Solving Bluetooth Deficiencies through Publish and Subscribe Systems

by

Jessica Yu-Tien Huang

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2004

Author . . . .
Department of Electrical Engineering and Computer Science
May 26, 2004

Certified by . . . .
Larry Rudolph
Ford Professor of Artificial Intelligence and Computer Science
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Solving Bluetooth Deficiencies through Publish and Subscribe Systems

by

Jessica Yu-Tien Huang

Submitted to the Department of Electrical Engineering and Computer Science
on May 26, 2004, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

Research in pervasive computing aims to fully integrate computing devices into our everyday environments in a seamless and efficient manner. Wireless technology such as Bluetooth takes us a step closer by replacing traditional cable connections with a more seamless communication transport, adding mobility and more human-centric computation.

However, if we are to fully integrate this technology, we must first address some of its shortcomings, particularly those with respect to areas with a high density of Bluetooth devices. Four of these shortcomings: susceptibility to anonymous attacks, poor power management, synchrony requirement, and lack of friendships stem from the tight coupling of device interactions during the discovery protocol.

One solution to this problem uses the advantages of publish and subscribe systems to decouple this interaction between smart mobile devices. Available devices can announce their availability to a central controller while devices interested in finding others can announce their interests. When a match occurs, the controller notifies both parties and provides information on how they can form a direct connection.

This solution preserves the functionality of the Bluetooth connection protocol while circumventing the four shortcomings. The assumptions it makes are reasonable when placed in a the context of personal computing environment. Future considerations include optimizations that utilize caching, improvements on performance, increases in system capacity, and solutions for including dumb devices.

Thesis Supervisor: Larry Rudolph
Title: Ford Professor of Artificial Intelligence and Computer Science

Acknowledgments

I thank my advisor, Larry Rudolph, for his guidance and patience during the year. I also thank Debbie Wan for providing information on her group keys research and I thank the other members of the Oxygen Research Group for creating a friendly and fun working environment. In addition, I thank my friends, Devon, Maya, Uttara, for the encouragement to finish this thesis.

Above all, I'd like to thank my mother and sister for their love and support during my years at MIT. Without them, I would not be where I am today.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In an era dominated by wireless technology, Bluetooth is fast emerging as the standard for short-range wireless communication. Due to its low cost, low power consumption, and operational robustness, this protocol has become widely recognized as the future replacement for cables. Unfortunately, the rising density of Bluetooth-enabled devices has revealed a number of problems associated with the technology. Therefore, before we can fully incorporate Bluetooth into everyday computing, we must address its major shortcomings.

This thesis deals with the shortcomings of the Bluetooth connection protocol, particularly the discovery procedure. First, the protocol allows anonymous attacks on discoverable devices, a trend that recent news has reported as spreading very fast among mobile Bluetooth devices. These attacks target devices left in discoverable mode and could result in the compromise of confidential data. Second, the protocol causes devices participating in the discovery procedure to expend power unnecessarily when the device they are looking for is not available. The available devices that reply to these searches waste power too because their relies are ignored. Third, the protocol requires devices to be synchronized in order to discover one another. If a device is not listening at the exact time and frequency that another is inquiring, then they cannot find each other. And fourth, the protocol requires devices who wish to be available to be so to everyone. There is no mechanism in the Bluetooth connection protocol that allows a device to be available to only its "friends".

One solution uses the publish and subscribe paradigm to address these shortcomings. In this scheme, responsibility for device discovery is shifted to a central entity, who handles information on availability and interests and notifies devices when there is a match. The system uses this middleware to provide a simple yet effective solution. The rest of this chapter describes the motivation for this thesis and gives an outline of later chapters.

1.1 Motivation

The field of pervasive computing has become a strong leader in computer research. Pervasive computing, also known as ubiquitous computing, aims to create the next generation computing environment, one with computer and communication technologies available at all times [1]. As the IEEE Pervasive Computing Organization states, the "essence of this vision is the creation of environments saturated with computing and wireless communication, yet gracefully integrated with human users" [2].

To achieve this goal, we must break the traditional interface between humans and computers. Typical interactions rely on physical closeness to a computer and are limited by wires and the connections they create. Computing in this environment revolves around the computer, forcing humans to adapt their lifestyles accordingly. Instead, computing should adapt to normal human activity, which requires mobility. With the introduction of cost efficient wireless technology, such as Bluetooth technology, we are now able to replace both short distance wires as well as less reliable forms of wireless communication such as infrared. This allows us to embed mobile devices seamlessly into our everyday lives.

1.2 Outline

Chapter 2 gives background information about Bluetooth, publish-subscribe systems, and related work. It provides an overview of Bluetooth technology and describes the shortcomings that this thesis will address. Next, the chapter gives an overview of publish-subscribe systems, including variations, and sites examples of different systems.

Chapter 3 lists the requirements necessary in a system that seeks to solve the Bluetooth shortcomings identified in Chapter 2.

Chapter 4 presents a solution in the form of a publish-subscribe system that improves upon the Bluetooth connection protocol. It details the protocol and assumptions of the system.

Chapter 6 discusses how the system meets the requirements listed in Chapter 3. It also describes design alternatives and why they were not optimal for this system. Next, it discusses the choices of parameters in the system and considerations for future work.

Chapter 7 describes the contributions of this thesis.

Chapter 2

Background

Understanding the problem requires a solid understanding of the Bluetooth technology, particularly those procedures related to device discovery and connections. This knowledge can then be used to identify the shortcomings of existing Bluetooth technology and to recognize the reasons why these shortcomings persist. The next step is to explore the possible solutions to this problem. This requires a solid understanding of the publish-subscribe paradigm. It also requires looking at different variations and examples of publish-subscribe systems.

This chapter provides background information on both Bluetooth and publish-subscribe systems. It details the core architecture of Bluetooth and the General Access Profile, which defines the procedures involved in device discovery and connection management. It also examines four shortcomings of Bluetooth associated with using the Bluetooth connection protocol: susceptibility to attacks, poor power management, synchrony requirement, and lack of friendships. Next, the chapter details the basic interaction model of the publish-subscribe paradigm. It describes the advantages and disadvantages of using different schemes (topic-based, content-based, type-based) and different architectures (centralized, distributed). It also sites examples of publish-subscribe systems. Lastly, the chapter looks at related work involving the creation of group associations among Bluetooth devices.

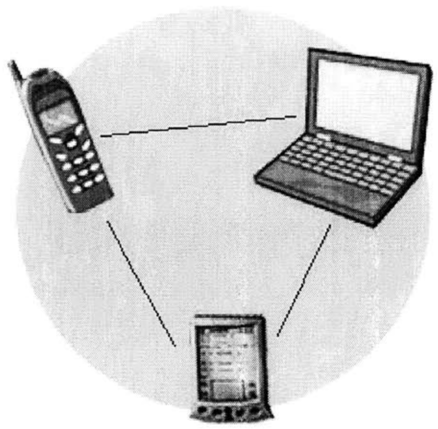

Figure 2-1: A Personal Area Network for communication between mobile devices

2.1 Bluetooth

Bluetooth technology is regulated by the Bluetooth SIG (Special Interest Group), which originally consisted of five companies, Ericsson, Nokia, Toshiba, Intel, and IBM, but has now expanded to thousands of companies. It is a privately held trade association dedicated to the development of wireless Bluetooth technology [3].

2.1.1 Overview

As described in the Intel Technology Journal, Bluetooth itself "encompasses a simple low-cost, low-power, global radio system for the integration of mobile devices"[4] . The basic Bluetooth network, known as piconet, is an ad hoc network that has a capacity of up to eight actively transmitting devices. Scatternets are formed by overlapping up to 10 piconets, resulting in the connection of up to 80 active devices. Bluetooth was developed for three main uses: to connect a computing device to a communication device, to connect a computing device with its peripherals, and to create Personal Area Networks (PAN) for mobile devices (see Figure 2.1) [5].

In the late 90's, supporters of the Bluetooth SIG publicly announced a long-term target mark of \$5 per chip. After a long struggle of about six years, it seems that Bluetooth chips are finally close to hitting this mark. Because it provides a low-cost replacement for cables, Bluetooth will soon become prevalent in every computing environment [6].

2.1.2 Core Architecture

Bluetooth radios operate in the unlicensed Industrial-Scientific-Medical (ISM) band at 2.4GHz. This band is divided into 79 different channels of 1 MHz size each. To reduce the effects of fading and interference from other radios, Bluetooth radios randomly hop through these channels while transmitting and receiving data at a bit rate of 1 Megabit per second (Mb/s). By default, radios transmit within a range of 10 meters, outputing 1mW of power. This range can be increased, up to 100 meters, by increasing the power output to 100mW [7].

The fundamental form of communication in Bluetooth is through a shared physical channel. This channel is accessible only to devices that are synchronized to the same clock and frequency hopping pattern. A frequency hopping pattern is a pseudo-random ordering of the 79 available frequencies. It is algorithmically determined according to a device's Bluetooth Device Address, a global identifier assigned to every Bluetooth transceiver. The clock offset of a device serves as the offset into this pattern. The device that provides this synchronization reference is known as the *master*, and all other devices on the channel are *slaves*. Together, this network of devices is known as a *piconet* [7].

Data transmissions on these physical channels use a Time-Division Duplex (TDD) scheme. In TDD, a single channel is shared by two streams of data going in separate directions. This is done by allocating different time slots during which the streams can send data [8]. The physical channel of a piconet is allocated in this same way; it is divided into time slots of 625 us during which a device can transmit a packet of up to 2,745 bits in length. Even time slots are typically reserved for master transmissions while odd time slots are for slaves [9].

Time slots on a physical channel are organized to create physical links between the master and each slave. This is done via Bluetooth frames, which consist of a master transmission followed by a slave transmission as shown in Figure 2.2. Unless they have negotiated otherwise, a slave may only transmit when the master addresses it in

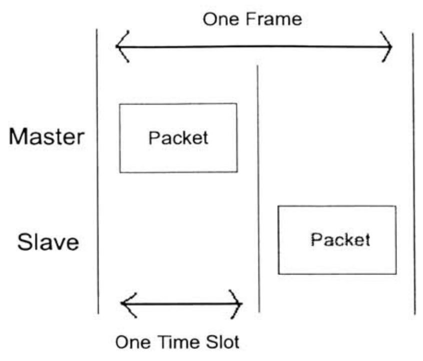

Figure 2-2: A single frame consisting of a master transmission in the first time slot and a slave transmission in the second

a frame, thereby forming a physical link from the master to the slave. Slaves do not form physical links directly to each other. After a frame, all devices in the piconet hop to the next frequency in the frequency hopping pattern. The typical time slot of 625 us therefore results in 1,600 hops per second [9].

Physical links support two types of data transfer: synchronous connection oriented (SCO) and asynchronous connectionless (ACL). A SCO link allocates periodic frames during which a slave device is free to transmit data without being requested by the master. A piconet can support up to three SCO links of 64,000 bits per second each. An ACL link only allows a slave to transmit data in response to a request from the master. Communication on the piconet typically occurs as follows: First, the master listens on the channel at frames separately reserved for SCO transmissions. Once that reserved period has passed, the master queries each slave with an ACL link to see if they have data to send. By default, all active slaves have an ACL link to the master [7] [9].

All Bluetooth devices can become the master of a piconet or a slave in another device's piconet, depending on its frequency hopping sychronization. In Bluetooth networks, devices can participate in more than one piconet; a device can be a slave in multiple piconets or a master in its piconet while a slave(s) in other piconet(s). However, a device cannot be the master in more than one piconet. Being master of multiple piconets would imply that these piconets all have the same synchronization

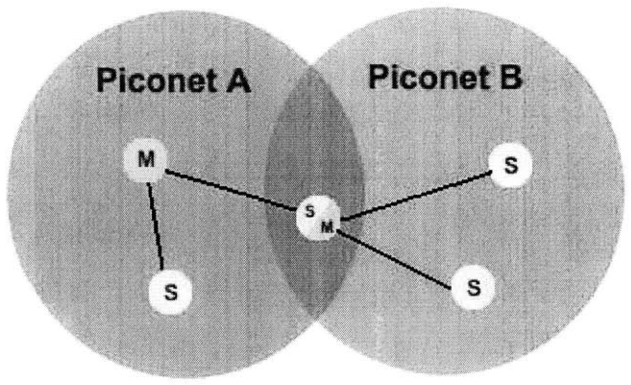

Figure 2-3: An example scatternet topology formed from two piconets A and B, where the master of A is also a slave in B

and therefore share the same physical channel. Figure 2.3 shows an example of a scatternet topology in which a device participates as a slave in one piconet and a master in another.

2.1.3 General Access Profile

The Bluetooth General Access Profile defines the set of operational procedures a device must follow in order to connect to another device. Because Bluetooth networks are ad hoc, the device must first discover neighboring devices that are available for connection. Then, the device targets an available device and connects by synchronizing that device to its piconet. The General Access Profile also defines the operational modes a slave device can enter. Ordered from highest to lowest power consumption, these modes are: active, sniff, hold, and park.

Discovery

A device performs the discovery or inquiry procedure to find out what nearby devices, if any, are available. In this procedure, an available or discoverable device listens for inquiries on a special physical channel, while an inquiring device actively sends inquiry requests on the same channel. When a request is heard, the available device responds

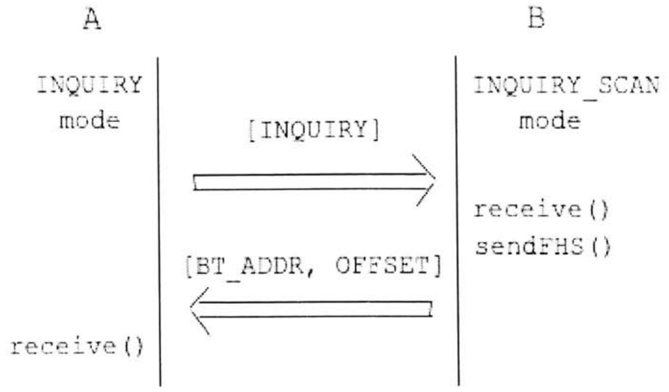

Figure 2-4: Transactions between an inquiring device A and a listening device B

by providing its Frequency Hopping Synchornization (FHS). A FHS packet is a special control packet that consists of a device's Bluetooth address and clock offset. Once this is obtained, the inquiring device can also ask for the device name and device class of the available device. Figure 2.4 shows the transactions of an inquiry procedure.

