Bluetooth Enabled Interaction in a Distributed Camera Surveillance System

Peter Rigole Yolande Berbers
Tom Holvoet
{peter.rigole, yolande.berbers, tom.holvoet}@cs.kuleuven.ac.be
K.U.Leuven, Department of Computer Science
Celestijnenlaan 200A, B-3001 Leuven
Belgium

Abstract
This paper reports on the challenge to extend an existing distributed camera surveillance security application with Bluetooth driven wireless communication using handheld PC’s. The main focus is on the engineering aspect of the development of BlueGuard. The paper gives an in-depth assessment of the chosen open software used and the choices made, on how Bluetooth support was designed in the BlueGuard application, and on how was coped with the technical difficulties encountered while implementing spontaneous ad hoc connections between handheld and PC by wireless PANs. Several new software components of BlueGuard are designed in such a way that they can be re-used in other applications for setting up ad hoc connections over IP.

The distributed camera surveillance security application was built using the SEESCOA (Software Engineering for Embedded Systems using a Component Oriented Approach) component-oriented methodology and is an example of re-use and modularity.

1 Introduction

Wireless personal area networks (WPANs) are short to very short-range wireless networks that can be used to exchange information between devices in the reach of a person. Bluetooth has become a popular communication protocol that can be used in WPAN’s.

BlueGuard is a camera surveillance security application which allows security guards to query suspicious images using their wireless handheld. These images are recorded by cameras scattered around the building; motion detection is used to characterize images as being suspicious or not; the security guards can query the suspicious images that were recorded during the last hours.

BlueGuard is an extension of an existing camera surveillance system; the extension consists of the functionality of the wireless handhelds using the Bluetooth protocol. Exclusively open frameworks and API’s (such as Java and the Bluez Bluetooth stack on the GNU/Linux OS) were used on all devices to implement this extension.

BlueGuard is an example of reuse and modularity: the existing application could be re-used and was easily extended. This was made possible through the SEESCOA component-oriented development, used both for the original camera surveillance application and for the BlueGuard extension.

SEESCOA1, a cooperation project between the KULeuven and three other Belgian universities, developed a component methodology for embedded systems. One of the merits of SEESCOA is establishing a formal definition of a component and introducing an approach in which different components are dynamically connected to form the application ([1]). In addition, a tool to support component-oriented design (the CCOM Composer tool: [2]) and a component runtime were developed.

This paper is organized as follows: section 2 gives an overview of the SEESCOA component methodology, necessary to understand the design of BlueGuard. Section 3 describes the architecture of BlueGuard, and discusses the choices that were made for hardware and open software. Section 4 goes into the details of the design of BlueGuard, introducing the new components, some of which were designed to be re-used in other applications for setting up ad hoc connections over IP. Section 5 zooms in on Blue-
Guard’s interface for the security guards. We conclude in section 6.

2 The SEESCOA component architecture

The SEESCOA project developed a reusable software engineering technique for embedded software using a component-oriented approach. The following sections will briefly present the methodology and the run-time environment.

2.1 The methodology

The SEESCOA modelling methodology offers the following concepts for constructing an embedded application: components, ports, port specifications, connectors and port contracts. The CCOM [2] development tool offers full-fledged support for these concepts in order to enhance the development process. The following overview describes the concepts:

components: are the entities composing the application. There are component blueprints, which are the design-time entities and there are component instances. The latter are instantiated at run-time and perform a certain subtask of the application. Components are isolated from each other, and their only way to communicate is via their port instances. Thanks to these properties, they are highly reusable.

ports: are design time entities which are used as entry points to components. They hold the interfaces to the component they belong to. SEESCOA defines two types of ports: single ports and multiports. Single ports can only accept one connector, while multiports can have multiple connectors attached.

port specifications: are attached to a port: they describe the specification of the port at four levels (see below).

connectors: are the wires between ports, and as such they define which ports can talk to one another. The physical realization of this wire is implemented by the run-time, thus offering a powerful abstraction at design time from the physical connection. Of course, ports can only be connected if their port specifications match.

port contracts: are attached to ports. They offer a way to represent design-time Quality of Service (QoS) specifications at run-time. The run-time system can use them to check and monitor the validity of the QoS requirements.

component contracts: are attached to components. They are analogous to port contracts, but they define QoS specifications relating to the component as a whole (such as the component’s memory footprint).

