Bluetooth technology based Network Support System for Development of Context-aware Applications

End of Degree Project

David Ferrández Bell

Directed By Dr. Pietro Manzoni

Faculty of Computer Science

Polytechnic University of Valencia
To my family, for the support you have always given me to pursue my dreams.

To Dad, wherever you are, I hope you will be always proud of your son.
Abstract

This project’s aim is to study and create a “Proof of Concept” for using Wireless Bluetooth Technology as a network support system for Context-aware applications. This Project will cover the basic and technical characteristics of Bluetooth, covering the most important parts of the Bluetooth Specification and its implications in everyday applications, as well as those relevant to the deployment of such Context-aware Systems. This project will also study the steps taken to create a Bluetooth based Context-aware application, difficulties aroused and remarks. Also many conclusions, ideas and personal opinions will be made upon the development of this project.
# Table of Contents

Abstract .................................................................................................................................................. 3

Table of Contents .................................................................................................................................. 5

Introduction ........................................................................................................................................... 7

Ubiquitous Computing and Context-aware Applications ........................................................................ 11

Introduction .......................................................................................................................................... 11

Network Support System: Bluetooth .................................................................................................... 14

Introduction .......................................................................................................................................... 14

Introduction to the Bluetooth Standard .................................................................................................. 15

Bluetooth Protocol Stack ......................................................................................................................... 19
  Bluetooth Radio .................................................................................................................................... 22
  Bluetooth Baseband ............................................................................................................................... 25
  LMP Link Manager Protocol .................................................................................................................. 35
  HCI Host Controller Interface ............................................................................................................... 39
  L2CAP Logical Link Control and Adaptation Protocol ......................................................................... 42
  RFCOMM .............................................................................................................................................. 44
  Bluetooth Audio .................................................................................................................................... 46
  SDP Service Discovery Protocol .......................................................................................................... 47

Bluetooth Profiles .................................................................................................................................. 50
  PAN Personal Area Networking Profile ................................................................................................. 53

Bluetooth Pros, Cons and Application Development Precautions ......................................................... 55

Interference Issues .................................................................................................................................. 60

Bluetooth Stack Implementations and Development Kits ...................................................................... 63
  Axis OpenBT ........................................................................................................................................ 65
  IBM BlueDrekar .................................................................................................................................... 66
  Nokia Affix ............................................................................................................................................ 67
  BlueZ: The Linux Open Source Kernel Bluetooth Support .................................................................. 81
  Why BlueZ? ......................................................................................................................................... 97

Implementation of a Context-Aware System based on Bluetooth Technology ....................................... 98

Aims ....................................................................................................................................................... 98

Basic System Description ....................................................................................................................... 100

Bluetooth Limitations ............................................................................................................................. 103

System Development, Process and Tools ................................................................................................. 105
  Museum Central Server ....................................................................................................................... 108

Museum Information Points .................................................................................................................... 114
  Museum Information Point Clients ..................................................................................................... 125

Results ................................................................................................................................................... 136

Further development and proposals ....................................................................................................... 146

Conclusions ............................................................................................................................................ 148

References and Bibliography ................................................................................................................ 149
Introduction

During the last ten years, technology and access to information has changed the lives of many people. It seems yesterday when trying to find information for our work or studies was as easy and intuitive as spending an afternoon going through the passage ways of our local libraries, taking hours to locate a specific piece of information through piles of thick manuals and bibliographic references. Today’s reality is that our everyday lives have changed radically. In seconds we can access vast amounts of information across the globe, in a manner impossible to imagine just a few years ago. The Internet has changed the way many of us work, play and have our personal relations. A whole new world of business opportunities have been born and are waiting for the new world of users to mature in the use of the Information Technologies.

If technology-wise the major advance has been the easy access to the Internet to our desks, the so called Information Super-Highway at our desks, the next step, that we can start to enjoy today and that will have major expansion in the years to come, is having access to our information with us, whenever, wherever. Wireless and cellular technologies allow us to experience the Net on the go. PDA’s and even mobile phones are getting smaller with more and more processing power, wireless networks are getting bigger, faster and cheaper. The difference between computation and communications is blurring. Today we can often see places where we can hook up our laptops or PDA’s to the Net without having to link up by cable. Whether its IEEE802.11 or cellular GPRS we can access our information easily from our current location. “Why should we have to disconnect now that our lives are starting to evolve on these technologies?” as people start to think.

More so, now that technologies and science are evolving so rapidly, why should we have to carry on doing many things manually? If the technology exists, why can’t it work for us and make things much easier and let us work and concentrate on more important things? Even the use of new technologies can be time consuming and tedious for an end user.
“A fundamental measure of progress in computing involves rendering it as an inseparable part of our everyday experience while making it disappear.”\(^1\) Now that computing and the use of the New Technologies is an everyday tool, why shouldn’t it evolve into something transparent, something we know that’s there, but that allows us to carry on working without noticing it. What’s more, today we have devices of considerable processing power everywhere, PDA’s, cell-phones, laptops, even some watches can give GPS functions, why can’t all these devices communicate and “do something together”?

This main change, which will be tomorrow’s world, is what we call Ubiquitous and Pervasive computing and Context-aware applications that use these new ideas of computing. Therefore we can see that the world of computing has evolved from one big machine in a corporation to a near future of many small or embedded machines that unnoticeably, but ever present, work in conjunction for an end user, these devices can be our personal PDA or embedded in many objects in the locality or even in our clothes. This is Ubiquitous computing.

Interconnected computing devices, along with various sensing technologies, from simple motion sensors to electronic tags to video cameras, are being used to make physical rooms and buildings “intelligent.” Interaction with computation can soon be an “environmental” or contextual and communal experience rather than a just a virtual and private one. Through these developments, computation is invading the fabric of our personal and social activities and environments.

Context-aware applications (or Pervasive Computing) are exactly what their name indicates, that is, applications that offer a flexible, adaptable service to a client or user and that are totally conscious of the user, what surrounds him, of its environment, or anything that might be useful in order to make a better experience for the end user. The application may change or adapt its functions, information and user interface depending on the context (by inferring or sensing it) and client of that precise moment.

Much progress has gone on in the Human-Computer Interactions area to help the definition of these types of applications, but still, Ubiquitous and Pervasive Computing remains an unexplored area with much to be investigated and developed.

On the other hand, for real Context-aware, Ubiquitous Computing, it is necessary for it to be based on a Mobile or Wireless network technology. Many definitions of Ubiquitous Computing state that it’s a combination of Pervasive Computing based on individual Context Awareness and Mobile computing. A good choice of network technology is Bluetooth; its versatile and flexible nature, along with many features like the ability to locate neighbour devices and find out what type of services it offers makes it a perfect network solution for Context-aware applications.

The purpose of this End of Degree Project is to study, design and develop a Network Support System for Context-aware Applications and give a proof of concept application for such system.

In this Project I will show a lot of what Bluetooth has to offer, along with practical installation and use information. I will make an introduction to the Bluetooth technology through the Bluetooth Standard, I will also show what Bluetooth implementations exist, going through the most important, showing how the pros and cons of each of them. The reason to choose the BlueZ Bluetooth implementation will be stated and the basic programming functions offered by the API that make the development of such applications quick and easy will also be presented.

After understanding the basic concepts of Context-aware applications, and how Bluetooth will support the whole system, I will explain how this all fits in the final development of a Context-aware application for a Hypothetical Museum.

The system will allow a quality service for the visitors to the museum, giving them precise information about what they are viewing at their level of knowledge in their natural language of preference and giving a possibility for the user to have a GUI adapted to his device, enhancing their experience. This will also help the maintainers of the museum, to reduce costs in guiding their visitors, to keep track of what are the preferred pieces of art, and so on. Using Bluetooth and/or embedded technologies this
can be developed quite quickly. Bluetooth offers some functions that directly can be used for these purposes.
Ubiquitous Computing and Context-aware Applications

Introduction

Ubiquitous or “Pervasive Computing is a term for the strongly emerging trend toward:

- Numerous, casually accessible, often invisible computing devices.
- Frequently mobile or embedded in the environment.
- Connected to an increasingly ubiquitous network infrastructure composed of a wired core and wireless edges.”

Pervasive/Context-aware applications will be the future of personal computing, mainly because it will not interfere with traditional computing and because it will enhance the use of information to a much higher level of quality.

“This concept implies the computer has the capability to obtain the information from the environment in which it is embedded and utilize it to dynamically build models of computing. The process is reciprocal: the environment can and should also become “intelligent” in that it also has a capability to detect other computing devices entering it. This mutual dependency and interaction results in a new capacity of computers to act “intelligently” upon and within the environments in which we move.”

The information obtained from the environment is what we call the context in the usage of a specific service. We can consider the context as a series of factors that determine a state of a user and therefore infer an enhancement on that user’s usage of services. These factors can be taken by sensors, RFID tags or other mechanisms like Bluetooth, taking into account certain information for a Context-aware application e.g.: The identity of the user, his position or location when he is actually using the service giving him information of the area nearby, or others more difficult to infer like knowledge or education of the user, his mother tongue or language of preference, his personal interests...

Using this information the application can offer an adapted version of its service, giving a very personalized view depending on who, where, how and when the service is

---

1 Bluetooth for Context-aware Applications, 2001
2 ACM Special Issue of Human-Computer Interaction, Volume 16, 2001
used, minimizing unwanted information. The application might also consider on giving dynamical information based on external events occurring in that precise moment, like weather, world news or interactions of others users of the same system which would also allow a group or community based experience to the services. All this leads to a very high quality service that can open many new businesses opportunities, as the cost to reach customers gets reduced drastically.

If the system can find out where the user is operating from, whether the device informs the service provider of the users’ position (using GPS or Galileo-like devices to find its current location) or whether the server is close to the user, it can also give location based information of the surrounding area, like the closest Chemist, selection of recommended restaurants, special sales or offers in the nearby Commercial Centre. All this location based information can be filtered in way to the interests and habits of the user (type of clothes, needed medicines, preferred food...) and also based on other external events like, only taking into account emergency Chemists if it’s out of commercial work-hours.

On the other hand we can define Ubiquitous as: "existing or being everywhere at the same time: constantly encountered: widespread"\(^3\)

Therefore we can understand “Ubiquitous Computing” as ever present, widespread computing. More so, we can fit this definition perfectly into our world of Context-aware applications, as for slick, full functionality, we need to be surrounded by devices, that work for us, even if invisible. These devices can provide us with a lot of functionality, as they can contact, retrieve and use information along with context-awareness can to provide us with much better or more refined services.

Ubiquitous Computing is an extension to Pervasive or Context-aware Computing; it takes for granted the concepts and extends it.

But as the necessary devices exist (PDA’s, laptops...) there is a capability to physically move computing services with us. As a result, the computer becomes a taken for granted, ever-present device that expands our capabilities to inscribe, remember, communicate, and reason independently of the device’s location. This is Ubiquity.

\(^3\) Merriam-Webster Online Dictionary
Ubiquitous computing along with the sensing of Context-aware applications also means that any computing device, while moving with us, can build and remember models of its various environments and configure its services accordingly.

Typically a user with a laptop computer, a PDA or a mobile phone can use this technology to:
1. Find a service of interest near the current location.
2. Connect to it.
3. Configure, access and use the service.
4. Disconnect.

With Ubiquitous Computing / Context-aware applications the search and use of services can be done with or without user intervention. Without user intervention the device requires only knowledge of the user profile, to be able to find services that the device knows of interest of its user, and auto-configure itself. Alternatively the user can guide the device in the search when specific information is needed. The device can also learn from past human interventions to refine its user profile for future needs, building its environmental model.
Network Support System: Bluetooth

Introduction

The choice of the use of Bluetooth is due to its low power consumption, short-range wireless network capabilities. It is designed to be small, and included in practically any device, even to be carried with the user. It has also been designed to keep costs low and to hopefully reach the $5 per chip mark. But what makes it a specially good solution is many of its own features, including device and service discovery which makes much more easier the deployment of Context-aware applications.

It can form a wireless “ad-hoc” network, where there is no infrastructure to support networking. Along with service discovery any device can create a connection on the fly without user intervention. This makes applications easier to use for the end user, and also reduces maintenance costs, being ad-hoc there is a possibility for each client to pass-on the service to other clients allowing to service to be used out of the reach of the servers range. Many possibilities and alternative set-ups are available, allowing interaction between thousands of devices, making many tasks and communications configuration automatic, eliminating cables, and tedious setup for the end user.

Thanks to inquiry we are able to find devices in our neighbourhood. Service discovery and browsing allows the user to find required services from those devices and all the necessary configuration and connection parameters which in turn gives the user an easy, transparent, interconnection of devices using Bluetooth, therefore Context-aware applications are made so much easier. For the service provider, these features allow identifying, locating and finding out what type of users are the ones nearby and giving them an according service.
Introduction to the Bluetooth Standard

Bluetooth is a new standard for short-distance digital radio connections. The technology was named after the great Viking chieftain Harald II, nicknamed Bluetooth due to a tooth disease (910-940 AD). Harald ruled during the 10th century in Denmark during a very troubled period of time. People settled their differences with a sword rather than trying to come to a common understanding. This was a time when Vikings ruled and the belief in Asatro was widely spread. Harald succeeded in bringing all free men to a council where they could make decisions and socialize with each other. Thus he was able to make peace in Denmark and friends in other countries.

The original idea of the Bluetooth came from two Ericsson Telephone Co. employees, Sven Mattisson and Jaap Haartsen in 1994. They became interested in the story about Harald Bluetooth and named the standard after him. In Bluetooth several technologies are united. The standard enables devices, such as laptops, mobile phones, and printers etc. to connect wirelessly in short distance networks. This is a new concept but is already being used in a number of applications.

Imagine the number of problems that arise when we start connecting devices to a computer, or household electronics to each other (TV to VCR or DVD to HIFI), the number of cables used and many different interconnections make it something not trivial for an end user. We can start to think if it wouldn’t be easier to use some other technology to bind them, like radio. Many proprietary solutions exist, putting receivers and transmitters on each device, but none are interoperable, all are incompatible between them. Except for Bluetooth. Other solutions like HomeRF seem to have disappeared leaving the market open to the Bluetooth solution.

What makes Bluetooth attractive is its open standard for all the Computer and Telecommunications industry. In the standard it is described how small (and not so small) devices should be interconnected, using radio frequency. Several companies are already developing Bluetooth products and the number of developers is increasingly rapidly. All Bluetooth certified products are able to work together with any other
Bluetooth product. To ensure this, the Bluetooth SIG (Special Interest Group) has published the Bluetooth Specification.

The specification arrived late 1998 with the collaboration between various telecommunications and computing industry giants which originally formed the SIG: Ericsson, Nokia, Intel, IBM, Toshiba, and Motorola; joined later by 3Com. Today there are more than 1600 companies that belong to the SIG, which should ensure a very good future for this technology. It is estimated that there should be more than 1000 million Bluetooth devices on the street by 2005, although Bluetooth’s growth has been slowed down by economic recession and slow adoption by the end user.

The Bluetooth Specification is divided into two Volumes: The Core and the Profiles. The Core describes the radio characteristics and protocol hierarchy and functionality. The Profiles describes profiles for using the Bluetooth protocols in different applications to ensure interoperability. This document focuses on the Core volume. However, parts of the Profiles volume are also included.

The latest Specification and Profiles Volumes are version 1.1, released February 2001. 4

One of the reasons for a great success is that the radio frequency is defined in the ISM band (Industrial Scientific Medical). This is free in all countries except France, Spain and other countries but they studying redefining their spectrum to allow Bluetooth and IEEE802.11, another great technology that is booming away today. This means that companies for the first time can develop products with radio communications that they can sell all over the world without modifications.

4 Bluetooth SIG Specification of the Bluetooth System Core (vol1) and Profiles (vol2).www.bluetooth.org
Some Technical features include:

- **2.4 GHz ISM Open band**
  - Globally free available frequency, 79 MHz of spectrum utilised on 79 channels
  - Frequency hopping and time division duplex

- **6 – 100 M range, personal “bubble”**
  - 8 active devices in each piconet (shared data rate)
  - 10 piconets in the same bubble (full data rate)

- **Power consumption**
  - 1 mW with 6 m range
  - 100 mW with 100 m range

- **1 Mbps gross rate (symbols per second) 723kbps real data rate.**
  - Future versions will provide higher speed

- **Simultaneous voice/data capable**
  - 432 Kbps (Full duplex), 723/56 Kbps (asymmetric)
  - or 3 simultaneous voice channels
  - or a combination of both

In what kind of applications can we use Bluetooth? At the beginning, Bluetooth was designed to be a cable replacement for home and work environments. But as it was developed the visions got greater and greater. Bluetooth is designed to supply a wide range of application with the need of wireless communication. It can be used in many different ways. Some examples could be:

Synchronise cellular phone with PC or Palm, if you add a persons address and phone number in you cellular, the PC will also get the address when the two devices gets into range, completely automatically. You don’t even need to pick up the phone from your pocket. Since the devices starts to talk to each other when they get into range, the Bluetooth can be used for access systems. When you get close enough the modules identifies each other, and in the case of this work, sends an identification number to a server application.
If you have a cellular phone with Bluetooth, you can automatically connect to the phone line when you are at home. You just need one phone and you don’t have to worry about the higher rates in the GSM net.
Bluetooth Protocol Stack

The Bluetooth Protocol Stack is quite complex, but this complexity also gives it a versatility that makes it perfect to work in heterogeneous environments. I will give a basic introduction to each layer.

![Simplified Bluetooth Protocol Stack](image)

**Fig. 1.- Simplified Bluetooth Protocol Stack**

The Bluetooth Radio (RF) layer forms the physical layer of the protocol stack. This layer incorporates the actual radio circuits. The layer is responsible for the transmission of data through the air.

The baseband layer forms the data link layer, which means that this layer either constructs packets to be sent over the physical layer or it receives packets from that layer. The baseband layer also takes care of packet acknowledgements, packet retransmissions and flow control. The baseband layer is physically implemented in the link controller, which is closely connected to the radio circuits. The link controller can be fed with audio data, which it passes on, in packets, for the RF layer to transmit. Audio feeds use high-priority packets when transmitted and follow ITU-T and GSM recommendations.

On top of the baseband layer is the LM (Link Manager) protocol, which forms the network layer of the protocol stack. This layer allows higher protocols on the protocol stack to invoke link set up, security and control commands. In order to execute
these commands peer LMs send PDU (Protocol Data Units) to each other, containing link set up, security or control information. LM PDUs, received from the baseband, are filtered out by the LM on the receiver side and are not propagated to higher layers. All layers up to the LM protocol layer are implemented in Bluetooth hardware. Higher protocols find their implementation in software running on a Bluetooth host.

To be able to access all the SAPs (Service Access Points) of the LM protocol and to be able to exchange data with the baseband layer such as such as setting up a connection or start an inquiry, the HCI (Host Controller Interface) protocol is used to provide an interface between the Bluetooth hardware and a host, using a physical connection between them. It also works in the other direction sending event packets back to the host, such as disconnection occurred or inquiry result events.

The HCI is independent on the hardware allowing transparency with the software, making programming easier.

The L2CAP (Logical Link Control and Adaptation Protocol) is the transport layer of the protocol stack. This protocol supports higher-level protocol multiplexing, packet segmentation and reassembly. The session layer can contain many different protocols. For instance, TCP/IP packets can be sent with Bluetooth and with RFCOMM a serial communication port can be simulated on a PC. It is possible for many more protocols to send their packets over Bluetooth.

On top of L2CAP, we have an option to use SDP (Service Discovery Protocol). This protocol defines the communication protocol and the data structures to use in order to provide or obtain a description, configuration and connection details to use a certain service that a server advertises. With all this information a client can almost automatically connect and use services nearby that are advertised by local devices. Most applications are defined in the Bluetooth Spec (Profiles) and therefore the use of SDP comes essential. Context-aware applications will use heavily this protocol, due to the almost invisible nature of Ubiquitous applications, it must be defined how devices can use services that are announced and connect without user intervention.
With all the layers of the stack comes along the definition of profiles (Volume 2 of the Bluetooth standard). Bluetooth Profiles try to guarantee interoperability in a wide set of applications. Using profiles simplify the implementation since the product only has to be fully compatible with the profile it is qualified for. The cordless phone may not need to talk to a printer or an industrial sensor. One device may support several profiles. The profiles are used for defining a common user experience. It is possible to register a new profile if you develop an application that does not fit into any existing profile.

In Bluetooth 1.1 there are a few profiles defined:

- Generic Access profile
- Transport profiles
- Telephony profiles
- Object exchange profiles
- LAN Access and Dial-up networking profiles
- Printing service profile
- And many, many more.

A profile describes which parts and functions of the Bluetooth stack that is in use by the application. Only these parts need to be implemented.
Bluetooth Radio

The Bluetooth radio operates in the 2.4 GHz ISM (Industrial Scientific Medicine) band. This band is unlicensed (with restrictions in France, Spain and Japan) and can be used by all kinds of radio equipment world-wide. The Bluetooth radio therefore has to deal with a lot of interference. To overcome this problem a frequency hopping algorithm is used. The frequency at which the radio is operating is changed every 645 µs. If a certain frequency is blocked by interference, then this will have a minor influence on the total data transmission rate of the Bluetooth radio. This is because after hopping very quickly to a new transmission frequency there will hopefully be no interference.