The special physical channel reserved for this procedure is called the inquiry scan channel. The inquiry scan channel has a slower hopping sequence and comprises fewer frequencies (32) than a piconet channel. A discoverable device listens on this channel by scanning through the frequencies, listening to each for 10 ms and hopping to the next every 1.25 seconds [4]. An inquiring device sends requests by also scanning through the frequencies, transmitting and listening for responses at each. However, because an inquiring device has no previous knowledge of this channel, it must pseudo-randomly hop through the possible frequencies. It takes advantage of the slower hopping sequence by hopping at a faster rate, increasing the probability that a request is heard [10].

Paging

Once a device discovers its neighbors, it can target an available device to connect to by. This is done by performing the paging procedure, which is similar to inquiring. In this procedure, a connectable device listens for pages on a special physical channel, while a paging device actively sends pages on the same channel. When a page is

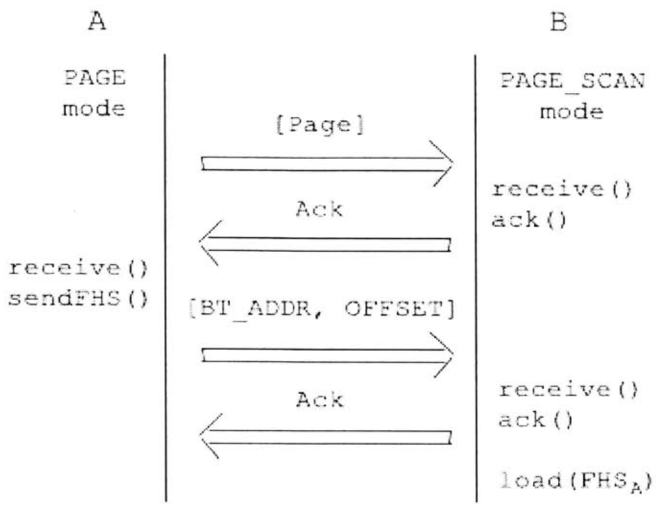

Figure 2-5: Transactions between a paging device A and a listening device B

heard, the connectable device responds with an acknowledgement. The paging device receives this and sends its FHS packet. The connectable device acknowledges this again to inform the paging device that it is now joining the piconet. The device then uses the FHS packet to calculate the frequency hopping pattern and synchronizes to the piconet of the paging device. Figure 2.5 shows the transactions of a paging procedure.

The special physical channel reserved for this procedure is called the page scan channel. Similar to inquiry scan channels, page scan channels consist of a unique sequence of 32 frequencies. A connectable device listens on this channel by scanning through the frequencies every 1.25 seconds and listening for 10 ms on each [4]. The paging device sends pages by transmitting and listening for responses at each frequency. If a paging device has no prior knowledge of this channel, it must psuedo-randomly hop through the possible frequencies in the same manner as in the inquiry procedure. However, prior knowledge of this channel can be obtained from the FHS packet acquired during a previous inquiry procedure. This packet can be used to calculate the frequency pattern of the page scan channel and its offset [10]. Because of this optimization, it takes much less time for a paged device to hear a page than for a discoverable device to hear an inquiry, given that the paged device was previ-

Operation Type	*Minimum Time*	*Average Time*	*Maximum Time*
Inquiry	0.00125s	3-5s	10.24-30.72s
Paging	0.0025s	1.28s	2.56s
Total(paging +inquiry)	0.00375s	4.28-5.28s	12.8-33.28s

Table 2.1: Minimum, average, maximum times of inquiry and paging

ously discovered. Table 2.1 shows the minimum, average, and maximum times for the inquiry and paging procedures [11].

Operational Modes

A slave device defines its level of participation in a piconet by entering one of four modes: active, parked, hold, and sniff. The active mode is the default for a slave while the other three are used when a slave wishes to enter a low-powered state or to define a period(s) of inactivity in order to participate in other piconets.

Active mode – In the active mode, a slave participates in data transmission and is assigned a 3-bit Active Member Address that gives it access to the piconet's physical channel. An active slave can particpate in ACL or SCO transfers with the master.

Parked – The parked mode is the lowest-powered mode for a slave. In the parked mode, a device gives up its AMA for an 8-bit Passive Member Address (PMA). A parked device does not have access to the channel and therefore cannot transfer data. However, it still listens on the channel at a beacon interval reserved for broadcasting to parked devices. The master can still communicate with a parked slave by addressing broadcasted packets to that slave. The situations in which the master uses this beacon interval are: to ask a parked device to become active, to ask if there are any parked devices that wish to be active, or to broadcast data to parked devices.

Hold – The hold mode consumes the next lowest power after the parked. In the hold mode, a slave listens but does not transfer data for a period previously negotiated by the master and slave. This mode is only entered once per invocation after which the slave returns to its normal mode. Slaves with a SCO links, which requires them to transmit at fixed periods, cannot enter this mode.

Sniff – The last mode is the sniff modes, which provides a power level less than the active mode but more than the parked and hold modes. The sniff mode allows a device to define a duty cycle with periods of presence and absence. A slave typically uses this mode to engage in activity in other piconets during periods of absence.

The master of a piconet may choose to park devices to expand its piconet capacity. Because the AMA address space only allows for seven active slaves (an address is reserved for the master), a new device can only be added to the piconet by parking an active slave and reassigning its AMA to the new device. The combination of AMA and PMA allows over 256 devices to belong to a piconet. However, only the eight devices assigned an AMA can transmit data [4].

2.1.4 Problems with Bluetooth

This thesis addresses four shortcomings of the Bluetooth technology: susceptibility to attacks, poor power management, synchrony requirement, and lack of friendships. All four shortcomings are a consequence of device interactions during the Bluetooth connection protocol, particularly the device discovery stage of the protocol. This section details each shortcoming and the protocol feature that causes it.

Attacks in Discoverable Mode

Mobile Bluetooth-enabled devices, such as cell phones and PDAs, have recently begun experiencing attacks when left in discoverable mode. Many popular models of Bluetooth-enabled devices contain vulnerabilities that make them susceptible to these anonymous attacks. According to an AL Digital website, vulnerable phones include: Ericsson T68; Sony Ericsson R520m, T68i, T610 and Z1010; and Nokia 6310, 6310i, 7650, 8910 and 8910i [12]. This section describes the two most common anonymous attacks, bluejacking and bluesnarfing, and how they are currently prevented.

Bluejacking is an increasingly popular phenomenon among Bluetooth cellular phone users. It occurs when one person anonymously sends a message to another without being connected. This attack aims to startle or unnerve a victim, usually by sending physical information about the victim, such as his appearance or current activity, in the middle of a crowded area. A victim receives this anonymous message, knowing he must be visible to the sender, but cannot identify the sender. To perform a bluejack attack, an attacker creates a new contact on his phone and places the message he wishes to send in the "Name" field. He then chooses to send this contact via Bluetooth and searches for Bluetooth-enabled phones within range. Finally, he picks a victim and sends the contact. Bluejacking has spawned a new craze and presents a possible new outlet for spammers. The only way to avoid bluejacking is to make sure a device is not in discoverable mode, ensuring that it can no longer be found by other devices [13] [14].

Bluesnarfing occurs when a person connects to another device without alerting the owner of the request. According to an AL Digital website, the attacker gains limited access to confidential stored data once connected, including "the entire phonebook (and any images associated with the entries), calendar, realtime clock, business card, properties, change log, and IMEI (International Mobile Equipment Identity, which uniquely identifies the phone to the mobile network, and is used in illegal phone cloning)" [12]. Bluesnarfing primarily targets mobile devices that are in discoverable mode. Therefore, to avoid the attack, device should not be discoverable to other devices. A few models remain vulnerable even when not in discoverable mode and have no option but to turn Bluetooth off to avoid bluesnarfing [15].

Power

One of the main concerns of wireless communication is power management. Unlike wired devices which typically connect to their own power source or to another device with a power source, most wireless devices require mobility and therefore run on batteries. Because of this limitation, conservation of power is a top priority for wireless devices.

One way to conserve power is to minimize the number of transmissions a Bluetooth device must perform. Because transmissions consume much power, unnecessary ones should be eliminated. Unfortunately, the Bluetooth connection protocol promotes just the opposite when used in Bluetooth-rich environments. During the inquiry procedure, all available devices must respond to an inquiry request. While these transmissions ensure that an inquiring device can discover its neighbors, they become particularly wasteful when the inquiring device is looking for a specific target. Because the responses of other available devices are ultimately ignored, the battery power of those devices is wasted. Even worse, an inquiring device might periodically repeat these inquiry requests if the target device is not present, resulting in repeated response transmissions. This poses a real power management issue in areas dense with Bluetooth-enabled devices participating in the inquiry procedure.

Another way to conserve power is to reduce the amount of time a device spends listening for transmissions. During the inquiry procedure, an available device must spend its time scanning the inquiry scan channel for inquiry requests. This level of activity consumes much power when there are no immediate inquiries.

Synchrony

While Bluetooth devices can connect freely with respect to physical location (they only have to be in range of each other), they are limited in synchronization during discovery. This requirement states that a device can only discover another device if it sends inquiry requests at the same time and on the same frequency that an available device is scanning. Both devices must be present and using the same communication channel. This requirement greatly limits the options of an inquiring device. If the specific target device is not present, an inquiring device must repeat its request periodically in hopes that the target shows up and is available at a time that coincides with its request. Therefore, staggered inquiries and inquiry scans can impede the discovery process.

Friendships

The Bluetooth connection protocol does not allow for the formation of friendships. A friendship is a privileged relationship where one device is exclusively available to another. A device can restrict its availability to a pre-determined set of friendly devices, but remain invisible to strangers. There are several reasons a device might like to do this: it might not wish to provide unique and traceable information about itself to unknown devices or it might wish to avoid attacks from unfriendly devices while remaining available to its friends. Unfortunately, the Bluetooth connection protocol does not support this option.

2.2 Publish and Subscribe System

A publish-subscribe system is a communication service that dynamically routes information from their sources to interested parties. The system consists of two components, publishers and subscribers, who exchange information through a server. The publishers, or information providers, publish information to the system while the subscribers, or information consumers, subscribe to information of interest. When information is generated by publishers, the system matches and delivers it to interested parties. This section describes the basic interaction model of the publish-subscribe paradigm. It details the different schemes and architectures along which publish-subscribe systems vary, and cites examples of different systems.

2.2.1 Basic Model

The basic interaction model of a publish-subscribe system allows subscribers to express interest in an *event* or pattern of events. Events contain information that a publisher produces and a subscriber consumes. When a publisher provides an event, the event is delivered to interested subscribers through a notification.

In this model, storage and management of subscription interests and publications are handled by an *event service*. The service receives events from publishers and

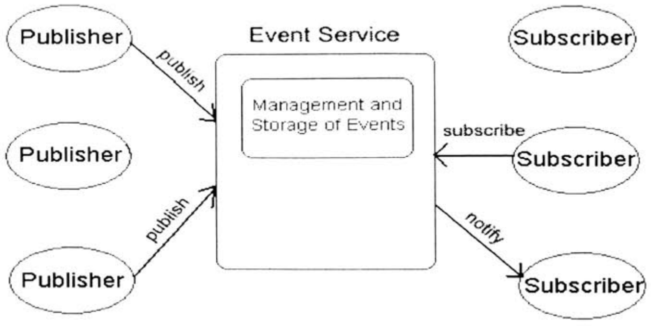

Figure 2-6: Interaction model of a simple publish-subscribe system

efficiently delivers them to interested subscribers, thus serving as a mediator between publishers and subscribers. Subscribers typically register their interest by calling a `subscribe()` operation on the event service. This operation is called without prior knowledge of where events are generated. Subscribers can also terminate a subscription with an `unsubscribe()` operation. Publishers generate an event by calling a `publish()` operation on the event service. The event service then propagates the event to all interested subscribers. In this model, every subscriber receives every event that conforms to its registered interest. Figure 2.6 illustrates this basic interaction [16]

The main advantage of the publish-subscribe paradigm is that it decouples interactions between publishers and subscribers along three dimensions: space, time and synchronization.

Space – Space decoupling is provided by the fact that end users do not need to know about each other. Publishers provide information directly to the event service and do not need to hold references to subscribers. Subscribers do not need to hold references to publishers either because they receive the information directly from the event service. Therefore, unless the information contains end-to-end semantics, publishers and subscribers remain anonymous to each other.

Time – Time decoupling is provided by the fact that publishers and subscribers do not need to be actively participating, i.e. present, at the same time. For example, a publisher might publish an event before a subscriber arrives or a subscriber might receive notification of an event after the publisher of the event has disconnected. This is because once an event is published, the event service takes control of it, either storing it or forwarding it.

Synchronization – Synchronization decoupling is provided by the fact that production and consumption of events do not have to occur synchronously. For example, a publisher might publish an event that is stored on the event service until a subscriber comes along that is interested in it. Even if no one is interested in the event initially, it can still be consumed by a subscriber that later registers interest.

There are two fundamental differences between publish-subscribe systems: the expressivity of the subscription language and the architecture of the event service. The scheme a system determines the expressivity of subscriptions. The implementation of the event service follows a centralized or distributed architecture, or a combination of the two [17].

2.2.2 Schemes

In publish-subscribe systems, subscribers are not interested in all events, but rather particular events or event patterns. In the very first publish-subscribe prototypes, e.g. TIB/Rendezvous and Elvin, which were developed over Local Area Networks (LANs), publishers would simply multicast information and leave it to each subscriber to filter out information that was not of interest [18] [19]. When publish-subscribe systems made the moved to Wide Area Networks (WANs), it was no longer practical to multicast. Instead, an event service took on the responsibility of filtering and delivering events of interest to subscribers. These are many possible ways to do this, but most fall under one of three schemes: topic-based, content-based, and type-based.

