THE UNIVERSITY OF BIRMINGHAM

Department of Computer Science

MEng Computer Science/Software Engineering

Bluetooth Database Bridge

James Harvey

Supervised by Dr. Behzad Bordbar

April 2005

DBlue

http://www.dblue.co.uk/
http://dblue.sourceforge.net/

Abstract

This report is concerned with the design and implementation of a J2ME API library and desktop server daemon allowing development of database-driven mobile computing applications over Bluetooth. It proposes a solution to the problem based on distributed system theory and discusses a proprietary Bluetooth messaging protocol.
The target audience includes developers and database/network administrators.

Keywords: Mobile, Bluetooth, Database, API, Java

Acknowledgements

First and foremost, thank you to my project supervisor Dr.Behzad Bordbar for his enthusiasm and friendship throughout the project. His guidance and advice has been invaluable to making it a success. I would also like to thank my family and friends for their care and continual support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Mobile computing devices have become a ubiquitous part of modern living, maturing substantially from their inception to become the fast and powerful platforms available today. Users no longer expect basic device functionality and increasingly look to them as convergence machines serving multiple purposes. For example, mobile phones have diversified their functionality by becoming mobile media centres enabling playback of audio and video files "on-the-move". This desire for enhanced functionality leads to increasingly advanced software to address these needs. Recognising the potential of the market, Sun Microsystems released a mobile version of their popular Java platform, J2ME. It has achieved mass-market adoption with over 267 million Java-enabled devices worldwide [29], supported by software written by thousands of developers.

At the heart of all software, mobile or otherwise, is information. Desktop applications are often database-driven, obtaining the information required by the user from a third-party data source. Mobile applications are unable to adopt this paradigm as they are often resource constrained and simply do not have the memory capacity to store all the information that could possibly be requested by the user.

The project aims to solve this problem by allow developers to produce database-driven mobile applications using wireless technology. Bluetooth is a wireless communication standard [27] that is found in millions of diverse mobile devices and enables data to be transmitted over a short range. Mobile applications will use Bluetooth to communicate with legacy databases residing on a database server. This will be achieved by creating a J2ME database library and an application to be installed on the database server enabling Bluetooth access, thus bridging the mobile device with the database.

Applications of this project are restricted only by the developers imagination. For example, a restaurant could distribute Bluetooth-enabled PDAs to its waiters which automatically check the database for the menu listing. If someone wishes to know the soup-of-the-day, the PDA automatically queries the database to find out what it is. Similarly, waiters can submit orders to a database accessible by chefs without having to return to the kitchen, improving productivity and reducing risk of lost orders. Another example could be an electronic message board whereby users submitting messages to a database. Once a mobile device is in Bluetooth range of the server, the software queries the database for any unread messages for that particular user.

The report assumes the reader is a competent computer scientist and has knowledge of software engineering and object-oriented programming principles. Background research material is outlined and explained, leading to an analysis of the system and derivation of a requirements definition. The design chapter outlines the decision making process and structure of the software, followed by a number of technical implementation details. Testing of the software and project management approach are also analysed. Finally, an evaluation of the final product is made and conclusions drawn. The appendices contain information referred to within the report and relevant supplementary documents.

Chapter 2

Background Research

2.1 Bluetooth

Bluetooth is an open specification for a short range, low-power radio technology designed to allow devices such as personal digital assistants (PDAs) and mobile phones to communicate in a wireless, ad-hoc fashion. [27]

Ericsson Mobile Communications commissioned a study in 1994 to investigate the feasibility of a cable-replacement solution for connecting it's phones to accessories. In 1998, the Bluetooth Special Interest Group (SIG) was formed[1] to take the proprietary wireless-technology developed by Ericsson and develop it as an open specification. [30]
The first specification version (1.0) was published in 1999. At the time of writing, the specification has matured to a fourth-generation version (1.2), with a fifth-generation (2.0) drafted. [18]

2.1.1 Network Topology

Bluetooth is a technology designed for ad-hoc networking (see 2.1). Such networks are limited by a device's radio power class and typically extend approximately 10 metres from the device. [27]
Networks of this nature are known as *Personal Area Networks* (PAN). A PAN is a network formed when two or more devices are connected in local space.
The PAN topology is also referred to in the literature as a *piconet*. This topology type is very similar to a conventional client-server architecture [4] and is illustrated (see Figure 2.1) by a simple example.

- A Bluetooth device registers itself as either a *master* or a *slave*
- If the device is a *master*, it must register zero or more services that is is making available to slaves
- If the device is a *slave*, it will search for all available local devices to find a *master* with the service is requires
- Once a *slave* finds a *master*, it will query the *master* for all available services
    - If the query finds the required a *service*, it will attempt to connect to that *service*
- Once connected, data is exchanged

[1]Initially founded by Ericsson, Intel, IBM, Nokia and Toshiba, it currently has hundreds of members

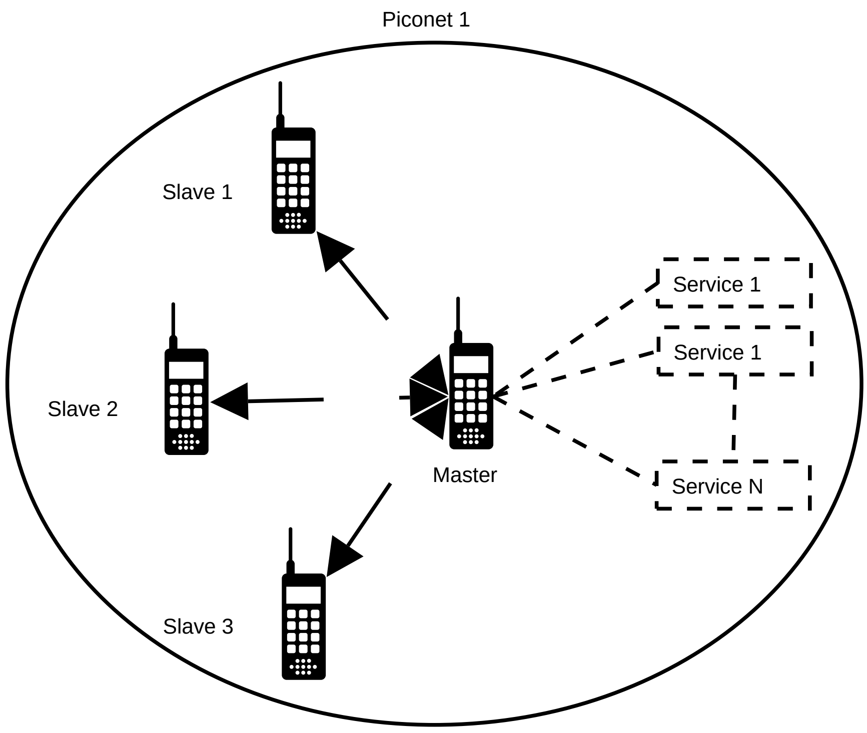


Figure 2.1: Example of a 3 slave piconet

- Each *piconet* consists of a single *master* with up to seven *slaves*

2.1.2 Specification Architecture

The Bluetooth specification describes the functionality of a compliant device in terms of a protocol stack (see Figure 2.2) similar to that of the *Open Systems Interconnection* (OSI) model. [4]
The *Host Controller Interface* (HCI) separates and provides communication between the host's software and the host's physical hardware. Application developers do not need to be concerned with the details of any hardware implementation. The protocol stack provides a number of strictly defined interfaces for developers to use. The highlighted boxes (see Figure 2.2) [18] [16] represent the protocols addressed by the *Java API for Bluetooth wireless technology* (JABWT) (see 2.3). An overview of these protocols is shown in Table 2.1. [18] [13] [27]

Protocol	Description
Object Exchange (OBEX)	Adopted protocol defined by the Infrared Data Association (IrDA). Allows an application to describe the structure of data using headers as well as its content.
Radio Frequency COM (RFCOMM)	Provides emulation of an RS-232 serial port. Allows applications to communicate as though devices are physically connected via a COM port.
Service Discovery Protocol (SDP)	Defines a standard method for the discovery and query of services (see 2.1.1) being made available by other local devices.

Logical Link Control and Adaption Protocol (L2CAP)	Shields higher layer protocols from lower level data presentation details. It is responsible for segmenting data into packets before passing to the HCI layer, as well as reconstruction of segmented incoming packets. This layer also provides functionality to synchronise grade of service level between connected devices. Details of this are beyond the scope of this paper. All data must pass through this core layer of the stack.

Table 2.1: Bluetooth Specification Protocols

In addition to the *protocol groups* (see Table 2.1), the Bluetooth specification also defines a number of *profiles*. A *profile* defines the protocols and features of the specification that can be used for a particular usage scenario. There are 16 profiles, each being a member of one of three major groups. An overview of these major groups is shown in Table 2.2. [27]

Profile Group	Description
Generic Access Profile (GAP)	Provides a number of generic functions that allow connections between two devices. Such functions include device discovery, link management and functionality for security. All other profiles are based upon GAP.
Serial Port Profile (SPP)	Allows devices to communicate as though connected by a serial port. This profile directly interacts with the RFCOMM protocol layer (see Table 2.1)
Generic Object Exchange Profile (GOEP)	Defines a number of set usage cases for applications requiring OBEX communication (see Table 2.1). These profiles define unique headers to describe data for each particular case.

Table 2.2: Bluetooth Specification Profiles

The interested reader should refer to Bluetooth Revealed [27] or the Bluetooth Specification [30] for further hardware, middleware and profile details.

2.1.3 Security

The Bluetooth Specification defines a security model based upon three components; *authentication*, *authorisation* and *encryption*. All aspects of security are managed by the *Bluetooth Control Centre (BCC)* and are not accessible by developers.

Bonding and Pairing

Bonding is the procedure in which one device authenticates itself to another by using a shared *authentication key*. If no key exists between two devices, one is generated based on a shared secret PIN. The PIN is usually exchanged verbally between users wishing to bond their devices. Once an *authentication key* has been generated, each device stores it for future use to allow devices to bond in the future without repeating the pairing process.

Encryption

Once devices are bonded, either of the pair may request the communications channel to be *encrypted*. The pair negotiate an *encryption key*, attempting to use the largest possible length supported by both devices. International encryption export restrictions limit this key to between 8-128 bits, a length that can theoretically be broken.

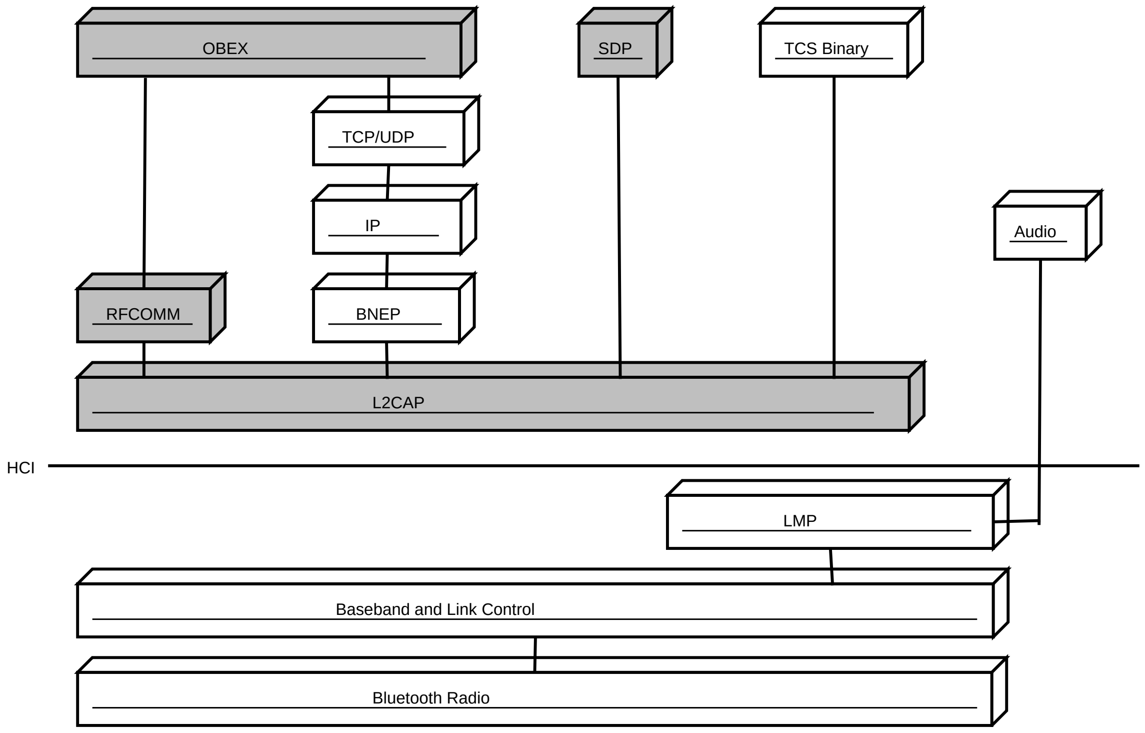


Figure 2.2: The Bluetooth Protocol Stack

Authorisation

Authorisation is the process of a *master* device permitting a slave to utilise a particular *service*. Access to the *service* may be granted on a temporary or permanent basis.

Security Modes

There are three security modes define in GAP (see Table 2.2).

- No Security
  - Communication is not *encrypted*, devices are not *authenticated* and *service* access is permanently granted. The most insecure mode.
- Service Level Security
  - Individual *services* determine the required level of security. This can be set by developers during *service registration*.
- Link Level Security
  - Any use of Bluetooth link must be secured using all three security model components before transmission will occur.

