Bluetooth Programming

- Software development for Bluetooth is very much similar (as far as semantics go) to TCP/IP software development.
- The basic steps can be summarized as follows:
    - Selecting a target device to communicate with
    - Selecting a communication (Transport) protocol for the communication session
    - Initiating an outbound connection
    - Accepting an incoming connection
    - Sending/Receiving data
    - Terminating the connection
- The following table shows a comparison for Bluetooth and TCP/IP programming steps for an outbound connection:

Outbound Connections	
TCP/IP Programming	**Bluetooth Programming**
Address/Name Resolution (DNS)	Search for devices (Device Inquiry)
	Query each device for its display name
	Select device with user-specified name
Select a Transport Protocol (TCP, UDP)	Select a Transport Protocol (RFCOMM, L2CAP, SCO)
	Search target device for Service Discovery Protocol (SDP) records matching a predefined identifier (UUID, name, etc)
Select a user-specified or well-known port number	Select a port number on matching record
socket (…) connect (…)	socket (…) connect (…)
send (…); recv (…)	send (…); recv (…)
close (…)	close (…)

- The following table shows a comparison for Bluetooth and TCP/IP programming steps for an inbound connection:

Incoming Connections	
TCP/IP Programming	**Bluetooth Programming**
Select a Transport Protocol (TCP, UDP)	Select a Transport Protocol (RFCOMM, L2CAP, SCO)
Use the service (open) port number	Use a hard-coded (open) or dynamically assigned port number
socket (…) bind (…) listen (…)	socket (…) bind (…) listen (…)
	Optionally, advertise service with the local SDP server
accept (…)	accept (…)
send (…); recv (…)	send (…); recv (…)
close (…)	close (…)

Selecting a Target device

- Each Bluetooth device is assigned a globally unique 48-bit address, (similar to an Ethernet MAC address) which is referred to as the **Bluetooth address** or **device address**.

- This device address is used through all of the layers of the Bluetooth stack, from the low-level radio protocols to the higher-level application protocols.

- In contrast, TCP/IP network devices that use Ethernet as the physical layer address only.

- In TCP/IP programming, the user will typically supply a host name, such as milliways.bcit.ca, which the client application must translate to an IP address using DNS query. In Bluetooth, the user will typically supply a user-friendly name (if known), such as “CellPhone”, which the application translates to a numerical address by searching nearby Bluetooth devices.

- **Device Names**

  - Bluetooth devices will almost always have a user-friendly name. For devices such as cell phones and personal computers, this name is configurable and the user can choose an arbitrary word or phrase.

  - Note that there is no central naming authority and names can sometimes be the same, the client program still has to translate from the user-friendly names presented by the user to the underlying numerical addresses.

  - In TCP/IP, this involves contacting a name server, issuing a query, and waiting for a result. In Bluetooth, where there are no name servers, a client will instead broadcast inquiries to see what other devices are nearby and query each detected device for its name.

  - The client then chooses whichever device has a name that matches the one supplied by the user.

Selecting a Transport Layer Protocol

- Both Bluetooth and TCP/IP programming techniques can select a specific Transport layer protocol, depending on the particular application's features and capabilities.

- In TCP/IP, many applications use either TCP or UDP, both of which rely on IP as the delivery protocol.

- While Bluetooth does not have exactly equivalent protocols, it does provide protocols which can be used in the same contexts as the TCP/IP protocols.

- **RFCOMM and TCP**

  - The RFCOMM protocol provides roughly the same level of service and reliability as TCP.

  - The main difference between TCP and RFCOMM from a network developer's perspective is the choice of port number. Whereas TCP supports up to 65535 open ports on a single machine, RFCOMM only allows for 30.

- **L2CAP and UDP**

  - In TCP/IP, UDP is often used in situations where reliable delivery of every packet is not a requirement, Specifically, UDP is used for its best-effort, simple datagram semantics. These are the same criteria that L2CAP satisfies as a communications protocol.