Topic-Based

The topic-based publish-subscribe scheme was the earliest scheme to emerge. Systems that follow this scheme use the idea of topics or subjects to categorize and filter events, resulting in a topic-based filtering scheme. The scheme essentially creates a communication channel for each topic and delivers information provided at one end of the channel to parties at the other end. Topics are identified by key words. With this scheme, clients in the system can publish events to individual topics and subscribe to individual topics. An individual topic can be viewed as its own event service to which clients publish and subscribe.

This scheme is ideal for systems that can statically categorize events into a fixed set of groups. It is an abstraction that is very easy to understand and requires little overhead or filtering at the event service. Unfortunately, this scheme does not provide expressive subscriptions. Subscribers who express interest in a specific topic must receive all events for that topic, often resulting in more filtering at the subscribers end. An improvement to the traditional topic-based scheme uses hierarchies. Hierarchical topics allow clients to organize topics based on containment relationships, providing more options for specifying topics. Topic names are also allowed to contain wildcard values, which can be thought of as describing an entire subtree or a specific level of topics in a hierarchy. IBM's Gryphon is an example of a system that uses hierarchical topics [18].

Content-Based

The content-based scheme allows more expressivity in subscriptions by filtering based on the content of the event. Unlike topic-based systems which use predetermined subjects, systems that employ this scheme filter events based on the dynamic properties of an event, either internal attributes or associated meta-data.

To express event interest, subscribers must provide a criterion or filter for expressing event constraints to the event service. These constraints serve as subscription patterns that can be represented several ways. Three representations are string, template object, and executable code.

Strings – Strings are the most frequently used representation because they are the simplest. String filters must conform to a subscription language that the event service understands. This subscription language is usually in the form

```
String filter = "news == 'REGIONAL' and location == 'BOSTON'
                 and month == 'MARCH' and year == '2004'";
Subscription sub = new Subscription(filter);
EventService.subscribe(sub);
```

Figure 2-7: Example code of using a string filter

of attribute-value pairs that use basic operations such as $=, <, <=, >, and >=$. These pairs are then combined using operations like AND and OR to form a string which is parsed by the event service. Figure 2.7 shows example code for using a string filter. More complex subscription grammars include SQL (Structured Query Language) and XPath [20] [21]. The Intentional Naming System (INS) is an example of a system that uses a simple language based on attribute-value pairs for resource discovery in dynamic networks [22].

Templates – Template objects are another representation for expressing a subscription pattern. Subscribers must provide a template object to the event service, indicating that it is interested in every event that conforms to the template. Events that conform are those whose attributes all match the corresponding attributes of the template, except for attributes carrying a wildcard value.

Executable Code – The last representation for a subscription pattern is in the form of executable code. Subscribers provide an object to the event service that is able to filter events at runtime. This representation is not often used because the implementation of the object is left to each subscrber, making it difficult to optimize the system as a whole.

The content-based scheme is the most common employed scheme because of the expressiveness it allows in describing event interest. Unfortunately, this requires a lot of overhead from the event service, who must pass events through filters in real time to match them with interested subscribers. Much research has been done on developing efficient and scalable matching algorithms.

Type-Based

The type-based scheme is a recently proposed scheme. It emerged from the observation that many topic-based systems were categorizing events based on both content and structure. Systems that employ this scheme filter events based on the structure or type of the event, which can also lead to a natural description of content-based filtering if types are defined by the content.

The advantages of this scheme are that it is simple to implement and preserves type encapsulation, unlike the template-based approach which considers event types to be dynamic properties. This scheme is particularly useful for object-oriented systems. An example is the Distributed Knight, a tool for synchronous, collaborative distributed modeling. It integrates a type-based publish-subscribe scheme with object-oriented languages to model events [23].

2.2.3 Architecture

Publish-subscribe systems follow either a centralized or distributed architecture. A centralized architecture consists of a central entity that manages messages using the client-server model. In this model, an event server receives, stores, and forwards events while clients publish or subscribe or both. Each system has exactly one event server, resulting in a star topology, as shown in Figure 2.8. Systems that follow this approach, like the IBM MQSeries queuing system, are built on a central database [24]. This architecture is suitable for a system that requires reliability, data consistency, or transactional support, but does not need high data throughput [17]. Unfortunately, it does not scale well due to the bottleneck and single point of failure that the central server presents. The distributed architecture follows a peer-to-peer model, which has no central entity. In this model, all nodes are equal and can act as publisher, subscriber, or event service. The role of an event service requires a node to store or forward events it receives. Because every node has some service functionality, there is no bottleneck or single point of failure. This architecture is advantageous for fast and efficient deliver of transient data, such as encountered on the Internet [16]. The

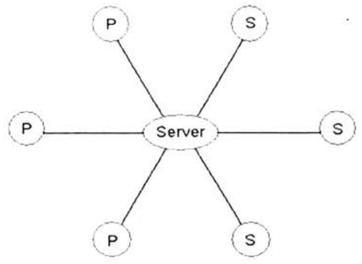

Figure 2-8: Star Topology

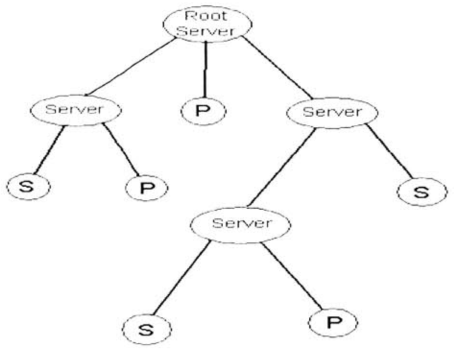

Figure 2-9: Hierarchical Topology

TIBCO Rendezvous example mentioned earlier also uses this distributed approach [18].

Systems can combine these two architectures to create a client-server models with multiple servers. This results in different topologies such as the hierarchical and ring topologies. Hierarchical topologies arrange event servers in a hierarchy tree, where every server except the root has a parent. These servers act as gatekeepers for their subtree, forwarding along only events that the subtree is interested in. Therefore, event servers follow the same protocol for their server links as they do for their clients. The ring topology arranges servers into peer-to-peer relationships in the form of a ring. Servers communicate to each other via a different protocol that allows them to exchange subscriptions and publications [16]. Figures 2.9 and 2.10 illustrates these two topologies.

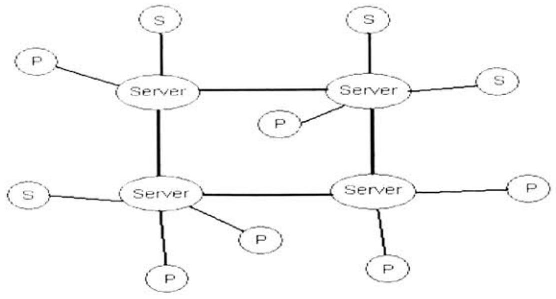

Figure 2-10: Ring Topology

2.3 Related Work

The Oxygen Research Group at the MIT Computer Science and Artificial Intelligence Laboratory is currently researching possibilities for creating groups among Bluetooth devices. The hope is to allow two Bluetooth devices that belong in the same group to be able to verify each other's membership and bypass the authentication step when setting up a connection. Currently, a protocol has been developed for creating groups, adding devices to groups, and verifying membership in groups. This section describes this protocol.

Groups are implemented using private and public keys. To participate in groups, each device generates four keys: a device_private_key, a device_public_key, a group_private_key, and a group_public_key. Private keys are known only to a device while public keys are available to everyone. In a group, there is exactly one owner device, which knows the group's private key and can add new members to the group. Using these keys, every device can be the owner of its own group [25].

Group membership is determined through an encrypted group token, which identifies the Bluetooth device and the group it belongs to. Tokens are obtained in the following manner: First, a device encrypts its unique Bluetooth address with its device_private_key to create a *device token*. Then, it sends the device token and its device_public_key to the owner of a group it wishes to join. The owner decrypts the token with the key to verify that it is speaking to the correct device. It then encrypts the device token with its group_private_key to create a *group token*. This token is sent back to the device and serves as a passport for proving group membership. If the

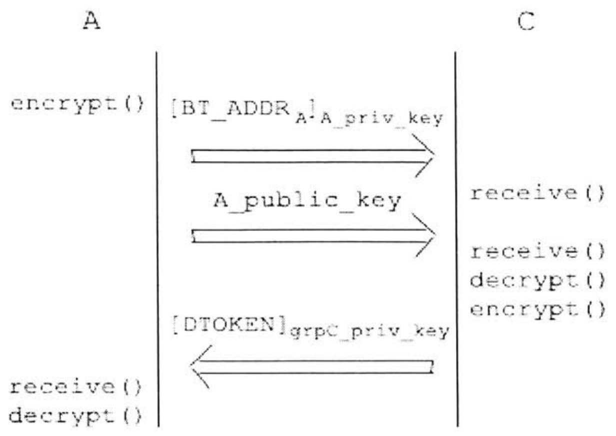

Figure 2-11: Transactions of adding device A to the group owned by device C

device wishes, it can ensure that it was speaking to the group owner by decrypting the group token with the group_public_key to obtain the original device token it sent. Figure 2.11 shows the transactions between a device A that wishes to join the group of device C [25].

The group membership of a device can be verified by any other device. This is done by first decrypting the group token with the group_public_key to obtain the device token. The device token is then decrypted with the device_public_key to obtain the Bluetooth address of the device. If this matches with the address of the communicating device, then membership is verified. Following the example in Figure 2.11, Figure 2.12 shows the transactions between devices A and B, where B is verifying A's membership in group C (owned by device C) [25].

Using this protocol, devices can acquire a group token for each group it belongs to. Tokens are typically kept with the corresponding group_public_keys to verify another device's membership in the same group.

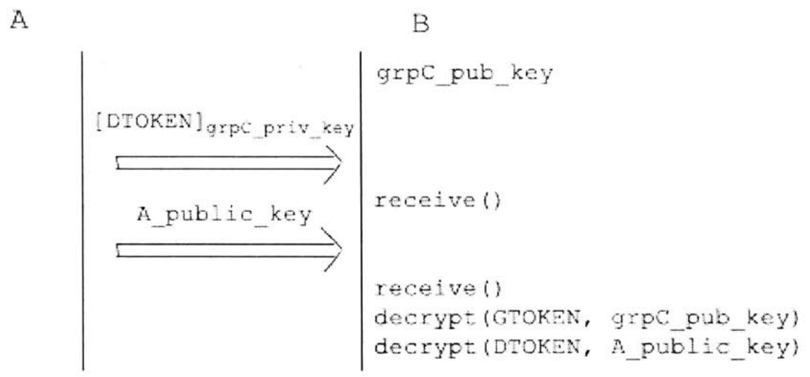

Figure 2-12: Transactions of device B verifying device A's membership in the group owned by device C

Chapter 3

Requirements

This chapter describes the requirements of a system that seeks to improve upon the Bluetooth connection protocol. Such a system would perform no worse than the Bluetooth connection protocol and would improve specifically upon the four shortcomings of the connection protocol as described in Section 2.1.4.

Perform no worse than Bluetooth connection protocol – The system should provide at least the same amount of functionality as provided by Bluetooth. It should allow all devices that woiuld have normally discovered each other on their own to discover each other through the system. In other words, an inquiring device that uses the Bluetooth protocol should be able to find the same discoverable devices through the publish-subscribe system, assuming that those devices have published their availability to the system. Also, the system should provide a subscribing device with no less information about available devices than the device would have received through a Bluetooth inquiry. Lastly, the system should provide devices with no less privacy than the Bluetooth connection protocol. The system should never provide more information about a device than would have been discovered during inquiry, unless the device specifies otherwise.

Prevent anonymous attacks in inquiry scan mode – The system should be able to prevent anonymous attacks. A device should not be vulnerable to bluejacking or bluesnarfing attacks when it wishes to be available to other devices.

Allow devices to expend a lower average power during discovery – The system should improve the power management of devices involved in the inquiry procedure when in Bluetooth-rich environments. On average, these devices should expend less power in order to discover other devices. In particular, the cases to consider are when a device is left in discoverable mode and when a device is inquiring whether a specific device is available. Both these cases expend an undesirable amount of energy in the existing Bluetooth protocol.

Provide asynchronous device discovery – The system should be able to decouple device interactions during discovery. Available devices and searching devices should be able to discover each other through the system, without requiring them to listen and ask for each other at the same time and place. In this case, the notion of place refers to a frequency channel. This requirement eliminates the synchrony criterion of the Bluetooth connection protocol which states that a device can only discover another device by inquiring at the same time and at the same frequency that another device is listening on.

Allow devices to restrict availability – The system should allow the formation of friendships by enabling a device to restrict its availability to a pre-defined set or sets of device groups. The available device must be group member or friend in order to restrict its availability to that group.

Handle inconsistent data – The system should have a mechanism to handle system states in which the information on a client device is inconsistent with information on the server. These can occur when a device disconnects from the system. Inconsistent data should not persist in the system. Information about publsihers that are no longer available should not be given to subscribers because the subscribers will waste resources paging while the unavailable device is not listening. Information about subscribers that are no longer searching for other devices should also be removed since a new publisher might be told to enter page scan mode (listen for pages) if its event matches this old subscription. In both cases, a device wastes power unnecessarily.

Chapter 4

Design

This chapter presents a solution to the shortcomings of the Bluetooth connection protocol described in Section 2.1.4. The purpose of this best-effort system is to set up Bluetooth device connections in a flexible and efficient manner while fulfilling the requirements listed in the previous chapter. Devices that participate in this system are smart mobile devices that can support multiple connections to different classes of devices. Examples are laptops, Personal Digital Assistants (PDAs), and some mobile phones. These devices require an application layer running on top of the standard Bluetooth HCI, or control, layer, in order to locally call HCI commands. These commands are sent remotely from a controller, rather than inputted by the user (as in the normal procotol). This system does not require any modification to the Bluetooth stack, but the controller in this system must have a device name that clearly indicates its role as a controller. Therefore, devices looking for a controller will know when it is found.

