else send error
*/
cstate = sdp_cstate_get(pdata);

SDPDBG("SvcRecHandle : 0x%x\n", svcrec);
SDPDBG("max_rsp_size : %d\n", max_rsp_size);

/*
* Calculate Attribute size according to MTU
* We can send only (MTU - sizeof(sdp_pdu_hdr_t) - sizeof(sdp_cont_state_t))
*/
max_rsp_size = MIN(max_rsp_size, (req->mtu - sizeof(sdp_pdu_hdr_t) -
sizeof(uint32_t) - SDP_CONT_STATE_SIZE - sizeof(uint16_t)));

/*
* pull header for AttributeList byte count
*/
pduform->data += sizeof(uint16_t);
pduform->buf_size -= sizeof(uint16_t);

if (cstate != NULL) {
    sdp_pduform_t *pCache = NULL;
    short bytesSent = 0;

    pCache = sdp_get_cached_rsp(req->bdaddr, cstate);

```

```

SDPDBG("Obtained cached rsp : %p\n", pCache);

if (pCache) {
pResponse = pCache->data;
bytesSent = MIN(max_rsp_size, (pCache->data_size - cstate->cStateValue.maxBytesSent));
memcpy(pduform->data, (pResponse + cstate->cStateValue.maxBytesSent), bytesSent);
pduform->data_size += bytesSent;
cstate->cStateValue.maxBytesSent += bytesSent;

SDPDBG("Response size : %d sending now : %d bytes sent so far : %d\n",
pCache->data_size, bytesSent, cstate->cStateValue.maxBytesSent);
if (cstate->cStateValue.maxBytesSent == pCache->data_size)
cstate_size = sdp_set_cstate_pdu(pduform, NULL);
else
cstate_size = sdp_set_cstate_pdu(pduform, cstate);
} else {
status = SDP_INVALID_CSTATE;
SDPERR("NULL cache buffer and non-NULL continuation state\n");
}
} else {
svcRec = sdp_svcrec_find(*BDADDR_LOCAL, svcrec);
status = extract_attrs(pdata, svcRec, attrSeq, attrDataType,
pduform, pduform->buf_size);
if (pduform->data_size > max_rsp_size) {
sdp_cont_state_t newState;

memset((char *) &newState, 0, sizeof(sdp_cont_state_t));
newState.timestamp = sdp_cstate_alloc_buf(req->bdaddr, pduform);
/*
* Reset the buffer size to the maximum expected and
* set the sdp_cont_state_t
*/
SDPDBG("Creating continuation state of size : %d\n", pduform->data_size);
pduform->data_size = max_rsp_size;
newState.cStateValue.maxBytesSent = max_rsp_size;
cstate_size = sdp_set_cstate_pdu(pduform, &newState);
} else
cstate_size = sdp_set_cstate_pdu(pduform, NULL);
}

// push header
pduform->data -= sizeof(uint16_t);
pduform->buf_size += sizeof(uint16_t);

if (status)
return status;

/*
* set attribute list byte count
*/

```

```

*(uint16_t *) pduform->data = htons(pduform->data_size - cstate_size);
pduform->data_size += sizeof(uint16_t);
return 0;
}

/*
* The single request for a combined "service search" and
* "attribute extraction"
*/
static int service_search_attr_req(sdp_req_t *req, sdp_pduform_t *pduform)
{
int status = 0;
char *pdata;
int scanned = 0;
GSLList *searchPattern;
int maxExpectedAttrSize;
uint8_t dataType;
sdp_cont_state_t *cstate = NULL;
char *pResponse = NULL;
short cstate_size = 0;
GSLList *attrSeq = NULL;
uint8_t attrDataType = 0;
GSLList *svList;
int index = 0;
int rsp_count = 0;
sdp_pduform_t localpdu_form;

pdata = req->buf + sizeof(sdp_pdu_hdr_t);

searchPattern = NULL;
scanned = sdp_extract_des(pdata, &searchPattern, &dataType, SDP_UINT16);

SDPDBG("Bytes scanned: %d", scanned);

pdata += scanned;
maxExpectedAttrSize = ntohs(*(uint16_t *) pdata);
pdata += sizeof(uint16_t);

SDPDBG("Max Attr expected: %d", maxExpectedAttrSize);

/*
* Extract the attribute list
*/
scanned = sdp_extract_des(pdata, &attrSeq, &attrDataType, SDP_UINT16);
pdata += scanned;