  - L2CAP, by default, provides a connection-oriented protocol that reliably sends individual datagrams of fixed maximum length. The default maximum length is 672 bytes, but this can be negotiated up to 65535 bytes

  - The main difference here is that L2CAP, unlike UDP, will establish a connection to transfer data.

  - The L2CAP specification actually allows for both connectionless and connection-based channels, but connectionless channels are rarely used in practice, since sending "connectionless" data to a device requires joining its Piconet.

  - Joining a Piconet is a time consuming process that is merely establishing a connection at a lower level, so in that sense connectionless L2CAP channels afford no advantages over connection-oriented channels.

- L2CAP can be configured for varying levels of reliability. To provide this service, the transport protocol that L2CAP is built on (Asynchronous Connection-Less logical transport), employs a transmit/acknowledgement scheme, where unacknowledged packets are retransmitted.

- There are three policies an application can use:

    - Never retransmit
    - Retransmit until total connection failure (the default)
    - Drop a packet and move on to queued data if a packet hasn't been acknowledged after a specified time limit (0-1279 milliseconds). This is useful when data must be transmitted in a timely manner.

- Note that adjusting the delivery semantics for a single L2CAP connection to another device affects all L2CAP connections to that device.

- If an application adjusts the delivery semantics for an L2CAP connection to another device, it must also ensure that there are no other L2CAP connections to that device.

- Additionally, since RFCOMM uses L2CAP as a transport, all RFCOMM connections to that device are also affected.

- While this is not a problem if only one Bluetooth connection to that device is expected, it is possible to adversely affect other Bluetooth applications that also have open connections.

- Ignoring the limitations of simplifying the delivery semantics for L2CAP, it does suffice as a suitable transport protocol when the application does not need the overhead and streams-based nature of RFCOMM, and can be used in many of the same applications that UDP is typically used in.

- The following table summarizes the usage of the different protocols:

Requirement	TCP/IP	Bluetooth
Reliable, streams-based	TCP	RFCOMM
Reliable, datagram	TCP	RFCOMM or L2CAP with infinite retransmit
Best-effort, datagram	UDP	L2CAP (0-1279 ms retransmit)

Port numbers and the Service Discovery Protocol

- Once the device address and the Transport protocol are known, the application needs to select a port number.

- The process of selecting a port is very similar to TCP/IP, but Bluetooth is uses a slightly different terminology.

- In L2CAP, ports are called **Protocol Service Multiplexers**, and can take on odd-numbered values between 1 and 32767.

- In RFCOMM, **channels** 1-30 are available for use.

- These differences aside, both protocol service multiplexers and channels serve the exact same purpose that ports do in TCP/IP.

- L2CAP, unlike RFCOMM, has a range of reserved port numbers (1-1023) that are not to be used for custom applications and protocols.

- This information is summarized in the following table:

Protocol	Terminology	Reserved/Well-known ports	Dynamically assigned ports
TCP	port	1-1024	1025-65535
UDP	port	1-1024	1025-65535
RFCOMM	channel	none	1-30
L2CAP	PSM	odd numbered 1-4095	odd numbered 4097 - 32765

- Bluetooth transport protocols were designed with many fewer available port numbers, which means we cannot choose an arbitrary port number at design time.

- Although this problem is not as significant for L2CAP, which has around 15,000 unreserved port numbers, RFCOMM has only 30 different port numbers.

- A consequence of this is that there is a greater than 50% chance of port number collision with just 7 server applications. Clearly, the selection of port numbers cannot be arbitrary. The Bluetooth stack mitigates this problem using the Service Discovery Protocol (SDP).

- Instead of agreeing upon a port to use at application design time, the Bluetooth approach is to assign ports at runtime and follow a publish-subscribe model.

- The host device operates a server application, called the SDP server, which uses one of the few L2CAP reserved port numbers.

- Other server applications are dynamically assigned port numbers at runtime and register a description of themselves and the services they provide (along with the port numbers they are assigned) with the SDP server.

- Client applications will then query the SDP server (using the well defined port number) on a particular machine to obtain the information they need.