The solution uses a content-based publish-subscribe system to facilitate device discovery between smart mobile devices. In this centralized system, a publish-subscribe controller acts as middleware between available devices (publishers) and devices that are looking to connect (subscribers). Publishers must provide self-descriptive information to the controller, which then packages it up into a publication. Publishers have the option of limiting access to their publication, by creating friendships. Only devices verified to be a member of an authorized group can unlock a restricted pub-

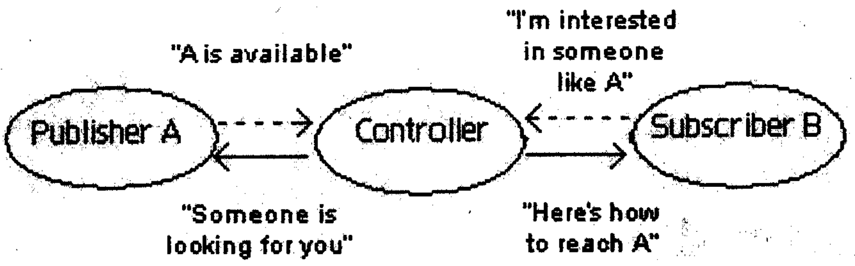

Figure 4-1: Basic interaction of the system. Publishers announce availability and subscribers announce interest in devices (dotted lines). The controller notifies both parties when there is a match (solid lines)

lication. Subscribers must provide information to the controller describing the types of devices that are of interest. This information is packaged into a subscription. In this system, there are two *event types*: publications and subscriptions.

Responsibility falls on the controller to match publications and subscriptions, resulting in the pairing of devices that are interested in connecting with each other. When these matches are made, the controller must *answer* these events by notifying both parties with information on how to form a connection. Figure 4.1 shows the most basic interaction model. Once these events have been answered, they are removed from the system. Open events are unanswered events that remain on the system because they have not yet been matched. Each time a new event or trigger event arrives, the controller seeks to match it against open events through the matching procedure.

The controller keeps track of events by assigning unique identifiers to each one. These identifiers are needed as multiple events can originate from the same device. When information is sent to a device regarding one of its events, there must be a way for the device to recognize which event is in question. These following are situations in which identifiers are used: to identify which event is being answered in a match notification, as acknowledgements for when an event is successfully added to the system, and to identify an event that has expired

The controller manages open events on the system by assigning an expiration date to each. When an open event remains unanswered on its expiration date, it is removed from the system and the originator is informed of the expiration. The originator can then resubmit an event to the system if it wishes. When devices disconnect, the controller also removes their events from the system. If a device disconnected unintentionally, it reconnects to the controller. Otherwise, the device reenters the state it was originally in before it connected to the controller.

This publish-subscribe system allows devices in the system to bypass the Bluetooth discovery stage when initiating connections to other devices (see Section 2.1.3). This saves on power in Bluetooth-rich environments and reduces the chances of an anonymous attack. It provides synchronization decoupling and a mechanism for forming friendships. This chapter presents in detail the protocol for device and controller interactions in the system. It also describes the assumptions this system makes and justifies why they are made.

4.1 Protocol

This section outlines the protocol for devices that comprise the system, including the controller. The controller in this publish-subscribe system is a non-mobile device that draws power from a constant source. It has a Bluetooth radio and is always both discoverable and inquiring about its neighbors. The protocol set forth in this section is for a system with exactly one controller, which is only in connectable mode when there is room for another device in its piconet. Devices in this system are smart mobile devices that have an extra application layer to handle their participation in the system. Section 5.4.3 discusses the possibilities for creatito ng a system that includes dumb devices, i.e. simple devices that only connect one other device and run on simple software.

To join the system, a device may connect to the controller via the standard Bluetooth connection protocol. The device inquires and discovers the controller, if in range. The device name that a controller provides during discovery must be descriptive enough to identify it as a publish-subscribe controller. If the controller is in connectable mode, the device can then page the controller and connects. The device and controller perform a role switch so that the controller becomes master and the device becomes slave.

Discoverable devices may also connect to the system without initiating the connection. Because the controller is continually inquiring, it will discover the new device and page it for its information. That way the device can seamlessly transfer the responsibility of announcing its availability to the controller.

4.1.1 Procedures

This section describes the procedures for publishing availability and subscribing to device types. Both procedures produce a new event, which triggers the matching procedure by the controller. If there are no matches, the event is left open on the system and the controller parks the device. When a match does occur (triggered by a later event), the controller reactives the device and notifies it of a match. The controller reacts differently depending on whether a publication arrived first or a subscription.

Publish

There are two ways to publish availability: directly and indirectly. In the first, the device initates a connection to the controller. After the connection is made, it calls a `publish()` operation on the controller and provides information on its availability. The controller then checks that the publication is valid (see Section 4.1.2). If so, the controller assigns it an event identifier and returns it to the publisher, who stores for later reference. This method is called a direct publication because the device is intentionally using the controller to publish.

In the second method, the controller is the one to initate a connection. As mentioned in the previous section, the controller continually sends inquiries and will find a new discoverable device. The controller then pages the device, obtains its information, and packages it into a publication. An event identifier is assigned and returned

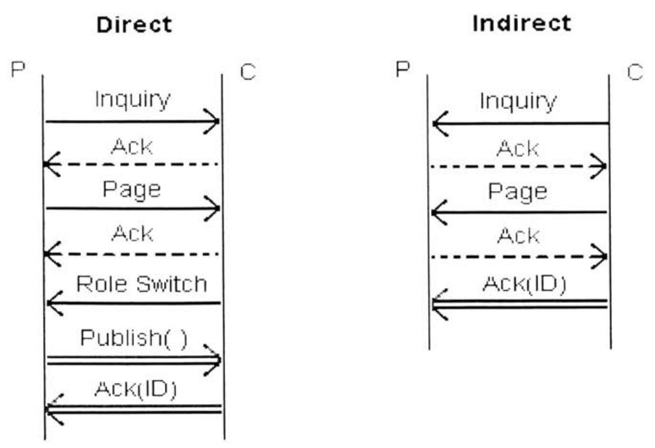

Figure 4-2: Message transactions for publishing directly and indirectly. Solid lines indicate standard Bluetooth actions, dotted line indicates automated responses (see Section 2.1.3), and double lines indicate system specific actions

to the device. This method is called an indirect publication because the device does not intentionally use the services of the controller. Unlike a direct publication, a indirect publication can specify no more information than what is obtained from the inquiry and page. Figure 4.2 shows timelines of message transactions for these two cases.

For both cases, the controller performs the match procedure to search for any open subscriptions that match the publication. If there are no matches, the publication is kept open in the controller's database until expiration. The controller will park the device to put it in power saving mode and will disable any scan modes if applicable. The matching procedure, identifiers, and expirations are described in more detail in Section 4.1.3, Section 4.1.4, and Section 4.1.5 respectively. At any time, a device can remove a publication by calling an `unpublish()` operation on the controller with the event identifier as an argument.

Subscribe

Unlike publishing, there is only one way to subscribe: the device initiates a connection by inquiring and paging the controller and calls a `subscribe()` operation with its subscription interest. The controller checks that the subscription is valid before

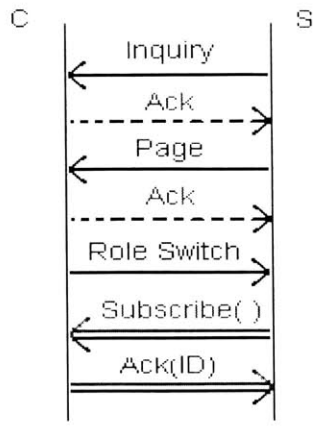

Figure 4-3: Message transactions for subscribing. Solid lines indicate standard Bluetooth actions, dotted line indicates automated responses (see Section 2.1.3), and double lines indicate system specific actions

assigning it an event identifier and adding it to the system (see Section 4.1.2). The event identifier is returned to the subscriber, who stores it for later reference. Figure 4.3 shows the message transactions for a subscribe procedure.

As with publishing, subscribing triggers the matching procedure to see if there are any open publications that match. If there are no matches, the controller keeps the subscription open in the system until expiration. The controller parks the device to put it in power-saving mode and disables any scan modes if applicable. At any time, a device can remove a subscription by calling an `unsubscribe()` operation on the controller with the event identifier for the subscription as an argument.

Parked Mode

When a device's publication or subscription does not result in a match, the controller stores the event and parks the device. As described in Section 2.1.3, parked devices give up their Active Member Address for a Parked Member Address. These devices remain synchronized to the piconet but cannot transmit. This mode is used to put the devices in power-saving mode. When the device needs to transmit, the controller will reactive the device using the beacon interval reserved for communication with parked devices.

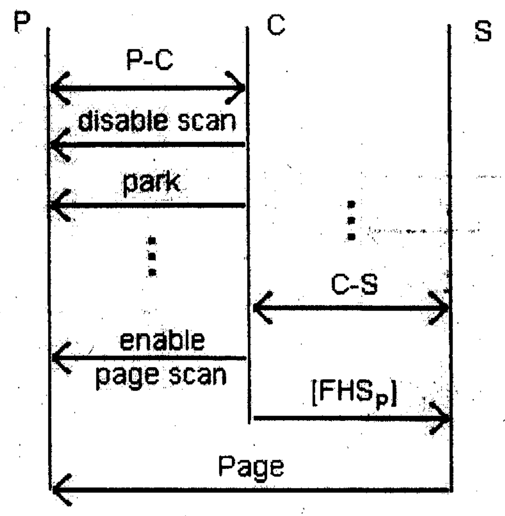

Figure 4-4: Message transactions for matching when the publication arrives first. P-C action indicates either direct or indirect publishing as shown in Figure 4.2 and C-S action indicates subscribing as shown in Figure 4.3

Notification

Matches occur when availabilities agree with interests. When a match occurs, the controller must notify both parties and provide information on how they can directly connect to one another. The subscriber is provided with information on how to page the publisher, while the publisher is told to listen for the page.

The controller reacts differently according to which event type arrived first. If the publisher arrives first, the controller stores the publication and parks the device. When the matching subscription arrives, the controller does not park the subscriber. Instead, the controller sends a message to the publisher to enable its page scan and a message to the subscriber containing the FHS packet of the publisher. Figure 4.4 shows the message transactions when the publication is indirect. A direct publication would also have the same result.

If the subscriber arrives first, the controller stores the subscription and parks the device. As previously described, the matching publication can arrive directly or indirectly. In the first case, the device actively publishes by connecting to the controller. Following this, the controller notifies both parties of the match. In the

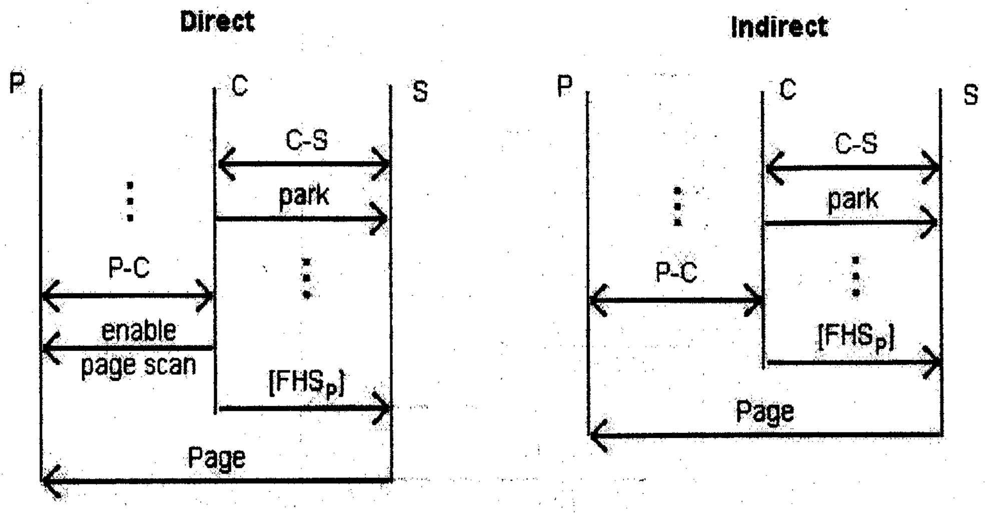

Figure 4-5: Two possible message transactions for matching when the subcription arrives first, one for a direct publication and the other for indirect. P-C action indicates publishing as shown in Figure 4.2 and C-S action indicates subscribing as shown in Figure 4.3

second case, the controller finds the discoverable device and receives its information indirectly through an inquiry and a page. In this case, no connection is needed between the publishing device and the controller. Instead, the controller only needs to send the publisher's FHS to the subscriber. Figure 4.5 shows the message transactions for both a direct and indirect publication.