<table>
<thead>
<tr>
<th><strong>ISM regulatory range</strong></th>
<th>2.400 - 2.4835 GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Number of Channels</strong></td>
<td>79</td>
</tr>
<tr>
<td><strong>Channel Spreading</strong></td>
<td>1 MHz</td>
</tr>
<tr>
<td><strong>RF channels</strong></td>
<td>2402 + k Mhz, k = 0...78</td>
</tr>
<tr>
<td><strong>Lower guard band</strong></td>
<td>2 Mhz</td>
</tr>
<tr>
<td><strong>Higher guard band</strong></td>
<td>3.5 MHz</td>
</tr>
</tbody>
</table>

Table 1: Basic Frequency usage

The Bluetooth radio defines 79 channels each separated by 1 MHz in frequency. The first channel starts at 2.402 GHz leaving a 2 MHz guard band, which insures that no transmission will take place outside the ISM band range. The upper channel, which is located at 2.48 GHz, leaves a guard band of 3.5 MHz.

The Bluetooth radio equipment is classified into three power classes. For class 1 -equipment power control is required. The power control is used for limiting the output power above 0 dBm. Equipment of class 1, with a maximum output power of +20 dBm, must be able to control its output power down to +4 dBm or less.

The sensitivity threshold of a receiving Bluetooth radio device is defined at the point where the BER (Bit Error Rate) of the incoming data becomes 0.1% or less.
This approximately occurs at an input level of -70 dBm. Using 0 dBm output power the input signal at the receiver side drops to -70 dBm close to a range of 10 meters. This range becomes around 100 meters if transmission takes place at +20 dBm.

The modulation used by the Bluetooth radio is GFSK (Gaussian Frequency Shift Keying). The next figure shows how a ”110” sequence is transmitted.

![Fig. 2: Modulation using GFSK of “110”](image)

The modulation of symbols is achieved by shifting the frequency at which the radio is transmitting. A positive deviation from the transmit frequency corresponds to a binary one, while a negative deviation corresponds to a binary zero. If a ”1010” sequence is transmitted, then the frequency deviation must every time be greater than 80% of the frequency deviation while sending a ”00001111” sequence. Furthermore, the
minimum deviation must never be smaller than 115 kHz and the maximum frequency deviation must lie between 140 kHz and 175 kHz. The symbol timing must be better than ±20 ppm and the zero crossing error, which is the difference between the ideal symbol period and the actual crossing time, must be less than ±1/8 of a symbol period. With the ideal symbol period being 1 µs, this corresponds to a bit rate of 1 Mbps.

Other Bluetooth radio details can be found in the 1.1 Bluetooth Specifications, as further details are considered out of the scope of this project.
Bluetooth Baseband

The complete Bluetooth system consists of a radio unit, a link control unit and a unit which is embedded with the LM and HCI protocols.

![Functional Blocks of a Bluetooth System](image)

The link controller carries out the baseband protocol and other low-level link routines. The link controller is controlled by the link manager, which receives its commands from LM PDUs or from packets send by the host using the HCI protocol.

The baseband is responsible for transmitting and receiving bytes belonging to packets, which are sent from one Bluetooth device to another, using flow control. The link controller controls the Bluetooth radio to create a physical channel. On the physical channel the baseband is able to make physical links to send packets across.

Besides packet transmission, the link controller also has security and channel control routines implemented. The channel control routines are, among other routines, a link set up routine and a routine, which scans for other Bluetooth devices.

One of the security routines is responsible for the exchange of a key between two Bluetooth devices. This process is called pairing. After a key is obtained it can be used in encryption and authentication routines, in order to encrypt transmitting data and to authenticate a Bluetooth device.

Bluetooth supports point-to-point and point-to-multipoint communication. If Bluetooth devices are being connected to each other when they are brought together, then they will form, what is called, a piconet. Up to eight devices can be active in a piconet, one device will act as master, while the other devices will act as slaves. Each slave will be connected point-to-point with its master. And a Master will have a point-
to-multipoint with its slaves. It is possible for a master to also be a slave in another piconet or for a slave to be connected to multiple masters. In this way piconets are combined to form a scatternet.

Fig. 4 a) Point-to-point and b) point-to-point piconets. c) scatternet.

A Bluetooth device can take two actions to become an active member of a piconet. The device can start a page-scan procedure; this will seek for a packet with paging information sent by another device. If the device is paged it will send back a response packet and after that a connection will be established, with the device becoming a slave in the piconet. The other more direct way of making a connection is to page another device. If a connection is established, then the paging device automatically becomes the master of the piconet. Before paging can take place it is necessary for the paging device to know the specific Bluetooth device address (BD_ADDR) of the device being paged. Prior to paging, an inquiry procedure can be started; this will wait for other devices in inquiry scan mode to send back inquiry response messages containing, among other values, their Bluetooth device addresses. Therefore we have inquiry to find other devices, inquiry-scan to let other devices find the local device, page to create a connection forming a piconet with another device, and page-scan to let another device page the local Bluetooth device.
A Bluetooth slave device can be placed in hold, park or sniff mode, as soon as it has a connection established. This apart from saving energy, as devices in hold, park or sniff, have different energy consumption rates due to they are less active on the piconet, only listening at certain intervals, it is also possible to increment the number of slaves associated to a master. With slaves in park mode it becomes possible to have more than seven Bluetooth devices connected to the master. The parked slaves cannot take part in active data transfer to the master, but still keep their connection. The master can unpark a parked slave to make it a member of the possible seven active slave devices in a piconet. Devices in hold mode also do not take part in active data transfer, but devices in this mode stay one of the possible seven active devices in a piconet. Devices in sniff mode lower the rate at which they seek for packets send by the master, in this way they lower the data rate between them and the master.

This makes it possible for easier communication between devices but also, has a negative effect on data-rate and higher level protocol connection times as the devices are not ready for communication when higher levels want it, so the master must “wake up” the devices, along with clock drifts and synchronization makes connection times reach considerable lengths of time.

The 79 (23 France and Spain) hopping channels implemented in the Bluetooth radio are used semi-randomly by the baseband to create a physical channel.

Every Bluetooth device has a running clock, which revolves approximately once every day. If a connection is established with another Bluetooth device, then the master sends its Bluetooth device address and its clock information to the connected slave. The slave compares the clock information with its own native clock and calculates an offset. This offset can then be used together with the native clock to calculate a clock synchronized to the master. This master clock and the master Bluetooth device address form the input for the frequency hopping algorithm, which chooses semi-randomly one channel out of 79. Because both slave and master use the same input for the algorithm they both send and receive at the same hopping frequencies and are therefore synchronized to each other, this is like that for all devices in the piconet, making each piconet have a different hopping sequence. A TDD (Time Division Duplex) scheme is used where master and slave alternatively transmit packets in slots, with slots starting
every time a new hop frequency is chosen, every 645 µs. If the master sends a packet to a slave, then only the slave that has the address on that packet sent to is allowed to send a packet back to the master in the following slot. Master and slave are able to send packets extending one slot. These multi-slot packets can either be 3 or 5 slots in length. All slots in a multislot packet transmission use the hopping frequency, which is calculated for the first slot being transmitted.

Between master and slave two physical link types have been defined.

- **SCO.** Synchronous Connection-Oriented, for high priority data such as audio.
- **ACL.** Asynchronous Connection-Less, for data.
Up to three SCO links can be supported by the master. The master sends SCO packets over the physical channel to a specific slave at regular intervals, guaranteeing synchronous transmission. The slave is allowed to send a SCO packet back in the following slot. The SCO link can be seen as a circuit-switched connection and is therefore used with time-bounded information like voice data.

ACL packet exchange, with any active slave in a piconet, takes place on a per-slot basis. An ACL link provides a packet-switched connection. Only one ACL link can exist between a master and a specific slave. ACL links are used for data transmission. Packet retransmission is applied to assure data integrity.

![Bluetooth Packet format](image)

**Fig. 7 Bluetooth Packet format**

Data transmitted on a piconet channel is conveyed in packets. The packet format is shown in Figure 7. Each packet can consist of the following 3 fields: access code, header and payload.

These fields can have the following contents:

- **Access code**
  - *Preamble*
  - *Sync word*
  - *Trailer*

- **Header**
  - AM_ADDR
  - Type
  - Flow
  - ARQN
  - SEQN
  - HEC

- **Payload**
  - Class of device
  - AM_ADDR
  - CLK
  - Page scan mode

- **ACL packet payload field**
  - L_CH
  - Flow
  - Length
  - Payload
  - CRC
The access code can either be a channel (CAC), device (DAC) or inquiry (IAC) access code. If the access code is a CAC, it will carry general ACL or SCO data link packets. If the access code is a DAC, it means they are packets used for synchronization between master and slaves. The payload field contains FHS packet data, which consists, among other things, of the master’s BD_ADDR and master clock information.

IAC is used during inquiry and inquiry response messages. There are two types of IACs defined. GIAC (General Inquiry Access Code) is used to make an inquiry into all the devices in the vicinity. DIAC (Dedicated Inquiry Access Code) is used to make an inquiry into a dedicated group of Bluetooth units that share a common characteristic.

- **Header**
  - **AM_ADDR**
    - The active member address (AM_ADDR) is a temporarily 3-bit address given to all the active Bluetooth devices in a piconet (3 bits, 8 devices). Slaves that are disconnected or parked give up their AM_ADDR. A slave receives a new AM_ADDR when the master sends it a FHS packet during a page response. The AM_ADDR field in the header carries the AM_ADDR of the active slave device that the packet was send to or received from.
  - **TYPE**
    - The type field determines the type of the packet. Five control packet types have been defined. One of these control packet types is the ID packet, which cannot be determined using the type field, because this packet does not contain a header and payload.
    - The other control packets are:
      - **NULL packet**
        - This packet is sent by the slave to the master if the slave does not have data to transfer.
      - **POLL packet**
        - This packet is similar to the NULL packet. It is used by the master to poll slaves, without having to send data.
Faculty of Computer Science

- FHS packet
  - This packet is used during page response and inquiry response. The payload contains synchronization information and other important device information.

Besides the control packet types, also packet types for packets carrying SCO or ACL link data in their payload are defined. These packets can extend over more than one slot. The packet type determines the length of the data in the payload for SCO links and if CRC is added for ACL links.

<table>
<thead>
<tr>
<th>Slots</th>
<th>SCO Link</th>
<th>ACL Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DM1, HV1, HV2, HV3, DV</td>
<td>DM1, DH1, AUX1</td>
</tr>
<tr>
<td>3</td>
<td>DM3, DH3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>DM5, DH5</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Packet types for SCO and ACL links

Please note that it is needed for SCO links to occupy only one spot at a time, in order to guarantee time-bounded communication of up to 3 concurrent SCO links.

- FLOW
  - This bit is used for flow control of packets sent over an ACL link.
- ARQN
  - This bit can either signal a positive or negative acknowledgment of a packet being received correctly.
- SEQN
  - This bit provides a sequential numbering scheme to order the packet stream. The SEQN bit helps the receiver to discard these retransmitted packets.
- HEC
  - This field contains header error check information to insure header integrity
• Payload fields. The following fields are what I considered the most important. For more details please refer to Bluetooth the Specification, Part B.

  o **FHS packet.**
    ▪ Parity bits
      • This field contains the parity bits of the sync word of the access code of the device that sends the FHS packet.
    ▪ Class of device
      • This field contains the class of device.
    ▪ AM_ADDR
      • This field contains the active member address, which the slave receiving the FHS packet obtains after the master sends this packet in a page response. If a slave sends this packet during an inquiry response to the master, then this field contains only zeros.
    ▪ CLK
      • This field contains the native running clock of the device that sends the FHS packet. For each new transmission of a FHS packet this field is updated so that it accurately reflects the real-time clock value.
    ▪ Page scan mode
      • This field indicates which scan mode is used by the sender of the FHS packet. Currently one mandatory scan mode and three optional scan modes have been defined.
o ACL data packets
  • L_CH
    • This field specifies to which logical channel the ACL data belongs. Two logical channels for ACL data have been defined. One logical channel is used for the LM (Link Manager). The other logical channel transfers ACL data to the L2CAP layer, this logical channel sends user ACL data between Bluetooth devices, in a very similar way to what TCP ports behave.

  • Payload
    • The payload contains the actual ACL data bytes. This can either be user data or a LM PDU (Protocol Data Unit) depending on the logical channel being used.

  • CRC
    • All the ACL link packet types add a CRC code to check the integrity of the payload data, except for the AUX1 packet type.

ACL data packets can achieve data rates up to 723.2Kbps, depending on the type of packet used. The packets differ in size (1, 3 or 5 slots) and form symmetric or asymmetric links. In the next table we can see the data rates Bluetooth offers (in theory).
As we can see, DHxy are packet types without Forward Error Correction (FEC), DMxy do present some FEC. The number that accompanies the Packet type name indicates the number of time-slots used. Therefore DH5 is the highest data rate that can be achieved with Bluetooth, but only in optimal conditions as interference as we will later see has a hard effect on the data rate.
LMP Link Manager Protocol

The LM sends LM protocol messages over the LM logical channel to other peer LMs. These messages facilitate link set-up, security and control procedures between Bluetooth devices.

![Diagram of LM transmission on Bluetooth Stack](image)

LM PDUs can be sent over an SCO link in DV packets, which contain beside voice payload fields also data payload fields. Usually PDUs are sent in DM1 packets over an ACL link.

The transaction ID bit is set to 0 when PDUs are sent by the master and it is set to 1 when a slave sends PDUs. The opcode field either determines which LM command should be invoked at the receiver side of the PDU or it determines which response message is sent back to the device.

![Diagram of typical LM protocol procedure](image)
Two general response PDUs have been defined, which are LM_accepted and LM_not_accepted. LM_accepted is used to tell the initiating LM that the LM command invoked has been accepted by the receiver LM and that the command has been carried out. LM_not_accepted is used to tell the initiating LM that the command was not accepted by the receiver LM. In the case of the response message LM_not_accepted a reason why the command was not carried out is given as a parameter.

A few of the most important procedures, which peer LMs perform, by sending command and response LM PDUs from one LM to another, are the following:

**Pairing**

Before authentication or encryption can take place between master and slave a link key should be agreed on. Both LMs must pair to achieve this. After pairing a link key can be exchanged. It is possible to change the current link key once two LMs have been paired.

**Authentication**

A challenge (random number) is exchanged from a verifier LM to a claimant LM, both sides calculate a result using a link key and the challenge. The claimant LM sends the result back to the verifier LM. If this result matches the result calculated by the verifier LM, then the verifier LM accepts the claimant LM as an authenticated Bluetooth device.

**Encryption**

At least one successful authentication should have taken place before encryption can be used. Before starting encryption two communicating LMs must first negotiate on whether or not to use encryption while sending data. If both LMs agree, then the master LM will negotiate with the slave LM on the key size used during encryption. If the key size has been agreed on, then the master can start and stop encryption.
Master-slave role switch

Master and slave can negotiate on switching their role of master and slave. If negotiation succeeds then the slave becomes the master of the piconet and the master becomes a slave.

Detach

A detach LM message is sent from one LM to another to inform that a connection has been closed. A reason of the connection being closed is given as a parameter in the parameter field of the LM PDU.

Hold mode

An ACL link connection between two Bluetooth devices can be placed in hold mode. The master can send a LM message, which forces the slave to enter hold mode. A parameter is contained in this message, which determines for how long the slave should stay in hold mode. Another way is to negotiate on the time the slave should stay in hold mode. Both the master and slave can start this negotiation. If the negotiation succeeds, then the slave is placed in hold mode for the duration agreed on. After the first time negotiation succeeded the slave is also allowed to force an ACL link to enter hold mode, but not longer then the hold time previously agreed on.

Sniff mode

The master can force a slave into sniff mode. Both master and slave can request sniff mode, negotiation takes place to determine the conditions, which will be applied when the slave enters sniff mode. Both master and slave can send a LM message to the LM on the other side in order to return from sniff mode.

Park mode

The master can force the slave to enter park mode by sending a LM message. Both master and slave can request the LM on the other side if it is acceptable for the slave to enter park mode. If devices are in park mode, then only the master LM can unpark these devices.
Power control

A Bluetooth device can increase or decrease the output power of the other device. To achieve this, the LM sends either a message to increase or decrease the output power of the LM of the other device.

SCO link

Master and slave can initiate a SCO link. After a SCO link is established both master and slave can change the SCO parameters. Finally both the master and slave are allowed to remove a SCO link.

ACL link

Before an ACL link is established, baseband page procedures take place to make the slave an active member of the piconet. After that, LM PDUs can already be sent over the physical channel. Before an ACL link is created for L2CAP user data the LM first sends a connection request message to another peer LM, which can be accepted or rejected by the peer. After a ACL link connection is accepted procedures for pairing, authentication and encryption can take place. When a link is fully set up, a setup complete message is sent between the LMs.
**HCI Host Controller Interface**

The HCI (Host Controller Interface) protocol is used to make it possible for the host to control the complete Bluetooth device.

HCI packets are sent from a host to a Bluetooth device, containing either user data or commands. HCI command packets can initiate LM procedures, initiate baseband procedures (page, page-scan, inquiry, and inquiry-scan) and control the overall working of the Bluetooth device. User data packets either contain ACL or SCO link data, which the baseband transfers over the piconet. HCI packets send from a Bluetooth device to a host either contain user data received from the piconet or event data. HCI event packets are sent back to the host after the host has sent a command packet. Such event packets contain the results of the commands, which have been executed. Event packets are also sent when an event occurs in the piconet, for instance after a connection has been made or a disconnection occurred or after an inquiry response is received from a slave.

Each physical connection type uses its own transport layer to carry packet across the physical media used. Three transport layers have been defined, one for UART, RS-232 and USB media. The UART transport layer can also be used on a RS-232 physical connection, it places a prefix byte in front of each packet being transmitted. This prefix determines the type of HCI packet, which follows.

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Packet Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>Command Packet</td>
</tr>
<tr>
<td>0x02</td>
<td>ACL Data Packet</td>
</tr>
<tr>
<td>0x03</td>
<td>SCO Data Packet</td>
</tr>
<tr>
<td>0x04</td>
<td>Event Packet</td>
</tr>
</tbody>
</table>

Table 5: HCI Packet Prefix and types

To send a command to a Bluetooth device a packet is built with in the opcode field the code of the command, which has to be executed.
Many commands have been defined, a few of them are:

- **HCI_reset**
  - Resets the Bluetooth device.
- **HCI_create_connection**
  - Creates an ACL connection.
- **HCI_read_BD_ADDR**
  - Reads in the Bluetooth device address.
- **HCI_disconnect**
  - Disconnects a link.
- **HCI_inquiry**
  - Invokes the inquiry.

Inquiry is especially important for the discovery of devices in the vicinity, if the nearby devices allow us to inquire them, through inquiry scanning, we can find the devices and connect to them if we wish. The inquiry results give us the Bluetooth address and clock information to be able to connect to that device.

An ACL data packet consists of a connection handle, which is received after a connection is made with another Bluetooth device. ACL data packets send over the same link should use the same connection handle.

The BC (broadcast) flag determine how the data in the ACL data packet is broadcast in the piconet. The data should be in L2CAP packet format, if else the baseband will not accept the packet. An L2CAP packet consists of a data length field followed by a channel ID field, which is used for higher-level protocol multiplexing. The channel ID field is followed by a field, which contains the actual user data.

SCO data packets have the same format as ACL data packets, except for BC flags are not used and that raw audio data is transferred in state of L2CAP packets.

Event packets start with an event code field, this determines the type of the event that has occurred. Two types of events are defined, which are sent after a command
packet is received by a Bluetooth device. These events are the CommandComplete and CommandStatus events.

The CommandComplete event is sent after a command has finished execution completely. The parameter field of a CommandComplete event packet contains status information and other results from the command, which has been executed.

The CommandStatus event gives the same information, but this event is used in conjunction with commands, which take longer to execute, because of the baseband or link manager being involved in the procedure.

A few of these commands are HCI_inquiry, HCI_create_connection and HCI_disconnect.

InquiryComplete, ConnectionComplete and DisconnectionComplete events are sent to the host when the execution of such commands is finished. Other events with results of the inquiry like InquiryResult are sent when the baseband receives inquiry response messages from other Bluetooth devices in the vicinity. The parameter field of an InquiryResult event packet contains, among other parameters, the BD_ADDR of devices found.

For further information on the LMP layer please refer to the Bluetooth Specification.
L2CAP Logical Link Control and Adaptation Protocol

This protocol supports higher level protocol multiplexing, packet segmentation and reassembly, and the conveying of quality of service information. The data is multiplexed onto the single Asynchronous ConnectionLess (ACL) connection between two devices and, in the case of a master device, directs data to the appropriate slave.

Protocol multiplexing is supported by defining channels. Each channel is bound to a single protocol in a many-to-one fashion. Multiple channels can be bound to the same protocol, but a channel cannot be bound to multiple protocols. Each L2CAP packet received on a channel is directed to the appropriate higher level protocol.

L2CAP abstracts the variable-sized packets used by the Baseband Protocol. It supports large packet sizes up to 64 kilobytes using a low overhead segmentation-and-reassembly mechanism.

It also segments and reassembles the data into chunks that fit into the maximum HCI payload. Locally, each L2CAP logical channel has a unique Channel Identifier (CID), although this does not necessarily match the CID used by the remote device to identify the other end of the same channel. CIDs 0x0000 to 0x003F are reserved with 0x0000 being unused; 0x0001 carrying signalling information; and 0x0002 identifying received broadcast data.

The stack layers that sit above L2CAP can be identified by a Protocol Service Multiplexor (PSM) value. Remote devices request a connection to a particular PSM, and L2CAP allocates a CID. There may be several open channels carrying the same PSM data.

