

rococo™



i m p r o n t o
.....
w h i t e p a p e r

rococo software 9-11 Upper Baggot Street, Dublin 4, Ireland
t: +353 1 660 1315 f: +353 1 660 1328 e: info@rococosoft.com
enabling mobile collaboration www.rococosoft.com

IMPRONTO WHITE PAPER 03.02 © COPYRIGHT 2001 ROCOCO SOFTWARE

INTRODUCTION

MOBILE COLLABORATION: CONNECT, SELECT, COLLABORATE

We are witnessing certain technical trends that have the potential to revolutionise the way we interact with each other and with our environment. These trends, such as pervasive mobile phones and pagers, hand-held computing muscle in PDAs and location based technologies are well understood and comprehensively discussed in technical and mainstream circles. It is, however, worth considering the macro picture. It's worth thinking about how the convergence of these technologies will change our world, and about the opportunities this provides all of us.

Consider two trends: the ubiquity of "computing devices" and the emergence of peer-to-peer computing. There are lots of statistics out there telling us how pervasive mobile phones and PDAs have become. Gartner says, "by 2007, more than 60 percent of people in the United States and European Union aged 15 to 50 will carry or wear a wireless computing and communications device at least six hours a day." According to Forrester Research, "by 2004, one-third of all Europeans - more than 219 million consumers -- will regularly use their mobile phones to access Internet services." Jupiter Media Metrix reports, "U.S. wireless Web users will increase to 96 million by 2005." We don't really need statistics to confirm what we are witnessing ourselves -- portable computing devices are everywhere. They're also getting much more powerful. For example, a Nokia 9210 Communicator has a 200MHz ARM9 32 bit CPU and 42MB of RAM. It's not that long ago that this would have been a spec to be proud of in a top-of-the-range PC!

Meanwhile, in the wired world, peer-to-peer computing is changing the way people interact. The high profile Napster phenomenon was one example. The recent emergence of the JXTA open source peer-to-peer from Sun Microsystems™ is another. These and other peer-to-peer frameworks allow the computers and applications of end users to directly interact and share information and services. They break down the dependence on central infrastructure. Central servers have been the mainstay of most computing technologies for many years -- DNS servers, web servers, database servers, DHCP servers, file servers, print servers, etc. Peer-to-peer computing holds out the promise of a computing framework that allows client devices to become first-class citizens in a network, communicating without the need for centrally-controlled servers.

Ubiquitous computing devices and peer-to-peer networking -- at the confluence of these trends lies *Mobile Collaboration*. Personal computing devices, including mobile phones, will collaborate together to provide a richer service to the user. Using our personal devices, we will collaborate with ticket vending machines to purchase tickets, interact with colleagues in meetings, play multi-player games, swap recipes and print information to whichever printer is close by (regardless of print drivers). To successfully achieve this, an application must be able to *connect* to nearby devices, *select* the services of interest and *collaborate* using those services to get something done. **Connect, Select, Collaborate.**

The trends contributing to mobile collaboration are analogous to the laying down of Ethernet infrastructure on which the web has flourished. The world is currently being "plumbed" for a new way of working. Central to this will be some key technology underpinnings: short-range wireless technology to facilitate robust wireless connections and software frameworks to facilitate compelling applications. Bluetooth and the Java platform are leading the way in these areas. Rococo Software's Impronto product range is a software toolkit that makes it easy to build compelling Java applications using Bluetooth wireless technology.

Two Sample Applications

In order to explore Impronto's benefits in building collaborative mobile applications, consider two sample applications. These applications reflect two common mobile collaboration scenarios. The first, a patient care application, is an example of a *Different Time, Same Place* scenario -- that is, two people collaborate in the same geographical space but at different times. The second, an instant messenger application, is an example of a *Same Time, Same Place* scenario. Both scenarios present their own technical challenges -- we will examine Impronto's role in addressing these challenges.