Using these concepts, a developer is able to construct an application by creating an instance model of his component blueprints and wiring them together by means of connectors. The CCOM tool helps the software engineer to complete the specification and contracts correctly, automatically checking the compatibility on both sides of a connector by using message sequence charts.

We want to stress the utter importance of the specification in SEESCOA since it embodies the run-time reliability and flexibility of the entire application. This is why four levels of specification are included in each port. These are:

Syntactic level: describes the message’s signatures. These consist of the message name and its parameters. In SEESCOA, messages are always asynchronous and thus the signatures never have a return type.

Semantic level: describes the invariants and the pre- and postconditions
Synchronization level: describes the sequence of the messages between two ports in a Message Sequence Chart (MSC).

QoS level: defines all sorts of non-functional properties related to the interaction (such as timing, reliability, resource consumption). Currently these properties are limited to timing constraints attached to the messages in the MSC [1, 3].

It should be noted that these levels extend the two traditional levels of specification of the object-oriented designs paradigm with a synchronization and a QoS level. A port specification example is given in Figure 1. The synchronization specification shows an initial StartSendingKeyStrokesRate message sent from component THIS to component OTHER, followed by a sequence of KeyStrokesRate messages in the opposite direction and finally a StopSendingKeyStrokesRates message finishing the interaction. The QoS specification, on the other hand, shows a periodicity timing constraint.

2.2 The run-time environment

The SEESCOA executing environment (middleware) is able to load, instantiate and link components in order to establish a run-time reflection of an instance model. The run-time also handles the communication between components, thus allowing each component to be independent of all other components. This idea of late binding enables the use of techniques such as re-wiring components or inserting monitors at run-time. Since two components do not know whether they are connected, the run-time system may implement any kind of physical wire between the two. An description of the runtime system, called Draco, can be found at [4]. When an application engineer is designing an application, he simply has to stipulate in the CCOM tool that two components must be connected. The result is a middleware environment that offers flexibility to the application, though it remains hidden for the application developer.

3 Overview of the surveillance system

The camera surveillance case was developed as a proof of concept design for the SEESCOA project. Its implementation proved that the methodology rests on a solid foundation, whose basics were given in section 2. As ongoing research continues to improve the quality and features of the middleware supporting the architecture, new technology and environments are being explored for deploying the applications. This paper is a report on the challenge that was undertaken to extend the existing camera surveillance case with Bluetooth driven wireless communication using handheld PCs. Below, we first discuss the technology and hardware that was used to realize the system, and then provide an overview of the surveillance system design.

3.1 Technology and hardware used

3.1.1 Hardware

The hardware used in the surveillance system basically consists of several PC/104 [5] embedded modules. These are equipped with a FireWire card to which a digital camera (Sony DFW-VL500) is attached that can be controlled (such as zooming, adjusting brightness and focus) through the FireWire link. The embedded module itself is quite small and would easily fit into a shoe box. Half of its 16 megabyte ROM memory is used by the system: a small GNU/Linux distribution, a 1.1.8 Java Virtual Machine, the component system (middleware) and the component-based application. The original 32 megabyte RAM had to be doubled, which did not alter the physical size of the hardware. Furthermore, a plain Ethernet connection links the embedded camera module to a central PC hosting a MySQL [6] database for storing selected images. In our test setup, this PC controls a Bluetooth device (a 3COM PCMCIA Bluetooth card was used) that supports the ad hoc link to the handheld devices. Due to the fact that the components of the application can easily be moved, any other host (e.g. the embedded PCs) could be supporting the Bluetooth link.