2.2 Java Micro Edition (J2ME)

Java Micro Edition (J2ME) is a highly optimised, light-weight Java runtime environment designed specifically for resource constrained devices such as mobile phones, PDAs, set-top boxes and other consumer goods. [29] [13] [18]

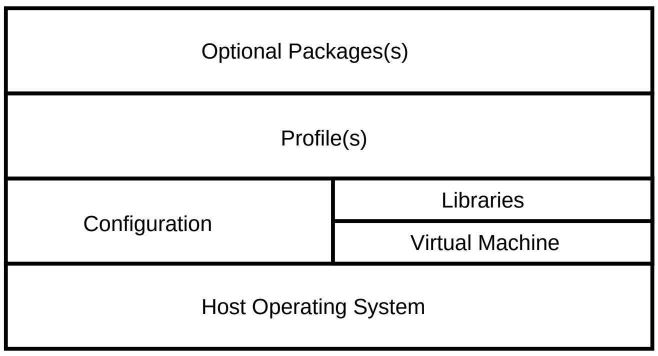


Figure 2.3: J2ME Framework

J2ME has achieved mass market adoption with over 267 million Java enabled handsets worldwide, with estimates placing this figure at 1.5 billion by 2007. [29] The platform is increasingly stable and has matured significantly as part of the *Java Community Process* (JCP) over the past five years.

Similar to its desktop and enterprise counterparts, J2ME defines a language and virtual machine (VM) specification. The functionality of this machine is implemented by a device's *original equipment manufacturer* (OEM) who must implement the VM specification to interact with their underlying operating system.

2.2.1 Configurations

To ensure performance, the all-encompassing base J2ME environment is optimised for sets of devices by the use of *configurations*. A *configuration* [16] defines the minimum functionality required by a VM and associated class libraries. Devices that share similar characteristics, such as available memory and processor speed, are grouped into *configuration sets*.
Currently there are two J2ME configurations; *Connected Limited Device Configuration* (CLDC) and *Connected Device Configuration* (CDC).

Connected Limited Device Configuration

The CLDC (JSR-30) defines a much-reduced subset of the J2SE class library. It also defines the specification of a *kilobyte virtual machine* (KVM). Devices that use the CLDC are those with limited memory, intermittent network connections, slow processors and require a VM with a minimal resource footprint. [32] Examples include mobile phones, low-end PDAs, pagers and other mobile devices.

Connected Device Configuration

The CDC (JSR-36) is an enhanced version of the base CLDC. It defines a larger subset of the J2SE class library, as well as a full Java virtual machine (JVM). CDC enabled devices have more available memory, greater network bandwidth, faster processors and are less resource constrained. Examples include set-top boxes and high end PDAs.

2.2.2 Profiles

In addition to *configurations* (see 2.2.1), the J2ME environment is further customised by *profiles*[1]. A *profile* defines a set of APIs used by a narrower category of device within a *configuration*, creating an environment targeted for a more specific device type. [18] For example, profile APIs may give the programmer a set of classes for GUI forms tailored to a specific device screen such as a mobile phone. Currently there are three profiles; *Mobile Information Device Profile* (MIDP), *Foundation Profile* (FP) and *Personal Profile* (PP).

- Mobile Information Device Profile
  - This profile is designed for mobile phones and pagers and is most often used with CLDC. It provides user interface (UI), network and storage classes.
- Foundation Profile
  - This is the base profile for CDC, and is designed for embedded devices without a UI. It cannot be used by CLDC devices.
- Personal Profile
  - Built on Foundation Profile, this profile is for high-end PDAs and game consoles. It provides all UI, network and storage classes required by CDC devices. There is also a subset of Personal Profile, *Personal Basic Profile (PBP)* which is targeted at CDC devices with limited graphical abilities.

2.2.3 Optional Packages

The final part of the J2ME framework are *optional packages*. These define features that can be included to give enhanced functionality to the environment. If an optional package is included, the OEM must implement whatever functionality that package provides with their hardware/operating system. An ever increasing number of packages are available to OEMs, however, the one of most relevance is the *Java API for Bluetooth wireless technology* (JABWT), JSR-82. (see 2.3).

2.3 Bluetooth and Java

The Java programming environment is a "write-once-run-anywhere" language designed to be platform and hardware independent. However, the Bluetooth specification (see 2.1.2) only outlines what protocols and functionality a compliant device should implement, not the interfaces they should provide. This makes it impossible for code using Bluetooth functionality to be portable as OEMs may implement functionality differently.
To solve this, the Java Community Process (JCP) released a Java specification (JSR-82) in March 2002 defining a set of Java APIs to be used when integrating Bluetooth functionality into a Java environment. JSR-82 defines functionality for:

- Device discovery
- Service discovery
- Service registration
- Serial Port Profile

[1]The reader should not that *profile* is a term used in both J2ME and Bluetooth contexts

- Generic Object Exchange Profile
- L2CAP layer manipulation

2.3.1 J2ME Bluetooth

An OEM distributing devices with a J2ME environment (see 2.2) may chose to include the JSR-82 optional package. If included, it must be implemented to work with *their* Bluetooth stack. At the time of writing, mobile devices implementing JSR-82 are limited. Many devices are not fully JSR-82 compliant and do not implement OBEX. The Nokia 6600 has been selected as a development platform for the project as it is JSR-82 compliant and is readily available to the author for programming.

2.3.2 J2SE Bluetooth

J2SE does not provide an implementation of JSR-82 as part of the J2SE. This functionality is provided by third-party *software development kits (SDK)* which act as an intermediary between the Bluetooth device software and J2SE. These SDKs are often limited to specific operating system environments. Currently there are six SDKs available, as outlined in Table 2.3.

Name	javax.bluetooth support	javax.obex support	Operating System	Price	Notes
Atinav	Yes	Yes	Win-32, Linux	$4995	100% Java, No education license available
Avetana	Yes	Yes	Win-32, Linux, MacOS X	25 Euro	100% Java, commercial license free for Linux
Blue Cove	Yes	No	WinXP SP2	Free	Early development stage
Harald	No	No	Any supporting J2SE	Free	100% Java
JavaBluetooth	Yes	No	Any supporting J2SE	Free	Software development inactive (Last release October 2003)
Rococo	Yes	Yes	Linux	2500 Euro	Free education license available

Table 2.3: J2SE Bluetooth (JSR-82) Implementations

In addition to SDKs, there are also a number of simulation development kits available. However, the project implementation is intended to use real hardware so these have not been researched.

2.4 Java Database Connectivity

The Java Database Connectivity (JDBC) API is a set of classes for J2SE/EE that provides Java applications with relational-database connectivity. It is SQL compliant, adhering to the SQL99 standard to ensure access to the broadest range of data sources. [7]

To enable connectivity to a data-source, the database must implement the JDBC specification as part of a *driver*. A *driver* is a database manufacturers implementation of the specification to work with their underlying database. The *driver* is then imported by a developer either at runtime or statically within their code to enable the application to use database functionality. There are currently 220 JDBC-compliant drivers available, including those for major industrial databases such as PostgreSQL, MySQL, Sybase and Oracle. [25]
Sun published the first release of the JDBC specification (1.0) in 1997; the latest version (3.0) was released in 2002. [7] The API is included as standard in J2SE/EE. A subset of the specification (JSR-169) is also included as an optional package (see 2.2.3) in J2ME as part of the CDC configuration.

2.5 Distributed Systems

A distributed system is a collection of networked computers which communicate by only message passing to achieve a common goal. [4] A well designed system should provide total transparency whereby the end user is completely oblivious to the distributed nature of the system.

2.5.1 Message Passing

Messages passed between networked computers are governed by *protocols*. A *protocol* defines a set of rules for interprocess communication including data syntax and semantics. It may also define miscellaneous attributes such as error correction. Protocols are either lower-level (e.g. specifications for raw bit manipulation) or high-level (e.g. more descriptive definitions).

2.5.2 System Architecture

The most common distributed system architecture is the client-server model [4]; a server makes a number of services available to its clients which connect and use the service. Typically, a client sends a message to the server, which computes a result and then sends it back to the client. This is known as the *request-reply protocol*. [4]

2.5.3 Java Remote Method Invocation

The request-reply protocol is implemented in Java by the Remote Method Invocation (RMI) API. [24] This implementation allows methods of remote objects to be called from other, possibly non-local, JVMs. Java RMI is available for J2SE and J2EE. A full implementation is also available for J2ME, however this is limited only to the CDC. This is because the CLDC does not support object serialisation, a core part of Java RMI.

Chapter 3

Analysis and Specification

3.1 Problem Analysis

Analysis of the core-technologies researched to implement the Bluetooth database bridge is undertaken to formulate a requirements specification and give the project a definitive direction.

The software was initially outlined to consist of two major components; a J2ME development library and a server-based application to enable Bluetooth access for existing databases. Research into the Bluetooth *piconet* topology (see 2.1.1) is based upon a *master* making services available to *slaves*, similar to the client-server topology found in the distributed systems research [4]. As identified (see 2.5.2), the request-reply protocol found in the client-server topology shifts processing loads from the client and onto the server. By applying the *request-reply protocol* to the project, load can be moved away from the resource-constrained mobile device and onto the more capable server machine.

Proposals for development in the J2ME environment have also been confirmed by the background literature. Java is the obvious choice for the project as it is based upon a virtual machine architecture and abstracts away from operating system specifics, enabling the software to operate on a number of different devices. It also has the widest support in the mobile device sector, providing an stable and established object-oriented environment. However, it provides two configurations, CLDC and CDC (see 2.2.1). Code developed for CDC is not backwards compatible with CLDC. Therefore the project should be designed and implemented for the base configuration (CLDC) to ensure maximum portability to a wider variety of devices.

Implementation of Bluetooth software in the desktop environment requires the selection of a third-party JSR-82 implementation SDK (see 2.3.2). The potential SDKs (see Table 2.3) were assessed using the following criteria:

- JSR-82 compliance to ensure standardised code
- Range of supported Operating Systems to ensure portability
- Cost and license availability

Blue Cove has not been selected as it supports only a limited environment (WinXP SP 2) which is unsuitable for this project.
Harald is not JSR-82 compliant and although any code written using this SDK would be portable to any J2SE runtime environment, it would be proprietary in nature.
JavaBluetooth supports partial JSR-82 compliance, but the SDK is still in the pre-alpha development stage and is too unstable to be used. The SDK is currently inactive with no future releases

planned.
Atinav passes the compliance and OS assessment criteria, however, at $4995 with no educational license available, it is not financially viable for this project.
Rococo provides complete JSR-82 compliance, as does Avetana. However, Rococo is limited only to Linux whereas Avetana provides support for Win-32, Linux and MacOS X. A full commercial license for Rococo is 2500 Euro compared to only 25 Euro for Avetana, which is free for Linux. If the project is deployed in a commercial environment, it is important that the SDK it relies upon is affordable. For these reasons, Avetana has been selected as the SDK to develop a J2SE application.

Investigations into database-driven desktop software showed JDBC to be the most significantly used Java API. If possible, the software should use this existing database technology to ensure a stability base for the project.

Based upon the background material and initial aims of the software, a specific requirements definition has been outlines. See 3.2 for details.

3.2 User & Non-Functional Requirements Definition

1. The system should provide a mobile development API and a desktop administration application
2. Mobile devices should be able to connect to a relational-database via Bluetooth
   (a) Mobile devices should be able to connect to multiple databases simultaneously
   (b) Multiple mobile devices should be able to connect to the same database simultaneously
   (c) The Bluetooth communication process should be completely transparent
3. Administration of wireless databases should be simple
   (a) Any legacy relational-database should be accessible by a mobile device
   (b) The bridge should require minimal user input
4. The developer API should be logical, intuitive and easy to learn
   (a) API should be well documented
5. The system should be stable and robust
   (a) If errors occur, the system should try and automatically recover
6. The system should be efficient
   (a) Resource usage (processor, memory and communication bandwidth) on mobile devices should be as small as possible
   (b) Database transactions should be efficient
7. The system should portable and operate in a variety of hardware and software environments
8. The system should be well designed to allow possible extensions
   (a) Source code should following object-oriented and software engineering principles
   (b) Code should be well documented

Chapter 4

Design

The design process is outlined in this section. Design decisions were based upon the requirements specification and the appropriate background material.

4.1 System Design

The requirements definition specifies that the system should provide two major components; a mobile development API and a desktop administration application (see 3.2). Each of these components is optimised and designed for their particular runtime environment and hence discrete in their nature. Herein, the mobile development API shall be referred to as the *Client Component*, and the desktop administration application the *Server Component*.

As shown in Figure 4.1, the system operates using a basic distributed system, RMI framework. [24] The *client component* creates a number of *stub* objects (M^n) which are pointers to real objects contained in the *server component* (S^n). Methods called by a third-party programmer on M^n result in a message being sent via Bluetooth to the *server component* to invoke the appropriate method on S^n. Acting on behalf of the *client component*, S^n then executes the required method on the database. The result is this invocation is then sent back to the *client component*.

All *server component* objects (S^n) are instances of JDBC classes which derive input from the *remote component*. This allows the server to use stable, well tested, existing code and build on top of

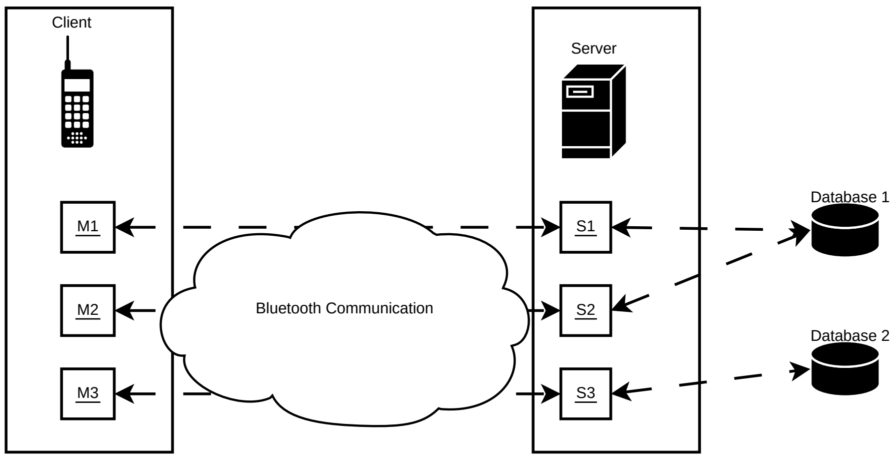


Figure 4.1: High-level System Architecture

it, improving overall robustness and reliability. It also allows the system to connect to a range of legacy-databases, assuming the appropriate driver is available.
As the *client component stub* objects simply send messages to the server, they can be programmed using the same method definitions as found in JDBC. This allows developers to port existing classes from the desktop/enterprise environment to a mobile device with very little code change. It also facilitates developer knowledge re-use as the software has an already strong JDBC documentation base.
Basing the software on a distributed JDBC platform allows devices to connect to multiple databases simultaneously thus fulfilling another of the requirements specification.
This design also allows for database enforcement of security. JDBC objects must supply credentials (such as User ID and password) to connect to a database. As *client component stubs* effectively "own" objects stored by the *server component*, the *client* must provide them with credentials to connect to a database. These credentials are verified by the database itself, meaning no additional security measures are required by the bridge. This reduces security risks as the project is no longer a "link" in the verification chain; it simply passes information for processing to other applications.