In our first scenario, Ilsa and Rick are both employees of the Casablanca Coronary Care Hospital. Ilsa is a busy doctor and Rick is an overworked nurse. They both have little time or patience for paperwork and gladly welcomed the hospital's new Patient Care application, built using Impronto. As Rick nurses Ugarte, one of the patients, he enters vital monitoring information into his Bluetooth-enabled Palm Pilot. This information is transmitted using Impronto to the Bluetooth access point on the wall above Ugarte's bed -- a virtual chart. He also provisionally orders tests that he thinks the patient might need, which the doctor needs to sign-off on before they go ahead. Later, Ilsa goes on her ward round. She has instant access, via her Compaq iPAQ, to all information on each patient. As she approaches Ugarte, his virtual chart detects the

doctor's iPAQ and a screen pops up on Ilsa's PDA requesting sign-off for the tests that Rick requested. Once approved, the virtual chart logs the tests with the hospital's central systems, the tests get carried out and Ugarte's insurance company is billed. Ilsa enters a change in medication through her PDA and Rick is automatically notified of this the following morning when he comes on duty.

Figure 1.0: Patient Care application.

	SAME PLACE	DIFFERENT PLACE
SAME TIME		
DIFFERENT TIME	X	

In our second scenario, Annie and Alvy are on a late night flight from Chicago to Washington and, as usual, they can't sleep in those economy class seats. Worse still, they didn't get seats together. They're passing the time by chatting with the Instant Messenger client on their PDAs (using Impronto this works over Bluetooth). Rob spots what they're doing and decides to join in. With a few keystrokes on his PDA he connects to their network, selects "Instant Messenger" and starts chatting.

Figure 2.0: Instant Messenger application.

	SAME PLACE	DIFFERENT PLACE
SAME TIME	X	
DIFFERENT TIME		

So, what kind of technical hurdles must we overcome in order to build these outwardly simple applications? In the case of the patient care application the following are some of the key requirements:

- **Discovery** -- a doctor/nurse must be able to discover the access point device (virtual device) once it's in range and vice versa
- **Authentication and Authorisation** -- only a doctor can change medication and only a healthcare worker with responsibility for Ugarte's ward should access his chart
- **Privacy and Integrity** -- because we don't want someone accessing, or modifying, private patient records as they're transmitted over the air, encryption technology is required
- **Cross Platform Client** -- the patient care client application works on both Palm Pilots and iPAQs (and we may want it to run on mobile phones and other devices)
- **Asynchronous Push** -- the access point can push notifications to client devices when it detects them in range

The chat application shares some of these requirements. Discovery and Authentication seem particularly relevant. It also imposes a requirement of its own, which is particular to this Same Time, Same Place scenario:

- **Coping with ad-hoc networks** -- client devices, like Rob's, might connect and disconnect at various times during the lifetime of the application. This is natural in the world of mobile collaboration and the application must be designed to cope with this.

We'll now examine the architecture of the Impronto product range and of Java APIs for Bluetooth Wireless Technology (JABWT). We will then return to the requirements of our sample applications to see how Impronto addresses them.

IMPRONTO ARCHITECTURE OVERVIEW

The **Impronto Developer Kit** and **Impronto Simulator** are designed to make it easy to develop Java applications that can take advantage of Bluetooth short-range wireless technology. They simplify the complex task of finding Bluetooth devices that are in range of your device, connecting to those devices (including authentication and encryption of the connections) and exchanging data and services between the applications running on the devices.

This is achieved using the standard Java APIs for Bluetooth Wireless Technology (JABWT). Rococo was a member of the Expert Group that set this standard in response to Java Specification Request 82 (JSR-82), and have provided the first commercial implementation of JABWT in the Impronto product range.

Impronto Developer Kit

The diagram below summarises the high-level architecture of the Impronto Developer Kit.

Figure 3.0: Impronto DK architecture diagram.



One of the key features of the **Impronto Developer Kit** is its ability to support multiple underlying Bluetooth Stacks on different Operating Systems, despite the different stack APIs. This is achieved by the Rococo Abstract C Stack (RACS). This is a C mapping layer that provides a common API to the Java components in JABWT. It's this portable kernel that is re-targeted to a particular Bluetooth stack/Operating System combination as required, without modification to the Java components. This makes the Impronto Developer Kit ideally suited to OEM use on various platforms. The RACS API is not exposed to the application developer -- it is used internally for platform portability. Future versions of Impronto may open up this API in order to give the same level of stack portability to C programmers.