The requirements for the choice of a handheld device were twofold. Firstly, it was obvious that it needed to be equipped with a Bluetooth device. Secondly, developing Bluetooth enabled applications must be well supported on the host operating system. Therefore, we chose to use the Compaq iPAQ 3870. This handheld has a StrongARM 32 bit RISC processor running at 206 MHz, 32KB ROM, 64KB RAM memory, and a built-in Bluetooth module. Since problems arose in finding a decent open programming interface for developing Bluetooth enabled applications with the pre-installed operating system, WindowsCE was set aside and a readily available GNU/Linux distribution (see also http://www.handhelds.org) was installed on the handheld instead. The Familiar Linux distribution that was used allows us to chose from among three configurations (see figure 2 for the software stack on the iPAQ):

Bootstrap Root Image: Offers the minimum environ-
Figure 2: The software stack on iPAQ

ment for booting the iPAQ with Ethernet, ppp, pcmcia and ssh. This configuration only offers a commandline interface.

X/GPE Bootstrap Image: This configuration has the Bootstrap Root Image, an X Window system, and a tiny Xserver, and it offers the GTK+ graphical libraries for GUI creation.

Opie Bootstrap Image: In addition to the Bootstrap Root Image, this configuration also has an alternative embedded graphical toolkit called Opie [7], which is based on Qtopia [8]. Opie directly accesses the Linux framebuffer interface, in contrast to X/GPE, which uses an Xserver.

Since common Java Virtual Machines (JVM) use an Xserver, the X/GPE option was chosen. Some alternative JVMs, however, directly access the framebuffer (such as Wonka AWT [9]). Among the many choices (Kaffe, Jeode, Wonka, Savaje, Japhar, Blackdown), the Blackdown JVM was chosen because it is a complete JVM that supports AWT/Swing and because it is well integrated in the Familiar project.

3.1.2 Bluetooth

Bluetooth [10], named after King Harald Blatand, a 10th century Viking who reunited Denmark and Norway, has become a popular communication protocol for short-range electronic data transport. It uses the 2.4 GHz ISM (Industrial-Scientific-Medical) band, which is available world-wide, and has an action radius of about 10 meters. It achieves its robustness in interfering environments (microwaves, wireless phones, etc.) by using frequency hopping. The transfer rate allows asymmetric channels of 723 Kbps or symmetric channels of 434 Kbps, which is good enough for most mobile applications. Bluetooth has some basic security mechanisms built-in, which minimize abuse and data sniffing in the environment of the Bluetooth device. A more exhaustive overview of Bluetooth can be found via [10].

3.1.3 BlueZ

BlueZ is a Linux-based protocol stack for Bluetooth which offers a partial implementation of the complete Bluetooth specification. The layered structure of BlueZ is shown in Figure 3. The bottom layers are the hardware and the driver layer. The Virtual Host Controller Interface (VHCI) can be used for testing without a physical device. The BlueZ core layer abstracts the hardware-dependent layer through a uniform interface. This interface is used by the different protocols such as L2CAP (Logical Link Control and Adaptation Protocol) which is a Reliable datagram protocol. SCO (voice socket) and HCI (Host Controller Interface) sockets are also provided. The applications are located in the top layer.

It did not seem that using these protocols directly from within the Java-based application would be a simple matter. There exists a standard Java API for Bluetooth, which is defined in the Java Specification Request JSR82 [11]. Unfortunately, there are no implementations available that are based on BlueZ, nor for other open Bluetooth stacks. In addition, Motorola (which has an implementation for JSR82) informed us that implementing the JSR would take about one man year. Even a partial implementation would require too much of our time, leaving not much choice but to look for another solution. The BlueZ PAN (Personal Area Network) profile offered an alternative. The use of BlueZ profiles overcomes the burden of implementation via the lower level protocols. It easily initializes a network between two or multiple Bluetooth devices, in which one device plays the role of controller and other device(s) play(s) the role of clients. These clients are called PAN Users (PANUs) and the controller Group ad hoc Network (GN) controller. Once discovery is ready and the peer-to-peer connections are set up, PAN makes it possible to transmit all data over IP. As soon as IP addresses are assigned, Java can open a plain socket for performing its communication tasks without knowing anything about Bluetooth. Although the PAN profile had only just been developed when we first used it at the end of 2002, only a few problems were encountered, the most important being the "initial ping" problem. This bug blocks all initial communication from the PANUs until the GN first sends some data to the PANUs.