Fig. 10 L2CAP on the Bluetooth Stack
Each Bluetooth defined layer above L2CAP has its own PSM:

- SDP – 0x0001
- RFCOMM – 0x0003
- Telephony Control Protocol Specification Binary (TCS-BIN) – 0x0005
- TCS-BIN-CORDLESS – 0x0007

L2CAP only deals with data traffic, not voice, and all channels, apart from broadcasts (transmissions from a master to more than one slave simultaneously), are considered reliable. But because of the nature of wireless communications, the links provided by the baseband are not reliable. Errors are caused by radio interference or fading of signals. There is a chance that two or more errors in a packet will combine to give a packet that contains errors but still has a correct checksum. The Bluetooth Special Interest Group (SIG) is considering implementing error correction at L2CAP, which would make such errors less likely to affect applications.

The L2CAP also has some very interesting features, like Quality of Service (QoS) and Group Addresses.

The L2CAP connection establishment process also allows the exchange of information regarding the quality of service (QoS) expected between two Bluetooth units. Each L2CAP implementation must monitor the resources used by the protocol and ensure that QoS contracts are reached. This can be, no QoS, “Best Effort”, Negotiated QoS, and Guaranteed QoS.

Group management provides the abstraction of a group of units allowing more efficient mapping between groups and members of the Bluetooth piconet. Group communication is connectionless and unreliable. When composed of only a pair of units, groups provide connectionless channel alternative to L2CAP’s connection-oriented channel.
RFCOMM

The RFCOMM (Radio Frequency COMM) protocol provides standard 9-pin serial port emulation on the functions given by the L2CAP layer channels. This protocol is based on the ETSI standard TS 07.10.

TS 07.10 includes the ability to multiplex several emulated serial ports onto a single data connection using a different Data Link Connection Identifier (DLCI) for each port. However, each TS 07.10 session can only connect over a single L2CAP channel and thus only communicate with one device. A master device must have separate RFCOMM sessions running for each slave requiring a serial port connection.

Up to a maximum of 60 simultaneous serial connections are supported by the RFCOMM protocol between two Bluetooth devices. Whether each device allows this quantity depends on the device and the RFCOMM implementation that device has.

Version 1.1 of the Bluetooth specification has added to the capabilities of the standard TS07.10 specification by providing flow control capabilities. This caters for mobile devices with limited data processing and storage capabilities allowing them to limit the incoming flow of data.

RFCOMM needs a complete communication path between two applications running on different devices. These can either be end-user applications or TCP/IP on the emulated serial port.

RFCOMM is intended to cover applications that make use of the serial ports of the devices in which they reside. Communication can be a simple segment between two Bluetooth devices forming a link (direct connect). When it’s a Bluetooth-Network communication, Bluetooth wireless technology itself is used for the path between the device and a network connection device just like a modem. RFCOMM can support other configurations, such as modules that communicate via Bluetooth wireless technology on one side and provide a wired interface on the other side; this is known as LAN access points (LAP). These devices are not really modems but offer a similar service. Basically two device types exist that RFCOMM must accommodate. Type 1
devices are communication end points such as computers and printers. Type 2 devices are those that are part of the communication segment; e.g. modems. Though RFCOMM does not make a distinction between these two device types in the protocol, accommodating both types of devices impacts the RFCOMM protocol.

RFCOMM offers RS-232 (EIATIA-232-E) serial ports and null-modem emulation. The emulation includes transfer of the state of the non-data circuits.

RFCOMM doesn’t limit baud rates (therefore data throughput) but will respect the limitations imposed by peer devices and higher layer protocols. RFCOMM supports emulation of multiple serial ports between two devices and also emulation of serial ports between multiple devices.

With RFCOMM, Bluetooth devices have another way of using all-popular higher level protocols such as TCP/IP, as serial ports can be mapped on to serial terminals and parsed with line disciplines and/or using PPP connections. But this can also be obtained using other profiles, as we will see later.
Bluetooth Audio

Bluetooth offers high priority audio support, through SCO packets. It is possible to use quite decent codec’s for this purpose, either a 64 kb/s log PCM format (A-law or μ-law, following ITU-T recommendations G. 711) or a 64 kb/s CVSD (Continuous Variable Slope Delta Modulation is used.

Types of SCO packets can be HV1, HV2, HV3 and DV.

HV packets are used for transmission of voice and transparent synchronous data; HV stands for High quality Voice. The voice packets are never retransmitted and need no CRC.

The DV packet is a combined data - voice packet. The payload is divided into a voice field of 80 bits and a data field containing up to 150 bits. The voice and data fields are treated completely separate. The voice field is handled like normal SCO data and is never retransmitted; that is, the voice field is always new. The data field is checked for errors and is retransmitted if necessary.

The next table will show the basic differences between these types of packets:

<table>
<thead>
<tr>
<th>Packet Type</th>
<th>Information Bytes</th>
<th>FEC</th>
<th>CRC</th>
<th>Payload</th>
<th>Speech rate</th>
<th>Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>HV1</td>
<td>10</td>
<td>1/3</td>
<td>No</td>
<td>240 bits</td>
<td>1.25ms @ 64kbps</td>
<td>1/2 slots</td>
</tr>
<tr>
<td>HV2</td>
<td>20</td>
<td>2/3</td>
<td>No</td>
<td>240 bits</td>
<td>2.50ms @ 64kbps</td>
<td>1/4 slots</td>
</tr>
<tr>
<td>HV3</td>
<td>30</td>
<td>No</td>
<td>No</td>
<td>240 bits</td>
<td>3.75ms @ 64kbps</td>
<td>1/6 slots</td>
</tr>
<tr>
<td>DV</td>
<td>10</td>
<td>Not Voice</td>
<td>16bits (Data)</td>
<td>80bits</td>
<td>--</td>
<td>1/2 slots</td>
</tr>
</tbody>
</table>

Table 6: Audio-SCO Packet types
SDP Service Discovery Protocol

Client devices are able to use services to seek out and find other complementary network devices, applications, and services needed to properly complete specified tasks. Other technologies exist in the same field but with different scopes. Sun has enriched Java with its introduction of Jini. Microsoft promotes Universal Plug and Play (UPnP). The IETF has adopted the service location protocol (SLP). Salutation, an independent consortium of leading technology companies, has also advanced a non-proprietary service discovery protocol. But Bluetooth offers a complex, low-level, hardware implementation of the given protocol.

The SDP defines how a Bluetooth client’s application can discover available Bluetooth servers’ services and their characteristics. The protocol defines how client can search for a service based on specific attributes without the client knowing anything of the available services. The SDP provides means for discovery of new services becoming available when the client enters an area where a Bluetooth server is operating.

SDP uses a request/response model where each transaction consists of one request protocol data unit (PDU) and one response PDU. Every SDP PDU consists of a PDU header followed by PDU-specific parameters. Generally, each type of request PDU has a corresponding type of response PDU. However, if the server determines that a request is improperly formatted or for any reason the server cannot respond with the appropriate PDU type, it will respond with an error PDU (SDP_ErrorResponse).

A service to be discovered may be implemented as software, hardware, or a combination of hardware and software. All of the information about a service that is maintained by an SDP server is contained within a single service record. The service record consists entirely of a list of service attributes. Each service attribute describes a single characteristic of a service. SDP has next service attributes: ServiceRecordHandle, ServiceClassIDList, ServiceRecordState, ServiceID, ProtocolDescriptionList, BrowseGroupList, LanguageBaseAttributeIDLList, ServiceInfoTimeToLive, ServiceAvailability, BluetoothProfileDescriptorList, DocumentationURL, ClientExecutableURL, IconURL, ServiceName, ServiceDescription, ProviderName.
These attributes describe different parts of a service, they can tell a client device information on how to connect to the server's service, what protocols are used, what ports, what type of service it deals with if defined in a profile (Audio gateways, PAN, LAN, DUN, SPP, etc.), where the client can download an executable that will understand the service, the services name and description in various languages.

Some attribute definitions are common to all service records, but service providers can also define their own service attributes in reserved fields.

A service attribute consists of two components: an attribute ID and an attribute value. An attribute ID is an unsigned integer that distinguishes each service attribute from other service attributes within a service record. The attribute value is a variable length field whose meaning is determined by the attribute ID associated with it and by the service class of the service record in which the attribute is contained. In the Service Discovery Protocol, an attribute value is represented as a data element. A data element is a structure that defines the type, length and value of a real piece of information.

The whole point of the SDP is to allow Bluetooth devices to discover what other Bluetooth devices can offer (what services). SDP allows this in various means. Searching and Browsing. Searching means looking for specific service, while Browsing means looking to see what services are actually being offered.

The Service Search feature allows a client to retrieve the service record handles for particular service records based on the values of attributes contained within those service records. The capability to search is provided for attributes whose values are Universally Unique Identifiers (UUIDs).

Important attributes of services that can be used to search for a service are represented as UUIDs. Service search pattern are used to locate the desired service. A service search pattern is a list of UUIDs (service attributes) used to locate matching service records.

Browsing however is similar to looking at a list of services to later choose one, if desired. In SDP, the mechanism for browsing for services is based on an attribute shared by all service classes. This attribute is called the BrowseGroupList attribute. The value of this attribute contains a list of UUIDs.
Interoperability between devices from different manufacturers is provided for a specific service and use case, if the devices conform to a Bluetooth profile. The Bluetooth protocol stack contains a Service Discovery Protocol (SDP), that is used to locate services that are available on or via devices in the vicinity of a Bluetooth enabled device. Having located what services are available in a device, a user may then select to use one or more of them.

SDP provides direct support for the following set of services inquiries:

- Search for service by service class.
- Search for service by service attributes.
- Service browsing.

A user can interact two ways with the SDP

- By performing the service searches on a particular device that a user “consciously” has already connected to.
- Or by performing the service searches by “unconsciously” connecting to a device discovered in the devices vicinity.

Both of the above approaches require that devices need first to be discovered, then linked with, and then inquired about the services they support. Service discovery is tightly related to discovering devices, and discovering devices is tightly related to performing inquiries and pages.

Service discovering in Bluetooth protocol realized only basic methods for searching and recognition of devices and proposed services, but for utilization and implementing mentioned above services need another, more sophisticated system based on native SDP procedures and using high-level technology.
Bluetooth Profiles

The Bluetooth profiles have been developed in order to describe how implementations of models of applications are to be accomplished. These application profiles describe a number of user scenarios where Bluetooth performs the radio transmission. A profile can be described as a vertical slice through the protocol stack. It defines options in each protocol that are mandatory for the profile. It also defines parameter ranges for each protocol. The profile concept is used to decrease the risk of interoperability problems between different manufacturers’ products.

A profile is dependent upon another profile if it re-uses parts of that profile, by implicitly or explicitly referencing it e.g. the Object Push profile is dependent on Generic Object Exchange, Serial Port, and Generic Access profiles.

Many profiles have been defined, but here is a short list:

GAP Profile: Generic Access Profile

The GAP Profile defines the generic procedures related to discovery of Bluetooth devices and link management aspects of connecting to Bluetooth devices. It is the core on which all other Profiles are based.

SDAP Profile: Service Discovery Application Profile

The SDAP Profile defines the features and procedures for an application in a Bluetooth device to discover services registered in other Bluetooth devices and retrieve any desired available information pertinent to these services.

CTP Profile: Cordless Telephony Profile

The CTP Profile defines the features and procedures that are required for interoperability between different units active in the 3-in-1 phone application case. This profile also shows how it’s possible to use wireless telephony in a residential or small office environment.
IP Profile: Intercom Profile

The Intercom Profile defines the requirements for Bluetooth devices necessary for the support of the intercom functionality. This is also refereed to as the 'walkie-talkie' usage of Bluetooth.

SPP Profile: Serial Port Profile

The Serial Port Profile defines the requirements for Bluetooth devices necessary for setting up emulated serial cable connections using RFCOMM between two peer devices.

HS Profile: Headset Profile

The Headset Profile defines the requirements that shall be used by devices implementing the so called ‘Ultimate Headset’.

DNP/DUN Profile: Dial-up Networking Profile

The Dial-up Networking Profile defines the requirements that shall be used by devices (modems, cellular phones) implementing the usage model called ‘Internet Bridge’.

FP Profile: FAX Profile

The Fax Profile defines the requirements for Bluetooth devices necessary to support the Fax use case. This allows a Bluetooth cellular phone (or modem) to be used by a computer as a wireless fax modem to send/receive fax messages.

LAP Profile: LAN Access Profile

The LAN Access Profile defines how Bluetooth enabled devices can access the services of a LAN using PPP. Also, this profile shows how the same PPP mechanisms are used to form a network consisting of two Bluetooth-enabled devices.
GOEP Profile: Generic Object Exchange Profile

The Generic Object Exchange Profile lays the basis (defines the protocols and procedures) for Bluetooth devices necessary for the support of the object exchange usage models. The usage model can be the Synchronization, File Transfer, or Object Push model.

OPP Profile: Object Push Profile

The Object Push Profile defines the requirements for applications providing the object push usage model. Typical scenarios covered by this profile involve the pushing/pulling of data objects between Bluetooth devices.

FTP Profile: File Transfer Profile

The File Transfer Profile defines the requirements for applications providing the file transfer usage model. Typical scenarios involve a Bluetooth device browsing, transferring and manipulating objects on/with another Bluetooth device.

SP Profile: Synchronization Profile

The Synchronization Profile defines the requirements for applications providing the synchronization usage model. Typical scenarios covered by this profile involving manual or automatic synchronization of PIM data when 2 Bluetooth devices come within range.

There are many more profiles, and the Bluetooth SIG is currently working on more. For the latest descriptions, and list of profiles, please check the Bluetooth-SIG website, at www.bluetooth.org
PAN Personal Area Networking Profile

Another very important profile is the PAN or Personal Area Networking Profile. This profile uses the Bluetooth Network Encapsulation Protocol to emulate Ethernet networks and to be able to establish various types con network configurations using it:

• Simple Ethernet Emulation by encapsulating Ethernet frames in Bluetooth packets
• Single-piconet IP PAN
• Master Forwarding as a Network Access Point

This means we can have dynamic ad-hoc IP-based personal networking, with support for IPv4 and IPv6 or use a peer of the network to act as a router or bridge between network technologies and effectively create a network access points where the network could be a corporate LAN, GSM, and other data networks.

The PAN profile also makes sure that smaller devices such as PDAs with limited storing and processing power also to be able to access and use this protocol. The result is a similar IP support to Wireless LAN.

The PAN profile depends on the Generic Access Profile, so any device or application that intends to implement the PAN profile must also implement the GAP Profile.

The PAN profile defines a set of device roles when creating the network:

• PAN user (PANU): Client of a NAP or client-type member of a GN.
• Group ad-hoc Network (GN) controller: Forwarding node in a peer-to-peer style network (Bluetooth Piconet). Interconnects up to 7 (active) PANUs to a real self-contained, peer-to-peer network.
• Network Access Point (NAP): Acts as proxy, router or bridge between an existing network infrastructure (typically LAN) and (up to 7 active) wireless clients (PANUs). In this scenario, the radio and host controller appear to be a direct bus connection to a network interface device with network access. Network access points will provide access to other networks technologies such as, ISDN, Home PNA, Cable Modems, and cell phones. The NAP can also
Faculty of Computer Science

configure its PANUs via DHCP so they can transparently configure themselves and access the network.

The NAP can act as a router, or as a bridge (level 2 vs. level 3 forwarding), configuring either is quite simple and certainly gives transparency to the clients. Further on in this project I will show some practical examples of configuring and using these features due to the high interest it proportions. Some purists also think that PAN can force DUN or LAP into inexistence, due to easier configuration and because it proportions easy access to IP networks.

Examples of Bluetooth Protocol Stack with NAP and GN:

![Diagram of Bluetooth Protocol Stack usage with PAN-NAP Profile.](image)

![Diagram of Bluetooth Protocol Stack usage with PAN-GN Profile.](image)
Bluetooth Pros, Cons and Application Development Precautions

As we have seen Bluetooth is a very complex system, this complexity gives it a versatility that makes adequate for many applications. But of course some of Bluetooth’s features have a negative effect depending on the requirements of each application.

Here are some things to take into account on application development:

Security:

Its radio frequency hopping at an incredible speed (1600 hops per second) makes it very hard for hardware sniffers to intercept the transmission. US military sources consider this speed of hopping to make intrinsically safe against sniffing. If the hardware was capable of sniffing in realtime, Bluetooth also offers authorization and authentication via PIN numbers and challenges and encryption through link keys. Though the encryption method used is not high security (the key it is only up to a 128bit random number) but is certainly sufficient, as devices can change the encryption key during the session.

Bluetooth also enables and disables the discovery and connection from devices, so if a service is to be used by a few devices, the master can be already known (through its address) and never located by other devices. Another option is not to allow more connections to that service once connection is set up. Also a device can ignore L2CAP connection-less datagram’s, which will make any device wanting to access a service to go through the authentication process.

Range and energy use:

The range of the three Classes of Device that the Bluetooth-SIG specification defines should be sufficient for most applications. As usual range and power are contrary to each other, meaning that to obtain higher ranges it is needed to transmit the signal with more power, having a negative effect on battery life. The most basic Bluetooth device has a range of 6 meters that should be enough for indoor, household applications. Class 2 devices extend this to 10 meters but to do so it’s needed 10 times the amount in power (1mW vs. 10mW). The top reaching Device Class is Class 1, with a range of 100 meters, but having to use up to 100mW to do so. Power
control/management can be the solution in intermediate cases, but the application must take into account that lowering the power devices that are further away will lose out and be unable to use the service. For most cases it should be sufficient, if more range is needed hybrid and/or fixed networks must be used.

Bluetooth otherwise offers very good energy consumption if compared to other wireless technologies. Its advanced energy saving states (hold, sniff and park) help to extend battery life much more than counterpart technologies, like IEEE802.11. This is essential if Ubiquitous computing is the end use of the technology, due to the very fact that many devices will be used and cannot be forever recharging.

**Speed:**

The maximum data throughput is an average problem and will depend heavily on the final application’s usage of the bandwidth. Where as 723kbps may be good enough for sending emails or surfing the web; we must think of the overall usage of this data rate, as this throughput is shared between all the members of the piconet. Interference is also a problem to think in as obtaining high data rates may be flawed as we will see in the next section. Therefore we cannot guarantee a minimum data rate for the end users. Multimedia applications must take care of the bandwidth it consumes to not overload the link’s capacity. It is planned for future versions of Bluetooth and the coming IEEE802.15 to have higher data rates, but for the moment we must use what Bluetooth offers.

**Free Spectrum Availability:**

This can help the success of a final application due to the very fact that Bluetooth can and will be sold everywhere. No worries about versioning for different countries. SDP also offers features for describing services in various languages and URLs for the final executable, leaving for the developer the opportunity to announce applications for any final user.

Being free spectrum space, it also means that Bluetooth will have to share it with many other technologies, causing interference with itself and with the others. This should be also studied if the application will run in an area of Wi-Fi or other
technologies there must be some way of keeping their clash the further away as possible.

**Price and Market Situation:**

Bluetooth is still a “not-cheap” alternative to other technologies due to its overdue arrival to the markets. Though the technology involved in developing and making Bluetooth technology is significantly cheaper than its counterparts, it is still more expensive as the production hasn’t reached a sustainable rate and still not enough to amortise the investigation and development invested in this technology.

Predictions and trends say that the final chip price of a Bluetooth solution can reach $5 by the end of the year, although this statement has been repeated for the last three years.

Slow user adoption is also a reason for slow progress in the end price. The technology is mature enough to enter the market and hopefully this time it will be successful. In the last months many leading companies have started to include Bluetooth support in their products: cell phones, PC adapters, laptop computers and even PC motherboards are starting to have Bluetooth as a standard option. In the next two years it is predicted for a real boom, not only in Bluetooth hardware consumption but also in software kits and end applications that use this technology.

The leading Japanese NTT-DoCoMo telecom already uses Bluetooth access points instead of 3G cell networks for access to *imode* kiosks.

Microsoft already sells wireless mice and keyboards, and there are many more examples coming.

**Inquiry and Discovery:**

One of the most celebrated features of Bluetooth is its ability to detect or find devices in the vicinity. Even more so, it is also capable of finding out what type of features and services that device can provide. This leaves many open markets for Ubiquitous computing, as rival technologies are not able to do this automatically and rely on higher level protocols and applications. This can accelerate Bluetooth’s entrance to the mainstream market as developers see it as an ultimate technology as Ubiquitous Computing applications take-off.
Latency and connection times:

This is one of Bluetooth’s greatest problems. Due to fast frequency hopping and device clock skew, devices tend to desynchronise easily with a potential master. If on top of that, the device that we want to communicate with is in an advanced energy saving state this device must be woken up, and synchronized with. Afterwards typically authentication challenges are taken place. All this can make connection times very long. Up to 33 seconds it can take for a simple connection, the next table will show how both inquiry and paging effect the overall connection time:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Minimum Time (sec)</th>
<th>Average Time (sec)</th>
<th>Maximum Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inquiry</td>
<td>0.00125</td>
<td>3 - 5</td>
<td>10.24 – 30.72</td>
</tr>
<tr>
<td>Paging</td>
<td>0.0025</td>
<td>1.28</td>
<td>2.56</td>
</tr>
<tr>
<td>Total</td>
<td>0.00375</td>
<td>4.28 – 6.28</td>
<td>12.8 – 33.28</td>
</tr>
</tbody>
</table>

Table 7: Minimum and maximum connection times

An inquiry train must be repeated at least 256 times (2.56s duration), before the other train is used. Typically, in an error-free environment, three train switches must take place. This means that 10.24s could elapse unless the inquirer collects enough responses and determines to abort the procedure. However, during a 1.28s window, a slave on average responds four times, but on different frequencies and at different times. Minimum Inquiry would correspond to two slices and maximum is defined in the specification as abort time.