The Java platform is actually a series of editions, each tailored to a particular class of device and application. There are three editions: Java 2 Platform, Micro Edition (J2ME - for constrained, often mobile devices), Java 2 Platform, Standard Edition (J2SE - for desktop and server systems) and Java 2 Platform, Enterprise Edition (J2EE - for enterprise class back-end server systems). The Impronto Developer Kit ships with both J2ME and J2SE support (in fact, the J2SE version works in J2EE, which is a superset of J2SE). One of the key differences between J2ME and J2SE is the absence in J2ME of a framework for calling native C methods from Java. Furthermore, providing support for asynchronous calls from C into Java (for example a device inquiry percolating up from a Bluetooth stack) is not a standard feature of J2ME. For this reason, the Impronto Developer Kit includes an Asynchronous Event Manager that provides the necessary support. This allows the application programmer to program in either a synchronous or an asynchronous manner as the application demands without worrying about the underlying infrastructure.

The Impronto Developer Kit provides a set of Java abstractions for use by a Java Bluetooth application developer. These components implement the standard JABWT APIs, making use of the underlying RACS infrastructure to achieve cross-platform support. In addition to the JABWT components, Impronto Developer Kit includes additional ease-of-use abstractions that improve the lot of the application developer. For example, a utility for managing UUIDs and a class for graphically browsing the Bluetooth services that a device supports.

Finally, the Impronto Developer Kit implements a Bluetooth Control Centre (BCC). This is the component that makes it easy to manage the key configuration elements of a Bluetooth device from within a Java application. In particular, it allows the application programmer to control the security aspects of the application without negatively affecting the other Bluetooth applications resident on a particular device. It is the BCC that is responsible for authentication, PIN pairing and encryption control.

Impronto Simulator

The diagram below summarises the high-level architecture of the Impronto Simulator.

Figure 4.0: Impronto Simulator architecture diagram.



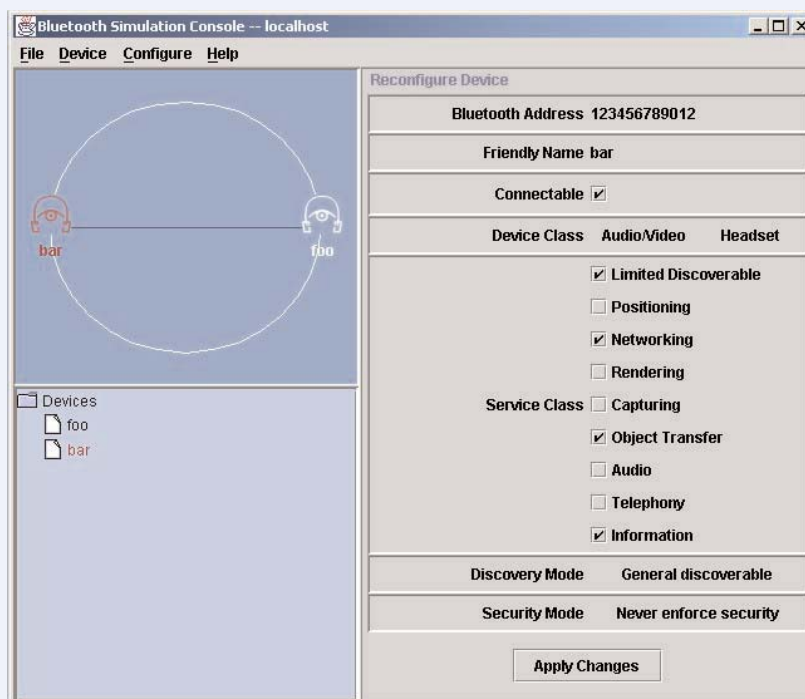
The **Impronto Simulator** is a complete implementation of the JABWT APIs in an emulated environment. Applications may be developed and tested using the Simulator and the same Java code will run unmodified on the Impronto Developer Kit. The Impronto Simulator is 100% Java and it has no dependencies on any Bluetooth Stack or on any Bluetooth hardware. This has obvious advantages for a Bluetooth developer wishing to build and test an application as productively as possible. Within minutes, an experienced application developer can be running Java Bluetooth applications in the Simulator without the distraction of stack and hardware installation and configuration.