- This raises the question of how do clients know which description is the one they are looking for. The standard way of doing this in Bluetooth is to assign a 128-bit number, called the Universally Unique Identifier (UUID), at design time.

- Following a standard method (RFC 4122) of choosing this number guarantees choosing a UUID which is different from those chosen by anyone else following the same method. Thus, a client and server application both designed with the same UUID can provide this number to the SDP server as a search term.

- Note that SDP is not even required to create a Bluetooth application. It is quite possible to revert to the TCP/IP way of assigning port numbers at design time and hoping to avoid port conflicts, and this might often be done to save some time.

- In controlled settings this approach is quite reasonable, however, to create a portable application that will run in the greatest number of scenarios, the application should use dynamically assigned ports and SDP.

Establishing connections and transferring data

- As it turns out, establishing connections and transferring data are the easiest components of designing and implementing Bluetooth applications.

- In writing a server application, once the transport protocol and port number to listen on are chosen, building the rest of the application is essentially the same type of programming most TCP/IP network developers are familiar with.

- A server application waiting for an incoming Bluetooth connection is conceptually the same as a server application waiting for an incoming TCP/IP connection.

- A client application attempting to establish an outbound connection appears the same whether it is using RFCOMM, L2CAP, TCP, or UDP.

- Once the connection has been established, the application will operate with the same guarantees, constraints, and error conditions as those in a TCP/IP application.

- Depending on the transport protocol chosen, packets may be dropped or delayed. Connections may be severed due to host or link failures.

- External factors such as congestion and interference may result in decreased quality of service.

The Bluetooth API using BlueZ

- Currently, BlueZ is a powerful Bluetooth communications stack with an extensive API that allows a user to fully exploit all local Bluetooth resources.

- Not that BlueZ does not have any official or unofficial documentation available. This is quite often enough of an obstacle to deter many potential developers.

Selecting a Target device

- The basic data structure used to specify a Bluetooth device address is **the bdaddr_t**. All Bluetooth addresses in BlueZ will be stored and manipulated as **bdaddr_t** structures.

```
typedef struct {
  uint8_t b[6];
} __attribute__((packed)) bdaddr_t;
```

- BlueZ also provides two convenient functions to convert between strings and bdaddr_t structures.

  int str2ba (const char *str, bdaddr_t *ba);

  int ba2str (const bdaddr_t *ba, char *str);

  - **str2ba** takes an string of the form "XX:XX:XX:XX:XX:XX", where each XX is a hexadecimal number specifying an octet of the 48-bit address, and packs it into a 6-byte bdaddr_t.

  - **ba2str** does exactly the opposite.

- Local Bluetooth adapters are assigned identifying numbers starting with 0, and a program must specify which adapter to use when allocating system resources.

- The following is an example of the output from the ***hciconfig*** command:

  [root@tempest ~]# hciconfig

  hci0: Type: USB

  BD Address: 00:0C:41:E2:7C:59 ACL MTU: 192:8 SCO MTU: 64:8

  UP RUNNING PSCAN

  RX bytes:663 acl:0 sco:0 events:19 errors:0

  TX bytes:319 acl:0 sco:0 commands:18 errors:0

- Usually, there is only one adapter or it doesn't matter which one is used, so passing NULL to **hci_get_route** will retrieve the resource number of the first available Bluetooth adapter.

  int hci_get_route(bdaddr_t *bdaddr);

  int hci_open_dev(int dev_id);

- It is not a good idea to hard-code the device number 0, because that is not always the id of the first adapter. For example, if there were two adapters on the system and the first adapter (id 0) is disabled, then the first available adapter is the one with id 1.

- In a system with multiple Bluetooth adapters present, to select the adapter with address "**00:0C:41:E2:7C:59"**, pass the char * representation of the address to **hci_devid** and use that in place of **hci_get_route**.

- This is done as follows:

  int dev_id = hci_devid ("00:0C:41:E2:7C:59");

- Most Bluetooth operations require the use of an open socket. **hci_open_dev** is a function that opens a Bluetooth socket with the specified resource number.