With Paging times we assume that mandatory paging scheme is used then the average time for connection should be 1.28s. The maximum time for connection is 2.56s.

All this lengthy time means that developers must take into account the possibility of actual connection times taking this long. In ubiquitous computing applications it must start from the very idea that users will not stand around forever waiting for their device to connect. After a couple of uses of the application, the user might get fed up and not use it again.

---

This will also limit mobility, as for the moment range is quite limited in some cases and there is no standard hand-over technique between access points. In other applications like Network access points it can be less relevant, as users don’t generally walk around to surf the net or read their emails. Much care should be taken in order to design a correct network-application combination.

**Voice and Audio:**

Bluetooth’s support of audio is quite a new and exciting idea. Although originally it has been designed for voice connections in a typical cell phone hands free set application, it can be extended to some other audio applications. But the end throughput is not enough for CD-quality music; therefore the developer must see what use of audio his application will use, and whether 64kbps CVSD voice is enough. If not and high quality is needed and bandwidth can be assured, the developer might consider using conventional ACL links to transmit highly compressed mp3-like audio streams in order to obtain it. This of course needs a lot of bandwidth and throughput studies around the model of the future application.
Interference Issues

As I have already mentioned, Bluetooth operates in the 2.4GHz band. This band is known as the Industrial Scientific and Medical (ISM) band. In the majority of countries around the world, this band is available from 2.40–2.4835GHz and thus allows Bluetooth and other wireless network technologies to be global. However being freely available makes Bluetooth vulnerable to interference from these technologies that use the ISM band:

- 802.11b/g
- Home RF
- Digital Enhanced Cordless Communications (DECT)
- Walkie-talkies

And also some household devices and others:

- Microwave ovens
- High-power sodium lights
- Thunderstorms
- Overhead cables
- GSM, CDMA
- Spark generators such as poorly suppressed engines

Also Bluetooth may have problems from signal fading due to distance or blockers such as walls, furniture, and human bodies.

As with any radio technology, Bluetooth technology is prone to interference from its co-residents in the ISM band and will produce interference to them. To achieve robustness to interference, the Bluetooth system utilizes a frequency-hopping scheme: Frequency Hopping Spread Spectrum (FHSS). Constantly hopping around the different radio channels allows packets that are affected by interference can be retransmitted on a different frequency, which will hopefully be interference free. Bluetooth radios hop in pseudo random sequences around all the available channels.

A Bluetooth FHSS system operating near an interfering signal can cope if a packet is hit by interference. The affected device simply retransmits the packet contents in the next
slot when it has moved to a different frequency which is no longer affected by interference.

This will impact on the throughput of an ACL link—the more interference, the more retransmissions, less real data rate. SCO data is not reliable, due to its intent of being in real time; retransmissions are not able to be performed, and so audio clarity becomes worse with any interference.

Therefore ACL is reliable at a price of lowering data throughput.

I have done some simple setups in order to see the effect of Bluetooth and IEEE802.11b (Wi-Fi) interference under simple FTP over TCP transmission.

The setup was to transmit via FTP the same file in and without presence of transmission by the other technology, and see the results. For Bluetooth I used a Bluetooth 3Com USB dongle and a 3Com Pc-Card. the Nokia Affix Bluetooth Stack for Linux and the PAN profile to be able to use IP. For Wi-Fi two Lucent Pc-Cards and a typical wireless LAN setup. The transmission was done in the same 10mx10m room and the devices are separate by 5 metres.

The next graph depicts the losses due to interference:

![Graph showing effects of interference on data rate between Wi-Fi and Bluetooth]

*Fig. 14 Effects of interference on data rate between Wi-Fi and Bluetooth*
As we can see Bluetooth’s throughput is halved from 480kbps to approximately 240kbps under interference from Wi-Fi. Wi-Fi seems to suffer less, losing only a tenth of its data rate, from ~5Mbps to ~4.5Mbps, when Bluetooth is close and transmitting.

In other setups, with 802.11 and Bluetooth devices on the same machine it was impossible to complete a set of transmissions as Wi-Fi cards started to produce errors, dropping packets, losing their channel configuration and refusing to transmit any data what so ever. While the Bluetooth device was transmitting at 100kbps, the Wi-Fi cards were malfunctioning and back up to 480kbps when the 802.11 cards stopped all together.

Most studies and papers in the area show similar results, for more information please read the chapter in the “Bluetooth Application Developer’s Guide” from Syngress.

Mobilian Corp. also has a set of interesting papers on the supposed coexistence between Bluetooth and Wi-Fi and possible hardware solutions.

Intersil Corp, does a similar study, but distinguishing between different types of Bluetooth packets and operation.

Therefore developers must take into account possible interference from other devices in the vicinity, whether its competing technologies or simple household appliances. Techniques can be used to try and minimise the effect Bluetooth can have on other technologies by power control using signal strength (RSSI) and adapting the transmission power accordingly.
Bluetooth Stack Implementations and Development Kits.

Many implementations of the Bluetooth stack are available, but not all of them are free or open source.

Proprietary development kits although very full featured and with excellent support are very expensive, limiting a lot of the home development that always have followed the open source community.

All Microsoft Windows implementations belong to these types of kits. Therefore due to lack of availability they will fall out of this project. Microsoft has announced future availability of its own development kit during the year 2003. At first it will be available for corporate developers and further on it will be released to the general public.

To name a few proprietary development kits for Windows and other popular Operating systems we have:

- Ericsson Bluetooth Application Toolkit.
  - Both software and hardware module testing and development kit.
- Digianswer Bluetooth Democard for Windows CE.
  - Software development kit for PDA’s running Windows CE 3.0.
- Digianswer Bluetooth Stack for Windows
- Socket Bluetooth Card SDK
  - SDK for PDA’s, PocketPC’s and TabletPC’s
- Meccel Bluetooth SDK
  - SDK for ANSI-C applications, designed for small embedded applications.
- Zucotto Whiteboard Bluetooth SDK
  - Java interfaced SDK for mobile equipment running J2ME.
- Extended Systems Bluetooth Software Kits
  - Bluetooth 1.1 compliant SDK for Windows and Windows CE.
- Palm Bluetooth SDK for Palm OS 4.0
  - Bluetooth-SIG v1.1 compliant SDK for Palm PDA’s
• Apple Bluetooth SDK for Mac OS X
• Cambridge Silicon Radio (CSR)
  o Casira: Embedded Systems Bluetooth Development Kit
  o Microsira: Platform for developing USB-connected host-side software
  o CompactSira: Compact flash card platform development for PCs and PDAs
  o BCHS BlueCore Host Software: Embedded systems Solution

Open source implementations are free with easy access to the API’s and source code. GPL licensing makes sure of always open future for that particular stack, guaranteeing many open source examples in the future. The main problem with the open-source variants is the lack of, or outdated documentation. Though support can be obtained through mailing lists, personal experience is that low level development questions remain unanswered and that there is a lot of a do-it-yourself attitude.

There are mainly four open-source stacks:
• The AXIS OpenBT Bluetooth Stack.
• IBM BlueDrekar Bluetooth Stack.
• Affix Bluetooth Protocol Stack for Linux
• BlueZ Bluetooth Stack for Linux

Each of the stacks, are readily downloadable and install with quite ease.

Throughout this chapter I will go through these popular, open-source and freely available Bluetooth stacks for the Linux Operating System.

The hardware used to test the stacks is:
• AMD K6-2 400Mhz standard PC running SuSE Linux 8.0. With 3Com CREB96 Bluetooth USB Dongle.
• Intel Celeron 500Mhz Acer DX-508 Laptop, running SuSE 8.0 and Windows 2000. With 3Com CREB96 Bluetooth PCMCIA card.
• In some cases I did some testing with a PocketPC running Windows CE and the 3COM Bluetooth drivers using the PC-card from the laptop.
Axis OpenBT.\(^8\)

 Originally developed at AXIS Communications, this Protocol Stack has been the first to arrive to the Open-Source movement, allowing early development of Bluetooth applications in the Linux scene.

 It has been developed for AXIS own hardware products, such as AXIS ETRAX Boards and AXIS Bluetooth access points, and extended to personal computers running Linux. It has a GPL license and is totally open-source.

 Its main features are:

- Not Certified as a Bluetooth Component, but compliant to the Bluetooth-SIG Specification 1.1.
- Used in Bluetooth Certified products compliant with the 1.1 Bluetooth Specification
- Basic L2CAP, SDP and RFCOMM support.
- The AXIS Bluetooth stack also supports, some Bluetooth 1.1 Profiles:
  - AP: LAN Access Profile
  - DUN: Dial-Up Networking Profile
  - SPP: Serial Port Profile
  - PAN Profile is currently being worked on.
- Support for UART Bluetooth hardware.
- Use of Bluetooth UART as Serial ports to create TTY’s, and line disciplines to deal with connection.
- XML based motor for SDP attributes database.
- Platform independent: x86, ARM, MIPS, PowerPC

 It does, however, have many flaws:

- Hardly any documentation. The little that’s available is outdated or submitted by other users.
- No use of RFCOMM other than PPP.
- No dynamic SDP registration, no SDP parser functions or libraries.

• No SCO support.
• The whole development process seems to have come gradually to a stop. There is much less movement on the CVS repositories and the last package available is dated from October 2001. No user development either.
• USB Bluetooth devices are supposed to be supported but no documentation whatsoever indicates that this statement is true. Questions that have asked this in the mailing list have not been answered. In the source code, options to activate USB have many commented lines indicated that USB is not yet supported.
• Many compilation problems solved by users. ⁹

All this, along with the fact that I did not have any UART based devices to play with made me abandon testing with this Stack. Most developers are tuning to Affix or BlueZ anyway, due to the fact that these stacks have more features and better support.

IBM BlueDrekar

This Protocol Stack has been developed by IBM. Its original purpose was to create a reference implementation for the UART transport for other Bluetooth-SIG compliant implementations. BlueDrekar only works for Ericsson UART modules. The Bluetooth protocol stack is proprietary and accessible only through a number of functions.

A part from that, the actual stack is not available due to expiry of the evaluation version in April 2002. IBM has stated that they do not have any more plans for the stack. Therefore I was also unable to test it.

The UART Transport driver source is available but without the Bluetooth stack it is useless.

The list of features did seem quite promising, with a complete implementation of the Specification 1.0b with: HCI, L2CAP, SDP, SDP server and dynamic SDP registration, RFCOMM, SCO and, through the UART transport driver, HCI-UART.

⁹Frank Kargl. http://medien.informatik.uni-ulm.de/~frank/bluetooth/bluetooth-howto.html#openbt
Nokia Affix.\(^\text{10}\)

Affix is a Bluetooth Protocol Stack for Linux developed by Nokia Research Center in Helsinki and released under GPL licensing.

Some of Affix’s features are:

- **HCI, L2CAP, RFCOMM, SDP, SCO**
- Modular implementation.
- Socket interface to L2CAP, SCO and RFCOMM protocols.
- Bluetooth module interface independence.
- SMP safe.
- Multiple Bluetooth devices support.
- Platform independent: i386, ARM, PowerPC, Sparc, and most platforms capable of running Linux.
- Affix also supports the following Bluetooth Profiles:
  - General Access Profile
  - Service Discovery Profile
  - Serial Port Profile
  - Dial-Up Networking Profile
  - LAN Access Profile
  - OBEX Object Push Profile
  - OBEX File Transfer Profile
  - PAN Profile
- Affix is also hardware independent.
- Brilliant documentation. This is something new as when I started to play with Affix there was even less documentation than BlueZ.
- Ethereal Plug-in

It does have, however, one major flaw, that has both people and developers in favor and against, that keeps it away from being the most popular Bluetooth stack for

\(^{10}\) [http://affix.sourceforge.net](http://affix.sourceforge.net)
Linux. Although it is Bluetooth Compliant, it is not as close to the Specification as other stacks, and understands Bluetooth as a network device which is not entirely correct as it is more similar to a USB device. This may make Affix attractive for some that know the Linux Networking queuing architecture, but it really doesn’t correspond to the Specification. Also the naming in the functions and API sometimes is confusing and over-simplified. I suppose Affix are trying to claim many developers that are used to network programming but do not want to go into detail on how Bluetooth works “inside”.

Affix is quite monolithic and only consists of the following packages:

- affix-kernel
- affix

Affix-kernel provides kernel modules, which implements protocols core protocols and Bluetooth device drivers. Kernel modules can be used separately from the kernel or can be linked statically into the kernel. The Affix package provides control tools, libraries, and server daemons.

The two packages are very simple to install. Let’s go through the basic steps:

The first to be installed must be the **Affix-kernel** package. The only dependencies it has are those related to kernel option being turned on, most of the out-of-the-box distribution kernels will have them by default. These options are:

- USB support in case of using USB Bluetooth devices.
- PCMCIA support for Pc-card Bluetooth devices.
- Devfs support, this is a new kernel option that allows a clear hierarchy of system devices. It also allows referring to devices other than through major and minor numbers. It is not an obliged option as many systems will not stand the option just yet, but it is recommended.
- Hotplug support to automatically load USB drivers on demand
• *Modutils* for auto-loading modules into the kernel from aliases other than `/etc/modules.conf`, also optional but will require manual editing of configuration files.

Once the dependencies are clear, it’s time to install the kernel package. The compressed package file should be copied to a source directory of some type like `/usr/src` or any other directory. Root permissions are needed in order for a correct installation as kernel modules are not allowed to be installed by normal users.

Copy to destination directory:

```bash
# cp affix-kernel-1.2.6.tar.gz /usr/src
# cd /usr/src
```

Unpacking:

```bash
# tar xfvz affix-kernel-1.2.6.tar.gz
# cd affix-kernel-1.2.6
```

Configuring and select the user options, kernel version, and other options:

```bash
# make config
```

Compile and install:

```bash
# make all
# make install
```

If *modutils* are installed, the kernel package installation would stop here, otherwise it is necessary for the user to edit the `/etc/modules.conf` file in order to tell the kernel what module to load when protocol usage is detected. These lines are simple aliases to the correct module to be added at the end of the file:

```bash
alias net-pf-27 affix
alias char-major-60 affix_rfcomm
```

Then to correct the current modules configuration:

```bash
# depmod -a
```
Depending on what system you have, especially if its either old or if you’ve been testing other Bluetooth stacks and using PCMCIA cards you will probably have to change the `/etc/pcmcia/config` file as new cards will be used, or to change the driver alias to aim at Affix’s own drivers. Usually reinstalling Affix will do this, but it’s possible for kernel modules and drivers from other stacks to remain in the system loading themselves on detection of devices being added to the system, deleting them will fix the problem. If the PCMCIA configuration is changed, the card manager must be restarted:

```
# killall -HUP cardmgr
```

Once installed on plugging-in Bluetooth devices to the system, the user should hear a “beep”. That is the card manager or hotplug system informing the user of the correct detection and loading of the modules. If not, it is possible for the user to manually add/change the modules to the kernel with `modprobe/insmod/lsmod`.

The modules available to install are:

- Basic Bluetooth stack support: `affix.o`
- Drivers and transport layers:
  - `affix_bt3c_cs.o` (3Com pc-cards)
  - `affix_usb.o` (USB dongles and adapters)
  - `affix_uart.o` (Serial adapters)
  - `affix_uart_cs.o` (Serial pc-cards)
- Profile and Other Bluetooth Layers
  - `affix_pan.o` (PAN Profile support)
  - `affix_rfcomm.o` (RFCOMM support)

Only install the necessary modules as more modules will just occupy memory and have an adverse effect as PAN connections disables inquiry and discovery.

After installing correctly the kernel package we will have Bluetooth support in our system, but still we are unable to use the devices. For that we must install and use the affix package. This package contains all the “user land” device control applications, libraries, scripts and programs to get the Bluetooth device up and running.
The steps for installing this package are very similar to the ones taken for the kernel package.

The dependencies that the user package has are:

- Affix-kernel package installed
- Open-OBEX installed if OBEX use is intended
- Python if use of the PIN code helper application is intended

Then:

Copy the package to a destination directory for unpacking:

```bash
# cp affix-1.2.6.tgz /usr/src
# cd /usr/src
# tar xfvz affix-1.2.6.tgz
# cd affix-1.2.6
```

Configuring and compiling:

```bash
# ./configure
# make all
# make install
```

That’s it! Now the system is ready to be played with. The Affix user package contains many interesting applications to get the device “to do things”. Let’s take a look at the applications we have:

- btctl: Main Bluetooth device control program
- btsrv: Main Bluetooth server application, SDP register and server and profile manager.
- btobex: OBEX server
- btftp: OBEX client
- btmodem: Modem emulator for DUN over LAN

The most interesting programs are the first two. Btctl allows us to totally control the device. For my own taste it’s too full featured and I must admit I prefer 10 small
well developed programs instead of one huge program doing 10 things, and not always well.

For example `btctl bt0 up` initializes the device and `btctl bt0 down` would bring the interface down. With the device up, if we were to execute `ifconfig`, the standard Linux call for viewing the current active interfaces we would see the BT0 net interface. 

Next I will go through the basic commands to allow us use the devices correctly.

```sh
# btctl ping <Bluetooth_device_address> 1000
Send a L2CAP echo packet of 1000 bytes to the Bluetooth device.

# btctl addpin pin-number
# btctl rmpin
Give the local device a PIN number for authentication. Remove it.

# btctl pair <address>
# btctl unbond <address>
Pair and unbound the local device to <address>, this will create a connection with authentication. It is necessary in some applications like cell-phones and headsets.

Some HCI commands like inquiry, discovery and local device parameters can be done with the next commands:

```sh
# btctl inquiry <length>
# btctl discovery <length>
These commands should inquire for all the discoverable devices in the vicinity for <length> seconds. Discovery will also return the remote name. Certainly the most interesting feature that Bluetooth offers.

# btctl role <allow|deny> <master|slave>
Allow role-switches during a connection with another Bluetooth device. It is essential for PAN/DUN/LAP and other networking profiles, as the peer server should be always master.
# btctl class <class>
Set the Device Class, this is important if offering a determined service as clients will inquire and filter out the servers they are looking for. In the Bluetooth-SIG website there are documents that define the assigned numbers to a service or group of services. The device should take a Class depending on the service it will offer.

# btctl pkt_type <pkt_type>
Set the available packet types. This will be taken into account on establishing a connection with other devices. Some packet types have a better average data rate than others, but usually Bluetooth devices try to use the most efficient packet type. It is still useful to study effects of packet types on connections though.

# btctl scan <+|-|><disc|conn>
To allow/unable to be discovered (inquiry scan) and/or connected to (page scan)

There are also RFCOMM commands that establish and destroy connections using the RFCOMM streams, binding them to virtual Terminals

# btctl connect <address> <service_type>
Connect to <address> and establish a <service_type> (LAN, DUN or Headset), a terminal will be bound to the connection.

# btctl status
To see what connections are established and to what terminal they are bound.

# btctl disconnect <line>
Disconnect and unbind the terminal <line>

Btctl also has SDP commands, to view services offered by other devices, and local services if local address is used:

# btctl browse <address>

To browse available services on a device with address <address>. It shows service name, provider name, description, protocol info (ports) and profile info if any.

# btctl search <address>
To search for all well known services on that server instead of showing everything.

OBEX commands also exist in order to get OBEX working under Bluetooth, enabling file and vCard push. The OBEX ftp command is practically a FTP, most commands are there and works in a very similar fashion.

```
# btctl ftp
```

To open an OBEX-FTP session with interactive command line. Once in, a connection must be opened to a server

```
# btctl open <address>
```

Btctl will do a SDP connection unless a channel number is provided.

Now most ftp commands can be used: close, ls, put, get, push, (using push profile), rm, cd, mkdir. Also, ls, push, get, put and rm can also work without creating connection if address and channel are supplied.

The PAN profile is also supported by this mega-program; it can establish and destroy connections and set the role for the PAN network. With PAN we can create small IP networks and create access points or bridges to other networks like corporate LAN’s. These commands are:

```
# btctl paninit [panu|nap|gn] <device>
```

To initialize a PAN interface (pan0, pan1, ..) and assign it role: panu, nap or gn.

```
# btctl panstop
```

To part from the PAN and release the network interface.

PAN features are one of the most interesting, as we can quickly establish an IP based network between Bluetooth devices. This is very useful for reuse of many TCP/IP programs. However due to fact that Bluetooth transmits on the air interface and is very prone to packet loss and interference, application developers must take care on creating applications, as it is frequent for many lost connections and timeouts.

However, PAN also offers the possibility of creating level 2 and level 3 bridging or routing between the PAN and other networks.
For the creation of a PAN with a NAP (Network access point) to give access we have to create connections with `btctl paninit`, one device would be the NAP and must have a second network interface, while the others (up to a maximum of 7) would be PANU’s (PAN users).

So the Access Point would execute:

```
# btctl paninit nap bt0
```

This would create an interface PAN0, PAN, etc. If there has been any problem, please check with `lsmod` to see if the “affix_pan.o” module has been loaded, if not load it manually with `modprobe` and try again.

Now as the interface is working, the NAP might want work as a router (level 3) for this its only necessary for the NAP to forward packets and act as a gateway for the PANU’s. If the PANU’s will have private IP addresses, it might be interesting to execute a DHCP server to give automatically IP addresses to the PANU’s. Also IP Masquerading might be needed in order to hide the private IP addresses to the rest of the public network. For IP Masquerading to work it must be activated in the Linux kernel, needing in many cases to re-compile the kernel.

Any firewall should be shut off or reconfigured to allow packets through, and the routing tables should be checked (`route –n`) to see if everything is correct.