The Simulator supports J2ME and J2SE application development. In either case, much of the Simulator infrastructure runs in a J2SE environment - lightweight J2ME support is provided in the form of JABWT "stubs" that allow applications to run in the Simulator framework. As the Impronto Simulator is 100% Java, it will run on any standard J2SE environment. It has been tested against JDK 1.3.1 on Windows and a number of Linux variants: RedHat, Suse and Mandrake.

In order to run in simulated mode, a Java Bluetooth application calls the same JABWT APIs. However, instead of being implemented over a Bluetooth Stack, the Simulator APIs are implemented in the Virtual Stack - the "vstack." The vstack is effectively a virtual Bluetooth device. In the J2SE Simulator, the vstack is collocated with the application code in the application's Virtual Machine (VM). In the case of J2ME, the vstack resides in a separate VM, with a lightweight J2ME XML-RPC stub collocated with the application code. In this J2ME case, a typical development scenario for PalmOS, for example, would involve running an Impronto application within POSE (the emulation tool for PalmOS development). Vstacks communicate with each other using the standard Java Remote Method Invocation (RMI) mechanism. In order to communicate, of course, virtual devices must discover other virtual devices. This is achieved by the application calling the standard JABWT discovery APIs - the implementation of these APIs uses the Impronto Simulator Discovery Daemon to locate other available virtual devices. After a vstack discovers another virtual device, all further communication with that device goes directly between vstacks, i.e. the Discovery Daemon is no longer involved.

During application runs, the runtime behaviour of the application can be monitored and controlled via the Impronto Simulator Management Console. This console presents a graphical representation of the Bluetooth devices that are "in range" and any connections between those devices. By selecting a device in the console, the user may view and modify the various configurable aspects of the device. By double-clicking a connection, they can monitor statistics on the Bluetooth communication link between two devices. The console also provides logging functionality, allowing the application developer to watch the Bluetooth network traffic - for debugging or optimisation purposes, perhaps. The Simulator, like the Impronto Developer Kit, also provides an implementation of the Bluetooth Control Centre, primarily for security management.

Figure 5.0: Impronto Simulator Management Console.



WIRELESS JAVA

Java 2 Platform, Micro Edition

Although Java started life as a programming language developed by Sun Microsystems, it has evolved into a fully-fledged development environment and computing platform. The success of Java - driven by its inherent portability, flexibility and ease-of-use - has made it the platform of choice for new development across many business verticals and across many diverse computing platforms. In 1999, Sun Microsystems divided the Java platform into three editions: Micro Edition (J2ME), Standard Edition (J2SE) and Enterprise Edition (J2EE). The recognition that one size does not fit all gave more focus to those involved in developing Java-based technology.

J2ME is targeted specifically at consumer devices and embedded devices. It consists of a set of configurations and profiles (not to be confused with Bluetooth profiles). A J2ME configuration defines the minimum set of Java class libraries and virtual machine features supported on a particular category of devices. The Connected Limited Device Configuration (CLDC) is the most relevant one for Bluetooth applications as it targets mobile devices with small memory budgets (160KB to 512KB) and wireless, potentially intermittent connectivity. It comprises a fast, small footprint virtual machine (the KVM) and a stripped down but fully-functional Java API subset. A J2ME profile is targeted at application developers. It is layered on top of a configuration and is the set of APIs available on a particular family of devices. Configurations target horizontal market segments whereas profiles target vertical segments. One important profile is the Mobile Information Device Profile (MIDP), which provides a set of User Interface components, a persistence mechanism and a HTTP connection capability for use in mobile phones, PDAs and other handheld mobile devices.