States

Figure 4.6 shows the state transition diagrams for participating devices. A subscriber has four main states. When it is not connected to the controller, it is in the disconnected state. To transition to the active state, the subscriber must initiate a connection. Once active, it can subscribe. If the subscription results in no matches, the device enters parked state. If there is a match, the device is given information to enter the page state. A publisher has states similar to the subscriber. When in the disconnected state, the device is not connected and is not in inquiry scan mode. The device can transition to active state by initating a connection. Another start state is the inquiry-scan state. The publisher enters the active state by hearing an

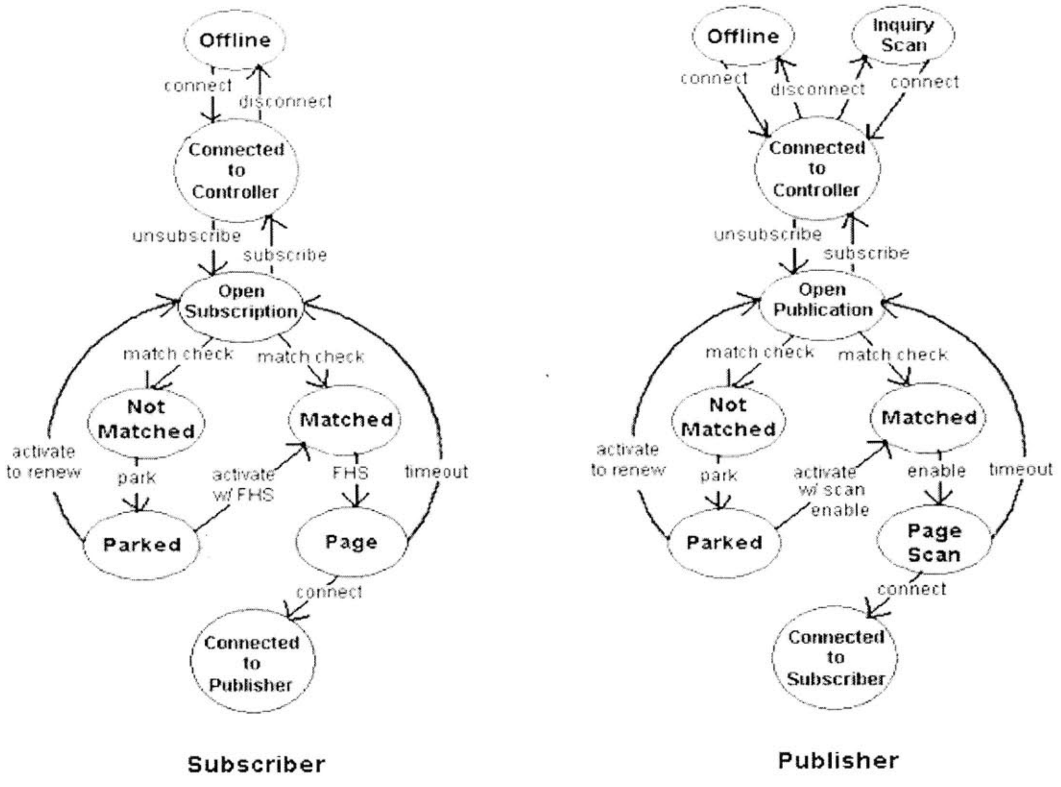

Figure 4-6: State transition diagram for a subscriber and a publisher.

inquiry from the controller and accepting the connection. From the active state, the publisher may publish its information. If there is no match, it enters park state. If there is a match, it enters page-scan state to listen for an incoming connection.

Flow Chart

The controller manages the system using a database of open events and a queue of events that are waiting to be processed. Events in the queue are processed on a First In First Out (FIFO) basis. This section describes the actions of the controller in ths system.

Figure 4.7 and 4.8 show the flow chart diagrams for the five main threads of the controller (cleaning, handling disconnections, handling new connections, handling existing connections, and processing events). The cleaning thread removes expired events from the system while the disconnection thread removes events left by devices that have disconnected. The new-connection thread handles the connection of new devices and the addition of their events to the event queue. The existing-connection thread handles the querying of devices that are already connected to the controller and are parked. This thread is necessary to check if a parked device wishes to renew, add a new event, or remove an event. Finally, the processing thread takes events off the queue on a FIFO basis and processes them by performing a match check. The results and matching procedure itself are described in detail in Section 4.1.3. Because these threads operate simultaneously and modify shared data, locks are used. The cleaning, disconnection, existing-connection, and processing threads all modify the database of open events while the existing-connection and new-connection threads modify the queue of events. A thread must wait for the appropriate lock to be free before executing its actions. When it is done, it must release the lock.

Figure 4.8 also shows the flow chart for the notification action. Devices are always parked after a notification. The only time a device is active for long in the system is when it is a new device that is waiting for an event to be processed. The thread for the processing of events will park all devices involved.

4.1.2 Subscription Language

This section describes the subscription language for publications and subscriptions. It gives the scheme and representation of the language and the reasons for these choices. It also describes the attribute-value pairs used to create an event.

Representation

To allow for expressiveness of events, this system follows the content-based scheme described in Section 2.2.2. Because events are essentially device descriptions, the system needs a language flexible enough to express all the different possible descriptions of a device or device type. Of the main schemes of publish-subscribe systems, the content-based scheme provides the most expressivity. A topic-based scheme is too limiting as it requires the system to categorize events into a fixed set of topics. This

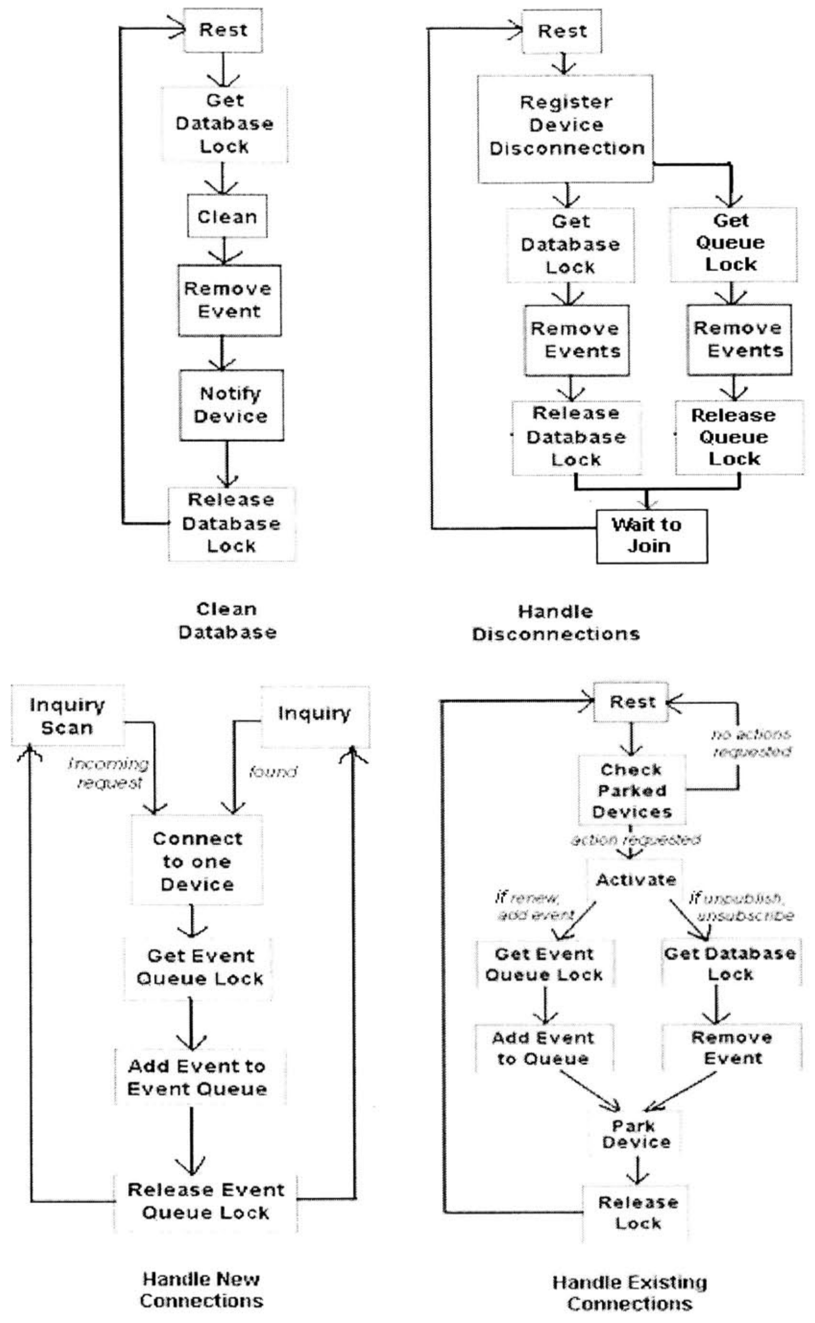

Figure 4-7: Flow chart diagram for the controller threads that handle cleaning, disconnections, new connections, and existing connections.

restricts the granularity at which a device can perform a search since a search can only vary according to one parameter: the topic. A type-based scheme is also unsuitable for this system due to the unnecessary complexity it adds. Publication events in

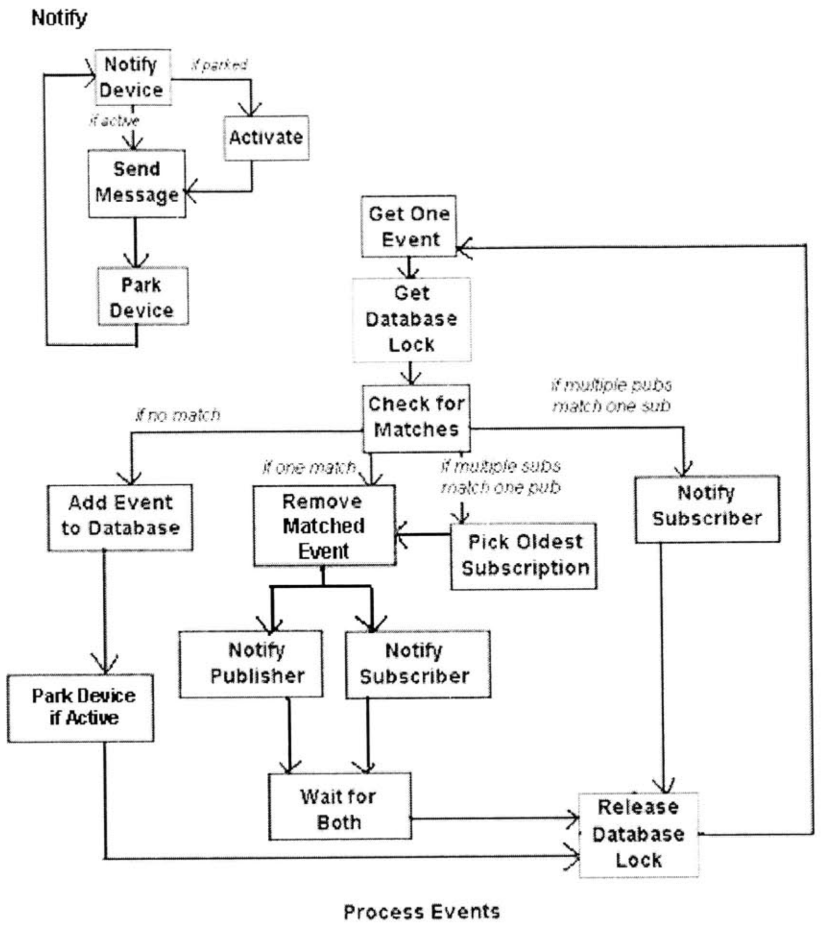

Figure 4-8: Flow chart diagram for the controller thread that handles event processing and the notification action

this system are device descriptions that do not fall into statically configurable event types. Therefore, filtering in the context of event types is not useful. The downside to a content-based scheme is that it places more overhead on the controller when performing matches. However, this is an allowable tradeoff since the controller has its own power supply and is assumed to have many resources (see Section 4.2).

Subscriptions in the system define event constraints using attribute-value pairs. This is ideal because information obtained during a Bluetooth discovery procedure is already in the form of attribute-value pairs. For example, a discovery procedure uncovers the name value, class value, and Bluetooth device address value of a device. Each piece of information can be easily formatted into an attribute-value pair and combined to form a subscription. With these pairs, it is then reasonable to follow the template object approach described in Section 2.2.2. Publications can be thought of as objects and subscriptions as templates for object types. Any object that conforms to the template is therefore of interest to the subscriber. A possible alternative representation uses strings. However, because this system incorporates the groups protocol described in Section 2.3 to allow for friendships, strings are impractical. The group verification protocol is too complex to express using strings.

Publications

The purpose of a publication is to provide information about device availability A publication consists of five attribute-value pairs: [KEYS=(group keys), BT_ADDR=(Bluetooth Device Address), OFFSET=(clock offset), NAME=(device name), and CLASS=(device class)]. In this system, a device is not allowed to publish multiple times because a device should be discoverable in only one way, just as there is only one inquiry response from each discoverable device participating in the standard Bluetooth inquiry procedure. Allowing multiple publications from the same device could lead to simultaneous matches. In that case, the controller would need to have a scheduling algorithm for answering matched subscriptions. Without one, the controller would answer all matches, resulting in simultaneous page requests from different subscribers sent on the same channel (see Figure 4.9). This greatly increases the chance of packet collisions. Introducing a scheduling algorithm, however, adds too much overhead and a level of complexity undesirable in a simple controller. Therefore, devices are limited to one publication each.

Publications contain device information that could have otherwise been discovered through a Bluetooth discovery procedure. This consists of the Bluetooth device address, clock offset, device name, and device class. All these parameters, except the clock offset, help identify a device. Typically, the user of an inquiring device chooses to connect to a device based on user-friendly attributes such as the name and class of a device. While the BT_ADDR is a unique global identifier for devices, it does not present information that easily describes devices. However, this attribute is included

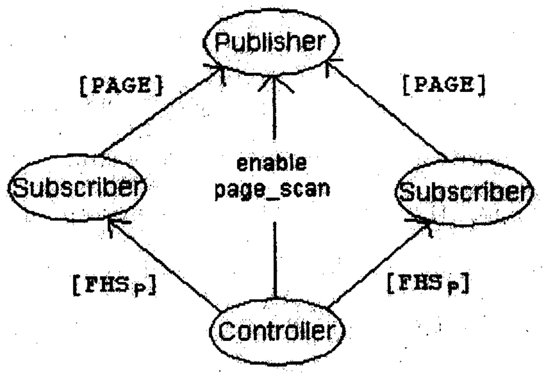

Figure 4-9: Allowing multiple matches to one publication results in packet collision

in the publication to leave the option of locating a device by its unique identifier. The clock offset is only included in the publication to keep device information stored in a single location for easy access by the controller. Though this attribute is largely ignored during a match, it remains in the publication alongside the BT_ADDR so that they can be easily packaged into a FHS packet when needed. This packet contains information on how to page and connect to a device.

A publisher also has the option of limiting or restricting access to its publications by specifying groups as describe in Section 2.3. By providing specific group_public_keys, the device indicates that it wishes to be available exclusively to members of those groups. Therefore, only subscriptions that have a key in common with a restricted publication can access it. Of course, the controller must verify both the publisher and subscribers membership in the group before allowing access. These steps are done during a matching procedure (see Section 4.1.4). Keys can also be represented as attribute-value pairs. Unfortunately, keys are different than the previously mentioned device properties in that they can have multiple instances. Since a publisher can belong to more than one group, it should be able to specify multiple keys to permit access to more than one group. Therefore, the value of the KEYS attribute is a list that contains either one or more group keys or a wildcard value (NULL). A wildcard value indicates an unrestricted publication. The reason for listing keys together rather than having multiple attributes (KEY1, KEY2, etc.) is that the system does not know how many groups a publisher belongs to or wishes to provide access to - different publishers might specify a different number of keys. Thus, a list is used to enforce a more consistent format across all events.