- Note that the socket opened by hci_open_dev represents a connection to the microcontroller on the specified local Bluetooth adapter, and not a connection to a remote Bluetooth device.

- This is due to the fact that performing low level Bluetooth operations involves sending commands directly to the microcontroller with this socket.

- Once the local Bluetooth adapter has been selected, the appliation is ready to scan for nearby Bluetooth devices.

- For example, the hci_inquiry function performs a Bluetooth device discovery and returns a list of detected devices and some basic information about them in the variable ii. On error, it returns -1 and sets errno accordingly.

int hci_inquiry(int dev_id, int len, int max_rsp, const uint8_t *lap,

inquiry_info **ii, long flags);

- **hci_inquiry** is one of the few functions that requires the use of a resource number instead of an open socket, so we use the dev_id returned by hci_get_route.

- The inquiry lasts for at most 1.28 * len seconds, and at most max_rsp devices will be returned in the output parameter ii, which must be large enough to accommodate max_rsp results. It is suggested that using a max_rsp of 255 for a standard 10.24 second inquiry will suffice

- If flags is set to IREQ_CACHE_FLUSH, then the cache of previously detected devices is flushed before performing the current inquiry.

- Otherwise, if flags is set to 0, then the results of previous inquiries may be returned, even if the devices are not within range or active anymore.

The **inquiry_info** structure is defined as

```
typedef struct {
   bdaddr_t    bdaddr;
   uint8_t     pscan_rep_mode;
   uint8_t     pscan_period_mode;
   uint8_t     pscan_mode;
   uint8_t     dev_class[3];
   uint16_t    clock_offset;
} __attribute__ ((packed)) inquiry_info;
```

- For the most part, only the first entry - the **bdaddr** member, returns the address of the detected device.

- Occasionally, there may be a use for the dev_class field, which gives information about the type of device detected (i.e. if it's a printer, phone, desktop computer, etc.) and is described in the Bluetooth Assigned Numbers specification.

- The rest of the fields are used for low level communication, and are not useful for most purposes.

- Once a list of nearby Bluetooth devices and their addresses has been found, the program determines the user-friendly names associated with those addresses and presents them to the user.

- The **hci_read_remote_name** function is used for this purpose.

  int hci_read_remote_name(int sock, const bdaddr_t *ba, int len,

  char *name, int timeout)

- **hci_read_remote_name** tries for at most **timeout** milliseconds to use the socket sock to query the user-friendly name of the device with Bluetooth address **ba**.

- Upon success, hci_read_remote_name returns 0 and copies at most the first **len** bytes of the device's user-friendly name into name. On failure, it returns -1 and sets errno accordingly.
- The example provided will detect Bluetooth devices in the proximity and then looks up the user friendly name for each detected device.
- Compile the program using ***gcc*** and link with the Bluetooth library:

    # gcc -o devscan devscan.c -lbluetooth

RFCOMM sockets

- Establishing and using RFCOMM connections boils down to the same socket programming techniques as those used for TCP/IP programming.

- The only difference is that the socket addressing structures are different, and we use different functions for byte ordering of multibyte integers.

Addressing Data Structures

- To establish an incoming or outgoing RFCOMM connection with another Bluetooth device, create and fill out the **sockaddr_rc** structure.

- Like the **sockaddr_in** structure that is used in TCP/IP, the addressing structure specifies the details of an outgoing connection or listening socket.

```
struct sockaddr_rc {
        sa_family_t rc_family;
        bdaddr_t    rc_bdaddr;
        uint8_t             rc_channel;
};
```

- The **rc_family** member specifies the addressing family of the socket, and will always be set to **AF_BLUETOOTH**.

- For outgoing connections, **rc_bdaddr** and **rc_channel** specify the Bluetooth address and port number to connect to, respectively.

- For a listening socket**, rc_bdaddr** specifies the local Bluetooth adapter to use, and is typically set to **BDADDR_ANY** to indicate that any local Bluetooth adapter is acceptable.