If level two forwarding is required, the NAP should act as a bridge. This is quite simple if the NAP machine has installed the Bridge-Utils package\(^\text{11}\). What is also required is 802.1d Ethernet Bridging support in the Linux kernel which is not on by default (recompile). This package has a utility that allows the creation of a bridge between network interfaces in a system. This utility is `brctl`.

What the first should be done is to create a virtual bridge device.

```
# brctl addbr br0
```

That is, add a bridge called `br0`. Next the interfaces used to forward the level two frames should be added to the bridge:

```
# brctl addif br0 eth0
# brctl addif br0 pan0
```

\(^\text{11}\) Bridge-Utils: http://bridge.sourceforge.net
Once added the bridge interface should be brought up

```
# ifconfig br0 up
```

Please note that it has the same IP address as eth0. Now the default gateway should be routed via the bridge interface, as internally the routing would be done through the bridge and not through the interfaces (although that is what the bridge does). Therefore with the `route` command the default gateway should be changed to be the bridge, removing any that was used before.

```
# route add default gw br0
```

Next we should allow the traffic coming through the Bluetooth/NAP interface by bringing it up:

```
# ifconfig pan0 up
```

Now the NAP, should be listening to any PANU wanting to connect, in order for the routing to work properly, the NAP should always be master, but by default the establisher of a connection is the master, so any PANU wanting to connect would become master and would have problems receiving frames coming from the NAP-side network. To solve this role switching must always be allowed, so the NAP would become master when a PANU connects to it:

```
# btctl role allow master
```

Now we have a functional bridge, what is needed is a real PAN network by adding PANU’s to it. Each PANU should allow role switching to slave and connect to the NAP:

```
# btctl paninit panu bt0
```

When the devices have connected, the users should hear a **beep** from the computer as the connection has been made. Once the connection is made, both sides should have an IP address, if not assign manually with:

```
# ifconfig pan0 10.0.0.x
```

Establishing a PAN-Group Network is much easier. This type of network is self-contained and doesn’t have any external networks to bridge/route to. The only
difference is that instead of a NAP we have a GN and the `paninit` command would look like this:

```
# btctl paninit gn bt0
```

Other very useful services are the DUN/LAN profiles, this can allow us to use Bluetooth devices as modems (Bluetooth enabled phone) or access points to corporate LAN (the difference here with PAN is the use of RFCOMM). These are usually the first profiles to be implemented on Bluetooth stacks; therefore I tried out interconnecting Affix with a Windows 2000 machine and an iPAQ PDA.

The first combination I tried was a LAN access from the Linux Laptop to the Linux Desktop. It took me quite a while to figure out how to establish the connection as at the time Affix hardly had any documentation what so ever. For the use of profiles it is necessary to use the `btsrv` utility that enables the use of SDP and coordinates profiles calling the correct scripts.

For a LAN access using Affix we need to configure both server and client. For the server we need to:

- Call the `btsrv` mega-daemon.
  ```
  # btsrv –d
  ```

From the client we need to establish a connection and create a PPP connection over the RFCOMM serial line TTY with:

```
# btctl connect <Bluetooth address> <TTY line number> LAN
# pppd bty<TTY line number>
```

The server should automatically accept the connection and call the `pppd` daemon to readily receive connections. If the Affix scripts were to fail this can be done manually with:

```
# pppd bty<TTY line number>
```

If it’s the case of scripts failing to call the `pppd` daemon, it can be fixed by adding a line to the `/etc/affix/btsrv.conf` configuration file:
“exec /usr/bin/btmodem”

After this the client should have an IP address and access to the LAN. The IP addresses can be assigned automatically via DHCP, through PPP configuration scripts or manually.

To quit the LAN connection the client can simply do:

# killall pppd
# btctl disconnect <TTY line number>

Please note that some versions of pppd have problems with devfs, as they require the old /dev/ structure when using Bluetooth terminal lines. This is because older versions do not allow symbolic name referencing that devfs offers.

DUN connections are done in a very similar fashion. DUN is defined for the use of Bluetooth devices as modems; this can be useful for internet connections using Bluetooth cell phones and PDAs or Laptops. The steps to create such connection are similar to the ones using normal modem dial-up PPP with the exception that first we must create a connection to our “modem”.

The first thing is to create a PPP account. In Linux this can be done in various ways and depends of the utilities included from each Linux distribution. Typically it can be done with the pppconfig utility, but in the SuSE distribution I was using it didn’t come. Instead I used the YaST2 configuration program. In the configuration the ISP name and telephone number should be used just like when configuring for normal modem dial-up.

After setting up the PPP account, we must create a Bluetooth connection in order to have a serial line for pppd to work on.

As before, this is:

# btctl connect <address> DUN

All needed now is to call the pppd account with the serial line just created.

# pppd <ppp_account> /dev/btty<line number>
That’s it! Connection should be established and access to the Internet service provided by the ISP.

As we can see DUN and LAN are practically identical and really can be considered interchangeable, as DUN can be used with a peer gateway and LAN to access an ISP.

For connecting Windows2000 machines to a Linux LAN server, it is very simple if compared to the steps before. On W2000 we have the 3COM Bluetooth Stack, driver and 3Com Bluetooth Connection Manager.

On the Linux machine all is needed is:

```
# btsrv –d
```

On the W2000 laptop, from the Connection Manager:

→ Create link to device.
→ Create Connection to Network → Direct line, Internet and Modems → Choose “3COM Bluetooth LAN modem” as modem, any telephone number, and rest of the option by default. → Connect!

That should be it. Please do not use any IP addresses for the W2000 machine as the Linux machine should assign one. I must say it did take me a few times to get it going as not always does the connection is established.

For connecting a PocketPC iPAQ, the steps are exactly the same. Mainly because both operating systems on the client side are Windows variants, and due to the driver and Connection Manager are practically identical.

One remark on the use of DUN/LAN profiles, it seems that RFCOMM introduces an important overhead on throughput. The TCP throughput for both Windows and Linux clients to the LAN/DUN server is quite smaller if compared to the data rate obtained using the PAN profile. In both cases using an ftp transmission we have that with DUN/LAN the data rate is approximately 340-350kbps on Windows clients and 370-380kbps on the Linux client, while with the PAN profile the average
data rate is 470-490kbps. This is certainly an aspect to take into account on choosing what profile is better for LAN access. For DUN we do not have an alternative.
BlueZ: The Linux Open Source Kernel Bluetooth Support

BlueZ is the Official Linux Bluetooth protocol stack for the Linux Kernel. It was originally developed at Qualcomm and actually is released to the general public under GPL license.

BlueZ is already part of the official kernel 2.4 tree and if you have a recent Linux distribution your standard kernel will probably have it built in or an option to do so.

Some of BlueZ’s features are:

- Support for core Bluetooth layers and protocols.
- It is flexible, efficient and modular implementation.
- Support for multiple Bluetooth devices
- Multithreaded data processing
- Hardware abstraction
- Standard socket interface to all layers
- PSM level security support
- Multiplatform: x86 (single and multiprocessor), SPARC, ARM, DragonBall
- Works on all Linux Distributions and currently being ported to *BSD
- Immense amount of hardware devices supported
- It has HCI UART, USB, PCMCIA and Virtual HCI drivers
- Support for L2CAP, SCO, RFCOMM and SDP
- It also supports the next list of profiles:
  - GAP
  - DUN
  - LAN
  - SPP
  - PAN
  - Head-set.
  - OBEX Object and File Push Profile
  - Third party utilities provide support for Microsoft Bluetooth Mouse and Keyboard, HID (Human Input Device) and printer Profile

http://bluez.sourceforge.net
• It comes with a Bluetooth Device (HCI) emulator, a Bluetooth Device Activity and Transmission Scanner and Configuration and testing utilities
• Very active development with a considerable rate of bug-fixing and more features in periodic releases of the Stack.
• Good collaborative mailing lists. Brilliant developers from the Linux Kernel and wireless networking world help with the development and support.
• Very close to the Bluetooth-SIG 1.1 standard. Naming to functions and data structures are practically identical.

BlueZ’s main flaw is the lack of documentation. Although many users have tried to create their own documents, none are as fully featured as the Affix. This can be solved by reading the archives of the continuous questions and answers of the BlueZ mailing list, but it is really a hassle.

BlueZ is distributed in a set of packages, though the main core protocol implementation relies in the kernel distribution. Updated patches from Marcel Holtmann’s website are available for older kernels and newer kernels that require extra functionality. The kernel support can be built as modules and loaded into memory as needed or fully integrated into the kernel, the latter giving extra performance. The BlueZ core takes care of making HCI-devices, the L2cap and LMP/LC protocols.

For 2.4.19 kernels it is possible to use the bluez-kernel-2.3 package, it contains the necessary to build the kernel modules needed for a fully functional Bluetooth stack without having to rebuild the kernel.

Apart from the kernel support, BlueZ distributes a series of packages depending on the final users needs; these are (with their current versions):

- bluez-libs-2.3
- bluez-utils-2.2
- bluez-sdp-1.0rc3
- bluez-pan-1.1pre4
- bluez-heidump-1.5

---

13 [http://www.holtmann.org/linux/kernel/](http://www.holtmann.org/linux/kernel/)  Marcel Holtmann’s Kernel Patches
• bluez-hciemu-1.0
• bluez-bluefw-0.9

The BlueZ-Libs is a package containing all the necessary libraries for Bluetooth user application development. These will also be needed by other BlueZ packages and applications that will dynamically link to the libraries.

The BlueZ-Utils package contains a series of control applications for the Bluetooth devices. They are needed in order to bring up Bluetooth interfaces, perform inquiries and general communications testing.

BlueZ-SDP contains exactly what its name indicates, all the SDP functionality: libraries, tools and the `sdpd` SDP server.

BlueZ-PAN package contains the necessary scripts, daemons and programs for the DUN, LAN and BNEP-PAN profiles.

BlueZ-HCIDump contains the HCI commands `sniffer`. It is very useful for debugging and studying the general working of the Bluetooth devices.

BlueZ-Emu is a Bluetooth HCI emulator. It allows developers and users to try out programs without having to buy Bluetooth devices.

And finally BlueZ-FW is a package with the firmware of various Bluetooth devices like the Broadcom Bluetooth PC-card.

As we can see BlueZ is quite a bit more modular and less monolithic than the Affix Bluetooth Stack. This has its pros and cons, a big monolithic package is very simple to install but will take much more on disk and memory, where as various small packages are more confusing for an end user but much more flexible and users will tend to install only the necessary.

The basic steps for installing the BlueZ Stack are very similar to the ones taken for the Affix Stack. For the best performance and up to date (less bugs) features is to install the new vanilla Linux kernel 2.4.20\(^\text{14}\) (or newer) and apply the kernel patches from Marcel Holtmann.

\(^{14}\) http://www.kernel.org/pub/linux/kernel/v2.4/linux-2.4.20.tar.gz
To apply a kernel patch it is necessary for the new kernel and patch to be downloaded, copied to /usr/src/, uncompressed and linked against /usr/src/linux

```
# cd /usr/src
# wget http://www.kernel.org/pub/linux/kernel/v2.4/linux-2.4.20.tar.gz
# wget http://www.holtmann.org/linux/kernel/patch-2.4.20-mh5.gz
# tar xfvz linux-2.4.20.tar.gz
# gunzip patch-2.4.20-mh5
# ln -s linux-2.4.20/ linux
```

It is now needed to apply the patch:

```
# patch -p0 < patch-2.4.20-mh5
```

Everything should be OK. Now it’s time to configure and compile the Linux kernel. First stop to make the configuration file. It is recommended to make the Bluetooth support on modules, so we can load them if needed. To make the configuration we have to follow the steps indicated in the documentation of the Kernel Sources, but basically to do it easily, from a Desktop terminal (X-Windows/KDE/GNOME) do:

```
# make xconfig
```

Choose the options corresponding to the hardware of the machine, save and exit. NOTE: make sure in USB options that Bluetooth support is NOT ON. This is an old Bluetooth USB driver that clashes with the BlueZ driver on detecting the addition of a USB device.

After configuring the kernel, it is necessary to calculate the necessary dependencies, compile and install:

```
# make dep
# make bzImage
# cp./boot/bzImage /boot
```

Now we have a brand new kernel, but the system doesn’t know how to use it unless we configure the boot loader to use it. For that we should add a entry in the /etc/lilo.conf file for the new kernel and execute Lilo so the system can install the entries on the boot sector of the main hard disk.

```
# /sbin/lilo -v
```
If everything is correct and no error messages have been printed on screen:

```bash
# reboot
```

After rebooting and choosing the new kernel from the Lilo menu we have a fully functional Bluetooth supported system.

The other way round if we have a 2.4.19 kernel and not particularly worried about not having cutting-edge Bluetooth support we can use the BlueZ-kernel package from the BlueZ website. Please note that no support is given towards the kernel package as it is considered deprecated.

The kernel package can be downloaded with the rest of packages and copied to `/usr/src`, from there we can uncompress each one and install them with:

```bash
# cd <BlueZ package>
# ./configure
# make all
# make install
```

This must be done for each package in the following order:

1. BlueZ-kernel (if used)
2. BlueZ-libs
3. BlueZ-utils
4. BlueZ-sdp
5. BlueZ-pan
6. BlueZ-hcidump

It is possible for the system to ask for the package libglibx.x-devel when configuring the Libraries package. Any Distribution software maintainer should be able to install (download if needed) the appropriate package.

There are a couple of lines we must add to the `/etc/modules.conf` for the kernel modules to be loaded properly when new services/devices are used.

```bash
alias net-pf-31 bluez
alias bt-proto-0 l2cap
alias bt-proto-2 sco
alias bt-proto-3 rfcomm
```
This should be it, to make sure everything is ok we can repeat a couple of calls that make install should do in order to keep libraries and modules well registered.

```
# ldconfig –v
# depmod –a
```

The BlueZ-kernel package has a habit of not using the correct directory for installing the kernel modules. If your kernel name/version is other than 2.4.19 (like 2.4.19-mdk or 2.4.19-4GB, this you can find out with `uname –a`) you will probably have the new BlueZ modules under `/lib/modules/2.4.19` instead of the real kernel module directory. It has a simple solution by copying the modules from one directory to another and reconfiguring the system to accept the modules:

```
# cp –r /lib/modules/2.4.19/* /lib/modules/<real name>/
# depmod -a
```

We now should have our system completely prepared to use and enjoy Bluetooth. To check it we can insert a USB dongle or PC-card and see the results. Once inserted we should hear a beep or two coming from the computer saying that the device has been loaded. We can check for problems by viewing the system logger with:

```
# tail -30f /var/log/messages
```

If the device is a USB dongle and using `lsmod` we see that bluetooth.o has been inserted into the kernel, means that we have the old Bluetooth USB driver. With it the device will not work and we must remove the module from the `/lib/modules/<kernel>` directory.

Now it is time to play around with what BlueZ offers. In the utilities package we have a good set of programs to make our Bluetooth devices work, with many, many
features. I will go over the most important and a brief introduction to those that I consider to fall out of the purpose of this document.

The most important tool is `hciconfig`. This tool is similar to the `btctl` of the `Affix` stack and allows controlling completely the Bluetooth device.

Its main call for the first Bluetooth interface is:

```bash
# hciconfig –a hci0 <option>
```

The `–a` option is to print every detail (verbose) about the device.

This acts like the `ifconfig` network command in Linux. `ifconfig` doesn’t work as the BlueZ team consider that Bluetooth is not a network device and therefore should not be added in the Linux networking architecture and queues. My own opinion is that BlueZ are right, and that the `Affix` approach, although not incorrect, is not the most desirable.

`<option>`:

- `up/down/reset`: Bring up, down and reset the `hci0` interface.
- `auth/noauth`: Enables and disables authentication.
- `encrypt/noencrypt`: Enables and disables encryption
- `piscan/noscan`: Enables/disables both, page scan and inquiry scan modes
- `iscan/pscan`: Set inquiry scan/page scan mode only
- `ptype <type>`: Sets packet type: DH1, DH3, DH5, DM1, DM3 and DM5
- `lm <mode>`: Get/Set default link mode, this can be to force the device to go into Hold mode, or perform a roles switch.
- `lp <policy>`: Get/Set default link policy. This will define the devices behaviour in connection latencies, peak bandwidth, periodic inquiries, etc... It is not generally a good idea to play around with it.
- `features`: Shows the device features, link support for scatternets and other, these values are in hexadecimal, and defined in the Bluetooth Specification
- `name <name>`: Get or Set a local symbolic name for the device. This can be good for naming devices “James’ Laptop”, “Robert’s Headset”, etc…
- `class <class>`: Get/Set class of device, this can be interesting to define as the Class of device defines shortly the type of services offered by the
Device prior to SDP. This can help peer devices short-circuit their search so it will not perform SDP on all devices.

- **version**: Display the device’s version information, chipset manufacturer, etc…
- **revision**: Display revision information, like version but much more detailed.
- **voice**: Get/Set voice settings, like channel configuration for SCO connections.
- **inqparms/pageparms**: Get/Set inquiry/page scan window and interval.
- **pageto**: Get/Set page timeout.
- **aclmtu/scomtu <mtu> <packet number>**: Set ACL/SCO MTU and number of packets.

That’s what `hciconfig` has to offer, as we can see there are many commands and a lot of functionality. We can control most of the devices features and its configuration when linked from the command line. One aspect I think that is both positive for a developer but overwhelming for the end user. Therefore it’s generally recommended for it to be an Administrator tool only. The interface can be set up without normal user intervention using scripts, leaving this tool for advanced users.

Another tool we can use is `hcitool`. This tool is especially good for allowing us to use the main features of the device like inquiry and connecting.

Its call from the command line is:

```
# hciconfig <option>
```

With `<option>` being:

- **dev**: Display list of local Bluetooth devices.
- **inq**: Inquire remote devices and return their details: Bluetooth Device Address, Clock offset and Class of Device.
- **scan**: Scan for remote devices.
- **name**: Get name from remote device.
- **info**: Get information from remote device, its address, name, protocol stack version, manufacturer, hardware chipset and features.
- **cmd:** Allow the user to submit arbitrary HCI commands. These commands have to be written in hexadecimal as stated in the Bluetooth Specification.

- **con/cc/dc:** Display connections, connect to device and disconnect from it.

- **cpt:** Change connection packet type. Like the command in the *hciconfig* command but during an active connection.

- **rssi:** Display connection RSSI, Received Signal Strength Indicator. This can be useful for detecting better device to connect to in case of roaming between devices, or for simply seeing how this value varies. It seems however that not all devices calculate it the same way which can lead to confusion.

- **lq:** Display link quality. This value is a vendor calculated indicator for link quality. Being vendor specific it loses any application value.

Other commands offered in the Utilities package are also very interesting:

- **l2ping:** Sends an L2CAP echo request to a peer device.

- **l2test:** Is a tool that allows transmission of data (recv’s and sends) using L2CAP sockets between two devices. It allows changing MTU’s and is good to evaluate bandwidth between devices.

- **rfcomm:** This utility allows to bind, create, disconnect, and show RFCOMM connections. This is necessary in some cases prior to DUN/LAN or other synchronization tools.

- **scotest:** Allows the user to try out SCO connections between two devices. Scotest allows playing audio files as an audio source prior sending it to the peer device.

- **hstest/hsplay/hsmicro:** similar to scotest, but for the testing headset devices.

- **rctest:** Is similar to l2test but using the RFCOMM layer instead of the L2CAP. RFCOMM also offers a socket interface to the layer. The transmission is stream based and is more reliable than L2CAP.

- **hciattach:** Is a tool needed for binding UART Bluetooth devices to a terminal. Without it the Bluetooth device will simply not work.
• /sbin/hcid: is the HCI daemon, it is usually called by the modules and drivers and it establishes a base configuration for the Bluetooth device. The configuration can be modified via the /etc/hcid.conf file.
• bluepin: Python program that gives a graphical edit line for introducing PIN numbers needed on establishing connections.

We also have additional tools and utilities from other packages:

From the BlueZ-hcidump package we have hcidump that offers a good sniffer for viewing traffic between the local device and remote devices. This can be very useful for debugging our own code seeing the remote devices responses during link setup.

From the BlueZ-SDP package we have a couple of utilities that are necessary for using profiles, whether it’s the PAN package or our own profile and application development. It also has libraries to be included by programs using SDP services; these should be installed to their normal library location. The package contains the SDP daemon, sdpd, which allows a server to announce its own services, adding and deleting them as needed. It also allows peer devices browse the SDP database looking for services it may need. This daemon stays in the background, unless indicated by the –n option.

To use the daemon, we must use another client utility, sdptool, which allows us to register services on it, and view services offered by remote devices. Although each service application should register its own services automatically, sdptool allows debugging and is a good source for programmers to start from.

# sdptool <option>

The main options that sdptool offers are:

# sdptool search --bdaddr bdaddr --tree <service>

Search for a service <service> on a device with <bdaddr>. The service can either be a mnemonic or a hexadecimal value for it defined by the Bluetooth Specification. The tree option is for showing the values in a tree format. This only returns the attributes pertaining to that particular service, if found.
For example, to find a DUN service on a cell-phone:

```
# sdptool search --service=DUN <phone bd address>
```

Browse all available services on the device with address `<bdaddr>` and view all well known attributes.

```
# sdptool browse --tree <bdaddr>
# sdptool add [--channel=<channel number>] service
# sdptool del <recordhandle>
```

Add a service for testing purposes on the specified channel number. This is not useful for end implementation as the services added are not really available unless the service daemon is started.