4.2 Module Identification

Major modules in each of the two components (*client* and *server*) will be identified and broad functionality outlined.

4.2.1 Client Component Modules

The *client component* has been designed to create the smallest possible resource footprint on the mobile device. Modules have been identified in such a way as to ensure strong cohesion and weak coupling, allowing for further future development.

API Module

The *API module* is the collection of database classes available to the mobile developer. These classes act as *stub* JDBC classes and generate messages to send to the server to invoke methods. This is the single point-of-entry into the API for the third-party developer. It provides total transparency to other levels of the system.

RMI Module

Due to the lack of distributed system support in the target CLDC environment (see 2.5.3), a proprietary RMI framework is required. The framework requires only basic distributed system functionality, most notably implementation of the *request-reply protocol* and *error correction* for stability. [4]

Communication Module

All Bluetooth communication is contained within the *Communication module*. It handles the process of *device discovery*, *service discovery* (see 2.1.1) and data transmission.

4.2.2 Server Component Modules

The *server component* has been designed to act as a *daemon*, running in the background on the server without any user interaction. This makes the bridge easy to setup and use. As with the *client component*, modules have been identified to ensure strong cohesion and weak coupling.

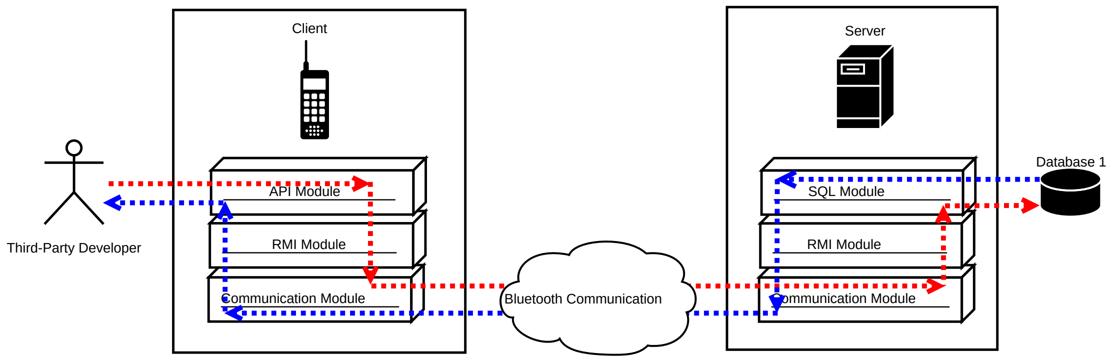


Figure 4.2: Client-Server Component Interactions

SQL Module

This module maps incoming requests from remote *stub* objects to locally stored JDBC objects. It is responsible for executing methods on backend databases and returning the result.

RMI Module

Although J2SE/EE provides native support for RMI [24], it is useless in this application due to the proprietary nature of the RMI model employed in the *client component*. Investigations into mapping incoming proprietary requests to the native Java RMI API were abandoned early into the design phase when it became clear that the *client component RMI module* could instead be optimised for the desktop environment. This maximises code re-use and reduces external entity dependency. (Java RMI).

Communication Module

As in the client-side design, communications have been separated from the rest of the system. This module handles service registration and data transmission.

4.2.3 Module Interaction

The modules identified in the *client* and *server* components interact in the following way to facilitate the RMI schema required (see Figure 4.2). The red lines show the flow of data for the *request*; the blue lines show the flow of data for the *reply*.

4.3 Client Component Module Design

4.3.1 API Module

Design Goals

- Provide a single point of entry to the *client component* for third-party developers
- Create *stub* objects for sending messages to *server component*
- Give the developer configuration options for the *client component*

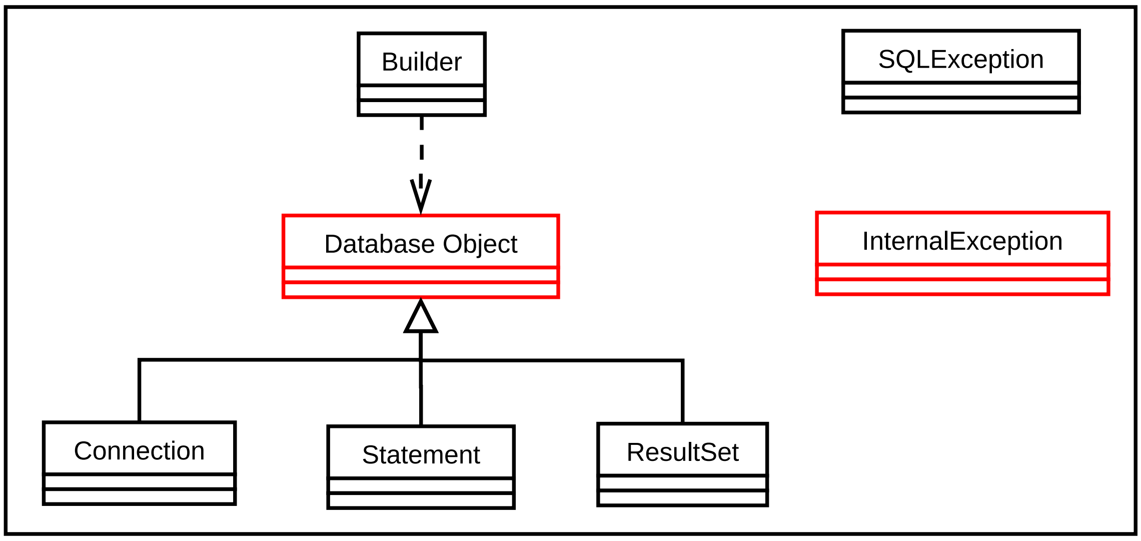


Figure 4.3: Client component API module architecture

Overview

This module is the single point of entry into the *client component*; all lower modules are transparent to the developer. The module consists of five core classes (black) for database connectivity which the developer is able utilise (see Figure 4.3). Each of these classes are *stubs* and form messages to be sent to the *server component* for remote method invocation. In addition, two helper classes (red) are defined for internal system use. Only the *Builder* object should be directly instantiated by the developer with all other objects being created from it.

Architecture

The *Builder* class is the core of the entire *client component*. It creates an instance of the *RMI module* (see 4.3.2) and the *communication module* (see 4.3.3) and facilitates inter-module communication. It also creates instances of any SQL objects and links them with the newly created RMI and Communication modules.
The developer can access use a single method from the *API Module*, *Builder.getConnection()*. This returns an instance of a *Connection* object, which is the key object in the database class hierarchy, with all other objects being instantiated from it.

The *Connection* class allows the developer to create a connection (or session) to a legacy database residing on the server. SQL statements and results are returned in the context of a *Connection*. Once the *Connection* object has been created, the developer is able to retrieve a *Statement* object from it. The *Statement* class is used for executing SQL statements and returning the results it produces. To access these results, the developer retrieves a *ResultSet* object from the *Statement* object. This class allows the developer to access the *Statements* results.

All database classes inherit from the *DatabaseObject* (see Figure 4.3) class. This base class provides functionality to format data from its sub-classes (*Connection*, *Statement* and *ResultSet*) before presenting it to the *RMI module*. Inheritance insures that the data is always presented in a standardised way and increases code re-use and reliability. All *DatabaseObject* methods call this superclass to format data to allow the *RMI module* to process it.

The *API module* objects implement a subset of the functionality provided by “real” JDBC objects.

This subset includes methods that return data which can be natively represented in J2ME. For example, CLDC does not support the *float* datatype and hence all methods returning *float* values are not implemented. Upon instantiation of any *DatabaseObject* sub-class, a message is sent from the *client component* to the server requesting it creates a "real", server-side object of the same type. This "real" object is assigned a unique identifier by the *server component* which is returned to the *client component* and assigned to the local object. Whenever that local object request a method to be invoked on the *server component*, this identifier is sent to indicate which server-object the invocation should occur on.

The module also provides the developer with a number of configuration options. The developer may set if the *client component* searches and connects to the *server component* instantly, or dynamically as determined by the API. For example, if the developer wishes to create an instance of the *Builder* object but then wait an arbitrary length of time for user input before making a database request, s/he may wish to search dynamically for the *server component* instead of an instant connection. This reduces the risk of the mobile device moving out of Bluetooth range from the server whilst waiting for user input.
Another developer option is to set the amount of resends the *client component* will attempt if a message fails to be delivered or is not correctly received by the *server component*. This gives the flexibility to limit the Bluetooth bandwidth usage as well as setting the error-correction vs system responsiveness balance according to the third-party applications use.
The final option is to set the *timeout limit*. This dictates how long the *client component* will wait for a response from the *server component* and is directly related to the *resend* option outlined above. A longer *timeout* and higher *resend* value may improve system reliability, but at the cost of performance. Hence, the decision is left to the developer.

4.3.2 RMI Module

Design Goals

- Take formatted date from the *API module* and create messages to be sent to the *server component*
- Interpret incoming messages and pass the result back to the *API module*
- Create a standardised messaging system for use in both *client* and *server components*
  - Enforce anti-deadlocking mechanisms to ensure software does not hang
  - Attempt to guarantee message delivery between the *client* and *server components*
  - Ensure the messaging system is flexible and easy to change

Overview

The purpose of the *RMI module* is to control and implement the *RMI framework*. Conceptually the module is arranged as a stack of classes which implement the *Message Exchange Protocol* (MEP), similar to the OSI model. This ensures that message processing is encapsulated within layers, thus reducing class inter-dependency and hiding the complexity and implementation of lower layers. Acting as middleware in the overall system design, it communicates with adjacent modules (*API* and *communications*) and hence has I/O interfaces at both the top and bottom of the stack.

The Message Exchange Protocol

The *Message Exchange Protocol* (MEP) is designed to create a standardised way of exchanging data between the *client* and *server components* and is XML-based. The protocol defines the syntax and

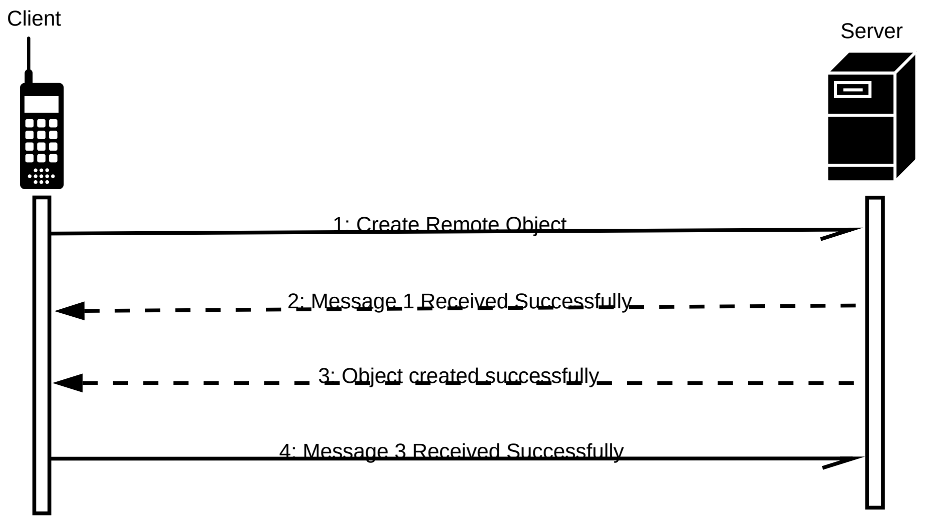

Client
Server
1: Create Remote Object
2: Message 1 Received Successfully
3: Object created successfully
4: Message 3 Received Successfully

Figure 4.4: A Simple Successful Message Exchange

semantics of messages and also defines a message acknowledgement protocol, ensuring messages are received. It also attempt tries to use the minimum amount of bandwidth possible. [9]

XML was chosen as it is easy to read and write in the resource-constrained *client component* environment [11] and is non-proprietary potentially allowing *clients components* written in programming languages other than Java to communicate with the *server component*.

Acknowledgement Protocol : The most important aspect of the MEP is ensuring that applications implementing it cannot become deadlocked by waiting for a message that will never be received. An example of this is a device sending a message that requires a return message (such as a remote invocation) but it is never received as the device moves out of Bluetooth communication range. In this case, the *client component* is deadlocked as it will indefinitely for a return message. To counteract this scenario, the MEP has been developed with anti-deadlocking mechanisms. Due to its complexity, a subset of its features will be outlined with illustrated examples. Each example is based upon the same scenario: a *client component* requesting a new object to be created by the *server component*.

Simple Successful Exchange - Figure 4.4 : All messages sent using MEP are stamped with a signature, otherwise known as a *transactionID*. This is shown in the diagram as the number preceding each message. Once a message is received, the *server component* sends an *acknowledgement message* for that *transactionID*, thus informing the *client component* it was successfully received. The *server component* then sends the return data for the original "create object" message. The *client component* responds by acknowledging that this return message was received. This simple exchange identifies a key point: all non-acknowledgement messages must be acknowledged.

