FORUM NOKIA

Games over Bluetooth: Recommendations to Game Developers

Version 1.0; November 13, 2003

Mobile Games

NOKIA

Disclaimer

The information in this document is provided “as is,” with no warranties whatsoever, including any warranty of merchantability, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification, or sample. Furthermore, information provided in this document is preliminary, and may be changed substantially prior to final release. This document is provided for informational purposes only.

Nokia Corporation disclaims all liability, including liability for infringement of any proprietary rights, relating to implementation of information presented in this document. Nokia Corporation does not warrant or represent that such use will not infringe such rights.

Nokia Corporation retains the right to make changes to this specification at any time, without notice.

License

A license is hereby granted to download and print a copy of this specification for personal use only. No other license to any other intellectual property rights is granted herein.

Contents

Change History

November 13, 2003	V1.0	Initial document release

1 Introduction

This document is intended as a guideline for game developers who want to write head-to-head or multiplayer games that use Bluetooth for data transport.

These recommendations apply to both native Symbian OS applications and Java™ MIDlets.

2 Bluetooth Overview

This chapter provides a short introduction to Bluetooth. If you are already familiar with the technology, proceed to Chapter 3, “Bluetooth Application Recommendations with Real Time Constraints.”

2.1 Connection Setup

Bluetooth devices need to exchange information before actual data transmission. This information exchange involves inquiry, service discovery, and remote name request.

2.1.1 Inquiry

Since Bluetooth devices are mobile and form networks dynamically, they need a way to find other nearby devices to connect to. This process is called the *inquiry* procedure in the Bluetooth Baseband Specification (see Chapter 7, “References”). From the point of view of an application, inquiry means collecting Bluetooth addresses and determining the class of device (CoD) in the vicinity.

The CoD value indicates the capability of the device. The CoD field shows if a device has rendering capabilities, like a printer, or is an audio device, like a headset. Some numerical values are reserved by the Bluetooth specification (see the *Assigned Numbers* page on the Bluetooth Special Interest Group (SIG) Web site at http://www.bluetooth.org/assigned-numbers.htm).

For inquiry, an access code is used. There is one General Inquiry Access Code (GIAC) to inquire for any nearby Bluetooth devices, and a number of Dedicated Inquiry Access Codes (DIAC) that inquire only for a specific type of device. The inquiry access codes are derived from reserved Bluetooth device addresses and are further described in the Bluetooth Baseband Specification (see Chapter 7, “References”).

Normally, inquiry is done with the GIAC. But the Java™ and Symbian Bluetooth APIs also offer the possibility of using a DIAC. In some papers, DIAC is also called Limited Inquiry Access Code (LIAC).

2.1.2 Service discovery

Inquiry is the first step; this is followed by *service discovery*, which is based on the Service Discovery Protocol (SDP).

In wired networks, there is normally a central infrastructure for managing connections; Bluetooth doesn’t provide this. Instead, connections are made dynamically, and devices must determine what services are provided on discovered devices. This is called service discovery, which is the process by which applications locate and gather information about other services in the vicinity.

2.2 Universally Unique Identifier (UUID)

Universally Unique Identifiers (UUIDs) are used in the SDP to search and provide services. They are assigned for each service class. A UUID is a 128-bit value that is guaranteed to be unique across all space and time. Some UUIDs for different Bluetooth protocols are defined by the Bluetooth specification. Other UUIDs may be defined by the application.

The UUID is written as the following series of hexadecimal digits:

UUID = ######## – #### – #### – #### – ############

32_bit – 16_bit – 16_bit – 16_bit – 48_bit_node

The generation of UUIDs does not require official registration from any registration authority for each single identifier. Instead, it requires a unique value over space for each UUID generator.

Thus, if you use the Bluetooth Device (BD) address of one of your Bluetooth devices for the 48-bit node, you can easily generate a unique UUID for the device as shown in Figure 1.

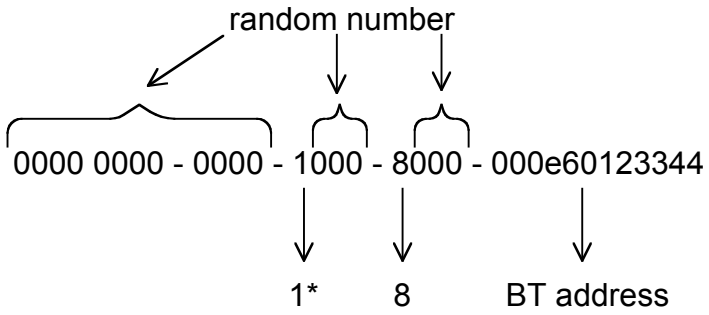

Figure 1: UUID generation example
1*: Only the two most significant bits are set to 10; the two least significant bits belong to the random number.

For the random number, you can choose any number, such as a date and time number. For more details about UUID, refer to the Bluetooth Baseband Specification (Chapter 7, “References”) and to the ISO-11578 entry in the *Assigned Numbers — References* page on the Bluetooth SIG Web site (https://www.bluetooth.org/foundry/assignnumb/document/assignnumb_references).

2.2.1 Remote name request

Inquiry just provides the user with bulky BD addresses. To obtain user-friendly names for remote devices, the remote name request procedure is used.

2.3 Piconet and Scatternet

A basic Bluetooth network is a *piconet*. The device that invites other devices into a piconet becomes a *master*, and a device that accepts such an invitation becomes a *slave*, although the roles can be switched later in some Bluetooth stacks. The master role does not imply any privileges, but means that the master device governs the baseband synchronization between devices. Figure 2 illustrates two kinds of piconet configurations, the first with a point-to-point connection (suitable for head-to-head games) and the second with point-to-multipoint connection (suitable for games with three to seven players). The slaves in the piconet only link to the master; there are no direct links between the slaves.

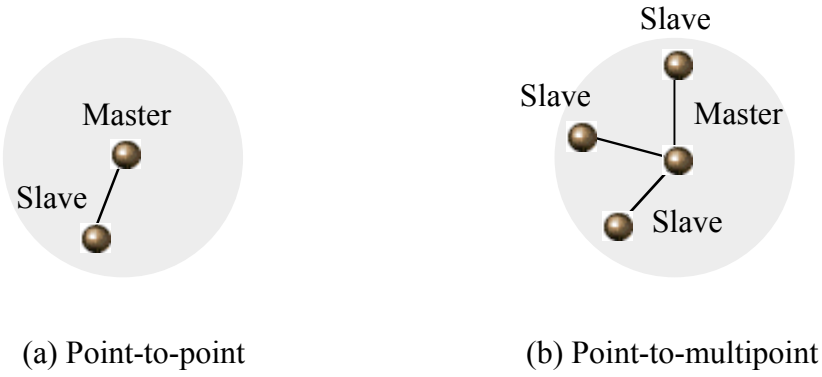

Figure 2: (a) Point-to-point and (b) point-to-multipoint piconets

The Bluetooth Baseband Specification limits the number of active slaves to seven in each piconet. It is also possible to establish a scatternet if the hardware and software stacks support it. Figure 3 illustrates two scatternet configurations.

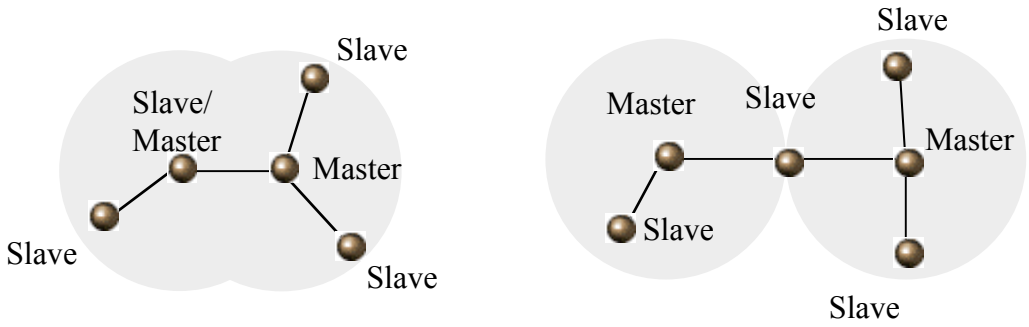

Figure 3: Scatternets

In a scatternet, the master of one piconet can be the slave of another piconet. Or a slave can be a slave in two piconets. The disadvantage of scatternets is that the piconets contend for bandwidth, so the effective bandwidth available to each is lower.

It is important to understand that actual data transmission is preceded by a series of operations for device and service discovery. Moreover, either a new piconet has to be established or the device has to join a piconet that already includes another peer.

At present, most Bluetooth stacks, including the Nokia Bluetooth stack, do not support scatternets.

2.4 Creating a Connection

Since most Bluetooth devices do not support scatternets, they are affected by piconet restrictions. In order to understand how these restrictions affect Bluetooth application development, it is important to understand how the piconet is organized.