The installation of BlueZ is very straightforward, both on PC and on the iPAQ. BlueZ is integrated into the Fa-
miliar distribution [12] for the iPAQ and is installed via precompiled packages. Apart from some minor problems, BlueZ seems to work seamlessly both on the PC and on the iPAQ. The one exception was that the 1 Mb/sec bandwidth was never reached with the iPAQ. This limitation is apparently a hardware issue on the iPAQ, where the Bluetooth device is accessed through a virtual COM port, which limits the maximum throughput.

3.2 Surveillance system design

The components depicted in Figure 4 are those that are deployed on the embedded camera devices. The *Camera* component plays a central role, grabbing and broadcasting images from the FireWire camera. This component uses native code for accessing the Linux FireWire subsystem by calls through a Java Native Interface [13]. This stream of images passes through *VideoStreamDecoder* components that transform the images to the requested format (size, color scheme, etc.). The *CameraMD* component analyses the images it receives and detects motion in order to select which images should be stored and which images should not. It sends an alarm message to the *Switch* component as soon as it finds suspicious images. When the *Switch* gets closed, images flow through its *OutputStream* port over the network towards the *BGVideoRecordIn* port of the *BGStorageController* component on the PC hosting the database (see Figure 5). The *BGStorageController* uses a database-specific *Storage* component for handling the storage and querying of the images. The *BGDesktopDistributor* component redistributes messages coming from the mobile Bluetooth enabled devices (via the *AdhocConnector*) towards the other components in the system; this aspect is discussed in section 4.

4 BlueGuard design

BlueGuard, the Bluetooth extension of the original surveillance application, provides additional components for managing ad hoc connections between the iPAQ and the (embedded) PCs. These are:

**AdhocConnector**: Manages the Bluetooth connection and data transfer.

**BGDesktopDistributor**: Injects the data coming from the PC-side *AdhocConnector* to the right PC-side component.

**BGIpaqDistributor**: Analogous to the *BGDesktopDistributor*, this component manages the distribution of data from the iPAQ *AdhocConnector* to the other iPAQ-side components.

**EventHandler**: This component is responsible for storing and consulting the alarm events on the iPAQ, whether they have already been confirmed by the user or not.

**BGClient**: Implements the graphical user interface for the iPAQ, which provides the core functionality on the handheld PC.

Figure 6 gives an overview of the components that are located on the iPAQ handheld device.

4.1 The AdhocConnector

Figure 7 gives a global overview of the interaction of the *AdhocConnector* (also called Bluetooth component) with its environment. Both the iPAQ and the PC have a master controller for loading, initializing and linking its components. The Bluetooth component sets up connections when possible and routes all data it receives to the combiner dispatcher component.

4.1.1 Technology independent

This combiner dispatcher component is application specific, the *AdhocConnector* on the other hand, can be reused in any other application for setting up ad hoc connections over IP. Thanks to the PAN profile, the component can work completely independently of the physical
Figure 4: Overview of the components on the embedded modules

Figure 5: Overview of the PC side components
datalink layer. The detection of other AdhocConnectors can therefore simply be performed using IP multicasting. UDP/IP packets are used for verifying the connection status, while TCP/IP is used for sending the messages.

4.1.2 Identification

AdhocConnectors, as designed here, must always work in connector pairs. In order to keep two AdhocConnectors belonging to different applications from connecting, an initial identification step is executed based on a connection name and the connector’s gender. Only a male and female connector having the same connection name can connect. This way, no difficulties will arise when the AdhocConnector component is reused in different applications. Its ports and their messages are defined as follows:

Settings port

in SetID(String:<ConName>, String:<Gender>): Sets the connection name and the gender of the port. Normally, this should be done only once when initializing the component.

Data port

in ReceiveData(Object:<Command>): Receives data from a connected AdhocConnector. The data is encapsulated inside the serializable Command object.

out SendData(Object:<Command>): Sends data to a connected AdhocConnector.

Status port

out ConnectionStatus(Boolean:<Status>): This message returns the connection status.