Non-Responsive Exchange - Figure 4.5 : This exchange illustrates the anti-deadlocking mechanism of the MEP. As shown, the *client component* sends the standard starter message to create a new object. However, after Time *t*, the message has not be acknowledged by the *server component* so it is re-sent using the same *transactionID*. Again, this re-sent message was not acknowledged so it is re-sent again. The *timeout* duration for message acknowledgement and the number of resends is determined by the implementor of the MEP. In this example, the message is acknowledged and the exchange outlined in Figure 4.4 is completed. However, had it reached the maximum number of resends without acknowledgement the protocol would infer that it was *never* going to receive

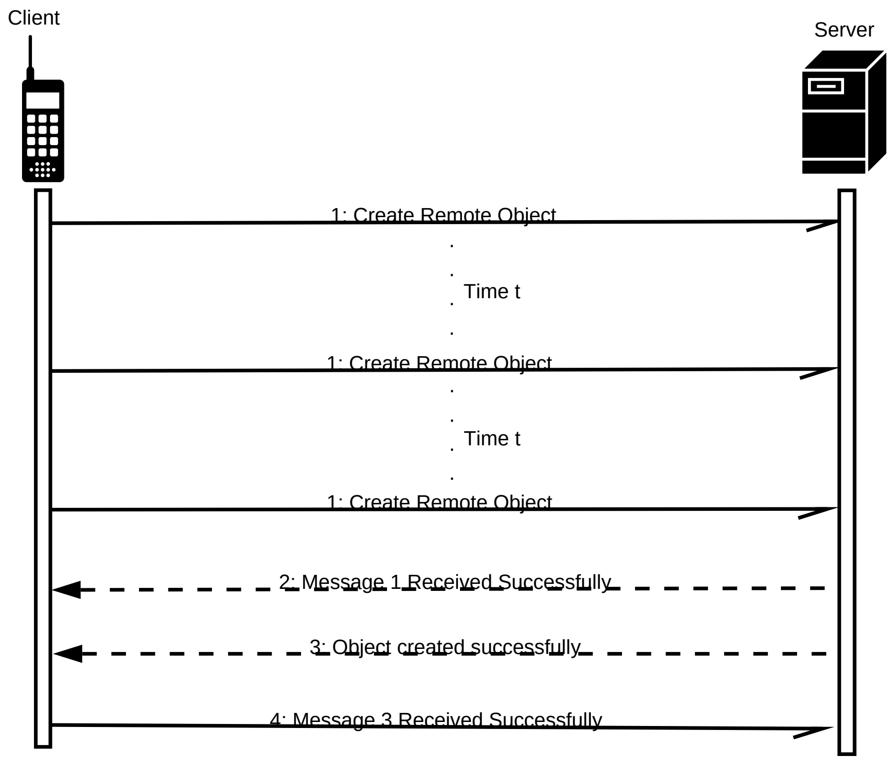

Client
Server
1: Create Remote Object
Time t
1: Create Remote Object
Time t
1: Create Remote Object
2: Message 1 Received Successfully
3: Object created successfully
4: Message 3 Received Successfully

Figure 4.5: A Non-Responsive Message Exchange

a message and it should stop attempting. In the example outlined in 4.3.2, the protocol would simply timeout and report that the connection had been lost.

Non-Responsive Exchange (Part II) - Figure 4.5 : This non-responsive exchange also illustrates the addition of *invocation semantics* to the protocol. Consider the exchange give in Figure 4.5, except each time a message is received from the *client component*, the *server component* acknowledges it but the *client component* never receives the acknowledgement. In accordance with 4.3.2, the original message will be re-sent by the *client component*. However, the *server component* has already created the new object as requested by the *client components* first message. Should it process the duplicate messages?
To solve this potential problem, *invocation semantics* are defined by the MEP and specify how the implementing application should handle duplicate messages. As standard, the MEP applies at-once-invocation; all duplicate messages result in an acknowledgement (to hopefully stop the other party sending duplicate messages) and the return message is resent back to the calling party; the duplicate message is not repeatedly invoked.

Simple Corrupt Exchange - Figure 4.6 : This example illustrates a simple case of message failure. Consider that the *server* is unable to extract the contents of a message sent by the *client component*. In this case, the *server* does not acknowledge the message, but *negatively-acknowledges* it. This causes the *client* to resend the message again. As in 4.3.2, the application determines how many times a message is re-sent before the protocol should stop.
If the *server* is unable to read the *transactionID* of the message, it sends special *negative acknowledgement* message asking for *all* messages sent from the *client* that have not been acknowledged to be re-sent.

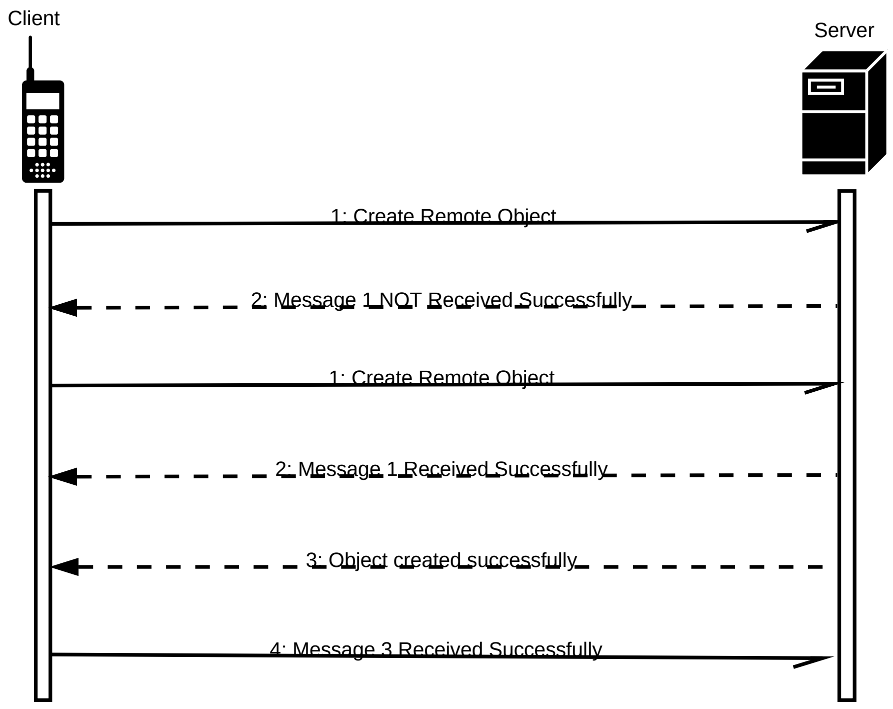

Client
Server
1: Create Remote Object
2: Message 1 NOT Received Successfully
1: Create Remote Object
2: Message 1 Received Successfully
3: Object created successfully
4: Message 3 Received Successfully

Figure 4.6: A Simple Corrupt Exchange

Disordered Message Exchange - Figure 4.7 : In this example, a return message is received by the *client component* before the message that caused the invocation has been acknowledged. In this case, the *client component* discards the message. This ensures that applications implementing the protocol MUST acknowledge all non-acknowledgement messages (see 4.3.2) and cannot have partial MEP implementation.[2]

Syntax and Semantics

The syntax and semantics of messages are governed by a *Document Type Definition* (DTD). DTDs are common within XML for defining valid document contents. The MEP DTD defines a set of shared tags that describe the semantics of the data contained within a message. These tags and their definitions are shown in Table 4.1.

Tag Name	Attributes	Definition
ACK	TRANSACTION_ID	**ACK**nowledges successful receipt of a message
NACK	TRANSACTION_ID	**N**egatively**ACK**nowledges receipt of a message
CRE	TYPE	Instructs the *server component* to **CRE**ate an object of the specified **TYPE**
DEL	IDENTIFIER	Instructs the *server component* to **DEL**ete the object with the corresponding **IDENTIFIER**
RET	TRANSACTION_ID, RETURN_TYPE, RETURN_DATA	Returns data from a remote invocation for a given **TRANSACTION_ID**, giving its **RETURN_TYPE** and the **RETURN_DATA**

[2]In this case, the *client component* would simply timeout waiting for an acknowledgement and resend the original message to the *server component*. This would cause the *server component* to re-send an acknowledgement *and* the return data, therefore not deadlocking.

METH	IDENTIFIER, METHOD, RETURN_TYPE, PARAMETER_NUMBER	Instructs the *server component* to invoke the **METH**od on the object with the corresponding **IDENTIFIER**. It contains **PARAMETER_NUMBER** parameters and expects the method should return the data of **RETURN_TYPE**. Note: this tag may contain 0 or more **PARAMETER** tags.
PARAMETER	PARAMETER_ID, PARAMETER_TYPE, PARAMETER_DATA	Instructs the *server component* the **PARAMETER_TYPE** and its **PARAMETER_DATA** of a **METH**od

Table 4.1: Message Exchange Protocol Tag Definition

RMI Stack

The stack (see Figure 4.8) implements the features required by the MEP (see 4.3.2). The left side of the stack deals with outgoing messages and the right, incoming. Note: the *Messaging* layer represents a package of classes that implement the tags defined in the MEP. At the top of the stack, requests from the *API module* to send messages are incoming and returned data from sent messages is outgoing. At the bottom of the stack, messages passed from the *communication module* are incoming and messages formed for sending are outgoing.
The dashed arrow represents raw data being passed into the module from the *communication module*, where M^i is the incoming message.

The *Receiver* class checks M^i for syntactic validity and strips the signature MEP data. It is then passed to the *Interpreter*.

The *Interpreter* class applies the at-once-invocation semantics to M^i. It then strips the MEP tag data and analyses the message type to decide which part of the module it should be passed to; this may be either the *Handshaker* or *Control* class.

The *Handshaker* processes incoming and outgoing acknowledgement messages in accordance with the MEP. It performs the following functions where M^i is:

- an incoming *acknowledgement*, sets the message the acknowledgement is for as acknowledged. Processing of M^i has now finished
- an incoming *negative acknowledgement*, resends the message the negative acknowledgement is for. Processing of M^i has now finished

The *Handshaker* also sends acknowledgements for messages received, where dictated by the MEP. It creates the data to be sent and then passes it to the *Control* classes which processes it as a regular message.

Control is core class of the *RMI module*. It accepts incoming requests from the *API module* above to send data and then passes it down the stack for processing. It maintains a list of sent messages and is responsible for messaging signing in accordance with MEP. Incoming data from the *Interpreter* layer is also processed, and if it is return data for a sent message, the result is returned to the callee from the *API module*.

The *Checker* implements the timeout and resend functionality described by the MEP. It periodically checks the *Control* class' list of sent messages for any that have not been acknowledged. It

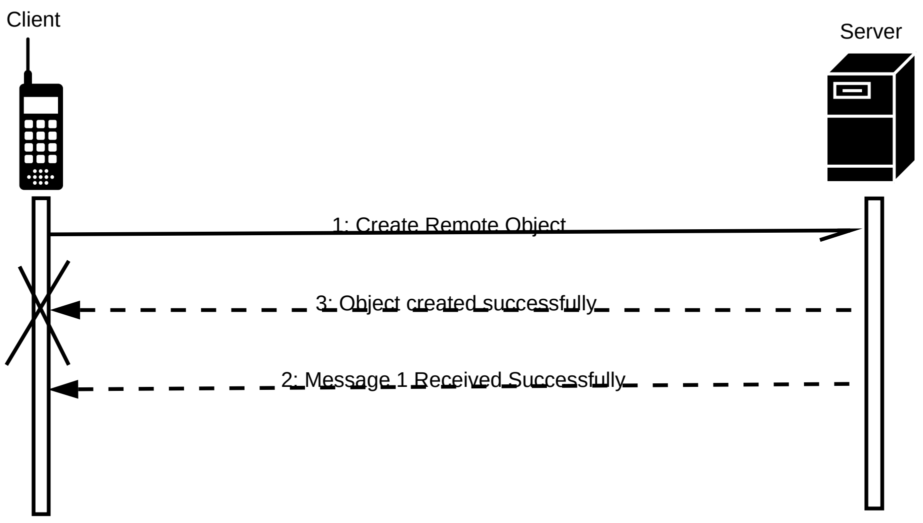

Client
Server
1: Create Remote Object
3: Object created successfully
2: Message 1 Received Successfully

Figure 4.7: A Disordered Message Exchange

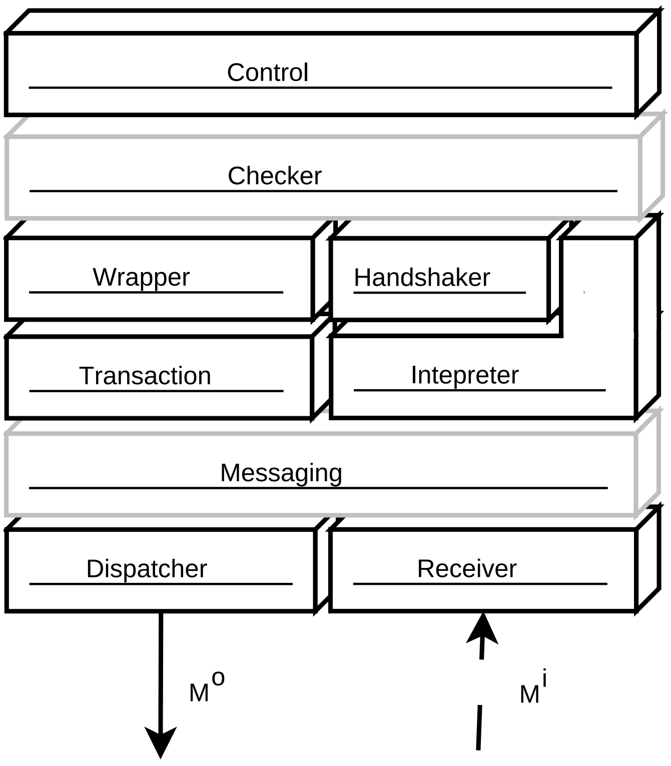


Figure 4.8: Client/Server component RMI stack architecture

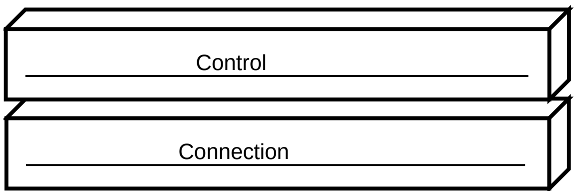


Figure 4.9: Client component Communication module architecture

then requests the *Control* class resends all messages currently not acknowledged.

The *Wrapper* class takes the raw data to be sent as a message and applies the MEP tag data. This creates a new message, M^o.
The *Transaction* class applies the MEP signature data to M^o and the *Dispatcher* passes M^o to the *communication module*.