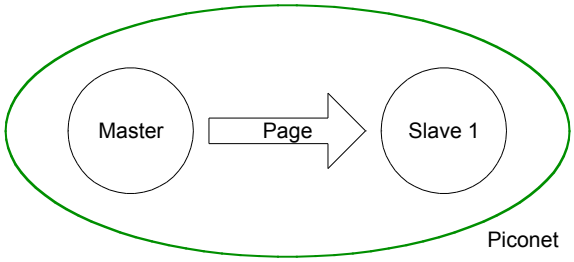

Figure 4: Point-to-point connection

The device that initiates creation of a piconet becomes a master; the device invited to the piconet (device that was paged and is in page scan mode) becomes a slave (see Figure 4). Only the master can add additional devices to the piconet.

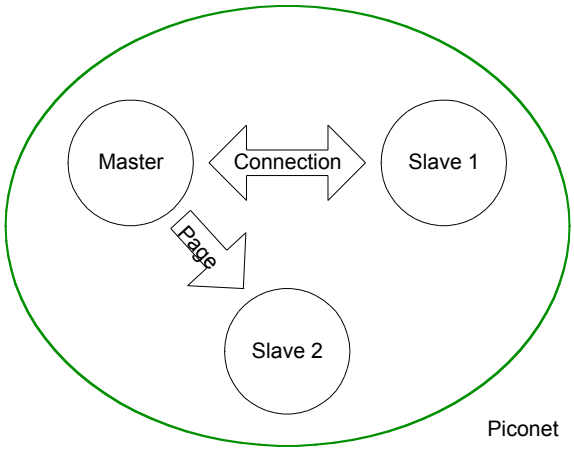

Figure 5: Point-to-multipoint connection

If the master wants to build a point-to-multipoint connection (see Figure 5), it adds the slaves one after another. In the piconet, it is not possible to send data directly from slave 1 to slave 2; data must instead be routed via the master.

In all other cases, the Bluetooth network turns into a scatternet. If a device does not support scatternet connections, the page procedure simply gets no reply from the addressed device. Thus, the creation of the connection fails.

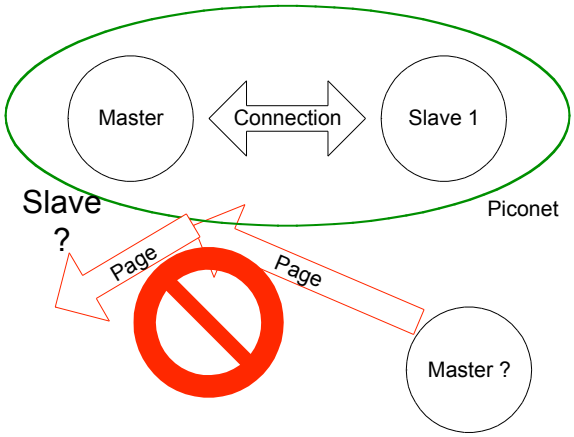

Figure 6: Failed scatternet initiation to master

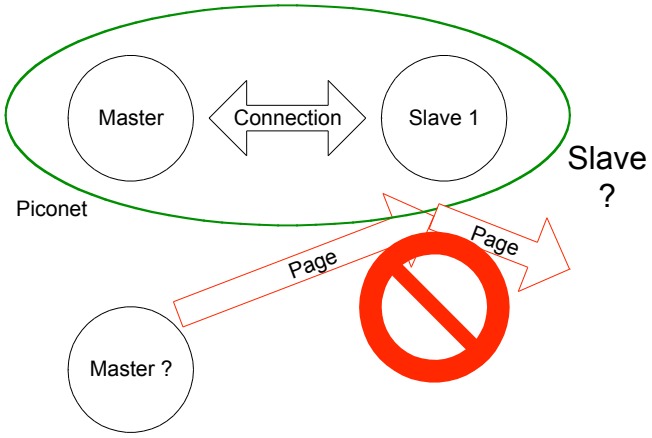

Figure 7: Failed scatternet initiation to slave

In most client/server applications including multiplayer games, clients initiate the connection to a server. This will not work in a Bluetooth game because only a master can initiate a connection, and if multiple devices appoint themselves as masters and try to initiate connections, some will fail because scatternets are normally not supported. Therefore, the method shown in Figure 5 is the recommended way to set up a point-to-multipoint connection (see also Section 3.1, “General Recommendations”).

2.5 Bluetooth Data Packets

Once a piconet is established, Bluetooth provides different packet types for data. Latency is important for most games, and latency varies with packet type.

DM packets (Data/Medium bandwidth) carry relatively small amounts of data, but provide error correction.
DH packets (Data/High bandwidth) carry more data, but only provide error detection, not correction.

Table 1 illustrates some of the different packet types.

Packet type	Maximum data size in bytes	Time consumption over air in microseconds
DM1	17	625
DH1	27	625
DM3	121	1,875
DH3	183	1,875
DM5	224	3,125
DH5	339	3,125

Table 1: Some Bluetooth data packet sizes

The number after the packet type (1, 3, or 5) shows how many time "slots" the packet spans (in Bluetooth terminology, one slot is equivalent to 625 microseconds).

"Maximum data size" shows the number of bytes contained in a packet of the indicated type. However, protocol header information needs to be included in a packet. As you will see in Section 2.6, "Different Bluetooth Protocols," Bluetooth provides two different data protocols, each requiring a different number of bytes for protocol data. But in essence, the actual amount of non-protocol data that can be included in a packet is equal to the packet's maximum data size, minus the number of bytes that need to be devoted to header information.

In general, the more data transmitted, the more time is consumed. In particular, in a disturbed environment, big packets have a great impact on data latency.

2.6 Different Bluetooth Protocols

Bluetooth provides a reliable connection; there is no need to add protocol headers for data correction. Corrupted packets are retransmitted until they are correctly received (even using DH packets). Additional protocols are defined for different purposes.

2.6.1 L2CAP

Logical Link Control and Adaptation Protocol (L2CAP) is the lowest-level Bluetooth protocol that can be accessed by an application. The protocol overhead for L2CAP is 4 bytes. L2CAP is recommended if you have a small amount of data and you need fast response times.

2.6.2 RFCOMM

RFCOMM is a Bluetooth protocol based on L2CAP. The protocol overhead for RFCOMM is between 4 and 5 bytes for small packets. For every 127 bytes of data, the header increases in size by 1 byte. So the overall protocol overhead is about 8 to 9 bytes for data less than 127 bytes (4 bytes from L2CAP and 4 to 5 bytes from RFCOMM).

3 Bluetooth Application Recommendations with Real Time Constraints

Since scatternet is not implemented in most modules, this document shows one way to establish a point-to-multipoint piconet that works with any Bluetooth hardware and software stack.

3.1 General Recommendations

Performing several Bluetooth actions at the same time does not speed the application.

All Bluetooth actions (such as Inquiry, Service Discovery, and Remote name request) will be executed one after the other at Bluetooth module (hardware) level.

To be more precise: It doesn't speed up things to start an inquiry and, when a device is found, immediately start the name request or service discovery without waiting for the inquiry to finish.

3.2 Connection Setup

All Bluetooth activities consume bandwidth, which leads to higher latency for the game. Therefore (depending on the latency requirements of the game), all Bluetooth activities that do not belong to the game should be canceled (for instance, a currently running Bluetooth audio link or a background file transfer). If the game application is unable to end the background activity, it should provide an alert to the user, asking the user to end or finish the activity before playing.

Then the user should be asked to select a game client or game host role.

Note: In a client/server architecture, the client uses a service, which is provided by the server. In the context of this document, the host provides the game service to the clients. Because clients want to retrieve information from a server, they normally actively connect to the server. Thus, very often the term "client" is used for the device that initiates a connection. But in this document the terms "client" and "host" are related to the role in the game. Since with Bluetooth only the master can connect the slaves, the master is inherently the server, and the slave devices inherently clients.

3.2.1 Host role

Typically, in small-scale multiplayer games, all devices run the same software, and any can act as host. Developers must take care to ensure that devices do not contend for the role of host, which would cause connection failures, as described in Section 2.4, "Creating a Connection." One way to do this is to allow any player to tell his or her software, "I wish to serve as host; issue invitations to other devices nearby." This is more or less how Quake works; any player can serve as host, with other players searching for and connecting to the first player's machine.

As illustrated in Figure 8, a host should perform the following actions:

1. Disable page scan and inquiry scan.
2. Register the service, which is unique for the game.
3. Start inquiry to obtain all Bluetooth devices with the appropriate CoD field in range.
   (The host/clients should perform an inquiry/inquiry scan on DIACs to reduce the number of responses.)
4. Perform a service search to find the devices that have started the game and to retrieve the player's name.