Subscriptions

The purpose of a subscription is to provide the system with information about the types of devices a subscriber is interested in. The format of a subscription is very similar to that of a publication. A subscription consists of five attribute-value pairs: [KEYS, BT_ADDR, OFFSET, NAME, and CLASS]. In this system, a subscriber is not allowed to add a repeated subscription. However, a subscriber is allowed to add up to six unique subscriptions to the system. When a match occurs, the subscriber becomes the master of the publisher (see Section 4.1.4). Since a device can only have up to seven active slaves in its piconet, the system allows it to look for no more than seven devices at any given time. Unfortunately, this does not ensure that a subscriber will not waste controller resources, since it can still add up to seven subscriptions even while the subscriber's piconet is at full capacity. However, this simple solution does limit the amount of waste to a reasonable extent for a system where controller resources are not scarce.

To access restricted publications, a subscriber can specify the groups it belongs to by providing a list of group keys as the value of its KEYS attribute. Doing so indicates that the subscriber is a member of those groups and can therefore access a restricted publication that is locked by any one of those keys. If the subscriber has no keys, it sets the attribute to a list containing the wildcard value (NULL). Again, the controller must verify these memberships before allowing access. The values of the BT_ADDR, CLOCK, NAME, or CLASS attributes can also be NULL value, implying that the subscriber does not care what a publication has for that value, i.e. the controller can match any publication value for that attribute. Subscriptions typically have an unspecified CLOCK value since it is not an identifier. If the controller were to check for matches on that attribute when a value is provided, then it is highly unlikely that a match would be produced. Therefore, the controller ignores the CLOCK attribute during a match (see Section 4.1.3).

4.1.3 Matching

The controller stores collections of publications and of subscriptions, or events. Every time a new event arrives, the controller checks for matches in the system. This new event is known as a trigger event because it triggers a matching procedure call. If the trigger event is a publication, the controller checks the publication for a match against the collection of subscriptions. If the trigger event is a subscription, it checks for a match against the collection of publications.

A match is defined as a consistent pairing of two events of opposite types, i.e. publication and subscription. To determine if there is a match, the procedure checks if the values of all attributes in a publication are consistent with the values of all corresponding attributes in a subscription. At any point, if two corresponding attribute-value pairs are inconsistent with each other, the match check fails for that element and the procedure moves on to next element in the collection. If all attribute-value pairs are consistent, then the procedure determines that the overall publication and subscription match and places it aside until it has found all matches in the collection.

A matching procedure can result in zero matches, exactly one match, or multiple matches. In the first case, the trigger event produced no matches and is added to the database as an open event. In the second case, there is exactly one match, so the controller can answer both events. This *answer* consists of informing the event originators how to communicate with one another in order to set up a connection. Once these events are answered, the controller removes them from the system. In the third case, the trigger event produced multiple matches. The controller reacts differently depending if the trigger event is a publication or a subscription. If it is a publication, the controller selects the oldest of the matching subscriptions and proceeds as if it had found an exact match. If it is a subscription, the controller returns the entire list of matched publications.

Matching Procedure

The matching procedure is called when a trigger event arrives at the controller. This procedure takes as its arguments the trigger event and a collection containing elements of the opposite event type. This procedure iterates through the elements of a collection, individually checking for matches between the trigger event and each element. Elements that are successfully matched are placed in a separate list during the iteration. At the end of iteration, the procedure returns this list to the controller.

A publication and subscription match when their attribute-value pairs are consistent. The first and most difficult attribute to match is the KEYS attribute. Matching this attribute is a two step process, involving the determination of common groups and the verification of membership in those groups. Checking for common group_public_keys determines common groups. However, because any device can claim to be in a group, the controller must verify membership before the attribute-value pairs are considered consistent. Only one commong group is necessary to gain access to the publication. This can be thought of as unlocking a room with n different doors, each locked by a different key. The first correct key unlocks the door and grants access to the room.

The first step in granting access to a publication requires checking for common keys in the KEYS attribute-value pairs. This occurs in three out of the five possible cases. In the first case, the value of the KEYS attribute on both sides is a list containing NULL. This case is consistent because the publication unlocked and therefore available to all subscribers. In the second case, the value on the publication side is a list containing NULL, but on the subscription side it is a list containing keys. This case is also consistent because the publication remains unlocked regardless of what keys a subscription owns. In the third case, both the publication and subscription contain a list of group keys and there is a common key in both lists, meaning that the publisher and subscriber claim they belong to at least one common group. The fourth case is exactly like the third except that the lists do not share a common element. This case is not consistent because the subscription does not belong to a common

Publication Keys List	*Subscription Keys List*	*Consistent*
(NULL)	(NULL)	True
(NULL)	(keyA, keyB, ... keyN)	True
(keyA, keyB, ... keyN)	(keyA, keyB, ... keyN) s.t. common key exits	True
(keyA, keyB, ... keyN)	(keyA, keyB, ... keyN) s.t. no common key exits	False
(keyA, keyB, ... keyN)	(NULL)	False

Table 4.1: Consistency for five cases of publication and subscription keys lists

Publication Keys List	*Subscription Keys List*	*Common Element*
(NULL)	(NULL NULL)	True
(NULL)	(NULL keyA, keyB, ... keyN)	True
(keyA, keyB, ... keyN)	(NULL keyA, keyB, ... keyN) s.t. common key exits	True
(keyA, keyB, ... keyN)	(NULL keyA, keyB, ... keyN) s.t. no common key exits	False
(keyA, keyB, ... keyN)	(NULL NULL)	False

Table 4.2: Common element for cases with NULL appended to subscription list

group. In the fifth case, the KEYS value on the publication side is a list of keys, but on the subscription side it is a list containing NULL. This case is also not consistent for the same reasons as the previous case. Table 4.1 illustrates these four cases and their consistencies.

One simple way of handling these different cases is to have the controller append a NULL value to a subscriptions key list when the event is first added to the system. Then when matching the KEYS attribute, the controller can disregard the different cases and just check if the two lists share a common value, wildcard value included. If they do, they are consistent and the procedure can verify these common keys. If not, access is denied and the procedure moves to the next event. Table 4.2 illustrates this solution.

Once the procedure obtains a list of common key values, it checks if the NULL value is present in that list. If so, this indicates that the publication had a NULL value for its KEYS attribute and is unrestricted. Therefore, the attribute-value pairs matche. If not, the procedure must verify a common group by running down the list and performing verification one by one. Verifying that a device belongs to a group requires the device's group token, the device_public_key, and the group_public_key, which was already provided by the KEYS attribute. First, the controller asks for both the publisher's and subscriber's device_public_keys. Then, it asks both devices for their group token corresponding to the first common group key. Once this is received, the controller can follow the group verification protocol described in Section 2.3. If verification succeeds, the attribute pairs match. If verification fails, the controller moves on to the next common group and asks for those tokens. This process repeats until there are no more common groups, at which time the attribute pairs fail to match and access is denied. This protocol allows the controller to ask for information on a need-to-know basis. Section 5.4.1 discusses the possibility of remembering verified group memebers on the controller to reduce the amount of necessary device transmissions.

For all remaining attribute-value pairs, except the CLOCK pair which is completely ignored in the match check, consistency occurs when the values of corresponding attributes are exactly the same or if the value on the subscription side is NULL. In the later case, the procedure allows any publication value for that attribute to match. It now becomes clear why the KEYS attribute is a special case. For friendships and access, it is the publication that controls the degree of specification. For all other attributes, it is the subscription that controls that degree. Lastly, all matches are case-insensitive. For example, the device names "BOB", "Bob", and "bob" are considered the same. Figure 4.10 shows pseudo-code for a match procedure.

Exact Matches

The matching procedure returns a list of elements that successfully matched with the trigger event. If the list contains only one element, then there is an exact match between a publication and a subscription and the controller takes the following action. First, it notifies the subscriber that its subscription has been matched. This notification consists of the event identifier for the subscription and the Bluetooth FHS

```
public class Controller {
// ...

   public boolean match(Publication pub,
                        Subsciption sub) {
      //get common keys
      List common = getCommon(pub.keys, sub.keys);
      //verify, if non verify then exit
      if (noneVerified(common))
         return false;
      if (sub.addr != NULL &&
          !pub.addr.equals(sub.addr))
         return false;
      if (sub.name != NULL &&
          !pub.name.equals(sub.name))
         return false;
      if (sub.class != NULL &&
          !pub.class.equals(sub.class))
         return false;
      return true;
   }
}
```

Figure 4-10: Match procedure for a publication and a subscription

packet of the matched publisher. This packet is easily taken from the publication by packaging the BT_ADDR attribute value with the CLOCK attribute value. Because the FHS packet contains information that allows the subscriber to calculate the paging channel of the publisher, the subscriber can now directly page the publisher. It enters PAGE mode and sends page requests on the publishers paging channel. The next step for the controller is to notify the publisher. This notification consists of an identifier for the publication and a "wake up" message. When the publisher receives this notification, it enters PAGE_SCAN mode and listen for page requests. From there, the two devices can start from the paging procedure of the standard protocol to set up a Bluetooth connection and exchange data. Both the publisher and subscriber will time out of the PAGE_SCAN and PAGE modes if there is no response from the other party. It is up to devices to determine the length of time they are willing to wait before they time out of a PAGE or PAGE_SCAN mode. However, the protocol specifies a minimum time that devices must wait before time out. Chapter 6.1 discusses choices for this minimum time.

After an exact match is answered, the controller erases both the subscription and the publication from its database. The subscription is erased because the subscriber has found the device it is looking for. The publication is erased so that there is no interference on the publishers paging channel while it sets up this connection with the subscriber. This could happen if immediately after a match, a new subscription was added that also matches with the same publisher. Two subscribers would then be trying to send requests on the same channel and result in interference. To prevent this, the publication is erased and the publisher has the option of reposting to the system once the connection is complete or the PAGE_SCAN has timed out.

There are two possible reasons for timing out of PAGE_SCAN and PAGE modes. First, the devices are out of range from each other. Even though two devices are in range of the controller, does not necessarily mean they are in range of each other. The second reason is that a device is uncooperative and decides not to send page requests or listen for pages. These situations are not the responsibility of the system. The system is a best-effort system that only guarantees both parties will be alerted of a match and given appropriate information for connecting. It does not guarantee device connection as a result of a match

Multiple Matches

When the matching procedure returns a list with multiple matches, there are two cases to consider: a publication matching with many subscriptions, or a subscription matching with many publications. When there are many subscription matches, the controller must notify many clients of one event. When there are many publication matches, the controller must notify one client of many events.

In the first case, the procedure returns a list of more than one matching subscriptions. The controller is now faced with the task of notifying multiple subscribers of one publication event. However, this task is more complicated than it seems. Traditional publish-subscribe systems decouple interactions between the subscriber and publisher, who are not aware of each other. As mentioned in Section 2.2.1, subscribers do not typically care where events are generated and publishers do not care where their events go. Therefore, those systems can send an event to all interested subscribers. However, this system differs in that it is concerned with future interactions between publishers and subscribers and therefore cannot allow a publication event to be sent to multiple subscribers. The purpose of this publish-subscribe system is to decouple device discovery in a way that can still lead to connections. If the system sent a publication event to multiple interested subscribers, it would most likely result in packet collision due to different page requests being sent on the same channel at the same time. This situation is similar to the one described in the previous section.

To handle this first case, the system removes the possibility of interference by selecting only one matched subscriber to notify: the subscriber that has been waiting longest for a match. The oldest subscription is found by looking at the time-to-live (TTL) values described in Section 4.1.5. The controller then treats that match as an exact match and proceeds as previously described (see Figure 4.11). If the publisher chooses to remain available to other devices after it connects with the subscriber, it must resubmit a publication. An alternate solution is to have a scheduling algorithm that assigns different times at which to send each matched subscriber the event notification. The publication remains open until the last matched subscriber has received its notification. Enough time can be allotted between notifications to give the publisher and subscriber a chance to set up a connection. This alternative was not used in this system because it adds too much complexity. It requires the controller to maintain a schedule and to split its time between managing the system and managing these notifications. It is also difficult to determine how much time should be allocated between notifications.

In the second case, the procedure returns a list of more than one matching publication. In response, the controller passes this list to the originator of the subscription event and erases the subscription (see Figure 4.12). Based on this list, the originator or subscriber can narrow its search by resubmitting a subscription that serves as a more specific template for the device it wishes to connect to. This case is more straight-forward than the previous because the system does not have to worry about future interactions. There is no exact match, so no publishers are told to listen for

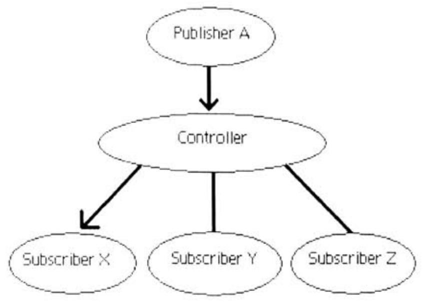

Figure 4-11: Controller chooses oldest matching subscriber to notify

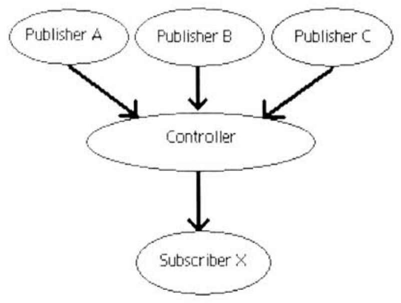

Figure 4-12: Controller notifies subscriber of all matching publications

page requests and no information is provided on how to page the publishers. Even if the subscriber could contact all matched publishers, it would be via different channels and would not therefore not result in packet collision. However, this is irrelevant since a subscription is meant to match with only one publication to constitute a device pair.