4.3.3 Communication Module

Design Goals

- Search for Bluetooth devices and query for appropriate *services*
- Create a communication channel for data exchange via Bluetooth.

Overview

The purpose of the *communication module* (see Figure 4.9) it to communicate with the *server component* via Bluetooth. The module provides a single I/O interface for the *RMI module* to use, thus reducing module inter-dependency. The *Control* class is responsible for maintaining any connections that the *client component* will have with the *server component*. It has been designed to spawn a *Connection* thread [3] which makes the Bluetooth connection to the server and acts a data channel. The spawning behaviour is determined by the developer at runtime (see 4.3.1).

The *Connection* implements a subset of functionality defined by JSR-82 (see 2.3.1). The thread is completely automated and will search for devices, query their services and connect to a device offering the DBlue service. It will also create a data stream to allow data exchange.
The module's I/O interface uses *synchronised* methods. This ensures that the data stream for data exchange cannot be interrupted during transmission resulting in data corruption.

4.4 Server Component Module Design

4.4.1 Communication Module

Design Goals

- Register Bluetooth *service*
- Create a communication channel for data exchange via Bluetooth
- Manage remotely connected *client components*
- Pass data to the *RMI module*

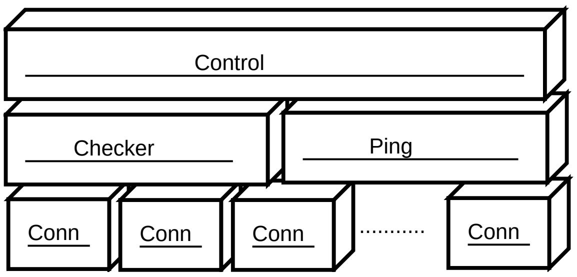


Figure 4.10: Server component Communication module architecture

Overview

The *server component's communication module* (see Figure 4.10) is similar in operation to the *client component* equivalent. It is responsible for maintaining all active *client component* connections and passing data sent from them to the *RMI module*.

The *Connection* class is responsible for acting a data channel between the *server* and *client components*. It implements a subset of JSR-82, notably service registration and opening a Bluetooth communication channel. Once created, the thread opens a data stream and blocks, waiting for incoming requests.

Each *client component* is connected to a different *Connection* thread. This allows the server to support multiple simultaneous *clients components* as no single one can block the service. Note: the number of simultaneous connections is limited by the Bluetooth piconet architecture (see 2.1.1).

When a connection is made to use the database service by a *client component*, the *Connection* it uses registers itself as active with the *Control* class. Similarly, if the connection is closed by the client, the *Connection* thread terminates and notifies the *Control* class that it is dead. Any incoming client data is passed from the *Connection* to the *Control* class. This hides the processing of data from the *Connection* and potentially allows it to be altered to become an generic communication channel for other communication links, eg IrDA, WiFi. *Connection* threads are created by the *Checker* class.

All communications are controlled by the *Control* class. Upon creation it spawns a *Checker* and *Ping* thread and starts their execution. The *Control* class' main role is to maintain a list of *Connections*. List manipulation is by synchronised methods to ensure concurrent access does not result in data corruption. When a *Connection* becomes active, the *Control* class creates an *RMI stack* for the *Connection* and binds the two together; all incoming data from a given *Connection* is passed to its associated stack. Dynamically creating a stack per connection ensures safety and robustness; if a *client component* becomes corrupt and attempts to bring down the *server component*, it is only possible for the *client* to attack the stack its *Connection* is bound to. This ensures safety by isolating each *client component's* user data. It also ensures that any possible bugs or crashes are confined to a single *RMI stack* that has no shared data with other clients creating a robust environment. The *Checker* thread runs periodically and acts as a *Connection pool manager*. At any time, the total number of *Connection* threads must equal the maximum number permitted, where C^t is the total number of connections, C^a the number of active connections and C^w number of waiting connections.

$$if(C^t \mathrel{!=} C^a + C^w) \text{ create new Connection thread}$$

Connection thread information is obtained from the *Control* class via synchronised methods (see above). The use of the *Checker* class ensures that the *server component* makes itself available to the

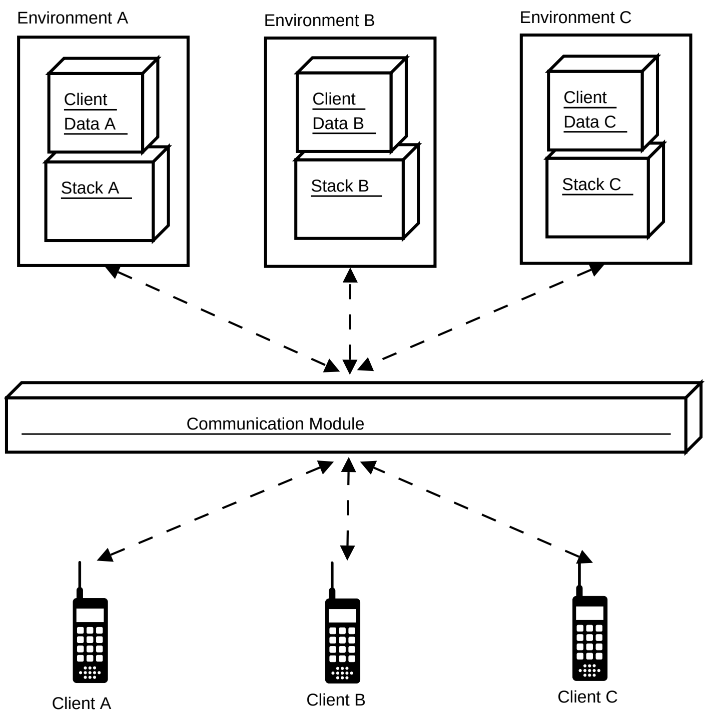


Figure 4.11: Server component RMI module isolation

maximum number of possible clients and makes the process of creating new *Connection* threads automated.

A *Ping* thread runs periodically to check to see if any *Connection* threads have become idle for long periods of time. It uses the synchronised list methods of *Control* to iterate through each *Connection* to send a message asking the *client component* if it is still alive. If a reply hasn't been received within an administrator set *timeout* limit, *Ping* asks *Control* to destroy the connection and its associated stack and user data.
Ping ensures that clients do not become idle and deadlock the service. With a maximum of only seven clients [27], it is important that access to the bridge is not restricted by idle clients.

4.4.2 RMI Stack

Design Goals

- Take raw data from the *communication module* for processing
- Process and interpret incoming messages and pass them to the *SQL module*
- Implement the MEP (see 4.3.2)

Overview

The *RMI module* controls and implements the RMI-framework as set out by the MEP. It is arranged as a stack of classes, the same as found in the *client component RMI Module* (see Figure 4.8). However, the *server component* implementation is optimised for the desktop and will differ in implementation of two layers of the stack, *Interpreter* and *Control*.

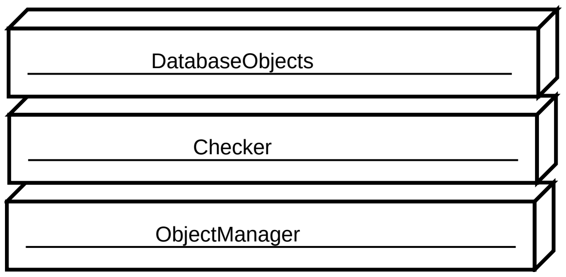


Figure 4.12: Server component SQL module architecture

The *Interpreter* class uses a different subset of MEP tags to the *client component* equivalent, specifically those for method invocation, object creation and deletion. Once the MEP tag data has been stripped from the message, it is passed to the relevant part of the *module*, as in the *client component RMI module*.
The *Control* class accepts requests for object management messages (creation, deletion, method invocation) and passes the request to the *SQL module*. The result returned from the *SQL module* is then passed down the left hand side of the stack (see Figure 4.8 to form a new message to be sent to the *client component*. It also maintains a list of sent messages and is responsible for message signing.

4.4.3 SQL Module

Design Goals

- Manage JDBC objects for *client component stub* owners
- Execute database functions and return data to the *RMI module*

Overview

This module executes requests from the *RMI module* and returns the result to be sent as messages to the *client component*. It is also responsible for maintaining information about remotely created objects.
Each *RMI module* has its own *ObjectManager*. Although this increases the *server components* memory overhead, it improves overall safety and robustness by isolating user data from other *client component* connections (see Figure 4.11).

At the core of the SQL module (see Figure 4.12) is the *ObjectManager* with all incoming and outgoing data passing through it. Its main function is to control the creation, deletion and usage of remotely accessible objects. Incoming creation requests result in a *JDBC-wrapper object* being created (see below) and assigned a *unique ID*. This ID and confirmation of creation is then sent back down to the *RMI module*.
A request to invoke a method on a remotely created object is subject to *method signature validation*. The *ObjectManager* checks the target objects method name and parameters (both number and type) before execution. If this validation fails, an error is thrown back to the *RMI module* which sends it as a failure message to the *client component*. This pre-verification of method calls ensures that faulty method calls cannot be sent to attempt to crash the *server component*.
The method call is then sent to the relevant object and its result returned to the *RMI module* to be dispatched as a return message to the *client component*.

Each Database object inherits from the *DatabaseObject* base class (see Figure 4.13). The func-

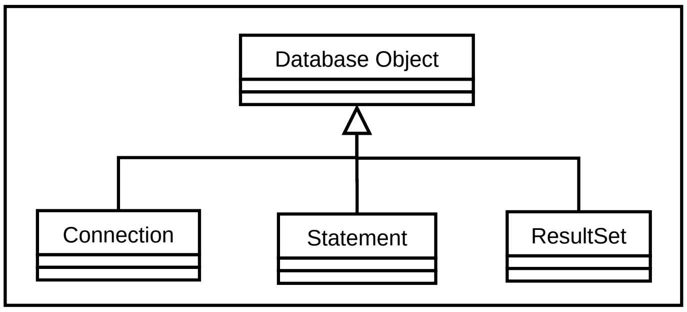


Figure 4.13: Server component SQL module classes

tionality provided by *DatabaseObject* is to scan the source code of the subclasses so give the *ObjectManager* a list of method signatures for pre-invocation verification. Each time a new subclass instance is created, the source code of that object is parsed and its method signatures loaded into memory. By scanning when objects are instantiated instead of at compile-time, it allows the functions that the database objects provide to be altered at runtime *without* any need to restart the *server component*. This improves service up-time and ensures that currently connected *client components* are not impacted if the backend database objects are altered.

ResultSet, *Statement* and *Connection* are simple classes that wrap around the JDBC *ResultSet*, *Statement* and *Connection* classes. These wrappers convert the internal data presentation and data syntax to that of JDBC. This allows presentation and syntax consistency throughout the rest of the *server component*, only being converted at the "final-stage".

Results from method invocation on the *DatabaseObject* are passed back to the *ObjectManager* which in turn passes the result to the *RMI module*.

Chapter 5

Implementation

The core implementation details of the project are outlined and, where appropriate, code extracts given.

5.1 Overview

The system was implemented using Java 2 Standard Edition SDK v1.5.0 (*server component*), Java 2 Mobile Edition CLDC configuration v1.0 (*client component*) and the J2ME Wireless Toolkit v2.2 (*client component*). The mobile test platform was a Nokia 6600 running Symbian OS; the server and development platform was a Gentoo Linux machine running v2.6.10 kernel using the Bluez Bluetooth driver. It relies upon a Bluetooth SDK (Avetana) to link the desktop Java environment to the Bluetooth hardware. The system adheres to good software engineering practices and is object-oriented throughout. Code has been implemented to be small and efficient whilst allowing it to be extensible. It also has extensive error handling to ensure reliability.

5.1.1 Package Structure

Components and modules have been separated into packages for improved code maintainability, as shown in Figure 5.1. A full class listing is available in Appendix B.

As shown, code is structured into main component packages; *Mobile* and *Server*. There is also another packages, *Shared*, which contains code that is common to both components. The *Messaging* package contains classes required for the implementation of the MEP and hence is common to both components. *Exceptions* are also common to both components due to the distributed nature of the system. *Util* contains a number of house-keeping classes common to both components.

The *Mobile* package contains implementations of the modules defined in the system design (see 4.2.1). It also contains a utility package which contains the same house-keeping classes found in the shared package, but optimised for the light-weight environment. These classes also provide functionality not included in J2ME, such as regular expression matching and support for the *Observer/Observable* paradigm.

The *server* package contains implementations of the modules defined in the system design (see 4.2.2). It should be noted that classes within the *comms* package define the operation of any communications mechanism. In this project, Bluetooth was implemented but the design allows for development of other communication mechanisms simply by implementing the defined interfaces.