The success of J2ME is evidenced by its widespread adoption by industry heavyweights such as Nokia, Motorola and Siemens as their preferred platform for open application development for their devices.

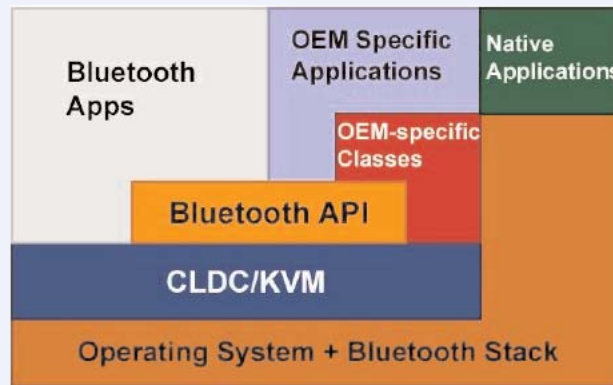
Java APIs for Bluetooth Wireless Technology (JABWT)

The Bluetooth specification is not an Application Programming Interface (API)-based specification. Instead, it standardises a set of protocol stack layers in such a way that the Protocol Data Units (PDUs) passed between stack layers and across the air are well-defined and interoperable. This is appropriate for a wireless technology specification. There are many implementations of the Bluetooth specification, or Bluetooth Stacks, available today. Each of these stacks by necessity has its own proprietary API - usually a C API. An unfortunate side-effect is the complexity involved in writing applications that use Bluetooth technology. A programmer's time will usually be taken up mastering the idiosyncrasies of the underlying Bluetooth stack API, figuring out how to initialise it, configure it and coax it into providing the desired behaviour. Some of the more mature commercial stacks provide some abstractions that aid usability, but these remain proprietary. It is only with the advent of JABWT that a standard API for programming Bluetooth applications exists. Better still, this standard is a Java standard. Many benefits come from this, among them:

- **Portability** - writing an application using JABWT guarantees that it works across all platforms that support this standard. In its best expression, Java achieves something called "write once, run anywhere." A given application once written can run on anything that supports Java according to the standard.
- **Extensibility** - when using a relatively open language and platform such as Java, application programmers have access to many third-party tools and techniques for extending the ability of their application.
- **Mobility** - Java code can be dynamically delivered OTA (over-the air) to mobile devices and then executed locally. This allows developers to develop applications that could be "served" to a customer via a mobile network, a web portal, or via another customer ("pass it on"). The ability to load up new applications "on-the-fly," perhaps disposing of them once used, is important for resource-constrained mobile devices.
- **Developer Community** - the estimated 2.5 million Java developers in the world form a hugely informed and helpful community that helps boost productivity of individual programmers. A 2001 study by the Evans Data Corporation found that over 30% of wireless developers were targeting the Java/J2ME platform.
- **Ease-of-use** - Java is well-suited to providing high-level programming abstractions that simplify complex tasks, thus increasing productivity and letting programmers focus on the application problem domain rather than the infrastructure.
- **Maintainability** - Java, as a language, generally leads to less error-prone code than some lower-level languages. Fixing the bugs that do remain is usually easier in such an inherently object-oriented, high-level programming language.
- **Net-enabled** - Java was designed with the Internet in mind. It is one of the only languages "plumbed" with Internet support at its core.

The JABWT specification was standardised under the auspices of the Java Community Process (JCP), which issues Java Specification Requests (JSRs). A JSR Expert Group is formed from industry leading experts and is tasked with providing both the standard itself and, importantly, a reference implementation of the standard. Standards emerging from this process are, by implication, "implementable." The JSR-82 Expert Group is tasked with standardising Bluetooth APIs for the Java platform. These APIs are an optional package that depends only on CLDC. Rococo Software is a member of the Expert Group, which is chaired by Motorola.

Figure 6.0: J2ME diagram and Bluetooth packages.