Cache of Matched Events

When an exact match occurs, the two matched devices try to connect via paging. As mentioned before, these devices might time out of the paging procedure because they are not in range of each other. If they both resubmit the same events to the system, the controller would match the events again and the devices would time out again. This results in a loop in the system.

To prevent such loops, the controller maintains a cache of recently matched event pairs. Each time an exact match occurs, the controller stores the matched pair of events in a cache along with a Time To Live (TTL) value. Using this cache, the controller can check if a match is a copy of a previously answered match. If so, the controller assumes that the previous match timed out and therefore the events should not be matched.

When the matching procedure generates a list of matches, whether exact or multiple, it checks the cache and removes all events from the list that have recently been exactly matched to the trigger event. This occurs during the "match check" action in Figure 4.8. It then continues with the matching procedure. If a cached pair is accessed, then the TTL value is reset. Section 5.3 discusses choices for the TTL value. It should be noted that a more effective way of preventing loops would be to have the devices that timed out notify the controller when the time out occurs. That way, the controller can maintain a cache of just the timed out matches, and not of all the matched events. This solution however requires more involvement from the device and is not necessary for this simple system.

4.1.4 Identifiers

Unique identifiers are used to reference events in the system. These event identifiers are necessary to distinguish between different events from the same device. This is applicable in the case that a device has multiple subscriptions or both a publication and subscription(s) active on the system (multiple publications are not allowed). The main beneficiaries of event identifiers are the client devices in the system, not the controller. When information is sent to a device regarding one of its events, there must be a way for the device to identify which event is in question. It is possible to have the controller send the event itself as an identifier, but this method transfers too much data unnecessarily. Since the event is already stored on the device, the controller should avoid resending all that data. Instead, events are assigned identifiers when are they are first added to the system. As acknowledgement of a successful addition, an identifier is sent back to the originator of an event, where it is stored alongside its associated event for later reference. These identifiers can later be included in messages from the controller to a device to identify the event in question.

Because publications and subscription events are managed in separate collections, they should also be separately assigned identifiers according to event type. The format of an identifier consists of a code indicating the event type and a numeric value that is unique within its collection. While a publication and subscription might have the same numeric value id, they have different codes for the event type, resulting in an overall unique identifier. To assign the numeric value of an identifier, each collection uses its own counter. Both counters begin at 0 when the system first starts up. Each time a new event arrives, the controller finds the next valid number by incrementally increasing the counter corresponding to the events type until no active event in that collection has the same value. For example, when the first publication arrives, the publication counter is increased to 1 and this value is used for its numeric identifier. The counter is then set to the latest assignment, in this case 1. Once the maximum number of the counter is assigned, it wraps back to the lowest value of 1 and continues from there.

4.1.5 Event Expiration

The system expires open publications and subscriptions that have been in the system for a fixed period of time. These events are deleted from the database and the originators are notified through an identifier and an expiration message. Expirations are implemented with time-to-live or TTL values. When a new event is added to the database, the controller attaches a TTL value after assigning an identifier. The controller updates the system by decrementing the TTL values of open events once every clock cycle. If a TTL value becomes 0, the controller deletes the event and notifies the originator of the expiration. If it wishes, the device can then resubmit the event to the system. Section 5.3 discusses choices of TTL values.

An alternative to using event expirations is to leave events on the system for the lifetime of a device's connection to the controller. This would not require devices to renew their events periodically. However, there must be some mechanism for syncing open events on the device side and open events on the controller side. It might be the case that what the device thinks is stored in the database is not actually stored or vice versa, resulting in inconsistent data. Unfortunately, syncing algorithms often require significant data transmission. This contradicts the requirement of minimizing the amount of data a device must send (see Chapter 3). Efficient syncing algorithms such as Rsync have been developed that minimize the amount of transmitted data. However, the tradeoff is significant computation at the device end which also contradicts a requirement of this system [26]. The use of event expiration makes the system as a whole much simpler. By timing out events, the system allows devices to renew an event as needed, which reduces the amount of data sent. There could be a short period of inconsistency but this is bounded by the maximum TTL value. The Java Message Service is an example of a publish-subscribe system that uses expiration to remove forgotten messages out of the system [27].

4.1.6 Disconnections

Devices either disconnect intentionally by explicitly closing the connection or leaving the range of the controller. When devices disconnect, they return to the state they were originally in before they joined the system. For example, a discoverable device enters a room with the controller, who detects the device and connects. Its availability is not matched and the device is parked and its inquiry scan disabled. However, the device stores its original state (discoverable mode). When the device exits the room, its connection to the controller times out because they are no longer in range. When this occurs, the device returns to its original state. Devices can also disconnect unintentionally due to interference or fading. To recover, the device may reconnect directly or indirectly, i.e. actively inquire or enter discoverable mode.

The system handles both types of disconnections by flushing the system of events that originated in the disconnected device. This ensures that no inconsistent data is left on the system. Unfortunately, removing these events requires the device to resubmit all its events when it reconnects. While this solution handles the inconsistent data requirement, it is not optimal for reducing the amount of data a device must send in this situation. Chapter 6.2.1 discusses an optimization for unwanted disconnections using a cache.

4.2 Assumptions

Plentiful resources – The controller is assumed to have near limitless resources. This is justified by the fact that the controller is a plugged in device. Because the controller is often a bay station, it does not need to worry about storage or power limitations as compared to mobile devices.

No idle devices – Devices will only remain connected to the controller for as long as they wish to use its services. In other words, devices that no longer wish to remain available or are not actively seeking other devices will disconnect from the controller. This implies that there are no malicious attacks on the controller. Attackers can be devices that purposely flood the system with nonsense or take up piconet capacity. This assumption is reasonable when the system is used in a personal computing environment, where devices are responsible and connect only when services are required.

Previously authenticated devices – Devices have been authenticated with one another and with the controller. This design assumes that all devices are trusted pairs and do not require user authentication. While the system still works when devices are not authenticating, it just requires more user interaction. However, this assumption is reasonable for a personal computing environment in which devices have already been paired. It should not be assumed when the system is used in a public setting.

Chapter 5

Discussion

The system presented in the previous chapter addresses the shortcomings of the Bluetooth connection protocol. However, it is a simple solution that still has room for improvements. This chapter discusses the solution itself and future considerations for improving on the solution. The first section discusses the system in the context of the requirements previously listed. It describes how the system satisfies these requirements and to what degree. The second section discusses three design alternatives and why they were not optimal for this particular system. These alternatives are based on communication paradigms similar to the publish-subscribe paradigm. The third section discusses how to choose system parameters and what factors to consider when choosing them. The fourth section discusses future considerations for the system. This includes different optimizations that use caching and key lists to reduce the amount of data a device transmits. It also includes a discussion on the bottlenecks of the system and ways to improve performance and scalability. Lastly, the section discusses a way to include dumb devices in the system.

5.1 Satisfied Requirements

The solution proposed uses the publish-subscribe paradigm to decouple device interaction during Bluetooth discovery. This section discusses whether the requirements described in Chapter 3 have been met.

Perform no worse than Bluetooth connection protocol – The system allows an inquiring device to find an available device through the matching of publications and subscriptions. Even when an inquiring device has no specific interest, it can post a subscription with NULL values for for all attributes, to which the system would return a list of all publications (unrestricted). This case is exactly like the normal Bluetooth discovery procedure in which an inquiring device asks for any available device in the area. Also, a discoverable device can use the system in a seamless fashion because the controller automatically detects the device and connects to it. It assumes responsibility of announcing the device's availability with little work from the device. The system goes beyond this by allowing devices to make specific inquiries according to their interests. The system also provides no less information to a subscriber and provides no more information about a publisher than would have normally been obtained in a Bluetooth discovery procedure.

Reduces the chances of anonymous attacks in discoverable mode – Because devices discover each other through the system, these types of anonymous attacks are less likely to occur. Both bluejacking and bluesnarfing, which target discoverable devices, are avoided when the device is not left in discoverable mode. In this sytem, a device is only discoverable long enough for it to publish or subscribe. Therefore, the chances of an attack are reduced.

Allow devices to expend a lower average power during discovery – This systems reduces the average power a device consumes to connect to another device. Currently, it requires the device to perform one inquiry and one paging procedure to connect to the controller, and then a second paging procedure to connect to a device. This does result in a greater power consumption when both devices are already present since they would have performed a single inquiry and page to connect under standard Bluetooth protocol. However, when devices arrive asynchronously, they consume much less power by using the system.

Consider first the power consumption of devices using normal Bluetooth connection protocol. For two devices to discover each other asynchronously, the device that arrived earlier must already be inquiring or scanning for inquiries when the second device arrives. Leaving a device in inquiry mode wastes a lot of power as the procedure requires periodic transmissions. Leaving it in discoverable mode also requires the tranmission of inquiry responses whenever a request is encountered.

The system decouples this interaction by allowing devices to announce availability and then go to "sleep", i.e. parked mode. Because the system takes on the responsibility of notifying devices of matches, they do not have to worry about conversing with other devices. Therefore, the average power consumed in this system for the purpose of discovery is lower. This is especially true in Bluetooth-rich areas where there are many devices in close proximity.

Allow devices to restrict availability – The system allows devices to restrict access to their publications. By providing group_public_keys, a device can specify access only to members of a common group.

Handle Stale States – The system limits the lifetime a parked device can remain in a stale state, i.e. a state that is no longer true, by expiring events that have been in the system for a prolonged period of time. The system also flushes those events whose originator has disconnected. Although these procedures address the problem of stale states, both these mechanisms require more overhead on the part of the controller. However, given the unlimited resources available to the controller, this is preferable to increasing overhead on client devices.

5.2 Design Alternatives

This section discusses three design alternatives and the reasons they were not chosen for this system. These alternatives use communications paradigms similar to publish-subscribe but do not provide the type of loose coupling required in the system.

Tuple Space – In DSM systems, hosts have the same view of a common shared space called the tuple space, which provides a simple yet powerful abstraction for accessing shared memory. A tuple space contains a collection of ordered tuples that act as the communication tool for hosts. Hosts can insert or remove tuples, or read them without changing their location. The advantages of this interaction model are the time and space decoupling it provides. Producers of tuples do not need to know what happens to the tuples in the future and consumers do not need to know what has happened to tuples in the past. The disadvantages of tuples are that there is no central management that controls the tuple space. For this system, computation and management should be handled by a central entity rather than by devices which do not have much power or resources. Also, tuple spaces do not provide the regulation mechanism neceessary for allowing friendships because any end user can read and access a tuple [17].

Observer Design Pattern – In these systems, subscribers register their interest directly with publishers while publishers notify subscribers asynchronously through a server. This management decouples interaction in synchronization but not in time and space. It does not work because the purpose of our system is to perform device discovery. Because devices do not know about each other, subscribers cannot directly register their interest in this manner. An adaptation of the paradigm might be to have interests delivered to a publisher through middleware, but this management would still place the burden of filtering on the publisher [17].

Message Queing – Message queuing integrates certain forms of publish-subscribe interaction. In this type of system, messages are put into a global space called a queue by a producer and removed by a consumer. However, consumers retrieve elements based on priority, or its location in the queue, rather than by structure. Therefore, this alternative fundamentally cannot be adapted for the system, which needs to deliver messages based on content. In other words, availability should not determined by the time a device publishes to the system [17].

5.3 Choosing Parameters

One parameter that affects performance is the initial TTL value assigned to open events (see Section 4.1.4). TTL values determine the length of time an unanswered or open event remains on the system. Small TTL values result in high turnover and require devices to renew events frequently, while large TTL values lead to higher chances of inconsistent data. Inconsistent data could result in missed matches, i.e. matches that should have occurred but did not. This consequence is not as harmful as requiring a device to retransmit data often. Therefore, larger TTL values should be preferred.

Another parameter that affects performance is the MIN_TIMEOUT value for devices participating in the paging procedure as the result of a match (see Section 4.1.6). When notification is received of an exact match, the subscriber goes into PAGE mode and the publisher goes into PAGE_SCAN mode. Because the controller cannot ensure that a device enters its respective mode, devices should time out of the paging procedure when there is no response from the other party. However, if devices time out too quickly, the other party might not have a chance to respond in time. This might result in re-publication and re-subscription which is costly for the system. Therefore, devices must agree to participate in paging for at least a certain amount of time. A reasonable choice for this value is the maximum time of a paging procedure. While the average and maximum inquiry times vary by as much as 27 seconds, the average and maximum paging times vary by only 1.28 seconds (see Table 2.1). Therefore, this is a reasonable time to wait. As shown in Figure 4.7, the controller will notify both devices simultaneously and then wait for both notifications to occur before moving on.

The last parameter to choose is the TTL value for the cache of matched events. As described in Section 4.1.3, this cache is needed to ensure that devices that timed out of a connection are not matched again, thus preventing loops in the system. This TTL value should be chosen such that devices who time out are not matched again. However, devices that matched and successfully connected should still be matched if they disconnect and return at a later time. Consider the first case. Two matched devices will try to connect for the MIN_TIMEOUT period previously discussed. Once they timeout, they must resubmit their events to the controller and wait for them to be processed. Depending on the number of events waiting in the queue, this could take a while. Therefore, a longer TTL value is desirable so that the record is still in the cache by the time the resubmitted events are processed. In the second case, two matched devices connect and communicate. They disconnect and return at a later time. At that time, the record of their match should have expired from the cache so that they may reconnect if desired. Therefore, a shorter TTL value is desirable. Altogether, a reasonable TTL value can be longer. The length of time for two devices to connect, communicate, disconnect, and return to the system should be at least several minutes. The length of time for the controller to process events should be much shorter (less than a minute).

5.4 Future Considerations

Future considerations for this system include different optimizations and scalability issues. The system currently meets the requirements listed in Chapter 3. However, it still has room for improvement, particularly in reducing data transmission and power consumption and increasing system capacity.

5.4.1 Optimizations

There are two possible optimizations for this system. The first uses caching and the second uses key lists to remember verified group members. Both result in less data transmission from the device and therefore saves power.