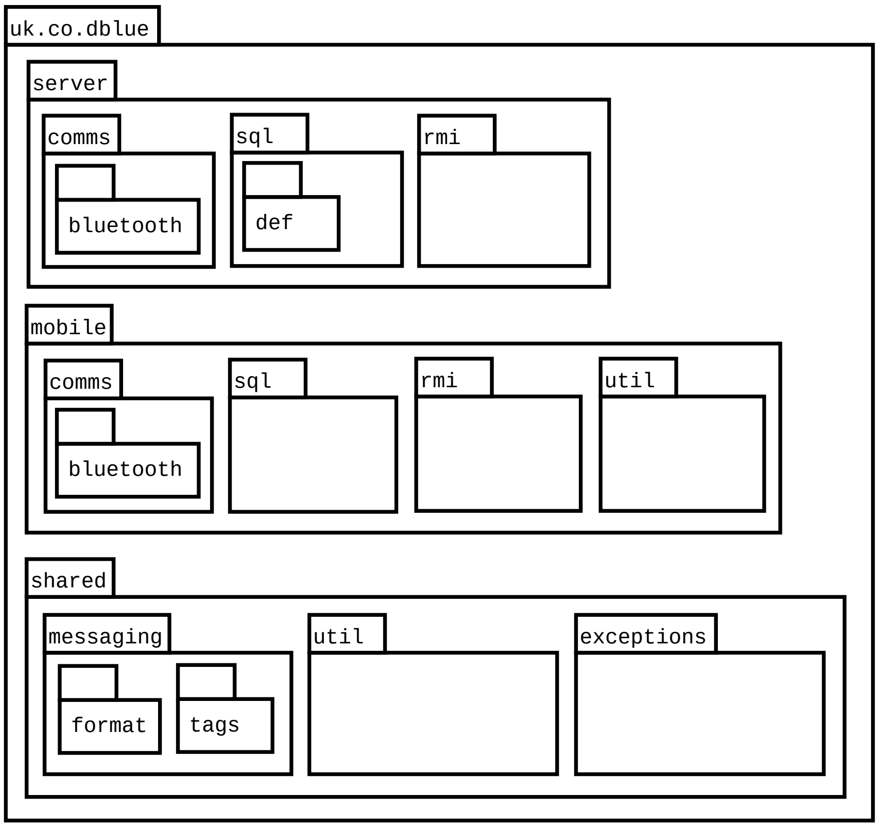


Figure 5.1: Project package structure

5.2 Client Component

5.2.1 API Module

Overview

All of the API classes (*Connection*, *Statement* and *ResultSet*) contain simple methods which wrap the data that needs to be transmitted to the server in a consistent internal format. This is performed by making a call to their superclass (*DatabaseObject*) which takes in the data that needs to be wrapped and presents in the systems internal data structure. The module then passes the data structure to the *RMI module* which continues processing. All developer accessible *API classes* simply act as pointers to objects stored on the *server component*.

Key Discussions

- System's internal data structure
- Data presentation for *RMI module*
- Object creation and deletion

Internal Data Structure

Data within the system flows through a number of modules and is modified by a number of different classes. It was decided to use a *Hashtable* for internal data representation to allow easy code maintainability and improved reliability. A *Hashtable* permits values to be accessed by reference (a key) instead of by direct access, eg array index. By ensuring that data is accessed by reference,

```
private static final String command = \"METH\";
private static final String[] attributes = {\"OBJECT_ID\", \"METHOD\", \"RET_TYPE\", \"PARAM_NUM\"};

public MethodTag() {
    super(command,attributes);
}

public String getObjectIDString() {
    return attributes[0];
}

public String getMethodString() {
    return attributes[1];
}
...
```

Figure 5.2: Code sample of MethodTag class

code changes can be implemented in a single class that do not depend upon other aspects of the system. For example, consider that data is represented by a string in the following, absolute, way:

Object:ID:Method:Parameter

If one part of the *RMI module* wishes to modify the syntax (say, by swapping ID and Object elements), this may cause other parts of the code to break. By abstracting away from ”hard” data representation to the *Hashtable*, the code has become more reliable and easier to maintain. However, this does have a slightly higher memory overhead but is considered acceptable[3].

Data Presentation

The *Hashtable* format relies upon referential access by unique key. These keys have been centralised (see 5.1.1) in the *messaging package*. As the package shows, there are a number of *Tags* that define the syntax for particular data types. Shown in Figure 5.2 is an extract from the MethodTag class which defines the syntax of data for a remote method call. Classes using the *messaging package* must use the accessor methods to access the key value. This ensures that the key can be changed at any time without breaking code reliant upon it.
The *DatabaseObject* superclass uses these tags as keys for use in the internal representation *Hashtable*. Shown in Figure 5.3 is a method called by all sub-classes which takes raw data and applies the internal representation before passing to the *RMI module*.

Object creation and deletion

As outlined in the module design, creation of API objects results in a message being sent to the *server component* to request the creation of the appropriate object. An identifier is returned to the client API object and is used to access the *server component* object. Upon instantiation, the API objects call a method in their superclass which creates a message using the internal representation format and sends a request to the *server component* to create the appropriate server-object. The *server component* objects contain a number of variables that are set at run-time. To allow the *client component stub* objects to use these variables, a copy of the variable is requested from the *server component* and is set at instantiation. This allows the third-party developer to use the variable within their code without a number of duplicate messages being sent from the *client component* to the *server component* to find the variable value. An example a *ResultSet* instantiation is shown in Figure 5.4.

[3]Hashtable is the only member of the J2SE Collections framework found in J2ME that allows referential element access

```
  public Hashtable makeMethodCallTable(Integer objectID, String method, String returnType, String parameterType[], String parameters[]) {
      Hashtable command   = new Hashtable();
      MethodTag meth      = new MethodTag();

          command.put(meth.getObjectIDString(), objectID.toString());
          command.put(meth.getMethodString(), method);
          command.put(meth.getReturnTypeString(), returnType);
          if (parameterType == null) {
              command.put(meth.getParameterNumberString(), "0");
              return command;
          }

          else {
          command.put(meth.getParameterNumberString(), new Integer(parameters.length).toString());
          Hashtable[] params  = new Hashtable[parameters.length];
          ParameterTag para   = new ParameterTag();

          for(int i=0; i<parameters.length; i++) {
              params[i] = new Hashtable();
              params[i].put(para.getParameterIDString(), new Integer(i).toString());
              params[i].put(para.getParameterTypeString(), parameterType[i]);
              params[i].put(para.getParameterDataString(), parameters[i]);
              command.put(para.getTag() + i, params[i]);
          }
      }

      return command;
 }
```

Figure 5.3: Code sample of DatabaseObject method for data presentation

```
  public ResultSet(Integer objectID, Control control) throws SQLException {
      this.objectID   =   objectID;
      this.control    =   control;
      this.CLOSE\_CURSORS\_AT\_COMMIT = getCLOSE\_CURSORS\_AT\_COMMIT();
      this.CONCUR\_READ_ONLY = getCONCUR\_READ_ONLY();
      this.CONCUR\_UPDATABLE = getCONCUR\_UPDATABLE();
      this.FETCH\_FORWARD = getFETCH\_FORWARD();
   ...
  }
```

Figure 5.4: Code sample of ResultSet instantiation

5.2.2 RMI Module

Overview

The *RMI module* implements the functionality defined by the MEP (see 4.3.2). It accepts incoming messages requests from the *API module* and blocks the call until a reply has been received from the *server component*, or the message has failed. XML messages are created from incoming data by incrementally adding syntax information to the raw data; this data is then passed out of the module for transmission. Incoming XML messages are parsed using a SAX-based parser and are internally validated. The content of return messages is passed back to the *API module* and the call unblocked.

Key Discussions

- Remote method calls
- XML message formatting
- Message parsing and validation

Remote method calls

At the heart of the client RMI module is the *Control* class. Its purpose was to accept incoming requests from the API module and send data down the stack for processing. It also takes data from the *Interpreter* and passes it back to the API object callee. During implementation it was decided to allow only a single request to be accepted from the *API module* at any one time (blocking). By doing so, it ensured that the third-party application could not get ahead of the database and message processing. For example, consider a fragment of third-party code shown in Figure 5.5

```
    ...
    Connection conn = builder.getConnection();
        conn.connect(\"postgresql\", \"localhost\", \"test\", \"postgres\", \"\");

        Statement st = conn.createStatement();

        ResultSet rs = st.executeQuery(\"SELECT * FROM testtable\");

                while (rs.next())
                {
                        String field1 = rs.getString(\"field1\");
                        String field2 = rs.getString(\"field2\");

                        f.append(\"Field1: \"+field1 + \", Field2: \"+field2 + \"\n\");
                }
        }
   ...
```

Figure 5.5: Third-party code example

```
    ...
    if (function.equals("delete"))
       return null;
    //return immediately to finalizer - don't need to wait for the server to respond as API object is being deleted by garbage collector

    transactionReturnDataSet = false;
    while (!transactionReturnDataSet) {}
      //check to see if its an exception otherwise return
        if (transactionReturnData instanceof SQLException)
            throw (SQLException)transactionReturnData;
        outgoingTransaction = null;
        return transactionReturnData;
    ...
```

Figure 5.6: Blocking Control class code

utilising a database. If the *RMI module* was not blocking, it could be possible for the third-party application to attempt to execute a query (line 7) on a *Connection* object (line 1) that had not yet been created on the server. This would cause a number of errors leading to a number of messages being sent between the *client* and *server components* to try and solve the problem. To achieve blocking behaviour, the method called by the *API module* database objects is shown in Figure 5.6. To achieve blocking, the method enters a loop. This loop is broken when data is set by the *Interpreter*, or if the *Checker* notices the message has timed-out.
The *Control* class was originally designed to maintain a list of outgoing messages. However, since only a single transaction can be processed, this was replaced by a globally accessible variable that stores the message. Access to this variable is by a number of accessor methods to ensure simultaneous access from the *Interpreter* and *Checker* classes does not cause data corruption.

XML message formatting

XML message formatting was completed iteratively, with syntactic information being added to the message through a number of stages. The *messaging package* used for internal data representation was also used to provide Strings for the XML tags. An example XML message is shown in Figure 5.7. Note: the message still contains the TIME_STAMP attribute as part of the COMMAND tag. Originally this was designed to allow for multiple messages to be sent simultaneously and processed according to their logical clock time. As outlined, only a single message is processed at any one time and the TIME_STAMP is redundant. However, it has remained to allow future code maintainers to modify the application to support multiple, simultaneous messages. The signature

```
<?xml version=\"1.0\" encoding=\"UTF-8\"?>
    <!DOCTYPE COMMAND SYSTEM ''commands.dtd>
        <COMMAND TRANSACTION\_ID=\"1\" TIME_STAMP=\"\$TIME\_STAMP\$\">
        <CONTENT>
            <CRE TYPE=\"ResultSet\">
            </CONTENT>
        </COMMAND>
```

Figure 5.7: Example XML message to create a server side ResultSet object

```
public String replace(String data, String rep, String value) {
    return data.substring(0, data.indexOf(\"\$\" + rep + \"\$\")) + value + data.substring((d.indexOf(\"\$\" + rep + \"\$\") +
    rep.length() + 2), d.length());
}
```

Figure 5.8: Code sample of regular expression function

part of the message (see Figure 5.7 line 1-3) was implemented to use a string constant, replacing parts of it with variable values where appropriate. For example, in the sample message, the value of TRANSACTION_ID was replaced as the message was created. This string manipulation is not present in J2ME and was implemented using a simple regular expression function that searches a string for variables within delimiter characters ($) and replaces them with a value.

Message parsing and validation

Although both of the system components use XML for data representation, they parse incoming data differently owing to their environmental strengths. Both perform the parsing in the *Receiver* class of the *RMI module*. The *server component* uses SAX to extract data from the XML as opposed to DOM. [14] SAX was chosen as it is able to process documents faster with a lower memory consumption than DOM. The XML is validated by using a DTD to ensure that no information is missing, as well as being validated by the parsing code itself.

A second reason for the use of SAX for the *server component* code is its portability to the *client component*. [11] Although J2ME does not support XML parsing, a third-party developer has produced a light-weight parser for mobile devices called kXML. The use of third-party code increases total memory footprint of the application, but it was decided for speed of development this would be acceptable. It provides a basic, event-driven XML parser similar to SAX. Although it requires some modification of the code developed for the *server component*, the principles of its operation were easily ported to the mobile environment. It does not included document validation (by DTD) but the XML is validated using the *messaging classes*. In the same way that messages are created using the attributes defined specified in the messaging *Tags*, the reverse is possible. Once a message has been parsed, the parser checks each of the attributes found in the XML message against the expected attributes listed in the relevant *Tag*. If all the attributes are present, the message is valid.

5.2.3 Communication Module

Overview

The *Communication module* implements the functionality defined by JSR-82 as well as creating Input/OutputStreams for data transmission. String formatted XML messages are passed from the RMI module above and sent “as-is” to the *server component*. Similarly, messages are received from the *server component* and passed “as-is” to the *RMI module*.

Key Discussions

- Datastream Issues

Datastream Issues

The *Connection* class implements the client-side Bluetooth functionality. It is threaded and implements *DiscoveryListener* (defined in JSR-82) providing methods for *device* and *service discovery*. It

```
while (connected) {
    int numUTFStrings = inputStream.readInt(); //reads in the number of messages expected to receive
    String msg = "";
        while (numUTFStrings-- > 0)
           msg = msg + inputStream.readUTF();
        incoming(msg);
}
```

Figure 5.9: Code sample of reading Bluetooth data

also creates input and output streams to send and receive data and uses globally accessible variables to indicate if the object is connected. These are used by the *Control* class to determine if the *Connection* is ready to send/receive data.

Problems during implementation were found when attempting to send and receive long strings of text. J2ME defines the *OutputStream/InputStream* classes which have methods, *writeUTF* and *ReadUTF*, allowing arbitrary length strings to be sent and received. However, on the mobile test device the J2ME implementation threw exceptions when using these methods with strings longer than 50 characters. To counteract this, long strings are divided into smaller segments. A message is sent to the remote device indicating the number of segments it should expect to receive, thus allowing the full message to be reconstructed (see Figure 5.9).

5.3 Server Component

5.3.1 SQL Module

Overview

The *SQL module* takes an input from the *RMI module* and executes the required function on a locally stored object. Each method invocation is verified to be correct before it is executed. Database objects are wrappers around řealĲDBC objects and convert the internal data representation to that required by JDBC. Database objects scan their own source code at runtime to create a list of valid methods for this pre-verification. This is performed by a custom built regular expression checker.

Key Discussions

- Data Presentation
- Regular Expression Checker

Data Presentation

To ensure consistency, data was presented in same way as found in the *client component*; by *Hashtable* to ensure referential access to data and reduce coupling between modules (see 5.2.1). Data is converted from the internal presentation method to that of the JDBC objects.

Regular Expression Checker

During implementation the regular expression checker was initially developed using a third-party parser, JavaCC. This tool uses the Java programming language grammar to create a number of parser classes that will scan a body of source code and perform operations on the code. However, although this parser solved the problem, it was too heavy for such a process that needed to be fast and memory efficient. A custom built parser was written that used functionality from the Java regular expression library to scan the interface definitions for each object. This extracts all method signatures which are then loaded into a *Hashtable* and used to verify methods before they were executed.