The JABWT APIs provide a number of key abstractions that simplify the tasks involved in building a Bluetooth application. In particular, they allow an application developer to initialise a stack, populate and read a Service Discovery database, perform device discovery, manage L2CAP and RFCOMM connections, manage Bluetooth security and exchange OBEX data as efficiently and simply as possible. To achieve this, JABWT uses well understood and widely used Java concepts, such as Listener classes for receiving asynchronous events and Universal Resource Locators (URLs) for identifying connection endpoints. The Input/Output (I/O) mechanism of the CLDC Generic Connection Framework (GCF) is extended by JABWT to include Bluetooth connection classes - `RFCOMMConnection` and `L2CAPConnection`. Since this is a standard networking framework used by all J2ME applications, programmers can quickly produce Java Bluetooth application by applying existing techniques and design patterns.

Sample Application Using JABWT

So, how can we use all of this technology to build our sample applications -- the patient care system and the instant messenger? Lets examine each of our highlighted requirements in turn, and illustrate how Impronto addresses them.

Discovery

There are two aspects to discovery. First, we need to be able to discover devices that are in range of us, and then to discover the services supported by those devices (e.g. a virtual chart service or an instant messenger service). Bluetooth supports this functionality as Device Inquiry and the Service Discovery Protocol (SDP) are core components of the Bluetooth specification. With Impronto, we can use familiar Java paradigms to simply and effectively make use of the power of the underlying Bluetooth wireless technology.

Here's a code sample that illustrates a Device Discovery call in Java using Impronto:

```

myListener listener = new myListener();
LocalDevice localDev = LocalDevice.getLocalDevice();
DiscoveryAgent agent = localDev.getDiscoveryAgent();
try{
    agent.startInquiry (DiscoveryAgent.GIAC,10,listener);
}
catch (BluetoothStateException ex) {
    System.out.println ("device inquiry failed");
}

```

This code sample asks the local device to inquire for devices in range for the next ten seconds and to notify the `listener` object about any devices found. The class `myListener` is also implemented by the application -- it contains callback methods such as `deviceDiscovered()` which allow us to be asynchronously notified about inquiry results. There are a number of things to note in this short code sample. First, notice that Impronto provides an object-oriented paradigm for accessing Bluetooth functionality. For example, the class `LocalDevice` encapsulates characteristics of the device in which the application is running. Using a `Listener` class for asynchronous callbacks is another example of a familiar Java programming pattern. Finally, we can make full use of standard Java exception handling (`try/catch` block) to make it easier to handle error conditions.

With Impronto, we can use this same style to access Bluetooth Service Discovery. Here's a code sample:

```
RemoteDevice devFound; // setup during device inquiry
uuidSet = new UUID[1];
uuidSet[0] = new UUID(0x1234); // identifies the service
int transactionId;
try{
    transactionId =
        agent.searchServices(uuidSet,devFound,listener);
}
catch (BluetoothStateException ex) {
    System.out.println ("service discovery failed");
}
```

Here, we specify the Universal Unique Identifier (UUID) of the service that we are searching for -- the patient care service or the instant messenger service. Impronto makes it easy to construct UUIDs and then to pass them to the `searchServices` method of the `DiscoveryAgent` class, asking it to go and find out which services are supported on the device that we discovered earlier. When the services are discovered, Impronto informs our `listener` object by calling the method `servicesDiscovered` on our `myListener` class. These code samples illustrate the productivity that is possible with Impronto. With more traditional C Bluetooth APIs, the application developer must write code that is longer and more complex (and therefore more error prone) than is possible with these Java abstractions.

Authentication, Authorisation & Encryption

Let's consider all of these security-related issues. The Bluetooth specification supports authentication of a remote device using a 128-bit key that is generated from a PIN code shared by both devices. If the PIN code on both devices doesn't match, the authentication fails. In Impronto, the ability to require authentication of Bluetooth devices is controlled primarily through optional parameters to connection URLs. Servers that are exporting some service use these URLs to identify a connection endpoint and clients use them to specify the service they wish to connect to. So, a server might include the following code, for example:

```
String serversConnString =
    "bt_spp://localhost:3B9FA89520078C303355AAA694238F07;authenticate=true";
try{
    notifier = (StreamConnectionNotifier)Connector.open(
        serversConnString);
    rfcnn = (StreamConnection)notifier.acceptAndOpen();
}
catch (IOException e) {
    /* handle any IOExceptions */
}
```

The "`authenticate=true`" parameter in this code fragment tells the server to require any connecting devices to authenticate themselves prior to connection establishment. The server then goes on to wait for clients to connect to this service, by calling `acceptAndOpen()`. A client connects to a server by calling the method `Connector.open()` with a URL as a parameter -- this URL may also contain the optional `authenticate` parameter.