/*
* Check if continuation state exists, if yes attempt
* to get rsp remainder from cache, else send error
*/

```

```

cstate = sdp_cstate_get(pdata); //this is my Continuation Information
svcList = sdp_get_svcrec_list(*BDADDR_LOCAL);

localpdu_form.data = (char *) malloc(USHRT_MAX);
localpdu_form.data_size = 0;
localpdu_form.buf_size = USHRT_MAX;
memset(localpdu_form.data, 0, USHRT_MAX);

/*
 * Calculate Attribute size according to MTU
 * We can send only (MTU - sizeof(sdp_pdu_hdr_t) - sizeof(sdp_cont_state_t))
 */
maxExpectedAttrSize = MIN(maxExpectedAttrSize, (req->mtu - sizeof(sdp_pdu_hdr_t) -
SDP_CONT_STATE_SIZE - sizeof(uint16_t)));

/*
 * pull header for AttributeList byte count
 */
pduform->data += sizeof(uint16_t);
pduform->buf_size -= sizeof(uint16_t);

if (cstate == NULL) {
/*
 * No Continuation State -> Create New Response
 */
for (;;) {
gpointer data;
sdp_svcrec_t *svcRec;

data = g_slist_nth_data(svcList, index);
if (data != NULL) {
svcRec = (sdp_svcrec_t *) data;
if (sdp_match_uuid(searchPattern, svcRec->pattern)) {
rsp_count++;
status = extract_attr(pdata, svcRec, attrSeq,
attrDataType, &localpdu_form,
localpdu_form.buf_size);

SDPDBG("Response count : %d\n", rsp_count);
SDPDBG("Local PDU size : %d\n", localpdu_form.data_size);

if (status) {
SDPDBG("Extract attr from svcrec returns err\n");
break;
} else {
if (pduform->data_size + localpdu_form.data_size < pduform->buf_size) {
// to be sure no relocations
sdp_append_to_pduform(pduform, &localpdu_form);
} else {

```

```

SDPERR("Relocation needed\n");
break;
}
}
SDPDBG("Net PDU size : %d\n", pduform->data_size);
}
}
index++;
} else
break;
}
}
if (pduform->data_size > maxExpectedAttrSize) {
sdp_cont_state_t newState;

memset((char *) &newState, 0, sizeof(sdp_cont_state_t));
newState.timestamp = sdp_cstate_alloc_buf(req->bdaddr, pduform);
/*
 * Reset the buffer size to the maximum expected and
 * set the sdp_cont_state_t
 */
pduform->data_size = maxExpectedAttrSize;
newState.cStateValue.maxBytesSent = maxExpectedAttrSize;
cstate_size = sdp_set_cstate_pdu(pduform, &newState);
} else
cstate_size = sdp_set_cstate_pdu(pduform, NULL);
} else {
/*
 * Continuation State exists -> get from cache
 */
sdp_pduform_t *pCache;
uint16_t bytesSent = 0;

/*
 * Get the cached rsp from the buffer
 */
pCache = sdp_get_cached_rsp(req->bdaddr, cstate);
if (pCache != NULL) {
pResponse = pCache->data;
bytesSent = MIN(maxExpectedAttrSize, (pCache->data_size - cstate->cStateValue.maxBytesSent));
memcpy(pduform->data, (pResponse + cstate->cStateValue.maxBytesSent), bytesSent);
pduform->data_size += bytesSent;
cstate->cStateValue.maxBytesSent += bytesSent;
if (cstate->cStateValue.maxBytesSent == pCache->data_size)
cstate_size = sdp_set_cstate_pdu(pduform, NULL);
else
cstate_size = sdp_set_cstate_pdu(pduform, cstate);
} else {
status = SDP_INVALID_CSTATE;
SDPDBG("Non null continuation state, but null cache buffer\n");
}
}
}

```

```

if (!rsp_count && !cstate) {
//we've not found anything
pduform->data_size = 0;
sdp_append_to_pduform(pduform, &localpdu_form);
sdp_set_cstate_pdu(pduform, NULL);
}

// push header
pduform->data      -= sizeof(uint16_t);
pduform->buf_size += sizeof(uint16_t);

if (!status) {
/* set attribute list byte count */
*(uint16_t *) pduform->data = htons(pduform->data_size - cstate_size);
pduform->data_size += sizeof(uint16_t);
}

free(localpdu_form.data);
sdp_free_slist(searchPattern);
return status;
}

/*
 * Top level request processor. Calls the appropriate processing
 * function based on request type. Handles service registration
 * client requests also
 */
void process_request(sdp_req_t * req)
{
    int test = 0;
    sdp_pdu_hdr_t *reqPduHdr;
    sdp_pdu_hdr_t *rspPduHdr;
    sdp_pduform_t rspPdu;
    char *bufferStart;
    int bytesSent = 0;
    int status;
    uuid_t *test2;