5.3.2 RMI Module

Overview

The *RMI module* implements the same functionality described in the *client component RMI module* (see 5.2.2). It accepts incoming messages from the *communication module* and parses them using a SAX parser before passing to the *SQL module*. The return data passed back from the *SQL module* is used to create XML messages by incrementally adding syntax information to the raw data; this data is then passed out of the module for transmission.

Key Discussions

- N/A - implementation problems were solved in the *client component RMI module* and simply ported to the *server component*

5.3.3 Communication Module

The *communication module* implements the functionality defined by JSR-82 as well as creating Input/OutputStreams for data transmission. String formatted XML messages are passed from the *RMI module* above and sent "as-is" to the *client component*. Similarly, messages are received from the *client component* are passed "as-is" to the *RMI module*. The module also performs connection maintenance by monitoring the status of connections.

Key Discussions

- Datastream Issues
- Client Pings

Datastream Issues

See 5.2.3.

Client Pings

To ensure that the bridge is available to the largest possible number of clients, idle or ̈deadc̈onnections are removed. The initial design required that a ping message was sent from the *server component* to the *client component* to check if it was alive. However, during implementation this caused a number of problems. If the *client component* was no longer alive, attempting to send a ping message would cause an exception to be thrown by the *OutputStream* as it could no longer find the other party it was communicating with. To solve this problem, the *Connection* class stores the last time the communication channel was active (data sent or received) in a globally accessible variable. Each time the *Ping* thread runs, it checks the value of the last used variable of each communication channel and subtracts it from the current time. If the result is greater than the timeout set by the administrator, the *Connection* and associated data is destroyed.
This method of checking if the *client component* is alive reduces Bluetooth bandwidth usage and lowers the exception rate caused by I/O stream errors.

Chapter 6

Testing

6.1 Introduction

Testing was applied using a bottom-up methodology and applied at all stages of the project. With a project of such magnitude, it is impossible to test every aspect of the system. At all stages, tests were selected using risk-driven analysis to ensure that the most critical parts of code have testing priority. This assessment is similar to risk analysis (see 7.3) and attempted to select testing by likelihood of the error occurring and possible impact.

6.2 Testing Strategy

During development, unit testing was applied using a variety of techniques. A number of crafted inputs were developed to test the unit code using functional testing methodologies. These tests were broken into three broad groups; boundary-value testing, decision tables and special value testing.
Boundary-value testing relies upon the principle that most errors occur at a boundary of a particular functions input domain. [5] These tests used negative testing by exceeding valid input boundaries in order to assess code robustness. Decision table tests were used in a number of isolated cases where multiple combinations of conditions could occur, eg nested statements. Special-value tests were crafted to exploit special cases that may not typically arise. For example, tests to ensure object casting was reliable were developed.
This process was applied continuously to code which was changed to reflect testing outcomes. The latest Java development kit provides a number of static-source-code-analysis (SSCA) tools designed to predict possible runtime errors before they occur. These were used during unit compilation to check possible erroneous run-time behaviour.

Integration testing was applied when units were integrated into modules. Dependencies between module output and inputs were first derived in order to structure testing in a systematic way and reduce test duplication. A number of functional tests were then produced, similar in nature to those used in unit testing. Regression testing was then applied. Stubs were created and added to a module to create an initial framework. Each stub had a pre-defined behaviour and in many cases returned a pre-computed value. Individual units were then added into the component, replacing the existing stub version. The functional tests were then applied; if the component failed, it could be assumed that the failure was due to an interaction between the recently added unit and another unit. This process was repeated iteratively until each component had no stub units.
Due to a large number of possible failure combinations as each unit was added, testing was heavily automated. Each component had a test harness designed which gave a random input from a

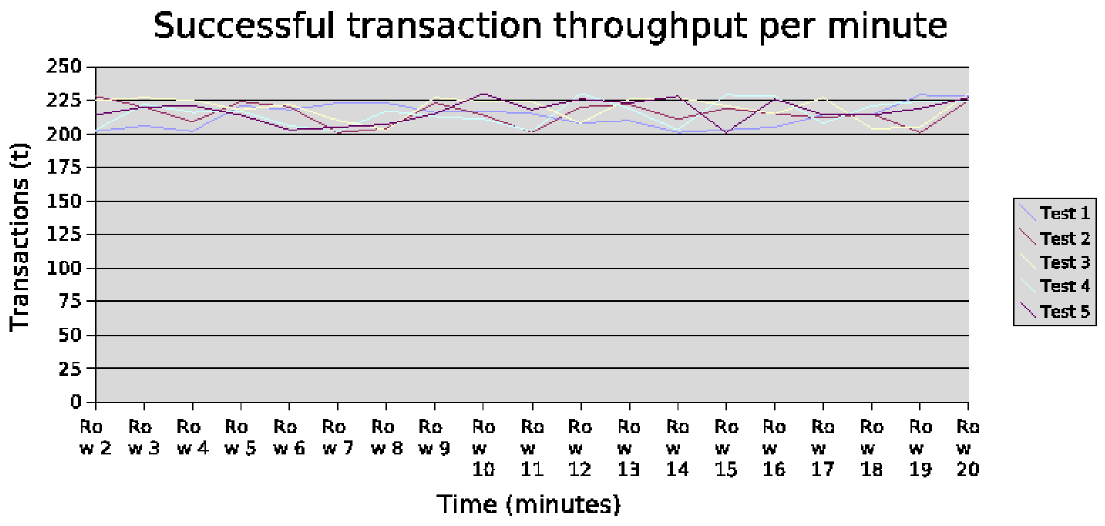


Figure 6.1: Graph of load testing performance

set of crafted possibilities. The output from the module was noted and compared to the expected result. This ensured consistent module testing and isolated failures to specific unit interactions.

Following integration testing, system testing was applied when modules were linked. Testing methodologies here relied upon testing the structural aspects of the project and ensured that specifications were being met. Although difficult to achieve, tests were developed to make no assumption of internal operation. As with integration and unit testing, critical outputs and inputs were identified and tested. This was achieved by writing a simple third-party test application using various aspects of the project and noting the results. Load testing was also applied to the system by sending a large number (16,000) of identical transaction requests from a mobile device to the server. As shown in Figure 6.1, each test only sent approximately 4,000 transactions before both the *client* and *server components* crashed. However, the crash was not due to a programming error (eg an uncaught exception), but an internal VM error. The server reported a HotSpot error and the mobile phone a Monty-Thread error. This behaviour is fully reproduce-able but its cause remains unknown. The bug has however been accepted as few mobile applications will request 4,000 transactions during the lifetime of a connection, effectively resetting the software. Note: the continual fluctuation of message throughput shown in the graph is due to variations in the available Bluetooth bandwidth; this cannot be affected by either the *client* or *server* software. Stress testing was also applied by affecting Bluetooth network usage, moving the mobile device into and out of range of the server machine.

Chapter 7

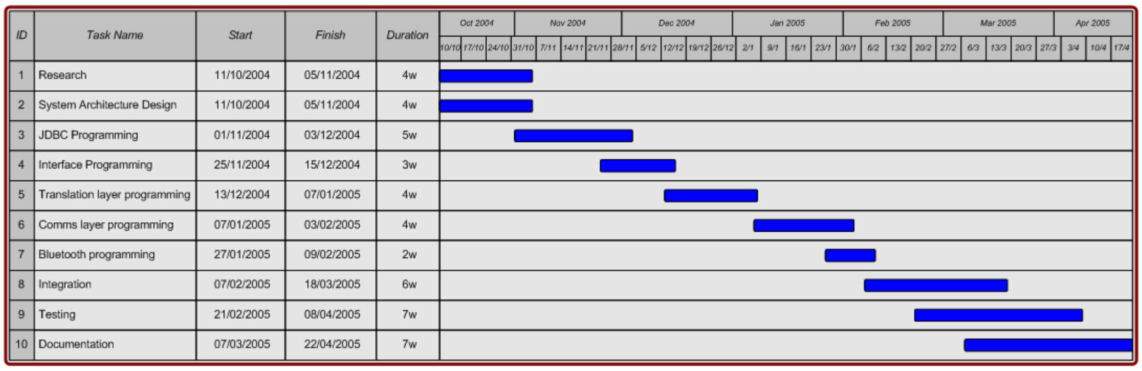
Project Management

7.1 Time Planning

With a significant amount of work to be completed over a seven-month period, organisation of time was critical for the project to succeed. During the planning phase, a Gaant chart was produced which outlined the estimated time to be spent on each phase of the project. Although useful as a guide, it proved to be unrealistic due to a lack in domain knowledge. It also assumed a linear, discrete phase, development cycle, but in reality a more iterative approach was taken. A significantly larger amount of time was spent researching and designing than originally assigned, with implementation taking much less time. However, the total workload remained balanced allowing the project to be completed on time. As the project progressed passed the research phase, time planning became more realistic and achievable.

7.2 Project Lifecycle

The software was developed using the iterative lifecycle model, with functionality being added during incremental version releases. This allowed requirements to be prioritised, implemented and researched accordingly. Mission critical requirements and high-risk areas were addressed in earlier releases, with less critical requirements implemented in later releases. This ensured that the minimum level of functionality was achieved before extensions were attempted. For example, core Bluetooth communication was established during the first month of the project and refined during later releases. This also ensured that the risk-driven testing strategy (see 6.1) was adhered to by testing releases with high-risk features before implementing additional functionality.



ID	Task Name	Start	Finish	Duration
1	Research	11/10/2004	05/11/2004	4w
2	System Architecture Design	11/10/2004	05/11/2004	4w
3	JDBC Programming	01/11/2004	03/12/2004	5w
4	Interface Programming	25/11/2004	15/12/2004	3w
5	Translation layer programming	13/12/2004	07/01/2005	4w
6	Comms layer programming	07/01/2005	03/02/2005	4w
7	Bluetooth programming	27/01/2005	09/02/2005	2w
8	Integration	07/02/2005	18/03/2005	6w
9	Testing	21/02/2005	08/04/2005	7w
10	Documentation	07/03/2005	22/04/2005	7w

Figure 7.1: Gaant chart of early time-scheduling

The iterative lifecycle also facilitated "just-in-time" background research. High-level research was conducted to produce an architecture design, but more detailed research was undertaken when the relevant problem area was addressed by a release. Due to the large volume of research material covered, it allowed focus to be given to specific areas and improved assimilation of information.

To reduce risk of data corruption and/or data loss, the project was managed using a combination of a version control system, CVS, and a proprietary backup system. This lent itself well to the release based nature of the lifecycle allowing software to be 'rolled-back' where appropriate.

7.3 Risk Identification and Management

High-level risk was identified early into the project and covered a number of general risks including data loss, failure to meet requirements, lack of relevant knowledge and poor system performance. Each of these risks was assessed and rated for probability of it occurring and potential impact; a strategy to handle each identified risk was then produced.
Each planned iteration of the lifecycle (see 7.2) was assessed and assigned an overall risk value based upon the criteria used above. This risk value was then used to determine in which release functionality should be implemented and tested.

Chapter 8

Appraisal

8.1 Assessment

Based upon the original specification (see 3.2), the project fulfilled each outlined requirement to a high-quality standard. The project also demonstrates significant application and integration of programming and computer science skills from a range of subject areas, as well as research into cutting-edge technologies.

The system has been developed with a focus on extendibility and attempts to use existing standards and technology where available. Where new problem solutions have been developed, specifically the MEP (see 4.3.2), they have been designed to use non-proprietary technologies such as XML. This enables future bridge developers to create custom implementations of the programmer API for use in non-Java environments such as Windows Mobile.
The software is completely modular, adhering to software engineering principles and designed to allow extendibility into a "wireless" database bridge. Assuming a suitable communication module is implemented by a bridge maintainer, communication between the database and a mobile application can move from using Bluetooth to another technology such as WiFi easily and quickly.

Basing the developer API library on an existing database connection platform (JDBC), the software allows developers to transfer development skills from the desktop to mobile applications with little new knowledge required. It also gives flexibility to port existing database dependent code from the desktop to the mobile platform with very few alterations. This improves application reliability by code re-use and reduces development time.
The API provides potential developers with a significant number (118) of database operations, including those core to all database systems. It has also been implemented to ensure compatibility with *any* J2ME compliant device. Finally, it is small (25KB) allowing it to be included in most mobile applications. However, if a developer finds they are not using all the database features provided by the library, the design allows them to easily customise it by removing redundant functionality reducing the memory footprint.

Enabling legacy databases to be accessed wirelessly is simple and requires no user input making the software perfect for running on as a daemon. The server application supports multiple device connections and isolates client data for security and reliability. The server software has also been designed to be "hot-swappable" allowing parts of the system to be altered during execution. This allows mobile clients to remain connected whilst maintenance is performed and does not result in "down-time".

8.2 Performance and Stability

The system shows good performance and a high degree of robustness. The code is fault tolerant and employs a number of error handling mechanisms throughout all parts of the system. Performance depends largely upon the target mobile device and the server machine hardware. As outlined in Testing (see 6.1), throughput of database transactions is heavily limited by Bluetooth's low transmission rate.

Chapter 9

Evaluation

9.1 Achievements Overview

The project has produced a stable and usable platform for the development of database-driven mobile applications. The API library is based upon existing Java database technology (JDBC) and is simple, intuitive and easy to learn. It requires developers to have *no* knowledge of Bluetooth or wireless communication technologies as its operation is completely transparent.
The server application enables mobile developers to connect to virtually any legacy database and is simple and easy to use, running as a service daemon on the database server.

The software has been designed to be highly extendable through good software engineering practices, enabling easy customisation for particular usage scenarios.

9.2 Extensions

With more time, the project could easily be extended and modified in a number of ways.

9.2.1 Further Testing

Although J2ME is based on a set specification, OEM implementations vary greatly between devices. Through personal experience, code that executes correctly on one device appears to fail on another. Although the software was tested extensively using the Nokia 6600, no other devices were used for testing. With access to a range of devices, bugs common to all environments could be established, as opposed to errors in OEMs Java implementations.
Similarly, the server application was tested heavily in Linux, but not in any other environments. However, Java's "write-once-run-anywhere" mantra should enable execution also in Windows and MacOS.
Finally, a persistent problem of VM crashes when exceeding 4,000 transactions should be addressed and solved.