5. Present the found devices that have the game running to the host user.
6. Allow the host user to select multiple devices he or she wants to connect to.
7. Connect the devices the host has selected.

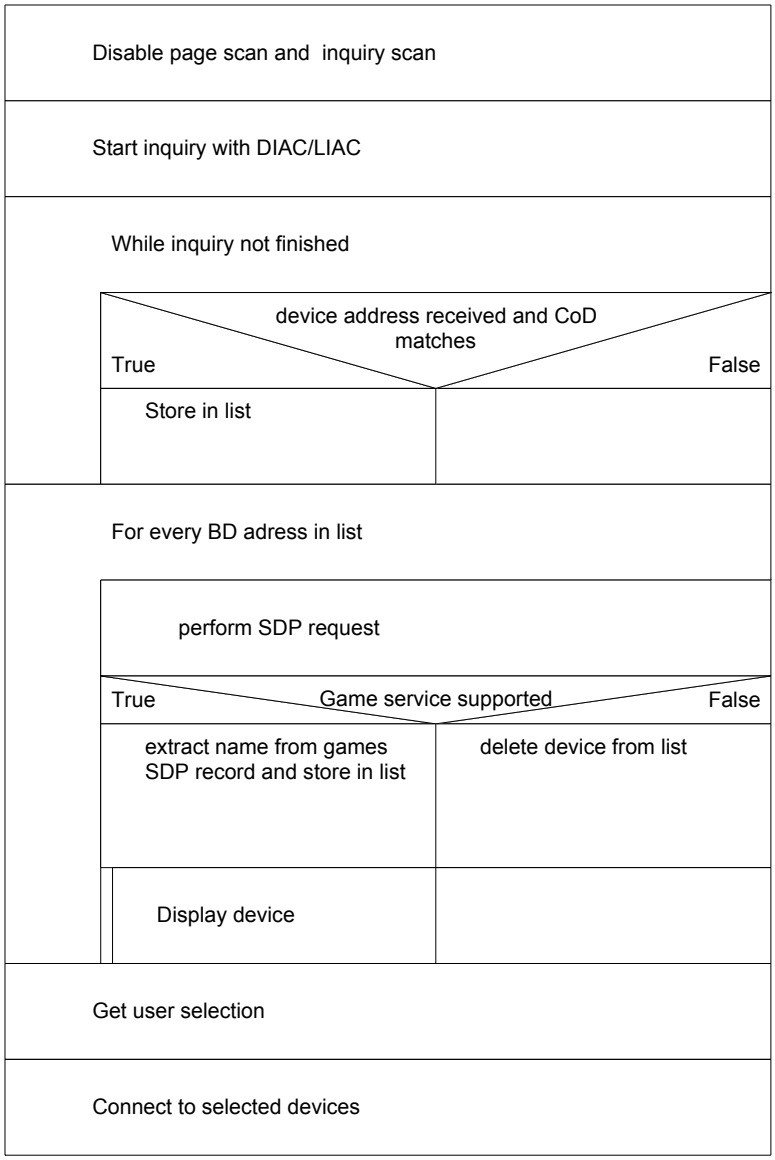

Figure 8: Connection setup on host side

3.2.2 Client role

As illustrated in Figure 9, the client should perform the following actions:

1. Register the service, which is unique for the game. Also create a player name as a service attribute. As default, use the Bluetooth device name.

2. Enable page scan and inquiry scan. (The host/clients should perform an inquiry/inquiry scan on DIACs to reduce the number of responses.)
3. Wait for incoming connection request.
4. Present to the user the choice to join or cancel the incoming connection request.
5. Set inquiry scan address back to GIAC.

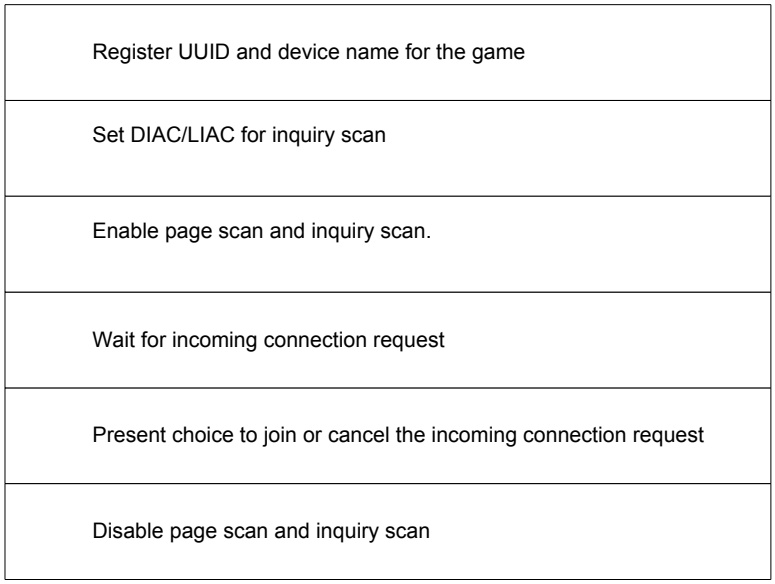

Figure 9: Connection setup on client side

3.2.3 Client and host role

The role of the client and host is to exchange necessary information for the game in a setup phase. The total number of players, data size of exchanged packets, and position of client data in the data packet can be negotiated. Once this is done, data needed to support game playing (such as joystick/position information) can be kept small and sent using small and speedy packets.

3.3 Recommendations for Games Once Started

Bluetooth provides a reliable connection; there is no need to add a custom protocol for data correction or data acknowledgement. Corrupted packets are retransmitted until they are correctly received.

Base the data exchange on a protocol that fits your application. Thus, if reducing latency is highly important (as it would be in a fast-action game, for instance) use a protocol with little overhead, such as L2CAP, and try to reduce the data that must be exchanged during game playing to small packets.

3.4 Recommendations for Game Shutdown

When the game is finished, it is important to:

Set the scan settings back to the values before the game has been started, and

Deregister the services in the SDP database.

3.5 Game Data Examples

There exist a number of different game models as follows:

Frame-based

Each time a device composes a new screen for display on the device, it also performs data exchange. This is only feasible when latency is less than 40 milliseconds (25 frames per second).

Dead reckoning

Data exchange and frame update are done separately. If data is missed, a client uses predictive algorithms to compose the screen, correcting it in future frames as necessary when correct data is received. This scheme works well for Internet-level latencies (100–200 ms for round trip of data).

Used in first-person shooter games.

Synchronous simulation

Data exchange and frame update are done separately. Every client has a complete model of the scene. If one client's data is delayed, the master updates the other clients when the delayed client finally does send update information to the master. This scheme works well for latencies of up to several hundred milliseconds.

Used in massively multiplayer online games.

Turn-based

The master waits for the client data and updates all clients after that data is received; used, for example, in chess.

Since players are likely to respond more slowly than the network, higher levels of latency are tolerable — several seconds is fine.

Since the frame-based model has the most stringent latency requirement, it is used for the next example.

3.6 Example Scenario

This example was selected to measure the round-trip time in conjunction with Bluetooth. This is a theoretical example, not a practical one, and was chosen to measure Bluetooth round-trip times; in practice, game developers are more likely to use one of the other models described in Section 3.5, "Game Data Examples," since the frame-based model is too critically dependent on extremely low latency.

Protocol: RFCOMM

Host: Transmits data to every client and collects data from all clients. When a client sends data to the host, it sends one packet. When the host transmits to clients, the packet it sends consists of the data received from the clients plus the host's data. If each client sends N number of bytes, the host transmits ([number_of_players] * N) bytes of data to each client.

Client: Receives data from the host, then sends its own data to the host.

Note: This differs from the typical behavior of many PC-based games. A game server receives data only from a subset of the clients during one update cycle, because the clients send typically only the changes the user has made — that is, if the keyboard/joystick position has not changed since the last cycle, there's no need to transmit anything. The game state is nevertheless transmitted to all the clients. In other words, this is a worst-case example, since all the clients are transmitting something. The advantage is that no prediction is necessary.

Data size: Data shall be exchanged in a *single* Bluetooth packet. If you need to transmit more data, switch from Dx1 to Dx3 or Dx5 — that is, increase the packet size. Since Bluetooth latency is a function of packet size, doing this multiplies the response and cycle time.

3.6.1 RFCOMM/L2CAP data example

Table 2 shows the amount of game data that the host can send about each player during a cycle, given a particular packet type and the number of players. It also shows actual, real-world latency — that is, the total amount of time required for clients to send data; the host to receive it and format it; and for the host to send data back to all clients.