The behavior of the AdhocConnector is characterized by the finite state diagram given in Figure 8. As soon as initialization is ready, it starts looking for a compatible ad hoc connector to connect with. The state pattern [14] was used for spreading the responsibilities of each state. One of the problems encountered while implementing this component was that the receive method on the IP sockets was blocking the threads executing them, which is unacceptable when this occurs with a thread allocated to the component by the underlying component middleware system. This is why it was implemented using an asynchronous thread change between a local component thread
and the usual system allocated thread. Another problem arose when the IP number of a specific network card had to be found out through Java. The JDK (Java Development Kit) 1.3.1 can only query the local machine address, which is not sufficient when more than one interface is available, such as a Bluetooth device alongside an Ethernet card. The solution was to use IP bridging (IEEE 802.1). This can be done very easily in GNU/Linux by adding a bridge network device that bridges one IP address to all other network devices. We learned, however, that the JDK 1.4 is able to acquire the IP address of any network device through the `NetworkInterface` class in the `java.net` package. A final problem was that Bluetooth did not detect broken connections in the IP-layer and this function therefore had to be performed manually. For this reason, UDP lifesigns are frequently sent out for monitoring the connection status.

4.2 The BG Distributors

The main task of the distributors is to encapsulate, decapsulate and route messages between the AdhocConnector and the other components. This encapsulating means adding information so that the receiving distributor knows to which port it has to deliver the message. This is the reason why the distributors are application specific. In the new version of our component system which is currently under development, called Draco [15], the AdhocConnector will sink from the application layer to the middleware layer in order to abstract the type of connection between components. This will avoid the need for a distributor component because the links between components are managed in the middleware layer, thus allowing the AdhocConnector to do the encapsulation task dynamically.

4.3 The EventHandler and BGClient

The discussion of these components will be limited since they simply contain the application logic at the iPAQ side. The BGClient displays the graphical user interface on the iPAQ screen and handles the major part of its functionality, which is to display data and to forward user input to the distributor component. The EventHandler’s job is to manage both the confirmed and the not yet confirmed alarm events for the user.

5 BlueGuard use and evaluation

The main aim of the Bluetooth enabled surveillance application is to achieve an easier and faster interaction for the security guards with the fixed cameras inside the building they have to protect. The combination of distributed hardware and flexible software makes this possible. The equipping of all embedded camera modules with an additional Bluetooth device automatically inserts location awareness into the system. As soon as a security guard comes nearby a camera, he can automatically request all alarm events having occurred during the last hour. The manual checking of the events the intelligent camera system found suspicious can be done easily and quickly on-site by the security guard. In the following sections, we...
explain how this is done on the iPAQ with the application.

5.1 Managing events

The BlueGuard graphical user interface, which is subdivided into three tabs, nicely fits on the iPAQ display. The first tab, which is shown in Figure 9, is used for managing alarm events. In the center lower section of the interface, a traffic light is depicted, showing the status of the Bluetooth connection. As soon as the light turns green, the user can update the list of events. As soon as this list is downloaded and displayed, the security guard can select an event and request to have a look at it. All events that the guard considers to be normal, can be confirmed. These events will not be shown on any later event update.

5.2 Viewing camera images

When a security guard requests to view a camera image, he has to switch to the View tab to see the downloaded image. See Figure 10.

5.3 Controlling a camera

In the final tab, as shown in Figure 11, the security guard can change certain settings of the camera, such as its brightness, zoom and sharpness. All changes are executed at once.

6 Concluding remarks

This paper described the support for mobile ad hoc interaction with a distributed camera surveillance application using the Bluetooth protocol: BlueGuard. The paper showed how existing open frameworks and API’s (such as Java and the Bluez Bluetooth stack on the GNU/Linux OS) can be used to successfully build an embedded and wireless application on top of off-the-shelf hardware. The paper also reports on the successful use of a component-oriented methodology: An existing application built using the SEESCOA component-oriented methodology was easily extended with new functionality. Several new developed components for BlueGuard were designed to be re-used in other applications for setting up ad hoc connections over IP.

Acknowledgement

We would like to thank Gert Bemindt and Peter Bracke who developed parts of the system described in this paper,
in the context of their master thesis.

References


[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Massachusetts, 1994.