9.2.2 Wireless Bridge

With the design architecture in place, the Bluetooth communication channel between the mobile application and the server could be changed for another technology such as WiFi, GPRS or IrDA. With more time, the software could be extended to "roll-back" between technologies. For example, if a mobile device was not in range of a Bluetooth server, it could automatically roll-back to use another technology to connect to the same server.

9.2.3 Logging

An optional extension would be to include logging functionality into the server application. Although not strictly required as it simply links to a database (which would log transactions), the addition of logging would aid in solving potential bugs should they arise.

9.2.4 Message Exchange Protocol Optimisation

Although the MEP provides anti-deadlocking functionality and a reliable way of sending messages, it could be extended to included error correction information. For example, if a message is partially-corrupted during transmission, the MEP could be extended to request only the corrupted part of the message is resent. This would improve bandwidth usage and overall system performance. Also, support for sending of multiple, simultaneous messages could be added.

9.2.5 Security

Currently the bridge provides no configurable security (beyond that afforded by Bluetooth) for server administrators. A further extension could implement client verification by lists of permitted or denied clients. Transmission of data could also be secured using the developers choice of encryption algorithm which could be added as a layer in the system architecture.

9.2.6 Non-Java Implementation

The software design is such that it is possible to be implementable in non-Java environments. A significant extension would be to implement it for non-Java environment, a further stand-alone project.

Bibliography

[1] benhui.net: the harmony of mobile development. http://www.benhui.net/.

[2] Bluez: Official Linux Bluetooth protocol stack. http://www.bluez.org/.

[3] Tom Axford. Real-Time Systems Programming. *Lecture Notes*, 2003.

[4] Behzad Bordbar. Distributed Systems. *Lecture Notes*, 2004.

[5] Ela Claridge. Software Testing. *Lecture Notes*, 2005.

[6] Oren Eliezer and Matthew Shoemake. Bluetooth and wi-Fi coexistence schemes strive to avoid chaos. *RF Wireless Connectivity*, 2001.

[7] Jon Ellis, Linda Ho, and Maydene Fisher. *JDBC 3.0 Specification*. Sun Microsystems, 2001.

[8] Martin Fowler. *UML Distilled Third Edition: A Brief Guide To The Standard Object Modelling Language*. Addison Wesley, 2004.

[9] Eric Giguere. Compressing XML for Faster Wireless Networking. *Sun Microsystem Java Developer Network Articles*, 2003.

[10] Eric Giguere. J2ME Tech Tips. *Java Developer Connection (JDC) Java 2 Platform, Micro Edition (J2ME) Tech Tips*, 2003.

[11] Eric Giguere. Parsing XML in CLDC-based Profiles. *Sun Microsystem Java Developer Network Articles*, 2003.

[12] Allen Holub. Programming Java threads in the real world. http://www.javaworld.com/javaworld/jw-09-1998/jw-09-threads.html, 1998.

[13] Bruce Hopkins and Ranjith Antony. *Bluetooth for Java*. Apress, 2003.

[14] Nazmul Idris. Should i use SAX or DOM? *developerlife.com*, 1999.

[15] Daniel Kappeli. JXTA over Bluetooth. Master's thesis, Information and Communication Systems Research Group, Swiss Federal Institute of Technology, Zurich, 2003.

[16] Andr N. Klingsheim. J2ME Bluetooth Programming. Master's thesis, Department of Informatics, University of Bergen, 2004.

[17] Jonathan Knudsen. Wireless Development Tutorial. *Sun Microsystem Java Developer Network Articles*, 2003.

[18] C Bala Kumar, Paul J. Kline, and Timothy J. Thompson. *Bluetooth Application Programming With The Java APIs*. Morgan Kaufmann Publishers, 2004.

[19] Laura Lemay and Rogers Cadenhead. *Teach Yourself Java 2 in 21 Days Second Edition*. SAMS, 2000.

[20] Qusay Mahmoud. Advanced MIDP Networking, Accessing Using Sockets and RMI from MIDP-enabled Devices. *Sun Microsystem Java Developer Network Articles*, 2002.

[21] Qusay H. Mahmoud. J2ME Low-Level Network Programming with MIDP 2.0. *Sun Microsystem Java Developer Network Articles*, 2003.

[22] Qusay H. Mahmoud. Wireless Application Programming with J2ME and Bluetooth. *Sun Microsystem Java Developer Network Articles*, 2003.

[23] Sun Microsystems. Java 2 Platform Standard Edition 5.0 API Specification. http://java.sun.com/j2se/1.5.0/docs/api/index.html.

[24] Sun Microsystems. Java rmi overview. http://java.sun.com/products/jdk/rmi/.

[25] Sun Microsystems. JDBC Overview. http://java.sun.com/products/jdbc/overview.html.

[26] Sun Microsystems. Working with XML. http://java.sun.com/xml/jaxp/dist/1.1/docs/tutorial/TOC.htm

[27] Brent A. Miller and Chatschik Bisdikian. *Bluetooth Revealed: The Insider's Guide To An Open Specification For Global Wireless Communications Second Edition*. Prentice Hall, 2002.

[28] Malcolm Orr. Bluetooth for Peer-to-Peer Communicator. Master's thesis, School of Computer Science, University of Birmingham, 2004.

[29] C. Enrique Ortiz. J2ME Technology Turns 5! *Sun Microsystem Java Developer Network Articles*, 2004.

[30] Bluetooth SIG. The Official Bluetooth Member Site. https://www.bluetooth.org/.

[31] Ian Sommerville. *Software Engineering 6th Edition*. Addison Wesley, 2001.

[32] Sun Microsystems. *Connected Limited Device Configuration (CLDC) v1.0 Specification*, 2003.

Appendix A

Project Proposal

A.1 Introduction

Today, the desire for a unified system of communication devices such as mobile phone's and PDA's is becoming increasing apparent. Bluetooth (IEEE 802.15x) addresses a large sector of this market by providing a low-cost, short-range wireless communication for low-power devices by creating the so-called "Personal Network".
Although the first specification was released only in July 1999, millions of devices ranging from mobile phone's to home appliances such as microwaves and refrigerators are Bluetooth enabled.

Imagine walking into a large, unfamiliar supermarket. You need only a few specific items but as you have never visited this store, you don't know where they are. You type into your Bluetooth PDA or phone a shopping list. Your device then gives you the location of each of the items. The simple example facilitates the use of a wireless database and presenting the data using a custom UI.

This project is to develop a Bluetooth bridge allowing devices to connect to any standard database and perform operations.

A.2 Project Overview

The bridge aims to:

- Failitate a range of SQL queries and update the database accordingly
- Create as small a footprint as possible on the mobile device, shifting the processing load to the more capable desktop machine
- Act as a platform for third-party applications
- Create simple programming interfaces to the core of the system

There are two main layers to the system; translation and communication, both having challenges associated with them.

The communication part of the system deals with how the Bluetooth device transfers date with the machine running the database. The device will attempt to connect its side of the bridge to the machines side to allow communication. This layer will perform operations such as authentication and socket creation.

The translation above layer will accept inputs in the form of database operations from third-party applications. It will then translate these into some form and pass the translated data to the connection layer to be send to the database machine. Similarly, on the machine side the translation layer provides an interface to the database connection bridge. Any statements will be executed and results sent back to the device. The device will then present the data the third-party application.

Target environments for this project are varied. Devices run a variety of operating systems and must connect to many different databases. The project will use JDBC and micro-Java. Micro-Java contains a platform independent Bluetooth API allowing the bridge to run on any device. Similarly JDBC is platform independent and can provide a database interface regardless of operating environment.

Appendix B

Source Code Listing

/uk/co/dblue/server
/uk/co/dblue/server/comms
/uk/co/dblue/server/comms/CommsControl.java
/uk/co/dblue/server/comms/bluetooth
/uk/co/dblue/server/comms/bluetooth/Connection.java
/uk/co/dblue/server/comms/bluetooth/Control.java
/uk/co/dblue/server/comms/bluetooth/Ping.java
/uk/co/dblue/server/comms/bluetooth/ConnectionChecker.java
/uk/co/dblue/server/sql
/uk/co/dblue/server/sql/Connection.java
/uk/co/dblue/server/sql/ResultSet.java
/uk/co/dblue/server/sql/Statement.java
/uk/co/dblue/server/sql/Method.java
/uk/co/dblue/server/sql/ObjectManager.java
/uk/co/dblue/server/sql/DatabaseObject.java
/uk/co/dblue/server/sql/def
/uk/co/dblue/server/sql/def/Results.java
/uk/co/dblue/server/sql/def/Database.java
/uk/co/dblue/server/sql/def/Query.java
/uk/co/dblue/server/rmi
/uk/co/dblue/server/rmi/commands.dtd
/uk/co/dblue/server/rmi/Dispatcher.java
/uk/co/dblue/server/rmi/Control.java
/uk/co/dblue/server/rmi/Checker.java
/uk/co/dblue/server/rmi/Receiver.java
/uk/co/dblue/server/rmi/Interpreter.java
/uk/co/dblue/server/rmi/Handshaker.java
/uk/co/dblue/server/rmi/RMIStack.java
/uk/co/dblue/server/rmi/Transaction.java
/uk/co/dblue/server/rmi/Wrapper.java
/uk/co/dblue/mobile
/uk/co/dblue/mobile/comms
/uk/co/dblue/mobile/comms/bluetooth
/uk/co/dblue/mobile/comms/bluetooth/Connection.java
/uk/co/dblue/mobile/comms/bluetooth/Control.java
/uk/co/dblue/mobile/sql
/uk/co/dblue/mobile/sql/Connection.java

/uk/co/dblue/mobile/sql/ResultSet.java
/uk/co/dblue/mobile/sql/Statement.java
/uk/co/dblue/mobile/sql/SQLException.java
/uk/co/dblue/mobile/sql/DatabaseObject.java
/uk/co/dblue/mobile/sql/Builder.java
/uk/co/dblue/mobile/sql/InternalErrorException.java
/uk/co/dblue/mobile/rmi
/uk/co/dblue/mobile/rmi/Dispatcher.java
/uk/co/dblue/mobile/rmi/Control.java
/uk/co/dblue/mobile/rmi/Checker.java
/uk/co/dblue/mobile/rmi/Receiver.java
/uk/co/dblue/mobile/rmi/Interpreter.java
/uk/co/dblue/mobile/rmi/Handshaker.java
/uk/co/dblue/mobile/rmi/RMIStack.java
/uk/co/dblue/mobile/rmi/Transaction.java
/uk/co/dblue/mobile/rmi/Wrapper.java
/uk/co/dblue/mobile/util
/uk/co/dblue/mobile/util/Queue.java
/uk/co/dblue/mobile/util/Observable.java
/uk/co/dblue/mobile/util/Regex.java
/uk/co/dblue/mobile/util/Observer.java
/uk/co/dblue/mobile/util/Serialize.java
/uk/co/dblue/shared
/uk/co/dblue/shared/messaging
/uk/co/dblue/shared/messaging/format
/uk/co/dblue/shared/messaging/format/FormatTag.java
/uk/co/dblue/shared/messaging/format/BodyTag.java
/uk/co/dblue/shared/messaging/format/SignatureTag.java
/uk/co/dblue/shared/messaging/tags
/uk/co/dblue/shared/messaging/tags/ReturnTag.java
/uk/co/dblue/shared/messaging/tags/NACKTag.java
/uk/co/dblue/shared/messaging/tags/Tag.java
/uk/co/dblue/shared/messaging/tags/DeleteTag.java
/uk/co/dblue/shared/messaging/tags/ACKTag.java
/uk/co/dblue/shared/messaging/tags/MethodTag.java
/uk/co/dblue/shared/messaging/tags/ParameterTag.java
/uk/co/dblue/shared/messaging/tags/CreateTag.java
/uk/co/dblue/shared/util
/uk/co/dblue/shared/util/Queue.java
/uk/co/dblue/shared/util/Regex.java
/uk/co/dblue/shared/util/Serialize.java
/uk/co/dblue/shared/exceptions
/uk/co/dblue/shared/exceptions/InvalidTransactionSyntaxException.java
/uk/co/dblue/shared/exceptions/TransactionNeverReceivedException.java
/uk/co/dblue/shared/exceptions/SQLException.java
/uk/co/dblue/shared/exceptions/InternalErrorException.java
/uk/co/dblue/shared/exceptions/NullTransactionCommandException.java
/uk/co/dblue/shared/exceptions/NullTransactionAttributeException.java

Appendix C

Guide For Running The Software

Note: all instructions are for Linux only.

C.1 Client Software

C.1.1 Pre-requisites

A mobile device with Java 2 Micro Edition Runtime with JSR-82 support. Without JSR-82, the software will not operate.

C.1.2 Starting the software

A demo application has been developed using the API. It connects to a PostgreSQL database running on the server machine with the userID *postgres* and no password. It then executes a number of simple database operations. If required, this simple application can be altered easily using the included the document programming API found on the distribution CD.

1. Copy all source code from the distribution CD or from *http://www.dblue.co.uk/* maintaining the directory structure
2. Change into the *DBlue-mobile/dist* directory
3. Install *DBlue-mobile.jar* onto the mobile device
4. Start the server daemon (see C.2
5. Start the DBlue-mobile software

C.2 Server Daemon

C.2.1 Pre-requisites

Execution requires Java 2 Standard Edition Runtime v1.5.0 or later. The software will not work with previous versions of Java.
In addition, the Linux Bluetooth driver (BlueZ) and userland software is required. The avetana JSR-82 driver should also be added to the systems $CLASSPATH variable.

C.2.2 Starting the daemon

1. Copy all source code from the distribution CD or from *http://www.dblue.co.uk/* maintaining the directory structure
2. Start the BlueZ userland software
3. Change into the *DBlue-server/build/classes* directory
4. To allow the daemon to connect to a database, edit *run.sh* changing *org.postgresql.Driver* for the appropriate JDBC driver(s).
5. Add the database driver to the $CLASSPATH variable
6. Check that the database will accept connections from the server machine
7. Execute run.sh with the following command

```
homer ~$ ./run.sh
```