Number of players		2	3	4	5	6	7	8
Net data size of client packets [bytes] Average round-trip time for collecting client packet(s) and sending host packet(s) [milliseconds]	Host uses DM1	4 [byte] 20.5 [ms]	2 24.0	2 33.4	1 51.1	1 79.4	1 99.4	1 121.8
	Host uses DH1	9 20.5	6 25.0	4 34.6	3 51.1	3 79.4	2 102.5	2 123.1
	Host uses DM3	56 30.0	37 31.5	28 38.2	22 62.8	18 117.0	16 161.0	14 202.3
	Host uses DH3	86 31.5	57 36.5	43 42.4	34 78.6	28 146.4	24 193.7	21 212.8
	Host uses DM5	107 34.0	71 39.0	53 55.3	42 90.3	35 173.5	30 222.1	26 269.2
	Host uses DH5	164 37.0	109 47.0	82 98.0	65 121.4	54 237.4	47 290.4	41 348.6

Table 2: Example net data and average round-trip time for different packet types with RFCOMM protocol

Legend:

Green cells:Client uses DM1 packet

Dark green cells: Client uses DH1 packet

Yellow cells: Client uses DM3 packet

Orange cells: Client uses DH3 packet

The second line of each cell in Table 2 shows the real, measured round-trip time for data exchange in this scenario. Note that even though one Bluetooth "slot" is 625 microseconds (0.625 ms), the actual round-trip times are not simply double this amount even for the smaller packet size. This is because other factors increase the effective latency, including polling times, additional MCL messages, and application delay. Thus, this table represents real-world latencies, not theoretical minimums (and the numbers result from performance of the experiment by Nokia researchers).

Note that latency does increase with packet size, since larger packets span multiple slots. It also increases with the number of players, since each additional player requires more data communication between the host and the clients, consuming more of the available bandwidth.

The top line of each table cell shows the number of bytes of data about each player that the host can include in its packet to each client. Take the DM1 line of the table as an example. As Table 1 shows, a DM1 packet contains 17 bytes. Using RFCOMM, which requires 9 bytes of protocol data (for packet sizes <128 bytes), leaves a mere 8 bytes for actual game data. In a two-player game, this means that the host's packet can contain 4 bytes of information about each player (2*4=8). In a three-player game, you can only include 2 bytes of data about each player (if you tried for 3 bytes that would be 3*3=9, which is more than your 8-byte budget). You can still provide 2 bytes of data for four players, but above that, you're down to 1 byte of data. If you had more than eight players, you couldn't use a DM1 packet at all — but Bluetooth only permits eight players (one master and seven slaves), so that's not a problem.

So the top line number in each cell is the maximum amount of data about each player that the host can include in its message to each client, given the indicated packet type. You can also view this as the amount of data each client is permitted to send the host with each update, since the host has to retransmit the same amount of data to each client.

Of course, clients are always sending less data than the host. Thus, even when a host is using a larger packet size in order to cram in all the player data, clients may still be able to get away with using a smaller packet size. The colors of the cells in Table 3 indicate this. With the bright green cells, clients can use DM1 packets. With dark green cells, DH1 packets are used; with yellow cells, DM3 packets are required; and with the dark orange cell, DH3 packets are used.

Number of players		2	3	4	5	6	7	8
Net data size of client packets [bytes]	Host uses DM1	6 bytes	4	3	2	2	1	1
	Host uses DH1	11	7	5	4	3	3	2
	Host uses DM3	58	39	29	23	19	16	14
	Host uses DH3	89	59	44	35	29	25	22
	Host uses DM5	110	73	55	44	36	31	27
	Host uses DH5	167	111	83	67	55	47	41

Table 3: Net data for different packet types with L2CAP protocol

Legend:

Green cells:Client uses DM1 packet

Dark green cells: Client uses DH1 packet

Yellow cells: Client uses DM3 packet

Orange cells: Client uses DH3 packet

Table 3 performs the same analysis when the L2CAP protocol, rather than the RFCOMM protocol, is used. The number of bytes that the host can transmit about each client is larger because the L2CAP protocol requires less protocol information overhead (4 rather than 9 bytes). Time information is not shown on Table 3, because there is no real-world difference in latency times when different protocols are used (that is, you can use the times from Table 2 when using L2CAP as well).

3.6.2 More detailed description for two players

Figure 10 shows the example scenario with two players.

Note: This scenario differs from many PC-based games. Normally, the frame rate and the game update rate are not directly bound to each other. (The "game update rate" is defined as the number of update messages exchanged between servers per second.) That is, the data throughput of the system doesn't determine how many frames per second the game can have. Typically, there are multiple frames between updates. The game object positions are extrapolated between the frames based on the information that was received in the most recent update.

If it is possible to have a data update every frame, extrapolations are not necessary. This reduces the CPU load for the game, but does mean that if there is some network problem, the game freezes. In practice, it is usually better to perform screen refreshes and data exchange asynchronously.

The figure assumes a two-player game — that is, there is one host and one client. Each has two layers:

1. The application layer
2. The Bluetooth layer

When the game is in play, the following happens on the *host* side:

Application layer:

The host reads its own game data and receives the client game data from cycle n.

The host sends its own plus the client data from cycle n to the client.

The host renders the cycle n animation frame based on the data it received from the client and its own data.

Repeat for cycle n+1.

Bluetooth layer:

The Bluetooth layer transmits all scheduled packages to the specified client and sends an additional message to the local device when the client has correctly received the package.

The Bluetooth layer schedules all received messages to the next layer in the local device.

When the game is in play, this is what happens on the *client* side:

Application layer:

The client receives its own and the host game data from cycle n.

It transmits the game data for the next cycle (n+1).

The client renders the cycle n animation frame based on the data received from the host.

Repeat for cycle n+1.

Bluetooth layer:

The Bluetooth layer transmits all scheduled packages to the host and sends an additional message to the local device when the package has been correctly received on the host side.

The Bluetooth layer schedules all received messages to the next layer in the local device.

Mi: Masters Joystick data of cycle i

Si: Slaves Joystick data of cycle i

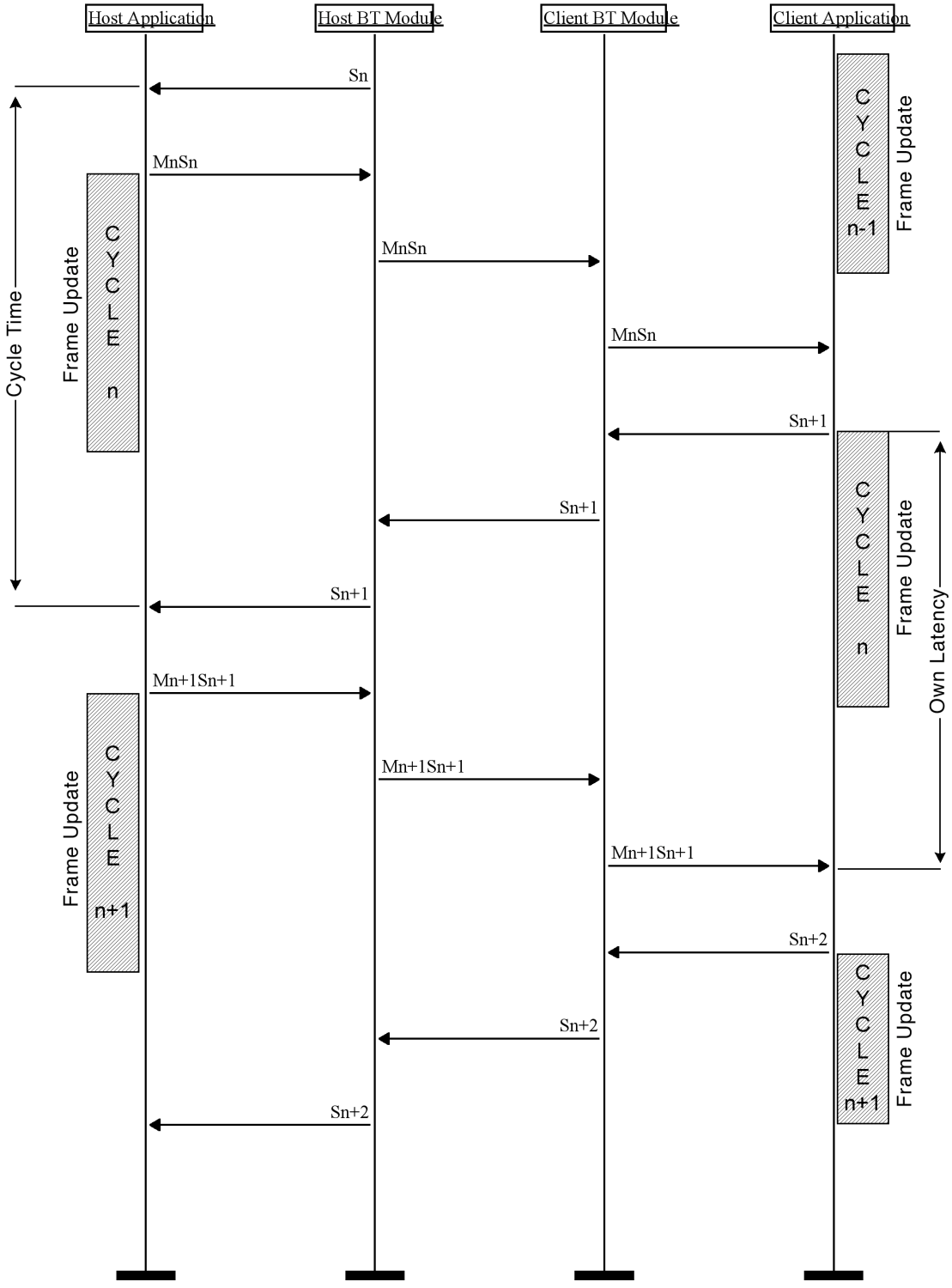

Figure 10: Data exchange sequence chart

3.7 Low-Power Mode

Bluetooth modules that reduce power consumption can affect game latency if developers are not careful.