Delete service identified by the record handle. The record handle can be viewed with a browse on the local device and identifies the instance of that service in the database.

```
# sdptool setattr and sdptool setseq
```

Add attributes/Attribute sequence to a SDP record. Again, it is only valid for testing and seeing the example source code for own development.

In the BlueZ-PAN package we have all the necessary to use three of the most used profiles in Bluetooth, DUN, LAN and PAN. These profiles allow users to use their Bluetooth devices as network cards or modems to be able to access remote LANs. Once the package is installed we can play around with the tools.

The first profile I recommend to try out is the PAN profile, as it is the easiest to use and one of the most popular due to its performance and versatility. Most steps are very similar to the ones taken in the Affix stack.

As we know the PAN profile specifies three roles that can exist in a PAN: NAP, GN and up to 7 PANUs. Both NAP and GN are service provides that coordinate the traffic in the PAN. NAP also is an access point to another network. The PANUs on the other hand are the clients or users of the PAN Service.
To establish a PAN with BlueZ we must start making sure that it is correctly configured. Please check that `/etc/modules.conf` has the next line:

```
alias bt-proto-4 bnep
```

If not add it, and run `depmod -a` to reconfigure the modules configuration.

It is also possible to load manually the module with `modprobe bnep`.

This is essential for the loading of the BNEP module. BNEP is an encapsulation protocol defined in the PAN specification, which allows Ethernet frames to work on Bluetooth links, making easy and transparent access to Ethernet and other networks. This module should auto-load on the execution of the PAN daemon `pand`.

The PAN daemon has many options as it is used by both clients and servers. Its options are:

- `--show/--list/-l` Show active PAN connections
- `--listen/-s` Listen for PAN connections which is usually used by servers
- `--connect/-c <bdaddr>` Create PAN connection to a remote device
- `--search/-Q[duration]` Search for a maximum of duration seconds and connect
- `--kill/-k <bdaddr>` Kill the current PAN connection
- `--killall/-K` Kill all PAN connections
- `--role/-r <role>` Local PAN role (PANU, NAP, GN)
- `--service/-d <role>` Remote PAN service (PANU, NAP, GN), which should be used by PANUs to define which service they prefer GN vs. NAP.
- `--device/-i <name>` Network interface name, in case of multiple devices
- `--source/-S <bdaddr>` Source bdaddr
- `--nosdp/-D` Disable SDP, no lookup and no announcing.
- `--encrypt/-E` Enable encryption
- `--master/-M` Become the master of a piconet
--nodetach/-n

Do not become a daemon, this will make *pand* print on screen connections and any problems instead of the system logger.

--persist/-p[interval]

Persist mode, if disconnected try to reconnect.

--cache/-C[valid]

Cache peer device addresses

Before executing *pand* a SDP server must be executing, so that *pand* can announce its service. Clients (PANUs) will at first look at the SDP server to get the port configuration needed, and then connect to the service. So we should run:

```
# sdpd
```

Now we need to run *pand* and configure it to be a NAP (GN), the next line is what I recommend, but many combinations of *pand’s* options should work.

```
# pand --listen --master --role nap
```

This tells *pand* to execute by listening to any connections, to perform a role switch to master if needed and to act as a NAP. We can change *nap* for *gn* if we prefer a GN to coordinate the PAN.

On the client side, search for 10 seconds:

```
# pand --Q10
```

Or, if we need a reliable connection:

```
# pand --search --persist
```

When the connections are made we should here a beep from both peers, this indicates that the connection has been established. Executing *ifconfig* we should see a new interface called *bnep0* or others. Usually that interface (on both sides) will not have an IP address. This can be assigned manually with *ipconfig*, or use DHCP on the server side and create some net scripts to initialize the interface. The script, which can be created quite easily should reside in /etc/sysconfig/network in SuSE systems. The script must be called *ifcfg-bnep0*, for the first BNEP interface.

If we want to get our IP via DHCP, *ifcfg-bnep0* could be like this:

```
DEVICE=bnep0
BOOTPROTO=DHCP
ONBOOT=no
```
If we want a static assignment, which makes things easier and more efficient if not dealing with many devices and roles are fixed, `ifcfg-bnep0`:

```
BOOTPROTO=static
IPADDR=10.0.0.1
NETMASK=255.0.0.0
STARTMODE=hotplug
```

In the last file we established that IP assignment should start on hotplug. Hotplug can be various things, USB, PCMCIA or Network so we must define that we want it on network interface creation. To do so we must modify the line:

```
'HOTPLUG_NET_DEFAULT_HARDWARE=net' in /etc/sysconfig/hotplug
```

This last configuration is also valid for the Affix stack, only that the Affix stack uses a `pan0` naming instead of `bnep0`, so files should be named accordingly.

Also the steps for creating a router or bridge on the NAP are practically identical to the Affix stack. Routing is exactly the same and bridging (with kernel support):

```
# brctl addbr pan0
# ifconfig pan0 10.0.0.1
# brctl addif pan0 eth0
# brctl addif pan0 bnep0
# ifconfig pan0 up
```

In the PAN package we also have the `dund` tool that allows us to use the DUN and LAN profiles. As I have already mentioned these two profiles are almost the same. However the `rfcomm` tool can create a conventional serial TTY if anybody wants to use it as a modem/PPP. The `dund` syntax and options are identical to the ones proportioned by `pand`, therefore I will not go over them again.

To start a LAN server we must execute:

```
# dund --listen --sdp --channel 1
```

The channel option is to indicate the use of the LAN profile, otherwise DUN will be used. SDP is optional here as the ports are well defined there is no obligation to check it up, although it is recommended.

On the client side we must connect to the server:
# dund --search --sdp

or

# dund --connect BD:AD:DR:ES:S --channel 1

Now we should have a connection to a peer device. *Dund* should assign IP addresses on both sides.

For the DUN profile the steps are exactly the same except for not specifying the channel. However we can do it manually with the *rfcomm* tool to bind a RFCOMM serial port (TTY) to a Bluetooth device and view the results.

```
# rfcomm bind 0 aa:aa:aa:aa:aa 1

# rfcomm

rfcomm0: AA:AA:AA:AA:AA channel 1 open
```

*rfccm* has created a terminal `/dev/rfcomm0` and bound it to the remote device on channel 1.

Any other application can now use "/dev/rfcomm0". Serial port will be automatically connected when applications open it and will be disconnected when it's not used. This serial port can be used for a modem access to an ISP through Bluetooth enabled cell phones using PPP:

```
# pppd /dev/rfcomm0 debug updetach

Using interface ppp0

Connect: ppp0 <-> /dev/rfcomm0

rfcomm0: 00:04:23:00:3D:06 channel 1 connected [tty-attached]

<snip>

local IP address 10.0.0.2

remote IP address 10.0.0.1
```

To remove a previously bound serial port we would do:

```
# rfcomm


# rfcomm release 0
```
Internally all theses *rfcomm* are done by *dund*, so the use of *dund* is recommended.

There is much more things that can be done, on the Projects webpage and in the mailing list there are many tricks, recipes and user added how-to’s, like connecting iPAQs, synchronizing Palm PDA’s and Sharp Zaurus’.

I must give a Special mention about the admirable hack of getting the Microsoft Bluetooth Keyboard and Mouse to work under Linux with BlueZ.

Also the effort made some BSD developers in porting BlueZ to the BSD variants.

With this I have concluded the study of the different Bluetooth stacks; I will now explain the reasons for why I choose BlueZ as the underlying network support system for the Context-aware application I will describe in the next section.
Why BlueZ?

BlueZ is my choice due to the very fact that it’s the standard Linux Bluetooth stack. It is already part of the main kernel tree and therefore every actual and future Linux installation is likely to have it ready for use. This simplifies the installation and makes applications developed with BlueZ reach more people than if it was created with the Affix or other stacks. If I was to develop with the Affix stack, which wouldn’t be too hard as the steps taken would be very similar, I would force future users of my application to download and install the Affix stack, which in a way interferes with new kernels and configuration that the user might have.

I have also found that the BlueZ development is more collaborative and has many top developers in the field of kernel, drivers and network development. There is mucho more movement on the mailing lists and development is fast. Although Affix has much more documentation, with BlueZ it is just as easy to find an answer through the mailing list’s archives. Every new BlueZ package version is full with new features.

Performance-wise both stacks are similar. None has a great advantage, in none of the connection types, over the over stack. I did find PAN in Affix more stable than in BlueZ, but later versions seem to be just as stable. BlueZ does have more functionality when it comes to options in the tools and utilities.

As I have already mentioned Affix is too monolithic and is further away from what the standard specifies. BlueZ’s naming scheme is practically identical to the Bluetooth-SIG Specification.

Therefore my main reason to choose BlueZ is due to its fast development, its inclusion in the Linux Kernel main stream it’s close to Specification naming and implementation, its GPL license and because of many end user developments which are improving the stack with many utilities day by day.
Implementation of a Context-Aware System based on Bluetooth Technology

Aims

The end application for this End of Degree Project is a Context-aware system based entirely on the functions and features that Bluetooth has to offer. These features have been presented in the former chapters to this Project.

The application is a hypothetical museum enhancement that takes use of Bluetooth technology to supply the needs of context-aware and client-server interaction. The application should give a high quality service for the visitors that either possess or hire a Bluetooth enabled device. After downloading (if needed) the client application and gone through a brief configuration process, the users can use the system as a guide or an information source of everything they see.

Thanks to Bluetooth and the network setup, the museum system can find out where every user is due to the logging of the users interaction with the system, as it will provide information of only what the user has in front of him. Knowing what the user has in front, the system can provide the user with detailed historic and cultural information of the piece of art and any other information considered interesting.

As many different types of devices exist, all with different technology and features, the system should be able to detect the type of device and present the information adapted to the technical capabilities of that device. E.g. minimize the graphical performance for smaller devices like cell phones to make better use of the audio system it provides or the other way round if it is a Tablet PC or Laptop with more powerful graphical subsystem.

If the user establishes on executing the device his age, mother tongue, tastes and other factors like knowledge level about the objects exposed at the Museum in the initial configuration process; it is possible for the system to refine the information presented. Therefore the user would visualize the information, in his language at his level and taking into account other information that the system might consider of interest for the user.
Dynamic information about the museum, congested areas, the start of special events, bargain souvenirs at the museum shop or that only 5 minutes are left before the Museum closes can also be given to the user.

Guide services can also be offered, showing the user a particular route around the Museum to cover the user’s needs by his tastes from the profile.

Taking into account all the user based information and where he is, in other words the context, to determine what information to be presented is better constitutes the very concept of Context-aware applications.

As we can see, the end developed Context-aware system, will have endless possibilities to be implanted in real Museums and for many future extensions.
**Basic System Description**

Any Context-aware system should be flexible and extensible enough to provide users with access to information wherever they are.

The Museum system will be a collection of network nodes that each represents one or more pieces of art or objects. These nodes will be connected and report to a Central Server. The technology used in this case can be any variant of TCP Local Area Network. As a typical Museum can reach easily hundreds of meters in extension, so due to range Bluetooth is not really an option, in this case my proposal can be a wired LAN. Wired, as IEEE802.11, the most common wireless technology has important interference issues with Bluetooth. It would be interesting to see how the 802.11a variant works with Bluetooth, as the former works on a different band there shouldn’t be any interference between them.

*Fig. 15: Basic System Setup*
Apart from the Central Server and the Object Representatives/Information Points, there will be one or more Museum Clients.

Between the Information Points and the Museum Clients the Network technology used will be Bluetooth. Bluetooth, as I have already gone over in this project, presents many advantages in detecting devices and obtaining information about them relatively quickly. This allows the developers to design applications for users to detect new servers and without intervention, obtain new information about what they are viewing. The application will then, have to be based entirely on Bluetooth’s Inquiry and Service Discovery Protocol, as inquiry allows detection of devices, and Service Discovery allows to detect the services offered by that device.

So the basic idea of the system is that a client with a Bluetooth equipped device, wonders around the museum. The client application will continuously search for new devices using Bluetooth’s Inquiry. When a new device is found it will then see if the peer device can offer any information of interest through a SDP search on that device, if so the user will be notified to whether he prefers to see the new information. In that case the device will connect to the Information Point and ask for the information.

But for the Information Point to know what type of information the user needs, it requires the context of the user. For that the user’s device will send the information point the profile that was entered in the beginning configuration process. As the Information Point has one Bluetooth device per Object represented, the Object is determined by the interface where the connection is made; therefore location data is also determined. Other approaches like triangulation and using signal strength to reduce the area where the user is, is just too much work and time (2 or three connections to do the triangulation) for something only valid in a very short period of time.

So after the Client has sent the Profile, the Information Point has all what’s needed in order to process the petition.

The Information Point would then add to the users profile the object the users is viewing and send it all off to the Central Server. There the petition would be processed, logged and returned to the Information Point, who would in turn send it to the client for viewing.
After viewing, the user can decide to search manually for more devices, or let the Client Application do it automatically. In any moment the user can change his profile if he finds that the information is too advanced or too basic, allowing futures views to be more in line with his own way of seeing things.

This is the very idea of Ubiquitous computing, as invisibly the system determines positioning, and without having to do nothing, the user’s device gets in contact with the system for personalized information, in other words: the whole system, servers and own devices work together for the users needs.

This gives a high quality service to the user, as he will rate very highly such a personal service. A high valued service is the beginning of good businesses.
But for the client-side communication Bluetooth is good for Context-aware applications but on studying its characteristics for this project I did find many limitations.

Bluetooth has the capability of using TCP/IP though either DUN/LAN or PAN profiles. Use of TCP/IP could help speed up the development as nearly every developer has experience with TCP/IP applications and the reliability that TCP gives.

But we have seen that there is an important bandwidth overhead in DUN due to the RFCOMM layer, but in both cases, DUN and PAN, Bluetooth has a limitation that is only worsened: Latency.

The latency that I have mentioned in earlier chapters is the high length of time Bluetooth can take in establishing a connection: up to half a minute between inquiry and connection (paging). If on top of that we have to set up the Profile, IP routes and gateways, and then do all the communication, too much time is lost. If it takes 20-30 seconds each time a client wants to view information, the user will most certainly want to stop using the system.

Also the use of profiles disables Inquiry and lower level Bluetooth operation. So if a user is connected and receiving information through an IP network by using one of the profiles it is going to be very difficult to be able to find other servers. Trade off methods could be used but the profile would have to be modified to cope with SDP and Inquiry.

Therefore no profiles could be used without an effect on the connection times. On the Bluetooth stack we have to use directly a lower layer: L2CAP.

Another limitation, but this time hardware based is audio. For audio-only devices like advanced headsets or cell-phones it is needed the use of SCO-audio connections and audio sources in order to give a service to a user that doesn’t have a graphical display to play with.

It would be very interesting for servers to give an audio description of objects, for both, invalid or blind users and for anybody with a Bluetooth equipped devices to give a real multimedia feeling to the information provided.
As most cell-phones will not be able to use a lot of Bluetooth software but it could be quite trivial to create an audio source for users to bind their phones to.

The hardware limitation is that SCO doesn’t work on certain types of USB Bluetooth adapters; more concretely no USB devices are supported for audio with the BlueZ or Affix stacks. The BlueZ team, however, are working on it. Though I did have a PCMCIA card that would work with SCO/audio, I would need another to actually try it out, but in my case all I had was one PC-card and one USB adapter.

The third limitation is bandwidth, meaning that the data to be passed on cannot be any size. Precautions should be taken to keep a good balance between the quality of the information given to the user and size. This will also help to maintain low connection times, as many simultaneous users can block the server or make it slower to react to the petitions, thus increasing latency. Keeping size low would also be beneficial to smaller devices without the processing power of today’s laptops.
System Development, Process and Tools

Now it's time to describe the whole development process: how the system has been implemented and what tools have been necessary.

The development of the Museum Context-aware system starts with comprehending the basic Bluetooth capabilities; this is what I have gone over in the first chapters of this Project. Once understood how Bluetooth works it is needed to find out how to implement an application and how to use the Bluetooth features. This comes by installing and going through the source code of the BlueZ protocol Stack, as no programming manual is available, except for a small snippet in the BlueZ “How-to”.

BlueZ has a great number of tools and utilities that work very well in testing a Bluetooth setup. But these utilities also show the basic use of the Bluetooth API. The include files give the function names and the utilities and stack implementation give the manual of “How to program using BlueZ”.

To understand how to program with BlueZ, I decided to implement one of the most simple network programs a programmer can do, an Echo Server and Client.

As I knew that higher layer protocols were not going to be an option in the final application and seeing that BlueZ gives a Socket interface to L2CAP I decided to use L2CAP as the layer to use mainly as it is reliable and uses segmentation and reassembly on ACL packets. L2CAP also multiplexes communications through the PSM abstraction that works basically as a TCP port. Lower levels like the BlueZ socket interface to the HCI, is not really interesting as the data received is raw, unprocessed and unreliable, meaning that all the treatment L2CAP gives, must be done by the application with hardly any performance gain.

By looking at the sources of l2test from the BlueZ utilities package I could quickly figure out how the Socket was setup, what options were available and how the application sent and received data. It was a couple of days before I had my first Bluetooth application. Using the BlueZ functions required me to include and link against the BlueZ shared objects. (/usr/lib/libbluetooth.so) This can be easily done by passing the “-L/usr/lib –lbluetooth” option to the compiler.
But this was a mere copy of l2test and I wanted to complicate it a little, so I added a Bluetooth Inquiry. The client would look for a device and then try to connect to the echo server. This was really exciting as I was not only programming with sockets using Bluetooth but I was also using some of Bluetooth’s features.

So I decided to start designing what the Final application should be like. But first I needed to know what graphical libraries and environment I would have to use for the Client. My GTK/QT knowledge was (and is) quite basic, and after playing around with the Glade environment and finding things wasn’t so easy I decided to go onto something I had experience with, Borland C++. But using Linux, I had to use Borland’s new Open Kylix, version 3 with C++ support.

Kylix is a very friendly environment to program with. It is very similar to the Borland C++ Builder saga for Microsoft Windows; therefore creating windows, buttons and lists is very easy. But Kylix has one main problem, it uses many Borland libraries. The minimum library that is needed for graphical management is a Borland version of the QT libraries. The QT libraries are readily available, as they are the very motor of the KDE desktop, so they are quite common on Linux installations.

Open Kylix is also free against what its bigger brothers like the professional edition. It is also GPL, and this forces any development with Open Kylix to be GPL as well. This really isn’t a problem for me, as the final result isn’t going to be released to the general public, but Kylix does insert a splash screen on every compiled program to remind us that it has to be GPL.

Now that I had clear what development tool I needed for the client, I had to decide how I would develop the servers. In this case the server’s code is much simpler that the Client code, due to it only has to answer petitions coming from the Clients. I decided to leave them as command line applications to make them as small and efficient as possible. In that moment I had also though about creating embedded systems for the Museum Information points. Standard C was used and compiled with gcc.

As I have mentioned there are two servers, one Central server without the need for Bluetooth support as it will typically be used via Ethernet variants. The other server,
or Museum Information Point, will need the Bluetooth functions as it will dialogue with the Bluetooth Clients.

I have already gone over what the communications are going to be like but let’s go into more detail, on how the whole system will work.

The first program I will mention will be the Museum Central Server, due to its simplicity. I will then go on to explain the Museum Information Point and its interaction with the Clients. And finally I will explain what the Clients actually have to offer. I will go through the functions that each program uses and how they all fit into the system. I will also give diagrams for anybody to be able to understand the basic working of the system.

The actual code is available and included in the End of Degree CD.
Museum Central Server

This program will start without any options and wait for a connection on the default Server Port, which has been defined as ‘9999’. If a connection is received on that port a new process is spawned to attend it and will receive a Code Operation (codop).

Depending on the value of the codop, the Central Server will do one of two things:

If the codop corresponds to a Configuration Request, the server will send to the Information Point the contents of the configuration file objects. In this file is an object name per line. All the names of the objects or pieces of art that reside in the Museum and for what the service are given will be listed in this file. Each line will correspond to a directory. In each directory there will be a structure of subdirectories that will correspond to information formatted for different devices, in various languages and at different levels of detail. The directory structure has this format:

![Museum Server Directory format]

*Fig. 16 Museum Server Directory format*
As we can see, the structure is replicated for each Object and information is adjusted for each profile option. Ideally we would be using a Relational Database, with Server and Information Maintenance Tools, but this End of Degree Project is based more on offering a proof of concept of Bluetooth as a Network support system.

Once the Museum Information Point has obtained the configuration file, the Main Server will disconnect, and the spawned process will end.

If the codop corresponds to an Information request, the Server will spawn another process for it to receive the profile and object name. For each profile option received the Server will ACK it to make sure errors are controlled.

Once all the profile is received, the Server will log the petition for security and for statistical information. This can help the Museum to know which are the preferred objects, which have had more visits and routes followed throughout the museum, helping the internal exposition and organization. The log file is called svrlog.

After logging the petition the Museum Server will access the Directory Structure taking into account each profile option, so if a petition is for an advanced description in English for the Mona Lisa and the clients device is a laptop, the information to send will be the information pertaining to the corresponding directory. This Information will be sent to the Information Point who in turn will resend it to its Client.

We must remember that a client profile consists of a Language of preference, which for the application I have reduced to two, but easily extendable to many more; a Device type which will allow information formatted to take into account the graphical capabilities of the Client and an educational level which will determine the complexity of the explanations taken. On top of this, what the Central server actually receives is the Object name which is added to the profile by the Information Point.

The information to be sent back to the Information Point is the contents of the directory which in the case of the laptop (no more clients have yet been created) are two photos, one of the actual object in view and another of the author, and three text descriptions. These text descriptions correspond to a description or interesting information of the object, another text will go over the life of the author and the third
will be related to the actual position of the object in the museum, giving the Client an idea of where he is and of where he can go.