    SDPDBG("");

    reqPduHdr = (sdp_pdu_hdr_t *) req->buf;

    bufferStart = (char *) malloc(USHRT_MAX);
    memset((void *) bufferStart, 0, USHRT_MAX);
    rspPdu.data = bufferStart + sizeof(sdp_pdu_hdr_t);
    rspPdu.data_size = 0;
    rspPdu.buf_size = USHRT_MAX - sizeof(sdp_pdu_hdr_t);
    rspPduHdr = (sdp_pdu_hdr_t *) bufferStart;

```

```

status = SDP_INVALID_REQ_SYNTAX;

switch (reqPduHdr->pdu_id) {

    /*-->Modifications had to be made here!
case SDP_SVC_SEARCH_REQ:
SDPDBG("Got a svc srch req\n");
status = service_search_req(req, &rspPdu);

        test = 1;
        rspPduHdr->pdu_id = SDP_SVC_SEARCH_RSP;
break;

case SDP_SVC_ATTR_REQ:
SDPDBG("Got a svc attr req\n");
status = service_attr_req(req, &rspPdu);
rspPduHdr->pdu_id = SDP_SVC_ATTR_RSP;
break;

case SDP_SVC_SEARCH_ATTR_REQ:
    SDPDBG("Got a svc srch attr req\n");
status = service_search_attr_req(req, &rspPdu);
rspPduHdr->pdu_id = SDP_SVC_SEARCH_ATTR_RSP;
break;

/* Following requests are allowed only for local connections */

case SDP_SVC_REGISTER_REQ:
SDPDBG("Service register request\n");
if (req->local) {
status = service_register_req(req, &rspPdu);
rspPduHdr->pdu_id = SDP_SVC_REGISTER_RSP;
}
break;

case SDP_SVC_UPDATE_REQ:
SDPDBG("Service update request\n");
if (req->local) {
status = service_update_req(req, &rspPdu);
rspPduHdr->pdu_id = SDP_SVC_UPDATE_RSP;
}
break;

case SDP_SVC_REMOVE_REQ:
SDPDBG("Service removal request\n");
if (req->local) {
status = service_remove_req(req, &rspPdu);
rspPduHdr->pdu_id = SDP_SVC_REMOVE_RSP;
}
}

```

```

break;

default:
SDPERR("Unknown PDU ID : 0x%x received\n", reqPduHdr->pdu_id);
status = SDP_INVALID_REQ_SYNTAX;
break;
}

if (status) {
rspPduHdr->pdu_id = SDP_ERROR_RSP;
*(uint16_t *) rspPdu.data = htons(status);
rspPdu.data_size = sizeof(uint16_t);
}

SDPDBG("Sending rsp. status %d", status);

rspPduHdr->transac_id = reqPduHdr->transac_id;
rspPduHdr->plen = htons(rspPdu.data_size);
/*
 * Now point back to the real buffer start and set the
 * real rsp length
 */
rspPdu.data_size += sizeof(sdp_pdu_hdr_t);
rspPdu.data = bufferStart;

/*
 * Stream the rsp PDU
 */

bytesSent = send(req->sock, rspPdu.data, rspPdu.data_size, 0);

SDPDBG("Bytes Sent : %d\n", bytesSent);

free(rspPdu.data);
free(req->buf);
free(req);
if(test) exit(0);
}

```

E.5 Bridge

```

/*
 * Bridge.java
 *
 * Created on June, 2002
 */

/**
 * @author skasper, lbuehrer

```

```

 * @version 0.1
 * @comment Bridge translator. A Bluetooth SDP request is translated and forwarded
 *          to the Jini lookup table.
 */

package myService;

import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.discovery.LookupDiscovery;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceTemplate;

import java.util.Vector;
import java.io.IOException;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
import java.io.*;
import java.lang.String;

public class Bridge implements Runnable {
protected ServiceTemplate template;
protected LookupDiscovery disco;
protected boolean subscribed = false;

public Bridge(Class[] types) throws IOException {
template = new ServiceTemplate(null, types, null);
// Set a security manager
if (System.getSecurityManager() == null) {
System.setSecurityManager(new RMISecurityManager());
}