After a period in which no Bluetooth data is received, the device enters a low-power mode called SNIFF mode. It checks for data transmission at a longer interval, powering back up when data is received.

In Nokia devices, the device enters SNIFF mode after 15 seconds of no data, and then only checks for data every 0.5 seconds. Thus, the first data exchange will suffer up to an additional 0.5 second delay.

You should either ensure that this is tolerable for your games or exchange the equivalent of a "no operation" message every few seconds to ensure that the device does not enter SNIFF mode.

3.8 Disconnection and Link Loss

The game should gracefully handle an unexpected link loss or a disconnection by a user. A link loss will occur when a device does not receive any packet[1] from the remote device for more than 20 seconds.

Players do drop out at times, and given the nature of the networking environment, an accidental loss is also possible.

A link loss on the Symbian OS and Java application side is only detected by a failing read or write operation on Bluetooth. There is no immediate way, when this occurs, to distinguish between a link loss and a purposeful disconnection.

One approach is to have the game application send a message to the host when a player purposefully quits or leaves the game, and assume that in the absence of such a message, a link loss has occurred.

[1] Packet refers to data packets and also to Bluetooth baseband packets, which are not visible to the application.

4 Special Notes for Series 60

4.1 Inquiry

Because your game application needs to get a list of BD addresses to perform service discovery, it is not possible to use the Symbian UI. Additionally, the Symbian UI blocks point-to-multipoint connections, which is a problem for games with more than two players. Thus, the class `CBTDeviceDiscovery` cannot be used if a connection is already established or the list of devices in range needs to be retrieved. Game developers need to create a UI that allows connection to multiple devices. For an example of how to create a point-to-multipoint connection, see Section 4.5, "Connecting to Multiple Devices."

Inquiry for retrieving the BD addresses can be done as shown in this code example. The inquiry is done on a dedicated inquiry address to reduce the number of responses. Call the function `StartDiscoveryL` to start the inquiry. The found BD addresses are stored in the list given as a parameter. To keep the example small and simple, no active objects are used.

If you want to display the addresses during the inquiry, the example must be changed to an active object.

```
#include <e32base.h>
#include <e32std.h>
#include <es_sock.h>
#include <bt_sock.h>
// link agains: esock.lib, euser.lib, bluetooth.lib

/*************************************************************
  Function:     AddToListL
  Parameter:    aEntry              the new entry to add to list
                iInqSockAddrArray   the list to add entry to
  Return:       ---
  Description:  Adds aEntry to the iInqSockAddrArray if not
                already in the list.
                Leaves if no memory for a new item is available.
*************************************************************/
void AddToListL ( TNameEntry& aEntry, RArray<TInquirySockAddr>&
aInqSockAddrArray )
    {
    TInt i;
    TInquirySockAddr& isa = TInquirySockAddr::Cast(aEntry().iAddr);

    // first check if BD address is already in list
    for( i=0; i<aInqSockAddrArray.Count(); ++i )
        {
        if( aInqSockAddrArray[i].BTAddr() == isa.BTAddr() )
            {
            return ;
            }
        }
    TInquirySockAddr *isa_ptr = new (ELeave) TInquirySockAddr( isa );
   CleanupStack::PushL( isa_ptr );
    User::LeaveIfError( aInqSockAddrArray.Append( *isa_ptr ) );
   CleanupStack::Pop();
    }
```

```
/*****************************************************************
  Function:     StartDiscoveryL
  Parameter:    *aInqSockAddrArray  the list pointer to add entry to
                aInquiryAddr        the address for doing inquiry
                                    default is the general inquiry
                                    address KGIAC
                aMajorClass=-1      major part of the CoD field
                                    default, not used
                aMinorClass=-1      minor part of the CoD field
                                    default, not used
                aServiceClass=-1    service part of the CoD field
                                    default, not used
  Return:       ---
  Description:  Starts an inquiry on aInquiryAddr address.
    The found devices are stored in the list aInqSockAddrArray, if
    the given major, minor and service class are matching. As default
    this filtering is inactive.
    aInquiryAddr can be set to KDIAC or KLIAC. Other BD addresses
    are possible but must be selected from the Bluetooth assigned
    numbers range.
    Leaves if no memory for storing address is available.
*****************************************************************/
void StartDiscoveryL( RArray<TInquirySockAddr> *aInqSockAddrArray, TUint
aInquiryAddr, TInt8 aMajorClass, TInt8 aMinorClass, TInt16 aServiceClass )
    {
    #define CAST_ISA(x) (TInquirySockAddr::Cast( (x)().iAddr ))

    RSocketServ      socketServer;
    RHostResolver    hostResolver;
    TProtocolDesc    pInfo;
    TInquirySockAddr addr;
    TRequestStatus   status;
    TNameEntry       nameEntry;

    User::LeaveIfError( socketServer.Connect() );
    CleanupClosePushL<RSocketServ>(socketServer);
    User::LeaveIfError( socketServer.FindProtocol( _L("BTLinkManager"),
pInfo ) );
    User::LeaveIfError( hostResolver.Open( socketServer, pInfo.iAddrFamily,
pInfo.iProtocol ) );
    CleanupClosePushL<RHostResolver>(hostResolver);

    addr.SetIAC( aInquiryAddr );
    addr.SetAction( KHostResInquiry );

    hostResolver.GetByAddress( addr, nameEntry, status );

    while( User::WaitForRequest( status ),
           (status == KErrNone || status == KRequestPending) )
        {
        TInquirySockAddr &isa = TInquirySockAddr::Cast( nameEntry().iAddr
);
        /* add to list, if filter matches */
        if(
            (aMajorClass < 0 || aMajorClass ==
(TInt8)isa.MajorClassOfDevice()) &&
            (aMinorClass < 0 || aMinorClass ==
(TInt8)isa.MinorClassOfDevice()) &&
```

```
            (aServiceClass < 0 || aServiceClass ==
(TInt16)isa.MajorServiceClass())
          )
            {
            AddToListL( nameEntry, *aInqSockAddrArray );
            }
        hostResolver.Next( nameEntry, status );
        }
    //---- finished search
    CleanupStack::PopAndDestroy(2); // socketServer, hostResolver
    }


/* The code part for calling the function can be like this: */

RArray<TInquirySockAddr> *inqSockAddrArray = new (ELeave)
RArray<TInquirySockAddr>;
CleanupStack::PushL( inqSockAddrArray );
/*---- start discovery on dedicated address */
StartDiscoveryL( inqSockAddrArray, KLIAC );
...
CleanupStack::Pop();
```

4.2 Inquiry Scan

To set the scanning to a dedicated address (DIAC) or to set it back to the general address (GIAC), use this function. It is important to restore the inquiry scan address to the general address when the application is terminated.

```
/************************************************************
  Function:     SetInquiyScanAddressL
  Parameter:    TUint aScanAddr
  Return:       ---
  Description:  Sets the inquiry scan address to aScanAddr.
                The address is valid until reset.
************************************************************/
void SetInquiyScanAddressL ( TUint aScanAddr )
    {
    RSocket           socket;
    RSocketServ       socketServer;
    TProtocolDesc     protocolInfo;
    TRequestStatus    status;
    TPckgBuf<TUint32> addr( aScanAddr );

    User::LeaveIfError( socketServer.Connect() );
    CleanupClosePushL<RSocketServ>(socketServer);
    User::LeaveIfError( socketServer.FindProtocol( _L("BTLinkManager"),
protocolInfo ) );
    User::LeaveIfError( socket.Open( socketServer, KBTAddrFamily,
KSockSeqPacket, KL2CAP ) );
    CleanupClosePushL<RSocket>(socket);

    socket.Ioctl( KHCIWriteDiscoverabilityIoctl, status, &addr, KSolBtHCI);
    User::WaitForRequest( status );
    User::LeaveIfError( status.Int() );

    CleanupStack::PopAndDestroy(2); // socketServer, socket
```

```
    }
```