Once finished sending the data to the Information Point the Central Server will disconnect and the spawned process that dealt with the communications will end.

A basic, rough, flowchart that can resume what has been mentioned above:

As we can see, the functioning of the main server is quite simple. On a real Museum implementation it would be needed a good set of options to ensure quality and
persistence of the information. Relational Databases, Server side Maintenance tools and Distributed Computing could be a solution. Later in this Project I will go on to talk to a whole set of enhancements for this application.

Now I will go over the main calls for this application. All command line applications start by executing their, main call:

```c
int main (int argc, char*argv[])  
```

Here most initialization is done, and space is left for future enhancements and command line options. This call then proceeds to call do\_listen().

```c
void do\_listen()  
```

This is the main program loop; in it the socket is made and bound to the Server Port 9999. It will remain listening until a connection is made to that port. If a connection comes in, it will be accepted and a new process will be spawned (forked) to deal with the connection. The Parent process will close the accepted socket and wait for other connections on the Server Port.

The first action to be done by the forked process is to receive an integer value, which will determine the service to be given. This integer is called a *codop*. Depending on the value of the *codop* the process will either Send the configuration file or receive the Clients profile for afterwards to send the required information. The call for obtaining the *codop* integer is:

```c
int do\_get\_codop(int s1)  
```

The parameter is the socket to receive on, and the return value is the actual *codop*.

If the received *codop* is a Configuration Request the process will first call do\_send\_obj\_config(int) and then disconnect and end.

```c
void do\_send\_obj\_config(int s)  
```

This function basically opens the configuration file, calculates the files size, sends to the Information Point its size and receives the size in form of an
Acknowledgment. After this the Server will send to the Client the configuration data. Once sent the socket is closed and the forked process ends.

Please note that I have had many problems with sending over the Bluetooth link. It seems that the BlueZ implementation is not 100% bug-free, and disconnections are frequent. The solution can be done by resending data and timeouts, but this really should be labor of the L2CAP layer. Therefore in my code many acknowledgements are made for debugging purposes. Also I have had problems with dynamic memory; as the server applications have frozen on *malloc* calls without returning any value whatsoever. This has made me implement using static memory allocations. Static memory could be a problem due to file sizes, but I have taken precautions to reserve decent sized arrays for memory allocation to make sure that there will be enough for the typical file size used.

On the other hand if the received *codop* is a Information Request, the Server will prepare itself and receive the Clients Profile and the object for which to retrieve the information for.

To receive the profile we would use the function:

```c
void do_get_profile(int s)
```

This function is a succession of receive and acknowledge for each profile data to use. First of all, the Museum Client’s Bluetooth address is received from the Information Point; this is mainly for logging purposes, to know what users have gone where. This could be seen as a privacy violation but is a good security feature for the museum. After the Bluetooth address is received the information to receive and acknowledge is the device type, user’s language preference and educational level. These values are typically integers, except for object name and the Bluetooth address that is received in a string format. Receiving the address in a string helps the main server to not to use the Bluetooth libraries. On the other side if needed BlueZ does proportion many functions to convert between strings and device addresses, as the addresses format is very similar to the Ethernet address type.

The acknowledgment is in form of returning the information for the information point to see if everything has arrived alright.
After receiving the profile the forked process in the do_listen() function logs the petition in the svrlog file. In this file the data is structured like this:

00:04:76:E1:AA:5E monalisa 127.0.0.1 Thu Jan 23 13:42:39 CET 2003

We can see that what is logged is the Client Bluetooth address, the object it asked for, the IP address of the Museum Information Point and the date/time of the petition.

This can allow quite a good control of what is happening in the museum.

After receiving the profile and logging the petition the Server changes its directory to the directory where the information is located. By using simple names to denote the profile option as we have already seen, it is quite trivial to obtain the correct information. Now it would be time to send the information back to the Information Point. To do so the Server has two functions in which its able to send text or images.

These functions do exactly what their name says:

```c
void do_send_txt1(int s, char *file)
void do_send_imgs(int s, char *img1name)
```

The process will send the data obtained from the directory with successive calls to these function. In the actual program it is something like this:

```c
do_send_txt1(s1, "./mytext1");
do_send_txt1(s1, "./mytext2");
do_send_imgs(s1, "./photo.jpg");
do_send_imgs(s1, "./author.jpg");
do_send_txt1(s1, "./mylocinfo");
```

The file names are standard and the actual object is only represented by its base directory name. Therefore in the listed calls above the program is sending a object description text (mytext1), an author information text (mytext2), a photo of the actual object (photo.jpg) and a photo or illustration of the author (author.jpg) and of course location based information surrounding the object (mylocinfo).

The sends are very similar to the previous receives, where acknowledgements are taken place quite often.

After sending the data the child process will end releasing the socket and the parent process will carry on waiting for more petitions.
Museum Information Points

This program starts taking at least two arguments; the first must be the local Bluetooth device, and the second the remote Museum Server IP address. Apart from the two obligatory parameters, there also 3 other options.

The first is a \texttt{--b <bytes>} option, this sets the Maximum Transmission Unit for the Bluetooth communications. It is there for testing purposes as different values should have different performance as the possible data rate could be increased but then again interference issues are there to be careful with.

The second is a \texttt{--P <psm>} option, this changes the standard PSM to register in the SDP database. This could allow various Information Points on the same machine, which reduce costs, but could have interference problems that would be interesting to study.

And finally there is a \texttt{--D} option to permit datagram sockets. This unfortunately doesn’t have any effect as I discovered and reported a bug to the BlueZ team as it didn’t work too well. The source Bluetooth addresses were not sent in the datagram packets thus creating problems when needed to send back data. Up to the moment I still have no news on whether it has been fixed.

Once executing, the program detects the users UID. This is needed as the program must be run by root, as the program will try to execute the SDP daemon in order to be able to register its own services.

The program will then go on to connect to the Museum server and retrieve the Server’s configuration file. This configuration file would be presented to the Information Point technician who would choose an object for that Information Point to represent.

Then the program would start the SDP daemon and register its service along with the appropriate attributes like titles in various languages and PSM values for the Clients to know where to connect to.

Once all the SDP registration has been done, the Information Point will create and bind L2CAP layer sockets and wait for connections coming from the Bluetooth Interface.
When a connection comes in on the Bluetooth Interface, the Information Point will spawn a child process to deal with the Museum Clients petition.

The Childs process’ first mission is to log the Clients Bluetooth Address and date of petition. This can also help debugging and Museum maintenance. Once the logging has finished the Information Point will receive the Clients profile.

The Information point would append the local object’s name and Clients Bluetooth Address and send it off to the Central Server. There it will get processed and send back the information that will be passed on to the Client. Once all the data has been sent to the Client it will disconnect and the child process will end. The Parent process will carry on waiting for further Client petitions.

A rough Flowchart of the Information Point’s execution life:

![Flowchart of the Information Point’s execution life](image-url)
Now, let’s get into more detail on how the Museum Information Point’s actually are implemented. This program now uses Bluetooth functions and data structures, therefore it is necessary to include the function headers and link against the Bluetooth and SDP libraries.

The BlueZ Bluetooth headers needed to be included are:

```
#include <bluetooth/bluetooth.h>
```

This include file contains the basic Bluetooth data types and the functions to manipulate them.

```
#include <bluetooth/l2cap.h>
```

This file has all the necessary to use the L2CAP layer like socket types, options and the actual libraries implements the mapping.

```
#include <bluetooth/hci.h>
#include <bluetooth/hci_lib.h>
```

These have the basic device manipulation functions, like getting and setting device options, packet types and other functions like Inquiry, etc.

```
#include <bluetooth/sdp.h>
#include <bluetooth/sdp_lib.h>
```

These contain the SDP calls and are necessary to register services, manipulate attributes and service records and invoke SDP searches and browses.

To link against the Bluetooth and SDP shared objects all what is needed is the linker option `–L /usr/lib –lbluetooth –lsdp`. Compile with `gcc` and that’s it.

The program, as I have already mentioned, starts in the `main()` function by checking the UID of the executor, this is certainly needed as the SDP daemon `sdpd`, needs root permission to execute, and without it, it would be impossible to register the Information Points services. Once that the program has checked that the UID is correct, it now checks that the Bluetooth Interface name that was passed on the command line is correct and gets the device identifier of that device. To do so, it uses the HCI call:
hci_devid(device_name);

This function uses the string name for a device e.g. hci0 and returns the device identifier. The identifier is needed for future operations of the device.

Now the program checks all the command line arguments passed to the program by the technician who will execute the Museum Information Point. As I had mentioned before the arguments are related to the MTU, PSM and Socket type.

Once these are defined, the Information Point is nearly ready to work, but it still needs an object to represent.

To do this it needs to know what objects it can effectively represent and choose one. As I have already mentioned it requires the Configuration file from the Central Server. The Information Point must connect to the Central Server, send the corresponding codop to indicate that it needs the objects list and finally receive the file. All this is done by the next sequence of functions:

\[ sck=do\_connect\_svr(); \]

This function creates the needed socket, connects to the Central Server and returns the socket identifier.

\[ do\_send\_codop\_svr(sck,1); \]

Here, \texttt{do\_send\_codop\_svr()} sends the codop corresponding to a Configuration Request. The Museum Server would now know that it is needed to send the configuration file to the Information Point.

\[ do\_get\_objs\_config\_svr(sck); \]

\[ close(sck); \]

Here the Information Point receives the configuration file and presents it to the administrator at the Information Point the list. Then the admin would then choose an object from the list. This object is what that Information Point will represent from then onwards to any Client petition. The connection to the Central Server is closed for the meantime.
Now that the Information Point has all the Information it needs to advertise its services in order for Clients to find the Information Point and connect to it. This means that the Information Point must register its services in a SDP server then the SDP server would give to all the soliciting Clients connection details and parameters. So the Information Point starts up the SDP daemon with:

\[\textit{do\_init\_sdp();}\]

In this function the SDP daemon is started by forking the main process and having the Child process to execute the \textit{sdpd} daemon.

Also the SDP server can give language based information, this is important for users that try out the service having different mother tongues. Clients could connect to the Information Point without having to search through the SDP database, and use predefined ports, but SDP gives much more flexibility, and the possibility for Multilanguage descriptions.

So for this the Information Point then registers the Museum Information Point in the newly created \textit{sdpd} database, using the next set of functions:

\[\textit{sess = sdp\_connect(BDADDR\_ANY, BDADDR\_LOCAL, 0);}\]
\[\textit{si.name = "Museum";}\]
\[\textit{do\_add\_museo\_sdp(sess, &si);}\]

Here the Information Point uses the SDP library functions to connect to the Local SDP server, the \textit{sdp\_connect()} call creates a L2CAP connection with the SDP Server on PSM 1. The SDP connection is usually called “session”. With the session, the Information Point calls \textit{do\_add\_museo\_sdp()}. In this function all the important data is stored on the SDP Server’s Database. The information to register is the Service Name, description of what languages are used, titles in those languages and the PSM value.

The \textit{do\_add\_museo\_sdp()} function is a very interesting example of how SDP Service Registration is used. It has been mainly adapted from Bluez’s \textit{sdptool} utility, but also BlueZ’s PAN and DUN implementation mimic it very closely.
Due to its interest, I will include the entire C code for the \texttt{do\_add\_museo\_sdp()} function:

```c
static int do_add_museo_sdp(sdp_session_t *session, svc_info_t *si){

    sdp_list_t *svclass_id, *apseq, *root, *proto;
    uuid_t rootu, gn, l2cap;
    sdp_list_t *proto[1];
    sdp_lang_attr_t lang[2];
    sdp_list_t *lSeq;
    sdp_record_t record;
    uint16_t lp = psm; //L2CAP_PSM
    sdp_data_t *psm_
    char mytitle[60];

    memset((void *)&record, 0, sizeof(sdp_record_t));
    record.handle = 0xffffffff;
    sdp_uuid16_create(&rootu, PUBLIC_BROWSE_GROUP);
    root = sdp_list_append(0, &rootu);
    sdp_set_browse_groups(&record, root);
    sdp_uuid16_create(&gn, MIP_SVCLASS_ID);
    svclass_id = sdp_list_append(0, &gn);
    sdp_set_service_classes(&record, svclass_id);

    sdp_uuid16_create(&l2cap, L2CAP_UUID);
    proto[0] = sdp_list_append(0, &l2cap);
    apseq = sdp_list_append(0, proto[0]);
    psm_ = sdp_data_alloc(SDP_UINT16, &lp);
    proto[0] = sdp_list_append(proto[0], psm_);

    lang[0].code_ISO639 = "en";
    lang[0].encoding = 0x08;
    lang[0].base_offset = 0x0100;
    lSeq = sdp_list_append(0, &lang[0]);
    lang[1].code_ISO639 = "es";
    lang[1].encoding = 0x08;
    lang[1].base_offset = 0x2000;
```

As we can see the code seems quite scrambled, this is due to the overloaded types in SDP. There are many of them and the information to be sent to the SDP Server has to be formatted in a specific way. Basically what the code does is create a Service Record and keep adding different data to it. The registration starts off by creating the Public Browse group which is needed for all Services as it allows browsing of the records if the Client doesn’t do a particular search. Then it adds the Class ID. This is an

```c
lSeq = sdp_list_append(lSeq,&lang[1]);
sdp_set_lang_attr(&record, lSeq);

aprot = sdp_list_append(0, apseq);
sdp_set_access_protos(&record, aproto);
sdp_list_free(proto[0], 0);
sdp_list_free(apseq, 0);
sdp_list_free(aprot, 0);

snprintf((char *)&mytitle,60,
        "Servicio de Punto de Informacion del Museo: %s",object);

sdp_attr_add_new(&record,SDP_ATTR_SVCNAME_SECONDARY,SDP_TEXT_STR8,
                 (void *)&mytitle);

snprintf((char *)&mytitle,60,
        "Museum Information Point Service: %s",object);

sdp_attr_add_new(&record, SDP_ATTR_SVCNAME_PRIMARY, SDP_TEXT_STR8,
                 (void *)&mytitle);

if (0 > sdp_record_register(session, &record, SDP_RECORD_PERSIST))
{
    printf("Service Record registration failed.\n");
    return -1;
}

printf("Museum Information Point service registered\n");
return 0;
```

Bluetooth for Context-aware Applications
Identifying number of the Service (MIP_SVCLASS_ID). It will be the main attribute a
Client will search for when looking for Servers to connect to. As this service has been
created by me, and no profile defined the ID value if well known between servers and
clients.

Next would come the definition of used profiles, as in this case I am using the
L2CAP layer and no profile is defined I have two options, either go through the lengthy
process of defining a profile, or omitting the profile data. I chose the latter to keep the
design and implementation simple. After the possible profile definition, it’s time to
define the list of Bluetooth Protocols used in the Service. Here I only use L2CAP so I
define it as L2CAP_UUID, which is a Specification defined value to indicate the use of
the L2CAP layer. No more protocol layers are used so I must define the basic PSM to
use on the L2CAP layer. This value is taken from the default value or updated at the
command line options. This is important for Clients to know for what PSM to use.

After the protocol setup, the language options are included. The code specifies
for what languages the service operates, defining encoding and offset for where Strings
indicating titles and descriptions will be located. The basic default language always has
its offset at 0x0100, and secondary offsets must be defined on both client and server
sides. With the offsets, the Information point can store description and titles at that
offset for that language, as I do for both English and Spanish, adding the Object name
as well so Clients can choose if they need the correct object in case of conflict.

Finally, after adding all the attributes the record that contains them must be
registered in the SDP database. The is done by the last call sdp_record_register().

In that moment until the device goes down, the SDP daemon is killed or the
attributes are removed the Service will remain advertised for all the clients.

Now let’s carry on with the rest of the code. The next action the Information
Point will do is to listen out for incoming connections on the Bluetooth interface with
the do_listen() call.
do_listen()

This function first establishes a correct Class of Device for the service. The class of device is important for future Clients, as they can check this number when they do an Inquiry. The Class of Device is formed by the Bluetooth-SIG assigned numbers; it is formed with three bytes which indicate the general functionality of the device. The Class of Device can indicate what type of services is the main aim for that machine. It acts a general description without concretizing exactly what that machine does. With the Class of Device Clients can decide to do a SDP query or not, as they have some information already about the device and can decide if the service they are looking for can be included in the description by that Class of Device.

This is done by the do_set_class(dev_id); function which in turn calls the two HCI functions:

```c
hci_open_dev(dev_id1);
```

```c
hci_write_class_of_dev(dev, cod, 1000);
```

What is done here is the opening of the Bluetooth device and establishing the Class of Device cod which contains the hexadecimal value “0x810100”.

The 10 most significant bits that form this value indicate the Major Service Class which generally defines the Service to be given. The types of services are Information, Telephony, Audio, OBEX, Capturing, Rendering, Networking, Positioning and others.

Bits 12 to 8 indicate the type of machine (Major Device Class) that has the device, like a miscellaneous device, Computer, Phone, LAN access Point, Audio/Video, General Peripheral, Imaging and Undefined.

The last 8 bits indicate the Minor Device class and generally refine the Major Device Class. It should be typically not specified.

The Class of Device that I define indicates Information and Positioning Services, on a unspecified Computer.

do_listen() function also creates the Bluetooth interface socket to listen on, and accept any incoming calls. The socket creation for the L2CAP layer is very similar to TCP/IP sockets. We have Datagram, Stream and SEQSockets to play with. Instead of ports we have PSM’s, instead of IP addresses we have Bluetooth device addresses.
(BD_ADDR). The big difference comes in the socket options. With Bluetooth we are able to define on the L2CAP socket whether to use, authentication, encryption or role switches. We are also allowed to set in and out MTU’s. Check the next snippet of code to see what I am on about, this code is not meant to do nothing but set up a server side Bluetooth Socket.

```c
s = socket(PF_BLUETOOTH, socktype, BTPROTO_L2CAP);
loc_addr.l2_family = AF_BLUETOOTH;
loc_addr.l2_bdaddr = bdaddr;
bind(s, (struct sockaddr *) &loc_addr, sizeof(loc_addr));

/* Set new options */
opts.omtu = omtu;
opts.imtu = imtu;
opt |= L2CAP_LM_MASTER;
opt |= L2CAP_LM_AUTH;
opt |= L2CAP_LM_ENCRYPT;
setsockopt(s, SOL_L2CAP, L2CAP_LM, &opt, sizeof(opt))
setsockopt(s, SOL_L2CAP, L2CAP_OPTIONS, &opts, opt);
```

In my own code the only options I use is the in MTU and define a role switch to enable the Information Point to be master. The role switch is important to ensure that the server is able to cope with various clients. With the socket set up the communication is done by simple `recv`s and `send`s on the socket.

The `do_listen()` function will then wait for any incoming connection. When this happens, the connection is accepted and a child process is spawned to deal with it.

The child process will log the connection and receive the profile from the Client with the `do_get_profile(s1)` function. This is very similar to the equivalent function on the Central Server, where the values of each profile parameter are received and acknowledged.
The next function, `do_connect_svr()` will connect to the Central server returning the Server side socket. With this socket the Information Point will send the corresponding `codop` to tell the Central Server that a profile is on its way. This is done with:

```
do_send_codop_svr(sck,2);
```

Then the Museum Information Point will resend the profile with the Clients Bluetooth Address and represented object. Just like when it received the profile it will send and receive acknowledgements from the Central Server. This function is:

```
do_send_profile_svr(sck);
```

Now that the Central server has the profile information the Information Point must receive the results of that profile.

The next successions of function calls represent the receiving of the Information from the Central Server, and resent being resent to the soliciting Museum Client. Please note that the information is stored locally in temporary files. Future versions could implement a cache to store this information with only updating dynamic information.

```
do_get_txt1_svr(sck,"./mytext1");
do_get_txt1_svr(sck,"./mytext2");
do_get_imgs_svr(sck,myfile1);
do_get_imgs_svr(sck,myfile2);
do_get_txt1_svr(sck,"./mylocinfo");
do_send_txt1(s1,"./mytext1");
do_send_txt1(s1,"./mytext2");
do_send_imgs(s1, myfile1);
do_send_imgs(s1, myfile2);
do_send_txt1(s1,"./mylocinfo");
```

After having successful resending of the information the Information Point disconnects from the Central Server and from the Museum Client. The forked process ends but the parent process continues to listen for more incoming connections.


Museum Information Point Clients

The Client side application is probably the most complex part of the Project. In part because of the difficulties I have had to develop it. The Client Application has been developed in Open Kylix C++ which proportions an easy interface to create graphical interfaced programs. Borland’s Linker \textit{ilink} had given me a good fortnight of nightmares to get the Client side SDP working, but finally it worked ok. The main problem came from using the BlueZ C functions in the C++ Project. For the general Bluetooth device, HCI and L2CAP there was no problem but the SDP library was simply not prepared to work with C++. To get everything working I had to modify the SDP library and embed it in the Client code. I hope everything stays just as readable, but no other solution was available at that moment. In the last days there has been a discussion in the BlueZ mailing lists about this problem and although it has not been released in package form, the fixed SDP library has been released in the BlueZ CVS.

The Museum Information Point, starts and works like any GUI application. But it needs the Borland Library that is included in the zipped package. To execute it:

```
./ProjectMIP.bash
```

This creates the necessary global variable to make the program work with the library, and also executes the actual file.

Once working the program creates itself on screen and waits for human intervention. What is needed is for the user to fill in his profile in the “\textit{Change…Profile}” menu, and if he prefers to change the language in the “\textit{Change…Language}” menu. Once all the data is filled in the user can now use the program to search for information on what he views. To do that the user must press the “\textit{Go Get Information}” button.