Bluetooth encryption is based on a stream cipher technique devised by Massey and Rueppel and is carried out at the Bluetooth Baseband level. With Impronto, an application can request that communication over a connection is encrypted using the same technique as for authentication. That is, both the server and the client may specify "`encrypt=true`" as a parameter to the connection URL. Note that encryption depends on authentication -- a connection can only be encrypted if it is a connection to an authenticated device.

Authorisation is also controlled using the same technique; "authorize=true" as the relevant parameter. This parameter specifies that a remote device may only access a server's service if that device is authorised to access the service. The Bluetooth Control Centre (BCC) manages the list of trusted devices that are authorised to access a service. Authentication is a prerequisite of authorisation -- only authenticated devices will be authorised.

In addition to the URL parameter technique of controlling the security of a Bluetooth connection, Impronto supports a method-based alternative through methods of the RemoteDevice class, namely `authenticate()`, `encrypt()` and `authorize()`.

Cross Platform Client

Impronto implements the standard JABWT APIs, which are based on J2ME CLDC. The practical consequence of this is that Impronto delivers on the Java promise of "write once, run anywhere." Any J2ME platform can host the JABWT APIs and an application can run unmodified from platform to platform across PDAs, mobile phones, pagers, or other devices. This is a major advantage to Bluetooth application developers who previously needed to port their application from one Bluetooth stack and Operating System to another.

Impronto achieves this portability through the Rococo Abstract C Stack (RACS). This component is the interface to the underlying Bluetooth stack and Operating System. In addition to unifying the interface to the Bluetooth stack, it is the RACS component that needs to be aware of the threading and event models of the Operating System. As a result, these issues don't directly impinge on the application developer, again simplifying cross-platform development.

Asynchronous Push

Our patient care application makes use of a key feature of Bluetooth and of ad-hoc networks in general. For example, the ability to detect that a device has come into range and to asynchronously push a notification to that device. This capability for location sensitive spontaneity is one of the distinguishing features of our mobile collaboration patient care example -- the virtual chart of our patient, Ugarte, asks Ilsa to approve the tests for him.

To implement this capability in Impronto, the application running on Ilsa's PDA must first make itself discoverable by other Bluetooth devices. Here's how:

```
LocalDevice localDev = LocalDevice.getLocalDevice();
try{
    localDev.setDiscoverable (DiscoveryAgent.GIAC);
}
catch (BluetoothStateException ex) {
    System.out.println ("failed to set discoverable");
}
```

The constant `DiscoveryAgent.GIAC` is the General Inquiry Access Code, as defined in the Bluetooth Specification. The Impronto application running on the virtual chart access point will be coded to periodically carry out an inquiry - using `DiscoveryAgent.startInquiry()` as shown earlier. When it discovers Ilsa's PDA, the access point can then check whether it supports the patient care service (using `DiscoveryAgent.searchServices()` as previously discussed). The next step is to authenticate and authorise the PDA. Now that the access point knows whom it is connected to, it can check whether it has any notifications for Ilsa. To send the notifications, it first establishes a data connection, for example an RFCOMM connection (L2CAP would work fine also). The following code sample establishes a connection to the PDA, using a `ServiceRecord` that was sent to the Listener object during the Service Discovery phase:

```
ServiceRecord record; //assigned during Service Discovery
String url = record.getConnectionURL(
    record.AUTHENTICATE_ENCRYPT, false);
StreamConnection conn =
    (StreamConnection)Connector.open(url);
```

This connection can now be used by the virtual chart access point to send an application message requesting the doctor to authorise the pending tests for our patient Ugarte.