To set the inquiry scan to a dedicated address, simply call the previously declared function like this:

```
/* Set scanning to dedicated/limited address */
SetInquiyScanAddressL ( KLIAC );

…

/* Set scanning back to general address */
SetInquiyScanAddressL ( KGIAC );
```

4.3 Get Local Device Name

To perform remote name requests, use this function:

```
/************************************************************
  Function:     GetLocalNameL
  Parameter:    aHostName, the name to be stored
  Return:       ---
  Description:  Stores the local name in aHostName
************************************************************/
void GetLocalNameL ( THostName& aHostName )
    {
    RSocketServ      socketServer;
    RHostResolver    hostResolver;
    TProtocolDesc    pInfo;
    TRequestStatus   status;

    User::LeaveIfError( socketServer.Connect() );
    CleanupClosePushL<RSocketServ>(socketServer);
    User::LeaveIfError( socketServer.FindProtocol( _L("BTLinkManager"),
pInfo ) );
    User::LeaveIfError( hostResolver.Open( socketServer, pInfo.iAddrFamily,
pInfo.iProtocol ) );
    CleanupClosePushL<RHostResolver>(hostResolver);

    hostResolver.GetHostName( aHostName );

    CleanupStack::PopAndDestroy(2); // socketServer, socket
    }
```

4.4 Service Discovery

4.4.1 Service advertising

The following code is drawn mainly from the Series 60 SDK for Symbian OS Bluetooth point-to-point and advertiser examples, which are included with Series 60 SDK for Symbian OS, Versions 1.0 and 1.2. The documentation of this class is delivered with the two examples. We have simply added a new service with its own UUID. The UUID is built from the real BD address of a Bluetooth device. This follows the recommendations of Section 2.2, “Universally Unique Identifier (UUID),” that a UUID should be built from the device’s BD address with additional digits to identify a specific service on that device.

For this new service, a new database record is created with its own member variable `TSdpServRecordHandle iRecord2`.

The game client has to advertise the service; then a host can search for it and know that the client is running the game.

```
#include <e32base.h>
#include <e32std.h>
#include <es_sock.h>
#include <bt_sock.h>
// link agains: esock.lib, euser.lib, bluetooth.lib

#include <btsdp.h>
//---- My settings
/* 0x0002ee0425b1,
   BD address of an old defective board. This address is used
   to build a UUID for my services.
   Please use one of your own board BD addresses for your service
*/
#define MY_UUID         0x00000000, 0x00001000, 0x80000002, 0xee0425b1
static const TInt KMyServiceId = 0x0100;
static const TInt KSerialClassId = 0x1101;

_LIT(KProtocolName, "BTLinkManager");


/*
    Class to set the local services which can be
    discovered by remote devices.
*/
class CBTModifiedServiceAdvertiser : public CBase
    {
      public:

    static CBTModifiedServiceAdvertiser * NewL();
    static CBTModifiedServiceAdvertiser * NewLC();

    ~CBTModifiedServiceAdvertiser();

    void StartAdvertisingL (TInt aPort, const TDesC8& aHostName);
    void UpdateAvailabilityL(TBool aIsAvailable);
    void StopAdvertisingL (void);
    TBool IsAdvertising (void);

protected:
private:
    CBTModifiedServiceAdvertiser();
    TBool IsLocalServiceOpen (void);
    void LocalServiceOpenL (void);
    void LocalServiceAddL (TInt aPort, const TDesC8& name);

    RSdp iSdpSession;
    RSdpDatabase iSdpDatabase;
    /*
        We have two records, one for RFCOMM
        and one for our own service with own UUID
    */
    TSdpServRecordHandle iRecord, iRecord2;
```

```
    TBool iLocalServiceVisible;
    TBool iLocalServiceOpen;
    TInt iRecordState;
    };


CBTModifiedServiceAdvertiser* CBTModifiedServiceAdvertiser::NewL()
    {
    CBTModifiedServiceAdvertiser* self =
CBTModifiedServiceAdvertiser::NewLC();
    CleanupStack::Pop(self);
    return self;
    }

CBTModifiedServiceAdvertiser* CBTModifiedServiceAdvertiser::NewLC()
    {
    CBTModifiedServiceAdvertiser* self = new (ELeave)
CBTModifiedServiceAdvertiser();
    CleanupStack::PushL(self);
    return self;
    }


CBTModifiedServiceAdvertiser::CBTModifiedServiceAdvertiser() :
        iLocalServiceVisible( FALSE ),
        iLocalServiceOpen( FALSE ),
        iRecord( 0 ),
        iRecord2( 0 ) //---- the new service record
    {
    }

CBTModifiedServiceAdvertiser::~CBTModifiedServiceAdvertiser()
    {
    if( IsAdvertising() )
        {
        TRAPD( err, StopAdvertisingL() );
        if (err != KErrNone)
            {
            User::Panic(_L("CBTModifiedServiceAdvertiser"), err);
            }
        }
    iSdpDatabase.Close();
    iSdpSession.Close();
    }

void CBTModifiedServiceAdvertiser::StartAdvertisingL (TInt aPort, const
TDesC8& aName )
    {
    if( !IsLocalServiceOpen() )
        {
        LocalServiceOpenL();
        LocalServiceAddL( aPort, aName );
        }
    iLocalServiceVisible = TRUE;
    }

void CBTModifiedServiceAdvertiser::UpdateAvailabilityL(TBool aIsAvailable)
```

```
    {
    TUint state;
    if (aIsAvailable)
        {
        state = 0xFF;  //  Fully unused
        }
    else
        {
        state = 0x00;  //  Fully used -> can't connect
        }

    //  Update the availability attribute field
    iSdpDatabase.UpdateAttributeL( iRecord, KSdpAttrIdServiceAvailability,
state );
    iSdpDatabase.UpdateAttributeL( iRecord2, KSdpAttrIdServiceAvailability,
state );

    //  Mark the record as changed - by increasing its state number
(version)
    iSdpDatabase.UpdateAttributeL( iRecord, KSdpAttrIdServiceRecordState,
++iRecordState );
    iSdpDatabase.UpdateAttributeL( iRecord2, KSdpAttrIdServiceRecordState,
iRecordState );
    }

void CBTModifiedServiceAdvertiser::StopAdvertisingL (void)
    {
    if( IsAdvertising() )
        {
        iSdpDatabase.DeleteRecordL( iRecord );
        iRecord = 0;
        iSdpDatabase.DeleteRecordL( iRecord2 );
        iRecord2 = 0;
        }
    iLocalServiceVisible = FALSE;
    }

TBool CBTModifiedServiceAdvertiser::IsAdvertising (void)
    {
    return iLocalServiceVisible;
    }

TBool CBTModifiedServiceAdvertiser::IsLocalServiceOpen (void)
    {
    return iLocalServiceOpen;
    }

void CBTModifiedServiceAdvertiser::LocalServiceOpenL (void)
    {
    if( !IsLocalServiceOpen() )
        {
        User::LeaveIfError( iSdpSession.Connect() );
        User::LeaveIfError( iSdpDatabase.Open( iSdpSession ) );
        iLocalServiceOpen = TRUE;
        }
    }
```

```
void CBTModifiedServiceAdvertiser::LocalServiceAddL (TInt aPort, const
TDesC8& aHostName )
    {
    CSdpAttrValueDES* attrValProt;
    CSdpAttrValueString* attrValString;

    //---- first make serial profil entry
    iSdpDatabase.CreateServiceRecordL( KSerialClassId, iRecord );

    //---- add a Protocol to the record
    TBuf8<2> channel;
    channel.Append( (TChar)(aPort<<8) );
    channel.Append( (TChar)aPort );

    attrValProt = CSdpAttrValueDES::NewDESL( NULL );
    CleanupStack::PushL( attrValProt );

    attrValProt
        ->StartListL()
            ->BuildDESL()
            ->StartListL()   //  Details of lowest level protocol
                ->BuildUUIDL( KL2CAP )
            ->EndListL()

            ->BuildDESL()
            ->StartListL()
                ->BuildUUIDL( KRFCOMM )
                ->BuildUintL( channel )
            ->EndListL()
        ->EndListL();

    iSdpDatabase.UpdateAttributeL( iRecord,
KSdpAttrIdProtocolDescriptorList, *attrValProt );
    CleanupStack::PopAndDestroy( attrValProt );


    //---- add our gaming service to a second record
    iSdpDatabase.CreateServiceRecordL( TUUID( MY_UUID ), iRecord2 );

    attrValString = CSdpAttrValueString::NewStringL( aHostName );
    CleanupStack::PushL( attrValString );

    // Add a name to the record
    iSdpDatabase.UpdateAttributeL( iRecord2, KMyAttributeId, *attrValString
);
    CleanupStack::PopAndDestroy( attrValString );
    }
```

To start advertising, create an object of `CBTModifiedServiceAdvertiser` and call the method `StartAdvertisingL`.

```
CBTModifiedServiceAdvertiser *pBTA = CBTModifiedServiceAdvertiser::NewL();
CleanupStack::PushL( pBTA );

TBuf8<sizeof(THostName)> *name8 = new (ELeave) TBuf8<sizeof(THostName)>;
CleanupStack::PushL( name8 );
THostName *name = new (ELeave) THostName;
```

```
CleanupStack::PushL( name );
GetLocalNameL( *name );
name8->Copy( *name );
CleanupStack::PopAndDestroy( name );

pBTA->StartAdvertisingL( 2, *name8 ); …
…
/* destroy pBTA, when service not needed any more */
CleanupStack::PopAndDestroy(); // all
```

4.4.2 Service search

For service search, please refer to the *Series 60 SDK for Symbian OS Bluetooth Discovery Example*, which is included with Series 60 SDK for Symbian OS, Versions 1.0 and 1.2.