// Only search the public group
disco = new LookupDiscovery(new String[] { "" });

// Install a listener
disco.addDiscoveryListener(new Listener());
}

//-----
//Basic Jini function

protected Object lookForService(ServiceRegistrar lusvc) {
Object o = null;

try {
o = lusvc.lookup(template);
} catch (RemoteException ex) {
System.err.println("Error doing lookup: " + ex.getMessage());
}
}

```

```

return null;
}

if (o == null) {
System.err.println("No matching service.");
return null;
}

System.out.println("Got a matching service.");

System.out.println("Status Message: " + ((JINIFTPServiceInterface) o).send());
System.out.println("Status Message: " + ((JINIFTPServiceInterface) o).receive());

if(!subscribed){
subscribed = true;
subscribe("JINIFTP");
}

return o;
}

// An inner class to implement DiscoveryListener

class Listener implements DiscoveryListener {
public void discovered(DiscoveryEvent ev) {
ServiceRegistrar[] newregs = ev.getRegistrars();
for (int i=0 ; i<newregs.length ; i++) {
lookForService(newregs[i]);
}
}
public void discarded(DiscoveryEvent ev) {
}
}

// This thread prevents the VM from exiting

public void run() {
while (true) {
try {
Thread.sleep(1000000);
} catch (InterruptedException ex) {
}
}
}

//-----
//Basic SDP functions, provided by BlueZ

```

```

public static void add (String service){
try{

Process c = Runtime.getRuntime().exec("sdptool add " + service);
InputStream in = c.getInputStream();
BufferedReader bu = new BufferedReader(new InputStreamReader(in));
while(bu.ready())
System.out.println(bu.readLine());

}catch (Exception e){}
}

//-----
//Bluetooth wants to discover Jini services

public static String get_bt_request () {
//Waits for a BT search or browse request. Returns the requested servicetype.
String service = "";
try{
Process d = Runtime.getRuntime().exec("sdpd -n");
InputStream in = d.getInputStream();
BufferedReader bu = new BufferedReader(new InputStreamReader(in));
boolean a = true;
while(a == true){
if(bu.ready()) {
service = bu.readLine();
boolean status = service.equals("JINIFTP Service requested!");
if(!status){
System.out.println("No JINI Service requested\n");
}
else {
service ="JINIFTPServiceInterface";
return service;
}
}
}
a = false;
}
}catch (Exception e){}

return service;
}

public static void subscribe (String jiniservice) {
//Adds the appropriate Bluetooth service to the Bluetooth record database.
try{
add(jiniservice);
}
}

```

```

}catch(Exception ex){}

}

//-----
//Usage and init functions

public static void usage () {
System.out.println("Usage: runbridge [option]\n");
System.out.println("option: additionally register a Bluetooth Service\n");
System.out.println("  FAX\n  DUN\n  LAN\n  OPUSH");
System.exit(0);
}

public static void init () {
System.out.println("*****\n");
System.out.println("SDP2Lookup Translation\n");
System.out.println("Waiting for an SDP request ... \n");
System.out.println("*****\n");
}

//-----
//The main class

//Creates a JINIFTPClient and starts its thread

public static void main(String args[] ) {
String classname = "No JINI Service requested";
String arg;

try{
arg = args[0];
}catch(Exception ex){ arg = "noService";
}

if(arg.equals("-help"))
usage();

init();

classname = get_bt_request();

try{
Process c = Runtime.getRuntime().exec("sdpd");
}catch (Exception e){}

```

```

if(arg != "noService"){
try {
Thread.sleep(5000);
} catch (InterruptedException ex) {}

add(arg);
}

if(classname.equals("JINIFTPServiceInterface")){
try {
Class t = Class.forName("myService."+classname);
Class[] types = {t};
Bridge hwc = new Bridge(types);
new Thread(hwc).start();
} catch (Exception ex) {
System.out.println("Couldn't create client: " + ex.getMessage());
}
}
}
}

```

E.6 Client Request

```

/*
 * JiniSDP.java
 *
 * Created on June, 2002
 */

/**
 * @author skasper, lbuehrer
 * @version 0.1
 * @comment Client routine in order to make SDP-Jini requests.
 */

import java.io.*;

public class JiniSDP {

    public JiniSDP () {
    }

    //-----
    //Basic SDP functions, provided by BlueZ

```

```

    public static void search1 (String service){
try{

    Process c = Runtime.getRuntime().exec("sdptool search " + service);
    c.waitFor();