Cache of Removed Events

The amount of data a device must send to the controller can be reduced by caching deleted events. Currently, the controller removes events when they expire, when a device disconnects, or when a matched event is answered. In the first case, it is likely that the event will be renewed since it had not been answered. In the second case, the disconnection might have been caused by factors such as interference or fading. If so, the device would like to reconnect as soon as possible. In the third case, the device might have timed out from an attempted connection and wish to resubmit the event. All three cases require the device to resubmit deleted events.

By maintaining a cache of events that have recently been deleted from the database, the controller only needs an event's identifier in order to renew it. For example, when a device receives an event expiration notice, it can call a `renew()` operation on the controller that takes the event identifier as an argument. If a device experiences an unwanted disconnection, it can call this `renew()` operation for each event that was deleted.

Similar to open events, cached events should expire from the system's cache. Therefore, the controller must also assign them TTL values (see Section 4.1.6). The TTL value for cached events should be smaller than TTL values for open events. This is because devices typically renew events shortly after expiration or resubmit them soon after they are able to reconnect. Given these two cases, a possible choice for this TTL value is the maximum time it takes for a device to reconnect to the controller (see Table 2.1).

This optimization reduces the amount of data transmission from the device, but requires much more controller overhead. However, this is not an unreasonable tradeoff given that a controller is assumed to have significantly more resources than devices. As shown in Figure 4.7, the controller could use the cache when it adds the event to the queue. The device can specify an identifier at that point and the controller could look it up and add the event to the queue.

Keys List

Another optimization is to maintain a list for remembering verified group memebers. When matching a restricted publication, the controller must check and verify if the publisher and a subscriber belong to a common group. This requires devices to send group tokens and their device_public_key. To avoid these transmissions, the controller can maintain a list for group members it has previously verified. Elements in this list associate Bluetooth device addresses with their verified groups, indicated by group_public_keys. Therefore, it must verify a device's membership in a group, the controller can simply check this list. If there is no listing for the group or device in question, the controller proceeds as described in Section 4.1.4. When new verifications are made, the controller stores them in the list. If desired, the controller can clean this list by flushing memberships that have not been checked in a long time. This can be implemented using TTL values that are reset every time the membership is checked. Group membership is changed by generating new group keys, requiring the owner to reissue group tokens [25]. Therefore, the system does not consider new group keys and old group keys a match. Old memberships in this key list will die out.

5.4.2 Performance

As shown in Figure 4.7 and 4.8, the controller uses five threads that interact sequentially to manage the system. This can hinder the performance of the controller. For example, when a new device publishes, the event must first be added by the new-connection thread. It must sit in a queue until the processing thread removes it. While it is in the queue, a matching subscription might be added to the queue by another thread. It is not until the processing thread finishes processing the publication and then begins processing the subscription that a match will be found. As the number of devices and the number of events to be processed increases, the possibility of the queue containing matching events increases too, causing the performance of the system to slow down dramatically.

One way to increase performance is to have two event queues, one for publications and one for subscriptions. When the processing thread gains the lock for the database, it can grab the next publication and next subscription off the queues and perform simultaneous match checks on each. This is possible because events are matched against open events of the opposite type and do not affect each other. The processing thread can then wait for the results of the match checks before taking the appropriate actions. These events can also be checked against each other or against events in the opposite queue if no events in the database match. In fact, multiple events from each queue can be checked for matches simultaneously, so long as no action is taken until these checks are completed.

Another factor that affects performance is the scheduling of threads. When two threads are vying for the same lock, it might not be desirable to be fair, i.e. to give the lock to whichever thread asked first. For example, the disconnection thread should be given higher priority over the processing thread because obsolete events should be removed before a match is performed. However, if the disconnection thread is always given priority, then the matching thread might never gain the lock if devices keep disconnecting. Another example is the cleaning thread. The cleaning thread has less priority than the existing-connection thread when they are both trying to remove an event. This is because the existing-connection thread is trying to remove an event due to an explicit command from a device, which is more important than declaring an event expired. Therefore, the scheduling of threads is an important consideration for performance.

5.4.3 Scalability

As mentioned in Section 2.1.3, the controller can support over two hundred slave devices in its piconet. Given this high piconet capacity, the capacity of the system really depends on when the performance falls sharply. As previously discussed, performance depends on how many devices are in the system and the traffic of events in the system. It might be the case that there are many devices in these system but the rate of event arrival and removal is very slow. It might also be the case that there are few devices in the system at a time but the rate of event arrival and removal is very fast. These factors might also change depending on what context the system is used in. It is likely that a home office has less than ten devices and has a slower traffic of events. A work office or space might have a higher traffic of events. Therefore, it is difficult to judge the capacity of the system without testing it in different environments. For now, a reasonable capacity can be set to around seven devices. This is because a piconet can only have seven active devices at a time. Since all slaves start out active, this is a good start for testing system capacity. Also, there are not usually much more devices in a 10 meter range.

Once the controller has reached its capacity, a possibility for increasing system capacity is to increase the number of controllers via a scatternet. However, one must keep in mind that the purpose of the system is to provide device discovery services so that devices can form connections. Introducing multiple controllers allows different points of access for a device to enter the system, which is dangerous. It makes no sense for devices to discover each other when they are out of range because they cannot directly form a connection. Therefore, multiple controllers must be grouped closely. Possible client-server topologies to use are the hierarchical and ring topologies (see Section 2.2.3). Hierarchical topologies are most applicable for this system because scatternets already have a hierarchical structure, i.e. a master in a piconet acts as slave in another. Ring topologies are not possible in a scatternet because they require peer-to-peer connections. However, they can be implemented with wired connections, which is acceptable since the controllers are grouped closely.

5.4.4 Dumb Devices

Dumb devices are used for voice or data communications and most peripheral functions. They support only one connection to a computing device and have limited software. An example of a dumb communicating device is the Jabra FreeSpeak BT200 headset. The Microsoft Bluetooth Wireless Intellimouse is an example of a dumb peripheral device. Because a Bluetooth dumb device can support only one connection, it must be paired with another device.

Pairing is the Bluetooth procedure that occurs after paging but before a connection, in order to authenticate two devices that are previously unknown to each other. Typically, it requires a PIN to be inputted on both devices by the user. Once two devices are paired, they can connect directly to each other at a later time without searching again. This is done by remembering the information of a device so that connecting requires only the paging procedure and not both inquiry and paging.

Pairing for dumb devices follows a simple procedure. Unpaired dumb devices are initially discoverable. A computing device will inquire and page a dumb device to connect to it. Because, the two devices are unknown to each other, they must enter a PIN to pair. Dumb devices have generic PINs that are assigned by the manufacturer such as 0000, as with the Jabra headset, or blank, as with the Microsoft mouse [28] [29].

Once pairing has occurred, the computing device connects to the dumb device and becomes the master. When the dumb device disconnects and reconnects at a later time, it directly pages the paired computing device. If the page is heard, it then connects and performs a role switch so that the computing device becomes the master. If the page is not heard, the dumb device periodically pages to check if the computing device has arrived.

Possible Solution

Dumb devices face only the shortcomings of poor power management and the synchrony requirement as described in Section 2.1.4. When the specific computing device they are paired to is not around, they waste power sending pages. Many devices have an on/off button to stop this when the device is not connected. They also suffer from the synchrony requirement because they must be looking for a specific device at the time that device is available. Dumb devices do not suffer from attacks because they do not store information. They also cannot benefit from friendships because they have limited software.

To create a system that decouples these interactions, the controller must be smarter. Because dumb devices are incapable of knowing about any participation in the system, the controller must be able to fake them into thinking they are interacting with a normal computing device. This section describes possible solutions for different cases of dumb device interactions.

In the case that the smart device arrives first and publishes, the publication remains open on the controller and the smart device is parked and put to sleep. At that time, the controller can take over responsibility of scanning on the smart device's page scan channel. When the dumb devices arrives, the controller hears its page requests and can wake up the smart device by telling it to listen for pages. The dumb device then directly pages the smart device and connects.

In the case that the dumb device arrives first and pages the smart device, the controller does not know the page scan channel of the smart device. However, it can pseudo randomly hop through the page scan frequencies in the same manner as an inquiry scan. While it takes longer, the controller will hear the page eventually and respond. To respond to the page and connect, the controller must be able to fake the smart device's unique BT_ADDR. This can only be done with additional hardware. Once the controller connects, a role switch occurs and the controller parks the dumb device until a smart device matching the BT_ADDR arrives. At that time, the controller does not need to connect to the discoverable smart device. Instead, it disconnects from the dumb device and ignores its pages long enough for the dumb device to page and connect to the smart device.

In order for this solution to work, the controller must be able to fake Bluetooth addresses using extra hardware and to pseudo randomly hop through the page scan channels. It is yet unclear how to do this in hardware and how scalable this solution is.

5.5 Conclusion

Bluetooth promises to be the next leader in wireless technology. It is simple, reliable, cost-efficient, and provides all the characteristics necessary to embed computation into our everyday lives. With Bluetooth, devices now have the mobility and communication to adapt to the natural behavior of humans, rather than requiring humans to adapt to them. Soon, Bluetooth devices will become prevalent in every environment.

For many Bluetooth devices to coexist and operate effectively in the same area, it is necessary to manage them in an efficient and power-saving manner. This thesis provides management in the form of a publish and subscribe system. The system introduces a middle man to handle device discovery, thereby shifting the power consumption of discovery from the device to the middle man. Such transfer of responsibility frees up device resources and added benefits include the prevention of anonymous attacks when in discoverable mode, the possibility of asynchronous connections, and the development of friendships.

The ability to form Bluetooth connections in a low-powered manner changes user interaction with these devices. The user no longer has to worry about turning devices off in order to save power when they are not in use. Devices can remain on and available without excessively draining their power, therefore requiring less action on the part of the user. With this system, searching for devices also becomes easier. When a device is absent/unavailable, searching for it requires asking only once rather than periodically, also requiring less action from the user. Finally, permitting friendships in this system opens up many possibilities for associating devices. These associations can be based on any parameters, including user, type, or content of the information handled. Because the system manages friendships, the user does not have to remember these associations when connecting devices. Instead, simple actions that are more intuitive to the user, such as touching two devices, can be used to manage associations.

The primary advantage of this system is the opportunity it provides for transferring the power consumption of discovering devices to a static middle man. This allows many available or searching devices to coexist for longer periods of time. The result is a more seamless interaction with the user, bringing us a step closer to a pervasive and ubiquitous computing environment.

Bibliography

[1] Centre for Pervasive Computing. Available on the Internet: http://www.pervasive.dk, September 2003.

[2] IEEE Pervasive Computing. Available on the Internet: http://www.computer.org/pervasive, May 2003.

[3] Bluetooth Official Website. Available on the Internet: http://www.bluetooth.com/about/, 2004.

[4] James Kardach. Bluetooth architecture overview. *Intel Technology Journal*, 2nd Quarter 2000.

[5] Albert Proust. Personal area networks: A bluetooth primer. *O'Reilly Network*, November 2000.

[6] David Carey. Bluetooth making good on price points. Available on the Internet: http://www.eetuk.com/bus/news/, May 2004.

[7] Bluetooth SIG. Specification of the bluetooth system. Available on the Internet: http://www.bluetooth.com, November 2003.

[8] TheFreeDictionary.Com. Time division multiplexing. Available on the Internet: http://computing-dictionary.thefreedictionary.com/, 2004.

[9] Curt Franklin. How bluetooth works. http://electronics.howstuffworks.com/bluetooth.htm.

[10] Bluetooth Designer. Bt designer: Glossary: F-j. http//www.btdesigner.com/ftoj.htm.

[11] palowireless Bluetooth Resource Center. Time taken to complete inquiry/paging procedures.

[12] Adam Laurie, Ben Laurie, and A.L. Digital Ltd. Serious flaws in bluetooth security lead to disclosure of personal data, May 2004. Available on the Internet: http://www.thebunker.net/release-bluestumbler.htm.

[13] Mark Ward. New mobile message craze spreads. *BBC News*, November 2004. Available on the Internet: http://news.bbc.co.uk/.

[14] Jo Best. 'bluejacking' hits the mainstream. *ZDNet UK*, November 2003. Available on the Internet: http://www.zdnet.co.uk/.

[15] Munir Kotadia. Bluetooth phones at risk from 'snarfing'. *ZDNet UK*, Feburary 2004. Available on the Internet: http://www.zdnet.co.uk/.

[16] Ying Liu and Beth Plale. Survey of publish subscribe event systems. Technical Report TR 574, Computer Science Department, Indiana University, May 2003. Available on the Internet: www.cs.indiana.edu/pub/techreports/TR574.pdf.

[17] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.

[18] TIB/Rendezvous Web Site. http://www.rv.tibco.com.

[19] Ben Segall and David Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proc. of the 1997 Australian UNIX and Open Systems Users Group Conferencing*, 1997.

[20] SQLCourse.com Web Site. Interactive online sql training. Available on the Internet: http://www.sqlcourse.com/.

[21] World Wide Web Consortium. Xml path language (xpath). Available on the Internet: http://www.sqlcourse.com/.

[22] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *Proc. 17th ACMSOSP*, Kiawah Island, SC, December 1999.

[23] Klaus Marius Hansen and Christian Heide Damm. Building flexible, distributed collaboration tools using type-based publish/subscribe - the distrbuted knight case. In *Proceedings of IASTED SE 2004*, 2004.

[24] Peter Houston. Building distributed applications with message queuing middleware. Available on the Internet: http://msdn.microsoft.com/library/, March 1998.

[25] Tehyih D. Wan. Personal correspondence on device groups in bluetooth, May 2004.

[26] Andrew Tridgell and Paul Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, The Australian National University, June 1996. Available on the Internet: http://cs.anu.edu.au/techreports/1996/TR-CS-96-05.pdf.

[27] Sun Microsystems. Java message service, version 1.0.2. Available on the Internet: http://www.javasoft.com, 1999.

[28] Jabra Corporation. Support: Phone-specific info for bluetooth phones. Available on the Internet: http://www.jabra.com.

[29] Amazon. Microsoft bluetooth wireless intellimouse explorer. Available on the Internet: http://www.amazon.com.