Change the `ConstructL` method to:

```
void CBTDiscoverer::ConstructL()
    {
    iSdpSearchPattern = CSdpSearchPattern::NewL();
    iSdpSearchPattern->AddL( TUUID( MY_UUID ));

    iMatchList = CSdpAttrIdMatchList::NewL();
//    iMatchList->AddL(TAttrRange(0x0000, 0xFFFF)); // get them all
    iMatchList->AddL( TAttrRange( KMyServiceId ) );

    iAgent = NULL;
    }
```

Use `MY_UUID` and `KMyServiceId` as defined in Section 4.4.1, “Service Advertising.”

In method `TBTAttributeValueLister::VisitAttributeValueL`, change `TBuf<32> buf;` to `TBuf<40> buf;`

4.5 Connecting to Multiple Devices

The following example shows how to connect to multiple devices.

Since the intention of this document is to show what rules should be followed to make Bluetooth games, this example just shows the principles involved in making point-to-multipoint connections. Thus, it does not use active objects, because that would make the example quite complex. In addition, the example does not have complete error checking, such as checking to see if a connection is already opened or already closed. This is left as an exercise for the reader.

The code snippet defines a new class for creating a connection. The `ConnectL` function opens a Bluetooth RFCOMM connection. The parameters are the BD address of the device to connect to and the port number for the RFCOMM channel number.

The `close` function closes a BT connection.

It is possible to connect to up to seven devices.

```
//---- the max number of connections
const TInt KConMax = 7;
```

```
class CConnector
    {
public:
    /*!
      @function CConnector

      @discussion the constructor, since no data is allocated no two way
construction is needed
    */
    CConnector();
    /*!
      @function ~CConnector

      @discussion Destroy the object and release all memory objects. Close
any open sockets
    */
    ~CConnector();

    /*!
      @function ConnectL

      @discussion Connect to a device
      @param aBtAddr the device address to the device to connect to
      @param aPort   the RFCOMM channel number to connect to
    */
    TInt ConnectL(const TBTDevAddr& aBtAddr, TInt aPort);
    /*!
      @function Connect

      @discussion close connection to a device
    */
    void Close();

private:

    /*! @var iSocketServer a connection to the socket server */
    RSocketServ iSocketServer;

    /*! @var iSendingSocket a socket to connect with */
    RSocket iSocket;

    /*! @var iState internal state, which shows if connection is open */
    TInt iState;
    };

CConnector::CConnector ()
    {
    iState = 0;
    User::LeaveIfError(  iSocketServer.Connect() );
    }

CConnector::~CConnector()
    {
    Close();
    iSocketServer.Close();
    }

TInt CConnector::ConnectL(const TBTDevAddr& aBtAddr, TInt aPort)
```

```
    {
    TBTSockAddr address;
    TRequestStatus status;

    User::LeaveIfError( iSocket.Open( iSocketServer, _L("RFCOMM") ) );

    address.SetBTAddr( aBtAddr );
    address.SetPort( aPort );

    iSocket.Connect( address, status );

#ifdef __WINS__
    User::After( 1 );     // Fix to allow emulator client to connect to
server
#endif

    User::WaitForRequest( status );
    iState = 1;
    if( status != KErrNone )
        {
        Close();
        }

    return status.Int();
    }

void CConnector::Close()
    {
    if( iState )
        {
        iState = 0;
        iSocket.Close();
        }
    }
```

The class can be used as shown in the following code snippet:

```
    /* Connect to device
       I assume the addresses to connect to are in a
RArray<TInquirySockAddr>& aInqSockAddrArray
       I also assume that the RFCOMM channel numbers are stored in
RArray<TInt>& aChannelArray
    */
        CConnector *connectors = new (ELeave) CConnector[KConMax];
        CleanupStack::PushL( connectors );
        TInt i;
        /* connect to multiple devices, because the function ConnectL
           waits until a connection is created, we can connect all devices
           in a loop. */
        for( i=0; i<aInqSockAddrArray.Count() && i<KConMax; ++i )
            {
            /* here some better error handling should be done */
            if( connectors[i].ConnectL( aInqSockAddrArray[i].BTAddr(),
aChannelArray[i] ) != KErrNone )
                {
                break;
                }
```

```
            }

/* Do what you want to do with the multipoint connections
   You have to check here, if all connections are established
   Than you can transmit data or receive data, or what ever ...
*/

        // close all connections
        for( i=0; i<aInqSockAddrArray.Count() && i<KConMax; ++i )
            {
            connectors[i].Close();
            }

        /* before destruct the classes, there should be some waiting time
           after close to be sure that everything has gracefully closed. */

        delete [] connectors;
        CleanupStack::Pop();
```

4.6 Preferred Bluetooth Protocol

Currently, the RFCOMM protocol is easier to use with Series 60 devices than L2CAP.

5 Special Notes on JSR-82

5.1 Slave Wait-Until-Connected Code

This code can be used for a client that waits until it receives an invitation to join a game.

```
// Obtain local device object
LocalDevice local_device = LocalDevice.getLocalDevice();
// Obtain discovery agent object
DiscoveryAgent disc_agent = local_device.getDiscoveryAgent();

// Set device into limited access mode. Inquiry scan
// will listen only to LIAC.
local_device.setDiscoverable( DiscoveryAgent.LIAC );

// Do the service search on all found devices.
// Note: don't use this UUID in your own MIDlets.
// You have to create an own UUID for each MIDlet that you write:
String service_UUID = "00000000000010008000006057028C19";

// Retrieve players name
// This could be a name that user has modified and stored to
// non-volatile memory. As default (when game is first started)
// the local friendly name could be chosen.
String player_name = local_device.getFriendlyName();

// Open connection, note: name is attribute ID 0x0100
String url = "btspp://localhost:" + service_UUID + ";name=" +
   player_name ;
StreamConnectionNotifier notifier = (StreamConnectionNotifier)
   Connector.open( url );

// Wait on someone to connect (note: you can cancel this wait
// only if you call notifier.close() from another thread.
// This is important if you want to offer a UI for the user
// to cancel connections setup.)
StreamConnection con = (StreamConnection) notifier.acceptAndOpen();

// open input stream
InputStream is = con.openInputStream();
// open output stream
OutputStream os = con.openOutputStream();



// Devices are connected now:
// Run the game / exchange data ...


// Multiplayer session is stopped: Disconnect the devices
// close input stream
is.close();
// close output stream
os.close();
```

```
// Close connection
con.close();
```