}catch (Exception e){}
}

    public static void search2 (String service){
try{

    Process c = Runtime.getRuntime().exec("sdptool search " + service);
    c.waitFor();
    InputStream in = c.getInputStream();
    BufferedReader bu = new BufferedReader(new InputStreamReader(in));
    while(bu.ready()){
System.out.println(bu.readLine()+"\n");
    }
}catch (Exception e){}
}

    public static void search (String service, String bt_address){
try{

    Process c = Runtime.getRuntime().exec("sdptool search " + service + " " + bt_address);
    c.waitFor();
    InputStream in = c.getInputStream();
    BufferedReader bu = new BufferedReader(new InputStreamReader(in));
    while(bu.ready())
System.out.println(bu.readLine());

}catch (Exception e){}
}

//-----
//Usage and init functions

    public static void usage () {
System.out.println("Usage: runjinisdp [Service]\nServices:\n"
" JINIFTP\n FAX\n DUN\n LAN\n OPUSH\n");
System.exit(0);
}

    public static void init () {

```

```

System.out.println("*****\n");
System.out.println("Request to Jini-Bluetooth bridge in progress ...\n");
System.out.println("Please wait patiently :-)\n");
System.out.println("*****\n");
}

//-----
//The main class

    public static void main (String args[] ) {
try{
    String test = args[0];
}catch(Exception ex){usage();}

if(args[0].equals("-help"))
    usage();

init();
search1(args[0]);
try{
    Thread.sleep(10000);
}catch(Exception ex){}
System.out.println("Response from the SDP Bridge, asked via sdptool:\n");
search2(args[0]);
System.out.println(" \n");
System.out.println("*****\n");
}
}

```

E.7 Scripts

Compile the Service

```

#!/bin/sh
echo compile the service ...
javac -classpath \
/usr/lib/jini1_2/lib/jini-core.jar:\
/usr/lib/jini1_2/lib/jini-ext.jar:\
/usr/lib/jini1_2/lib/sun-util.jar:\
/home/skasper/service \
-d /home/skasper/service \
/home/skasper/myService/JINIFTPServiceInterface.java \

```

```

/home/skasper/myService/JINIFTPService.java

mkdir /home/skasper/service-dl/myService
cp /home/skasper/service/myService/JINIFTPServiceProxy.class
    /home/skasper/service-dl/myService/

echo compiling finished

```

Start a HTTP Server

```

#!/bin/sh
echo Webserver 8080 running ...
java -jar /usr/lib/jini1_2/lib/tools.jar -port 8080 -dir /usr/lib/jini1_2/lib -verbose

```

Start the RMI Daemon

```

#!/bin/sh
echo rmi running ...
rmid -J-Dsun.rmi.activation.execPolicy=none

```

Start the Lookup

```

#!/bin/sh
echo lookup started
rm -r /tmp/reggie_log/
JINI_HOME=/usr/lib/jini1_2
HOSTNAME=pc-3637
POLICY=${JINI_HOME}/example/lookup/policy.all
JARFILE=${JINI_HOME}/lib/reggie.jar
CODEBASE=http://$HOSTNAME:8080/reggie-dl.jar
LOG_DIR=/tmp/reggie_log
GROUP=public
java -jar $JARFILE $CODEBASE $POLICY $LOG_DIR $GROUP

echo lookup running ...

```

Start the Service

```

#!/bin/sh
echo running the service ...
java -cp \
/usr/lib/jini1_2/lib/jini-core.jar:\
/usr/lib/jini1_2/lib/jini-ext.jar:\
/usr/lib/jini1_2/lib/sun-util.jar:\

```

```

/home/skasper/service \
-Djava.rmi.server.codebase=http://pc-3637:8085/ \
-Djava.security.policy=/home/skasper/myService/policy \
myService.JINIFTPService

```

Compile the Bridge

```

#!/bin/sh
echo compile the bridge ...
javac -classpath \
/usr/lib/jini1_2/lib/jini-core.jar:\
/usr/lib/jini1_2/lib/jini-ext.jar:\
/usr/lib/jini1_2/lib/sun-util.jar:\
/home/skasper/client \
-d /home/skasper/client \
/home/skasper/myService/JINIFTPServiceInterface.java \
/home/skasper/myService/Bridge.java

echo compiling finished

```

Run the Bridge

```

#!/bin/sh
clear
/usr/lib/j2sdk1.4.0/bin/java -cp \
/usr/lib/jini1_2/lib/jini-core.jar:\
/usr/lib/jini1_2/lib/jini-ext.jar:\
/usr/lib/jini1_2/lib/sun-util.jar:\
/home/skasper/client \
-Djava.security.policy=/home/skasper/myService/policy \
myService.Bridge $1

```

Run the Client Request