In that moment the Client Application will proceed to do a Bluetooth Inquiry. If no devices with the correct Class of Device are found, it will show a message to the user to move closer to an object. If at least one device is found, it will proceed to do SDP searches on each of the devices found. If only one device was found the Client will connect to that device, if not the application will create a list of found objects with the title of the object in the users preferred language.
Once the Information Point has been chosen, either because it was the only one or because the user had chosen it from the list, the application will connect to the device. Once connected the Client will send the stored profile to the Information Point and receive the information. Once all the information is received the application will show it to the user. After the first retrieval of information the entire process of inquiring, SDP searching, profile sending and information receiving is done by a separate thread, that will do this automatically. The thread will act pseudo intelligently, as it will not always update information, as it would be annoying for the user. It will only do so when new servers are found and/or the current information is no longer available.

Next is the basic Flowchart of the Client application:

**Fig. 19 Basic Client Application Flowchart**
Now let’s see a lower level description of the Client application implementation. The application source is in the form of a Kylix/C++ project. The application consists of the next list of files:

- **Project File:** ProjectMIP.bpr
- **Resource Files:** ProjectMIP.res, Unit1.xfm, ProfileFill.xfm, RellenePerfil.xfm, MainAppEn.xfm, AboutEn.xfm, AppFormCpp.xfm, InqDlgc.xfm, Choosec.xfm
- **Headers:** AboutEn.h, AppFormCpp.h, Choosec.h, glob.h, InqDlgc.h, MainAppEn.h, MFUpdateThread.h, ProfileFill.h, RellenePerfil.h, Unit1.h
- **CPP Files:** AboutEn.cpp, AppFormCpp.cpp, Choosec.cpp, InqDlgc.cpp, MainAppEn.cpp, MFUpdateThread.cpp, ProfileFill.cpp, RellenePerfil.cpp, Unit1.cpp

Please take into account that for BlueZ applications to work with Kylix it is needed to modify the project file. The shared object BlueZ library must be included in the `<ALLLIB>` section. All we need to add is `libbluetooth.so` to the library list.

The starter application file that is where everything starts is the `ProjectMIP.cpp` file. There it defines all the Classes and forms used. The main form that will be the center of the application is the first form to be loaded in the `ProjectMIP.cpp` file, this is the `AppFormCpp.cpp` file.

`AppFormCpp.cpp` is the base file around what everything works around. Its form `Appform` is the first and main form that the user will see. The first thing this Form does using its `FormShow()` event is to initialize some variables. After this the program sits waiting for human intervention. The first action a user should take to make the program function properly is to enter his profile and change the program. Without the profile the program could assume defaults but this can be unreal and both user and system would not have the full picture. So to oblige the user to fill in the profile, the program will not inquire for devices until this is done.

The profile can be changed in the “Change...Profile” menu. This calls either the `FormProfile` or `ProfileEs` forms, depending on the actual language. The two forms have
their own Class/cpp file called ProfileFill.cpp and RellenePerfil.cpp respectively. These forms simply have a series of edit lines where the user can enter his name, surname (these for future use as at the moment the system only uses the basic profile data), and educational level. At exit from these forms the values are stored.

The language, however, can be changed by the “Change...Language” menu, there the user can simply choose between the two available languages, English and Spanish. The form and cpp file are FormLang and Unit1.cpp. On changing the language this value is saved and the program menus are changed to be in the selected language.

Now we have all the data needed for requesting information with the profile. But please note that actual device type information is given from the executable. So there should be an executable for each type of device, then the device type value should be hard coded into it.

From AppForm the user can do another 3 things, exit the program with the “File...Close” option, see the credits/developer about page through the “Help...About” menu and the AboutEnDlg form / AboutEn.cpp file or actually do the programs main purpose: Find Museum Information Point and retrieve information about the objects they represent.

For this the happen the user must press the “Go get Information!” button or its equivalent in Spanish. This will trigger the “Click” event for that button, this event is the TAppForm::ButtonGetInfoClick() function from the AppFormCpp.cpp file.

This function checks that the user has completed the profile and if the local device is up and running it will proceed to invoke a Bluetooth HCI Inquiry with:

```
num_rsp = hci_inquiry(dev_id, length, num_rsp, NULL, &info, flags);
```

This function from the BlueZ libraries takes the local device, the length of time to inquire for and the maximum number of responses to allow. It returns the actual number of responses and in the info structure, the Bluetooth Device Addresses, Clock Offset of those devices and Class of Device. During the execution of the Inquiry the FormFindMIP dialog (InqDlgc.cpp file) will be shown, telling the user to wait as search for an Information Point is being carried out. The function will then go on to check the actual number of devices found, to do so it filters out the responses with
Classes of Device other than the one searched for. If no correct responses were received the user will be informed. Otherwise each correct CoD device found will have a SDP search on it with

\[mip\_sdp\_search(&bdaddrloc, &bdaddr, &my\_list\_sdp[i])\]

What this function does is take the Bluetooth Device Address and search the Museum Service Class ID in it. If the attribute is found it will be added to the temporal list \textit{my\_list\_sdp}. As this function is quite interesting as a SDP search example its entire code is included:

```c
int mip_sdp_search(bdaddr_t *src, bdaddr_t *dst, 
                mylist_t *list_sdp) {

    sdp_list_t *srch, *attrs, *rsp = NULL;
    sdp_session_t *s;
    uuid_t svclass;
    uint16_t attr;
    int err;
    char str[60];
    uint32_t range = 0x0000ffff;

    sdp_uuid16_create(&svclass, MIP_SVCLASS_ID);
    srch = sdp_list_append(NULL, &svclass);
    attr  = SDP_ATTR_PROTO_DESC_LIST;
    attrs = sdp_list_append(0, &range);

    s = sdp_connect(src, dst, 0);
    if (!s) {
        fprintf(stderr,
                "Failed to connect to the SDP server.%s(%d)\n",
                strerror(errno), errno);
        return 0;
    }

    err = sdp_service_search_attr_req(s, srch, SDP_ATTR_REQ_RANGE, 
                                 attrs, &rsp);
    // Assume that search is successful
    //   if at least one record is found
```
if (!err && sdp_list_len(rsp)){
//FOUND MIP
list_sd->addr=*dst;
sdp_record_t *rec = (sdp_record_t *) rsp->data;
sdp_list_t *protos;
sdp_list_t **langSeq;

if (!sdp_get_access_protos(rec, &protos)) {
    int ch = sdp_get_proto_port(protos, L2CAP_UUID);
    if (ch > 0) {
        list_sd->psm = ch;
    }
    ch = sdp_get_string_attr(rec,
        SDP_ATTR_SVCNAME_SECONDARY, (char*)str, 60);
    memcpy(list_sd->titleSpa,str,strlen(str));
    ch = sdp_get_string_attr(rec,
        SDP_ATTR_SVCNAME_PRIMARY, (char*)str, 60);
    memcpy(list_sd->titleEng,str,strlen(str));
}
    sdp_close(s);
    return 1;
}
    sdp_close(s);
    return 0;           //NO MIP
}

What the code basically does is construct the Service Class it’s looking for (MIP_SVCLASS_ID) and adds it to a list. It also tells the SDP server that it’s wants the Protocol List (SDP_ATTR_PROTO_DESC_LIST) for that service.

Then it connects to the SDP Server on the remote device using the SDP library function:

    s = sdp_connect(src, dst, 0);
With the SDP session number that this function returns, the Client will go on to search the Service Class. So it uses:

```c
err = sdp_service_search_attr_req(s, srch, SDP_ATTR_REQ_RANGE, attrs, &rsp);
```

Which using the defined lists from before, it searches for the Service Class, asks for the Protocol List and tells the server to check the whole range of service records in its search. The returned value is the status in the `err` variable and in `rsp` we have the actual response list if everything has gone OK.

The program will then check to see if there has been any success. If so it will get the information it needs from the Service Record. What the Client will need is the Protocol list and the used PSM value for the L2CAP protocol. This is done with:

```c
sdp_get_access_protos(rec, &protos);
ch = sdp_get_proto_port(protos, L2CAP_UUID);
```

These SDP functions get the Protocol list and from that list it retrieves (through parsing) the port number for the `L2CAP_UUID` protocol. Therefore we have in `ch` the correct port number for the Museum Information Point Service on the remote device. This value along with the Bluetooth address and with the Language description titles that come next, are appended to my own defined list for later presentation to the user, this list `my_list_t *list_sdp` will have the necessary information to maintain an idea of what devices surround the Client and the parameters to connect to it.

For the Service Description it simply retrieves the titles in the two defined languages used on the system. Both values will be stored in the list and only one will be presented to the user (if needed) depending on his own language preference. To retrieve the object/service title the SDP functions used are:

```c
ch = sdp_get_string_attr(rec, SDP_ATTR_SVCMNAME_SECONDARY, (char*)str, 60);  
ch = sdp_get_string_attr(rec, SDP_ATTR_SVCMNAME_PRIMARY, (char*)str, 60);  
```

These functions get the Service Names strings at the Primary and Secondary Language offsets. Please note that the Secondary offset is defined for both Server and
Client whereas the Primary offset is Bluetooth Specification defined. Both strings are added to the list and the function returns the corresponding status code.

How the ButtonGetInfoClick() event will process the Information Point Status’ and connect to the Service. It will check to see the amount running Information Points. If only one Information Point is in the vicinity it will assume that it’s the desired Information Point, and the main viewing form will be called in order to retrieve the information and present it to the User.

If more than one Information Point is available it will call the FormChooseMIP dialog form from the Choose.c.cpp file.

FormChooseMIP loads the list_sdp from the previous form and creates a list view with the available Information Points. This list contains the Bluetooth Device Address (for debugging purposes) and the Service Name and Object name in the User’s language. Then the User can choose one of the Information Points corresponding to the object it is viewing.

Either if it’s the only server from the Appform form or either the user chose the object from the FormChooseMIP form. The connection parameters will be taken and the main viewing form MainFormEn will be shown. Also being the first execution a separate thread MainFormUpdateThread will be created that will update the information in regular intervals.

Let’s see how the MainFormEn works. Being located in the MainAppEn.cpp file, this form on showing takes the connection parameters taken from the SDP search and connects to the selected Information Point with them. This Form has similar functions to the other applications that form the system, as what is done now is the sending of the profile and waiting for the Information Point to send back the Objects information. These functions are located in this same file and are called:

```
int do_connect()

do_get_txt1(int s)

do_get_imgs(int s)
```
void do_send_profile(int s)

They do the same as the equivalent functions in the other applications, that is connect, send profile and receive data with the usual acknowledgements. Once all the data is received the information will be presented to the User in the spaces for the text and images in the MainFormEn form.

After the first viewing by the Users the systems turns itself into a pseudo-automatic mode until the user closes the MainFormEn. This automatic mode is done via the MainFormUpdateThread I have already mentioned (MFUpdateThread.cpp file). This thread is created by the previous forms when connecting to an Information Point for first time.

This thread basically does the user steps automatically. First it will wait for a few seconds to allow the user to visualize the first information retrieved, and then will check if the MainFormEn is visualized. If so, it will then proceed to do a Bluetooth Inquiry, Class of Device Filtering, SDP searching and then with the results it will:

- Do nothing if there is no more new Information Points Available
- Change the Information if there is one more new Information Point available different from the previous one.
- Show the Choose dialog for the user to choose if there is more than one Information Points available. This will not close the information form, so the user can carry on reading the previous information if he is not ready to choose another Information Point.

And back to the beginning! As this is the main life of the thread until the program ends. If the thread sees that the user has closed the Main Information viewing window it will not have any effect until the user decides to search for Information again.

This is all the main code of the Client application. For more details please see the actual source code, compile and execute the programs to get a better idea of how everything actually works.
Now I’ll present a diagram of how the actual message passing works between Client, Information Point and Central Server, this is only a basic example with one Information Point:

![Diagram](image)

*Fig. 20 Basic Message passing Diagram*
As we can see, it basically describes the communication between the three types of devices in the Museum System. More detail can be obtained by tracing the execution of the different applications.

The rest of the C files that are part of the Client project correspond to the SDP library source that was included to get around the Open Kylix C++ vs. C problem. Next release of the BlueZ SDP libraries should be fixed and only *libsdp.so* should need to be added to the Project file in order to get the SDP library linked with the Client Application.
Results

In this section I will show the results, screenshots and how the applications work. But first the applications must be compiled, the Bluetooth interface must be up and running and the applications should be executed.

To compile the Central Server, first the sources must be unzipped and later compiled simply with `gcc`.

```
# unzip Museum_Server-1.0.zip
# cd Museum_Server-1.0
# gcc -o museumsvr museumsvr.c
```

For the Information Point:

```
# unzip Museum_Master-1.0.zip
# cd Museum_Master-1.0
# make
```

And finally the Client must be compiled with Kylix (CTRL-F9). The resulting application can be located between the sources.

For the application to work the Bluetooth interface must be up and running on all machines that will be used in the Museum. Please refer to the BlueZ chapter and follow the steps, plug the Bluetooth device in and use `hciconfig hci0 up`. With `hciconfig -a` we should see the Device Address and features. If anything goes wrong during the whole process, please check the system logger (as root) with:

```
# tail -30f /var/log/messages
```

Once everything is in place the applications must be started. First the Central server with:

```
# ./museumsvr
```
We should then see something similar to the following screen:

![Central Server Start up](image)

**Fig. 21 Central Server Start up**

There we can see the museum server starting and waiting on its default port. It will stay like that until a Museum Information Point connects to it.

When an Information Point is started up the first thing it does is to connect to the Central server and retrieve the configuration. Then it will wait for Clients to connect to it and serve the object chosen from the configuration file.

To start an Information Point it is needed to execute the file indicating the correct parameters, that is, the local Bluetooth device name and remote Central Server IP address:

```bash
# master hci0 192.168.0.1
```
The start-up and choosing the object can be seen in the next screenshot:

![Image]

**Fig. 22 Configuration and Choosing of Object**

Now what is needed, are Clients to access the newly created Information Point, we must therefore execute the Client application.

If the deployed Binary and Source package is used:

```bash
# unzip Museum_Client-1.0.zip
# cd Museum_Client-1.0
# ./ProjectMIP.bash
```

If everything is OK and the Borland library has been properly linked with the executable, we should now be able to see the Client Application load up with the Borland GPL licensing message.
The application will now require user intervention. The application should load and resemble the next screenshot:

For the User to carry on playing with the system it is necessary for a User profile to be introduced. This is done through the Profile Change screen. The data to be introduced there is the User’s name, surname and educational level.

The profile to be sent off also consists of a device type which is application defined and the User’s language of preference. The latter can be changed by going into the Language Change screen, at the moment only Spanish and English is defined for the System although it is trivial for it to support more languages.
Screenshots of both are followed:

**Fig. 24 Profile Change Screen**

**Fig. 25 Language Change Screen**
Now the User is ready to go around searching for Information Points and obtaining interesting information about the objects the User is seeing.

To find an Information Point the User must press the “Go Get Information” button. Then the application will do a Bluetooth Inquiry and SDP search on the devices found. If more than one device is found, a menu indicating the Information Points available to the User is shown.

This menu will give the information in the user's own language as we can see in the next images:

![List indicating available services in English](image)

*Fig. 26: List indicating available services in English*

The equivalent Menu in Spanish can be seen in figure 27. This is able due to the SDP registration done by the SDP server and the SDP search made by the Client. Much more information can be stored in the SDP attributes, like Executable URL, webpage, full service descriptions in various languages, etc.
After having selected the desired Information Point, or automatically chosen if there is only one available server, the Client Application will go one to retrieve the information from the Information Point and show it to the User.

Figure 28 and Figure 29 show the screens on the Museum Central Server and Information Point when a Client petition occurs. In these screen we can see debugging and logging information about the petition like profile received or the Clients Bluetooth Device Address.

Figures 30 and 31 show the two Client screens for when the “Mona Lisa” object is chosen for different profiles. More exactly Figure 30 shows the obtained information for an English, Laptop, Advanced petition for the “Mona Lisa”, and Figure 31 for a Spanish, Laptop, Introductory petition. Please note that the Profile is written as a title in the text areas for information and demonstration purposes.
Fig. 28 Museum Central Server under Client/Information Point petition

Fig. 29 Museum Information Point after configuring; receiving Client petition
Fig. 30 Client Main Screen, with English, Laptop, Advanced profile for “Mona Lisa”

Fig. 31 Client Main Screen, with Spanish Laptop, Introductory profile for “Mona Lisa”
As I have stated before the Information will be updated by a thread and will carry on until the user closes the main window. With this I conclude the section with the credits/about screen that can be viewed through the appropriate menu option:

Fig. 32 Museum Client About/Credit screen
**Further development and proposals**

This project wanted to study Bluetooth as a contender Context-aware application network support system. It has certainly been sufficient. But in the development of the application it has been Bluetooth centric, without taking many real implementation aspects that would make the application more feasible for a real museum implantation.

During the development of the application I came across many problems and which would be better solved with more high level solutions, but I opted to keep things simple and center my work on Bluetooth’s role.

Next I will give some ideas I think that would contribute to a end application with real chances for being sold as a Museum Information Assistant Application.

**Caching and Persistence of Information:** Much network traffic would be reduced if Caching techniques were introduced. This would reduce the number of transfers between the Information Point and the Central Server. But it would also create a valid information problem as updates in the Central Server may not be reflected in the Information Points; but this can be resolved with notifying the Information Point and other techniques.

**Use of Web services or ASP/PHP dynamic WebPages; with HTTP over the L2CAP link:** The Client could simply use a modified web browser to access a web server to retrieve the information. The Dynamic pages on the server would do most of the profile formatting and adaptations.

**Distributed computing:** Give a higher level abstraction of the system and see it as a whole: This could allow rubbing out many of the Client/server model limitations. The technology is there and CORBA or .NET/SOAP could be implementation environments to work on.

**Databases instead of text files in the Central Server:** This would give a much higher level of security for the information and a much better way of storing it. It would also make it easier to create maintenance applications on the Server side.
**More Profile parameters:** This would allow a much higher level of quality if the information is adapted much more to a User, also taking into account the history of a User, where he goes and seeing a pattern in his preferences could also infer a better adaptation of the information.

**Instant messaging and employees’ assistance:** Give the User an opportunity to contact an employer of the Museum in case of problems or any other request of Information, thus allowing the User to have the Information Office in his hand.

**Audio and music:** This hasn’t been included due to the lack of SCO support over USB in the current BlueZ implementation. Hopefully in the future it will be possible to add it to the application.

**Use of a world valid GUI library:** Use of Borland’s adaptation of QT would limit the deployment on other systems. A move to QT without the Borland’s extension would be easy and would even extend the possibility for an application on Windows in the future.
Conclusions

I know this current deployed system is not perfect, but it is designed to be only a proof of concept of how Bluetooth technology can be used to design, develop and deploy Context-aware applications. Although BlueZ is still under development and many features are missing (like scatternets or advanced hold-park administration) it is mature enough to be the underlying technology for these types of applications, with a very optimistic future and which I hope it will be a mainstream success. In the end I do think of Bluetooth as a contender for the Title of Network support system for Context-aware Applications King.

In the development I did undergo many problems as I have mentioned, throughout the project but it has certainly been gratifying to solve some of them and frustrating to not solve others when it has been due to the underlying protocol stack, like L2CAP datagrams. Hopefully the implementation in the months to come doesn’t cease to better as it has been doing so in the months I was working on the Project.

But, still it has been very nice to work with new technology and with a new type of applications that everybody is starting to talk about.

Much investigation must be done before we will be able to see good Ubiquitous Computing applications. At the moment just a few experiences and developers are there to prove the basic concept but not yet implemented on a high scale basis. This is likely to come from the telecommunications companies offering location based applications via 3G, or computerized housing applications in a foreseeable future. But still there is a long way to go.

Recently IBM stated that Pervasive Computing is going to be the Future of Computing\textsuperscript{15}, I certainly agree.

And finally I must thank Carlos Calafate for the help he has given me setting up tests and with some implementation problems I had. Without his help I would have probably taken much longer in finalizing this project.

\textsuperscript{15} \url{http://www.silicon.com/news/500007/1/2679.html?et=search}
References and Bibliography

Bluetooth Products, www.bluetooth.com
BlueZ Bluetooth Implementation for Linux, bluez.sf.net
Affix Bluetooth Protocol Stack, affix.sf.net

T.G. Xydis et al. Security Comparison: Bluetooth Communications vs. 802.11
Bluetooth SIG security group Bluetooth Security Whitepaper
D. Kammer et al. Bluetooth Application Developer’s Guide. Syngress
ACM Special Issue of Human-Computer Interaction, Volume 16, 2001
ACM Communications of the ACM. December 2002
Mobilian. “Wi-Fi and Bluetooth: An Examination of Coexistence Approaches”
J. Zyren “Reliability of IEEE 802.11 Hi Rate DSSS WLANs in a High Density Bluetooth Environment”.
Hani Naguib et al. “Middleware support for context-aware multimedia Applications”
Ioannis MATHES, et al. “Context Aware Services in Exhibition Environment- The mEXPRESS approach”
Peter Brown. “Research issues in context-aware retrieval: a working paper on overall research issues”
Scott Mitchell et al. “Context-Aware Multimedia Computing in the Intelligent Hospital”
F. Siegemund “Context-based Bluetooth Scatternet Formation for Ubiquitous Systems”
Henrik J. Asplund. “Pervasive Computing and Bluetooth”
Eugene A. Gryazin. “Service Discovery in Bluetooth”