5.2 Master Connect-All-Slaves Code

```
// Obtain local device object
LocalDevice local_device = LocalDevice.getLocalDevice();
// Obtain discovery agent object
DiscoveryAgent disc_agent = local_device.getDiscoveryAgent();

// Disable page scan and inquiry scan
local_device.setDiscoverable( DiscoveryAgent.NOT_DISCOVERABLE );


// create inquiry listener object
InquiryListener inq_listener = new InquiryListener();

synchronized( inq_listener )
{
    // start a limited access inquiry and install inquiry listener
    disc_agent.startInquiry( DiscoveryAgent.LIAC, inq_listener );
    // Wait
    inq_listener.wait();
}

// Do the service search on all found devices
UUID[] u = new UUID[1];
// Note: don't use this UUID in your own MIDlets.
// You have to create an own UUID for each MIDlet that you write:
u[0] = new UUID( "00000000000010008000006057028C19", false );
int attrbs[] =
{ 0x0100 };
// Retrieved service record should include player
// name (service name)
Enumeration devices = inq_listener.cached_devices.elements();
ServiceListener serv_listener = new ServiceListener();

while( devices.hasMoreElements() )
{
    synchronized( serv_listener )
    {
        // on each device do a service search
        disc_agent.searchServices( attrbs, u, (RemoteDevice)
           devices.nextElement(), serv_listener );
        // Wait
        serv_listener.wait();
    }

}

// Now all devices which offer the requested service (game)
// can be found in the serv_listener.FoundServiceRecord list.

// This list would now be presented to the user and user can filter
// this list/ select one or more devices to connect to.
// Or in other words: remove all devices from FoundServiceRecords
```

```
        // list that you don't want to connect to.
        // Here we just print all the (remote) player names
        int number_of_players = serv_listener.FoundServiceRecords.size();
        String player_name;

        for( int i = 0; i < number_of_players; i++ )
        {
            // Retrieve player name which is contained as service name
            // (0x0100) in the service record
            player_name = (String) ((ServiceRecord)serv_listener.
                FoundServiceRecords.elementAt(i)).getAttributeValue(
                0x0100 ).getValue();
            // print name
            System.out.println( player_name );
        }

        // After filtering these devices will be connected:
        String url;
        StreamConnection con[] = new StreamConnection[number_of_players];
        InputStream is[] = new InputStream[number_of_players];
        OutputStream os[] = new OutputStream[number_of_players];

        for( int i = 0; i < number_of_players; i++ )
        {
            // Retrieve url for one device/service
            url = ((ServiceRecord) serv_listener.FoundServiceRecords.
                elementAt( i )).getConnectionURL( 0, false );
            // Open connection
            con[i] = (StreamConnection) Connector.open( url );
            // open input stream
            is[i] = con[i].openInputStream();
            // open output stream
            os[i] = con[i].openOutputStream();
        }


        // Devices are connected now:
        // Run the game / exchange data ...


        // Multiplayer session is stopped: Disconnect the devices
        for( int i = 0; i < number_of_players; i++ )
        {
            // close input stream
            is[i].close();
            // close output stream
            os[i].close();
            // Close connection
            con[i].close();
        }


    // For every found device the deviceDiscovered method is called.
    // If inquiry is finished (about 10 seconds) the inquiryComplete method
    // is called.
    private class InquiryListener
    implements DiscoveryListener
```

```
{
    // List for storing the found devices
    public Vector cached_devices;

    /**
     * Constructor
     */
    public InquiryListener()
    {
        cached_devices = new Vector();
    }

    /**
     * Called when a device is found during an inquiry.  An inquiry
     * searches for devices that are discoverable.  The same device may
     * be returned multiple times.
     */
    public void deviceDiscovered( RemoteDevice btDevice,
        DeviceClass cod )
    {
        // Filter CoD: Ie. only store device in case the device is
        // a phone.
        // Note that cod in Concept SDK is incorrect.
        int major = cod.getMajorDeviceClass();
        if( major == 0x0200 )
        {   // It's another phone, so store it in the list
            if( ! cached_devices.contains( btDevice ) )
            {   // But only if it's not already in the list
                // (same device might be reported multiple times)
                cached_devices.addElement( btDevice );
            }
        }
    }


    /**
     * Called when an inquiry is completed.
     */
    public void inquiryCompleted( int discType )
    {
        // Inquiry is finished.
        synchronized( this )
        {
            this.notify();
        }
    }

    /**
     * Not used for inquiry, but need to be declared:
     */
    public void servicesDiscovered( int transID, ServiceRecord[]
        servRecord )
    {}
    public void serviceSearchCompleted( int transID, int respCode )
    {}
}
```

```
// Listens on servicesDiscovered events
private class ServiceListener
implements DiscoveryListener
{
    // holds all the found service record
    private Vector FoundServiceRecords;

    /**
     * Constructor
     */
    public ServiceListener()
    {
        FoundServiceRecords = new Vector();
    }


    /**
     * servicesDiscovered.
     * Is called when a service was found.
     * Sets the FoundServiceRecord so that service search will stop.
     **/
    public void servicesDiscovered( int transID, ServiceRecord[]
        servRecord )
    {
        DataElement sn;  // for service name

        // A Service was found on the device.
        // Only the first is interesting (ie. there is only one because
        // we searched only for one)
        // Add service to list:
        FoundServiceRecords.addElement( servRecord[0] );
    }

    /**
     * serviceSearchCompleted.
     * Is called when the service search is ready.
     */
    public void serviceSearchCompleted( int transID, int respCode )
    {
        synchronized( this )
        {
            this.notify();
        }
    }

    /**
     * Not used for service discovery, but need to be declared:
     */
    public void deviceDiscovered( RemoteDevice btDevice,
        DeviceClass cod )
    {}
    public void inquiryCompleted( int discType )
    {}
}
```

5.3 Detecting Point-to-Multipoint Capabilities

To discover the point-to-multipoint capabilities of the local device, use the following example:

```
int n ;
String s;

s = LocalDevice.getLocalDevice().getProperty(
    "bluetooth.connected.devices.max" );
n = Integer.parseInt( s );
```

With the `getProperty` method, you retrieve the number of devices your local device can simultaneously connect to. `getProperty` returns a `String`, so you have to convert it to an `int` to use it. If the result is 1, your device has no point-to-multipoint capability. This means it can only connect to one other device at a time, limiting you to two-player games.The maximum number you should expect here is 7, which limits your game to a maximum of eight players.

6 Terms and Abbreviations

Term or abbreviation	Meaning
CoD	Class of device; for more details see Section 2.1.1, “Inquiry.”
Cycle	A data exchange between the host and *all* clients.
DH packet	One of the packet types supported by Bluetooth. “DH” stands for “Data/High bandwidth.” DH is normally followed by a number, which corresponds to the number of “slots” a packet of this size will span. DH packets are larger than DM packets, but provide only error detection, not correction.
DIAC	Dedicated Inquiry Access Code. Used during the inquiry process to query for the existence of nearby Bluetooth devices that offer a specific service. Sometimes called LIAC.
DM packet	One of the packet types supported by Bluetooth. “DM” stands for “Data/Medium bandwidth.” DM is normally followed by a number, which corresponds to the number of “slots” a packet of this size will span. DM packets are smaller than DH packets, but provide error correction as well as detection.
Frame rate	The number of video frames displayed per second by a device; with a computing device, this can be slowed if computing graphics and displaying them takes too long.
GIAC	General Inquiry Access Code. Used during the inquiry process to determine the existence of all nearby Bluetooth devices.
Inquiry	The inquiry process is the process by which Bluetooth devices determine what devices are nearby (see GIAC and DIAC).
L2CAP	Logical Link Control and Adaptation Protocol. This is the lowest-level Bluetooth protocol; the protocol overhead is just 4 bytes. Recommended for use if you need quick response times.
Latency, cycle	The amount of time required to receive data from all slaves, and distribute data to all slaves for one game cycle.
Latency, own	The amount of time required for two devices to exchange data for one game cycle. In a head-to-head game, this is equivalent to cycle latency; in a three- or more player game, it is shorter than or equal to cycle latency, and when talking about any master-slave pair, may depend on the order of polling.
LIAC	Limited Inquiry Access Code. Same as DIAC.
Master	The host of a Bluetooth piconet.

Piconet	A small network consisting of one master and multiple slaves with no direct connection between slaves; in Bluetooth, a piconet is limited to eight machines (one master and seven slaves).
RFCOMM	A Bluetooth serial line emulation protocol based on L2CAP. The protocol overhead is between 4 and 5 bytes for small packets, plus an additional byte for each 127 bytes of data, in addition to the L2CAP overhead of 4 bytes, so that for less than 127 bytes of data, the total overhead is 8 to 9 bytes.
Round-trip time	See Latency, cycle
Scatternet	A network consisting of multiple piconets linked through nodes that are themselves slave to multiple piconets, or slave to some and master to some. Although Bluetooth can in principle support a scatternet architecture, at present most Bluetooth stacks do not support it.
Service Discovery	The process by which Bluetooth devices determine what services are offered by nearby devices.
Slave	Clients connected to a master in a piconet.
Slot	In Bluetooth terminology, one slot is 625 microseconds. This is typically the amount of time required to send the smallest size of packet.
SDP	Service Discovery Protocol. Protocol that is defined in the Bluetooth Core Specification (see Part E, Service Discovery Protocol).
UUID	Universally Unique Identifier. UUIDs are used in the service discovery process (see Bluetooth Core Specification, Part E, Service Discovery Protocol); a UUID is assigned for each service class. It is a 128-bit value that is guaranteed to be unique. Some UUIDs are defined by the Bluetooth specification; others may be defined by the application.

7 References

Specification of the Bluetooth System, Version 1.1, Volume 1: Core
http://www.bluetooth.org/spec/

Specification of the Bluetooth System, Version 1.1, Volume 1: Core, Part B: Baseband Specification
http://www.bluetooth.org/spec/

ISO/IEC 11578:1996 Information Technology — Open Systems Interconnection — Remote Procedure Call (RPC)
http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=2229

Java™ APIs for Bluetooth (JSR-82)
http://jcp.org/aboutJava/communityprocess/final/jsr082/index.html

Symbian C++ APIs for Bluetooth
http://www.symbian.com/developer/