- For listening sockets, **rc_channel** specifies the port number to listen on.

Byte Ordering

- Unlike network byte ordering, which uses a big-endian format, Bluetooth byte ordering is little-endian, where the least significant bytes are transmitted first.

- BlueZ provides four functions to convert between host and Bluetooth byte orderings.

  unsigned short int htobs(unsigned short int num);

  unsigned short int btohs(unsigned short int num);

  unsigned int htobl(unsigned int num);

  unsigned int btohl(unsigned int num);

- Like their network order counterparts, these functions convert 16 and 32 bit unsigned integers to Bluetooth byte order and back.

- They are used when filling in the socket addressing structures, communicating with the Bluetooth microcontroller, and when performing low level operations on transport protocol sockets.

Dynamically assigned port numbers

- The rc_channel field of the socket addressing structure used to bind the socket is simply set to 0, and the kernel binds the socket to the first available port.

- The following function illustrates how to do this for RFCOMM sockets.

```
int dynamic_bind_rc(int sock, struct sockaddr_rc *sockaddr,
uint8_t *port)
{
   int err;
   for ( *port = 1; *port <= 31; *port++ )
    {
         sockaddr->rc_channel = *port;
         err = bind(sock, (struct sockaddr *)sockaddr,
sizeof(sockaddr));
         if ( ! err || errno == EINVAL )
               break;
   }
   If ( port == 31 )
   {
         err = -1;
         errno = EINVAL;
   }
   return err;
}
```

- The process for L2CAP sockets is similar, but tries odd-numbered ports 4097-32767 (0x1001 - 0x7FFF) instead of ports 1-30.

RFCOMM summary

- Advanced TCP options that are often set with setsockopt, such as receive windows and the Nagle algorithm, don't make sense in Bluetooth, and can't be used with RFCOMM sockets.

- Aside from this, the byte ordering, and socket addressing structure differences, programming RFCOMM sockets is virtually identical to programming TCP sockets.

- To accept incoming connections with a socket, use bind to reserve operating system resource, listen to put it in listening mode, and accept to block and accept an incoming connection.

- Creating an outgoing connection is also simple and merely involves a call to connect. Once a connection has been established, the standard calls to read, write, send, and recv can be used for data transfer.

- The examples show a simple Server and Client application. The code will establish a connection using an RFCOMM socket, transfer some data, and disconnect. For simplicity, the client is hard-coded to connect to "**00:1B:41:01:61:C7**".

- Most of the code is similar to TCP/IP programming. As with Internet programming, first allocate a socket with the socket system call. Instead of AF_INET, use AF_BLUETOOTH, and instead of IPPROTO_TCP, use BTPROTO_RFCOMM. Since RFCOMM provides the same delivery semantics as TCP, SOCK_STREAM can still be used for the socket type.

L2CAP sockets

- As with RFCOMM, L2CAP communications are structured around socket programming.

- By default, L2CAP connections provide reliable datagram-oriented connections with packets delivered in order, so the socket type is **SOCK_SEQPACKET**, and the protocol is **BTPROTO_L2CAP**.

- The addressing structure, **sockaddr_l2** differs slightly from the RFCOMM addressing structure:

```
struct sockaddr_l2 {
   sa_family_t    l2_family;
   unsigned short  l2_psm;
   bdaddr_t       l2_bdaddr;
};
```

- The **l2_psm** field specifies the L2CAP port number to use. Since it is a multibyte unsigned integer, byte ordering is significant.

Maximum Transmission Unit

- Occasionally, an application may need to adjust the maximum transmission unit (MTU) for an L2CAP connection and set it to something other than the default of 672 bytes.