Ad-hoc Networking

Bluetooth is an excellent physical transport for building "ad-hoc" networks (networks without any central administration, whose nodes can dynamically, arbitrarily and continually connect and disconnect). Nodes in an ad-hoc network are usually mobile. Applications written for ad-hoc networks need to use software frameworks that cope with nodes joining and leaving the network at any time. Impronto is such a framework.

Consider our instant messaging sample application -- remember that Rob wants to join Annie and Alvy in their chat room. Taking advantage of Bluetooth capabilities via Impronto, Rob's application does not need to know anything about Annie or Alvy's devices in advance. It doesn't need device addresses, IP addresses, a DNS server, or any other non-ad-hoc artefacts. Instead, if Annie and/or Alvy have set their devices to be discoverable, Rob's application can perform a device inquiry and a service discovery to get all the information it needs to establish a connection to the instant messenger service and start chatting. Similarly, if one of the three friends decides to switch off their device (thus disconnecting from the chat room), the applications on the remaining two devices find out via a `IOException`.

Bluetooth, Jini, Javaspaces, OSGi & JXTA

The Java APIs for Bluetooth Wireless Technology (JABWT) are part of a bigger picture of emerging software infrastructures being designed for ad-hoc, peer-to-peer networks. Many of these infrastructures are Java based. This section contains a brief overview of some of these technologies, all of which would benefit from an underlying Bluetooth network. Impronto is a building block for these types of systems and future Rococo product offerings will build on this platform to provide some of the functionality discussed below.

Jini is a technology for building distributed systems that consist of federations of services. A service can be anything - any functionality that a device, computer or person wishes to expose over a network. It facilitates late binding between a client and a service, as opposed to a client having pre-canned knowledge of a service. This is a key feature of effective mobile collaboration frameworks. Jini also addresses the "finding stuff" problem - how a client bootstraps itself by discovering services that are useful to it. This problem is addressed by the Jini Lookup Service, which allows clients to discover services based on various properties of those services. Other Jini features, particularly the ability of a client to "lease" a service and the event model of communication contribute to making this technology a good candidate for ad-hoc software development.

Javaspaces is closely related to Jini technology. Javaspaces is a Jini service that contains entries. An entry can be any component that makes sense in a problem domain. What the Javaspaces infrastructure provides is a mechanism to do two things: distributed persistence and distributed workflow algorithms. A Javaspaces client can read and write entries to a space, and take entries from a space. The entries are persisted by the space that they are in, transparently to the application. This paradigm is well suited to workflow problems. Consider an application that allowed an office worker to "take" an order entry from the "customer space," to validate it, and then to "write" entries to the "engineering space" for order fulfilment and the "finance space" for billing the customer. Engineering can move the entry on to the Test department, and so on. This paradigm has many uses for both wired applications and, perhaps more interestingly, wireless applications, where an entry could be "taken" from one physical location by a wireless device, and used in some other location.

The **JXTA** platform was originated by Sun Microsystems, who have made it an open source community-based platform. It is designed as a platform that can be used to build standard, interoperable peer-to-peer networks. Currently, peer-to-peer systems, such as instant messaging technologies or file-sharing services, use proprietary discovery technologies and communications mechanisms. JXTA provides a framework that can be shared by such services. JXTA is interesting in the context of ad-hoc networking because of the opportunities it provides for building distributed systems with central coordinating servers.

OSGi - the Open Services Gateway Initiative - is an open Java based specification for delivering managed services to networks in homes, cars and other environments. It defines a software framework based on "bundles" of "services" with the lifecycle, dependencies, configuration and provisioning of those bundles handled by the framework. The transport neutral platform that OSGi provides is a useful abstraction for building mobile applications over Bluetooth.

CONCLUSION

These high-level software frameworks need a platform that will marry them to a wireless infrastructure that will deliver on their potential - infrastructure such as Bluetooth. Rococo's Impronto provides this platform. By allowing developers to develop Bluetooth application in Java, quickly and efficiently, Impronto paves the way for the rapid roll-out of mobile collaborative applications.