```

#!/bin/sh
clear
java JiniSDP $1

```

Bibliography

- [1] TIK Homepage:
<http://www.tik.ee.ethz.ch>
- [2] ETH Homepage:
<http://www.ethz.ch>
- [3] Sun's Jini Homepage:
<http://www.sun.com/software/jini>
- [4] Infrastructure for Networking at the Edge:
<http://www.sun.com/software/jini/whitepapers/jiniEDGEwp052501.pdf>
- [5] Sun's Java Homepage:
<http://java.sun.com>
- [6] Bluetooth' Homepage:
<http://www.bluetooth.com>
- [7] Bluetooth Protocol Specification:
http://www.bluetooth.com/pdf/Bluetooth_11_Specifications_Book.pdf
- [8] Bluetooth Profile Specification:
http://www.bluetooth.com/pdf/Bluetooth_11_Profiles_Book.pdf
- [9] Bluetooth at red-m:
<http://www.red-m.com/products/technology/bluetooth/Default.asp>
- [10] Zucotto Wireless Technologies:
<http://www.zucotto.com>
- [11] Jini Surrogate Architecture Specification:
<http://www.jini.org/standards/sa.pdf>
- [12] Jini in a Cellular Telephone (Master Thesis):
<http://www.efd.lth.se/~d97fh/jini.pdf>
- [13] Jini IP Interconnect Specification:
<http://www.jini.org/standards/sa-ip.pdf>

-
- [14] Microsoft's Universal Plug and Play Homepage:
<http://www.upnp.com>
- [15] Salutation's Homepage:
<http://www.salutation.org>
- [16] Mapping Salutation Architecture API to Bluetooth Service Discovery Layer:
<http://citeseer.nj.nec.com/251752.html>
- [17] Bluetooth ESDP for UPNP:
http://www.bluetooth.com/pdf/ESDP_UPnP_0_95a.pdf
- [18] Bluetooth PAN Profile:
http://www.bluetooth.com/pdf/PAN_Profile_0_95a.pdf
- [19] IP over Bluetooth:
<http://opensource.nus.edu.sg/projects/bluetooth/others/lcn99-bt1.pdf>
- [20] Jini Projects:
<http://www.jini.org/servlets/ProjectList?type=Projects>
- [21] Bluetooth Network Encapsulation Protocol:
http://www.bluetooth.com/pdf/BNEP_0_95a.pdf
- [22] Java Protocol Stack for Linux:
<http://sourceforge.net/projects/bluej>
- [23] Jini Lookup Service Specification:
<http://www ldc.usb.ve/docs/java/jini/lookup.pdf>
- [24] D. Flanagan. *Java in a Nutshell*. O'Reilly, 1999.
- [25] H. Wong and S.Oaks. *Jini in a Nutshell*. O'Reilly, 2000.
- [26] W. Edwards. *Core JINI*. Prentice Hall, 1999.
- [27] Howtos on Handhelds Homepage:
<http://www.handhelds.org/minihowto/>
- [28] Howto Configure BlueZ on a Desktop PC:
<http://bluez.sourceforge.net/howto/>
- [29] Howto Configure BlueZ on iPAQ:
<http://www.handhelds.org/z/wiki/IpaqLinux>
- [30] BlueZ: Bluetooth Stack for Linux:
<http://bluez.sourceforge.net/>

-
- [31] Blackdown JRE:
http://www.tml.hut.fi/~jsantala/feed/blackdown-jre_1.3.1-RC1_arm.ipk
- [32] Suse's Homepage:
<http://www.suse.com>
- [33] Debian's Homepage:
<http://www.debian.org/>
- [34] Familiar Linux:
<http://familiar.handhelds.org/>
- [35] Forte for Java:
<http://forte.sun.com/eap/>
- [36] LateX Typesetting System:
<http://www.latex-project.org/>
- [37] Emacs Editor:
<http://www.gnu.org/software/emacs/emacs.html>
- [38] Adobe Tools:
<http://www.adobe.com>
- [39] Compaq's iPAQ:
<http://www.compaq.com/products/handhelds/index.html>
- [40] Skiff Cluster Project:
<http://www.handhelds.org/projects/skiffcluster.html>