- In BlueZ, this is done with the getsockopt and setsockopt functions.

```
struct l2cap_options {
   uint16_t    omtu;
   uint16_t    imtu;
   uint16_t    flush_to;
   uint8_t     mode;
};

int set_l2cap_mtu( int sock, uint16_t mtu )
{
      struct l2cap_options opts;
      int optlen = sizeof(opts), err;
      err = getsockopt ( s, SOL_L2CAP, L2CAP_OPTIONS, &opts,
&optlen );
      if( ! err )
      {
            opts.omtu = opts.imtu = mtu;
            err = setsockopt( s, SOL_L2CAP, L2CAP_OPTIONS,
&opts, optlen );
      }
      return err;
};
```

- The **omtu** and **imtu** fields of the struct l2cap_options are used to specify the **outgoing MTU** and **incoming MTU**, respectively.

- The other two fields are currently unused and reserved for future use. To adjust the connection-wide MTU, both clients must adjust their outgoing and incoming MTUs. Bluetooth allows the MTU to range from a minimum of 48 bytes to a maximum of 65,535 bytes.

Unreliable sockets

- Multiple L2CAP and RFCOMM connections between two devices are actually logical connections multiplexed on a single, lower level connection (**ACL connection**) established between them.

- The only way to adjust delivery semantics is to adjust them for the lower level connection, which in turn affects all L2CAP and RFCOMM connections between the two devices.

- Unfortunately, BlueZ does not provide an easy way to change the packet timeout for a connection. A handle to the underlying connection is first needed to make this change, but the only way to obtain a handle to the underlying connection is to query the microcontroller on the local Bluetooth adapter.

- Once the connection handle has been determined, a command can be issued to the microcontroller instructing it to make the appropriate adjustments. The example provided (set-flush.c) shows how to do this.

- On success, the packet timeout for the low level connection to the specified device is set to timeout * 0.625 milliseconds.

- A timeout of 0 is used to indicate infinity, and is how to revert back to a reliable connection. The bulk of this function is comprised of code to construct the command packets and response packets used in communicating with the Bluetooth controller.

- The Bluetooth Specification defines the structure of these packets and the magic number 0x28.

- The examples provided demonstrate how to establish an L2CAP channel and transmit a short string of data.

- The htobs function, described earlier, is used here to convert numbers to Bluetooth byte order.

Service Discovery Protocol

- The process of searching for services involves two steps - detecting all nearby devices with a device inquiry, and connecting to each of those devices in turn to search for the desired service.

- Searching a specific device for a service also involves two steps. The first part, shown below, requires connecting to the device and sending the search request. Note that the example searches for a device with **UUID** = **0xABCD**

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

int main(int argc, char **argv)
{
    uint8_t svc_uuid_int[] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0xab, 0xcd };
    uuid_t svc_uuid;
    int err;
    bdaddr_t target;
    sdp_list_t *response_list = NULL, *search_list, *attrid_list;
    sdp_session_t *session = 0;

    str2ba( "01:23:45:67:89:AB", &target );

    // connect to the SDP server running on the remote machine
    session = sdp_connect( BDADDR_ANY, &target, SDP_RETRY_IF_BUSY );

    // specify the UUID of the application we're searching for
    sdp_uuid128_create( &svc_uuid, &svc_uuid_int );
    search_list = sdp_list_append( NULL, &svc_uuid );
```

```
    // specify that we want a list of all the matching applications'
attributes
    uint32_t range = 0x0000ffff;
    attrid_list = sdp_list_append( NULL, &range );

    // get a list of service records that have UUID 0xabcd
    err = sdp_service_search_attr_req( session, search_list, \
            SDP_ATTR_REQ_RANGE, attrid_list, &response_list);
    .
    .
```

- The uuid_t data type is used to represent the 128-bit UUID that identifies the desired service.
- To obtain a valid uuid_t, create an array of 16 8-bit integers and use the sdp_uuid128_create function, which is similar to the str2ba function for converting strings to bdaddr_t types. sdp_connect synchronously connects to the SDP server running on the target device.
- sdp_service_search_attr_req searches the connected device for the desired service and requests a list of attributes specified by attrid_list.
- It is easiest to use the magic number 0x0000ffff to request a list of all the attributes describing the service, although it is possible, for example, to request only the name of a matching service and not its protocol information.

- The second part, involves parsing and interpreting the search results. The following code fragment shows how to parse and interpret an SDP search result:

```
sdp_list_t *r = response_list;

    // go through each of the service records
    for (; r; r = r->next ) {
        sdp_record_t *rec = (sdp_record_t*) r->data;
        sdp_list_t *proto_list;

        // get a list of the protocol sequences
        if( sdp_get_access_protos( rec, &proto_list ) == 0 ) {
        sdp_list_t *p = proto_list;

        // go through each protocol sequence
        for( ; p ; p = p->next ) {
            sdp_list_t *pds = (sdp_list_t*)p->data;

            // go through each protocol list of the protocol sequence
            for( ; pds ; pds = pds->next ) {

                // check the protocol attributes
                sdp_data_t *d = (sdp_data_t*)pds->data;
                int proto = 0;
                for( ; d; d = d->next ) {
                    switch( d->dtd ) {
                        case SDP_UUID16:
                        case SDP_UUID32:
                        case SDP_UUID128:
                            proto = sdp_uuid_to_proto( &d->val.uuid );
                            break;
                        case SDP_UINT8:
                            if( proto == RFCOMM_UUID ) {
                                printf("rfcomm channel: %d\n",d->val.int8);
                            }
                            break;
                    }
                }
            }
            sdp_list_free( (sdp_list_t*)p->data, 0 );
        }
        sdp_list_free( proto_list, 0 );

        }

        printf("found service record 0x%x\n", rec->handle);
        sdp_record_free( rec );
    }

    sdp_close(session);
}
```

- Obtaining the protocol information requires digging deep into the search results. Since it's possible for multiple application services to match a single search request, a list of service records is used to describe each matching service.

- For each service that's running, it is (not usually done in practice) possible to have different ways of connecting to the service.

- Therefore, each service record has a list of protocol sequences that each describe a different way to connect.

- Furthermore, since protocols can be built on top of other protocols (e.g. RFCOMM uses L2CAP as a transport), each protocol sequence has a list of **protocols** that the application uses, only one of which actually matters.

- Finally, each protocol entry will have a list of **attributes**, like the protocol type and the port number it's running on. Thus, obtaining the port number for an application that uses RFCOMM requires finding the port number protocol attribute in the RFCOMM protocol entry.

- The example uses several new data structures:

```
typedef struct _sdp_list_t {
    struct _sdp_list_t *next;
    void *data;
} sdp_list_t;

typedef void(*sdp_free_func_t)(void *)

sdp_list_t *sdp_list_append(sdp_list_t *list, void *d);
sdp_list_t *sdp_list_free(sdp_list_t *list, sdp_list_func_t f);
```

- BlueZ uses a linked list data structure and called it **`sdp_list_t`**. For now, it suffices to know that appending to a `NULL` list creates a new linked list, and that a list must be deallocated with `sdp_list_free` when it is no longer needed.

```
typedef struct {
     uint32_t handle;
     sdp_list_t *pattern;
     sdp_list_t *attrlist;
} sdp_record_t;
```

- The **`sdp_record_t`** data structure represents a single service record being advertised by another device.

- The internal details are not important, since there are a number of helper functions available to get information in and out of it.

- In this example, **sdp_get_access_protos** is used to extract a list of the protocols for the service record.

```
typedef struct sdp_data_struct sdp_data_t;
struct sdp_data_struct {
     uint8_t dtd;
     uint16_t attrId;
     union {
          int8_t    int8;
          int16_t   int16;
          int32_t   int32;
          int64_t   int64;
          uint128_t int128;
          uint8_t   uint8;
          uint16_t  uint16;
          uint32_t  uint32;
          uint64_t  uint64;
          uint128_t uint128;
          uuid_t    uuid;
          char     *str;
          sdp_data_t *dataseq;
     } val;
     sdp_data_t *next;
     int unitSize;
};
```

- Finally, there is the **`sdp_data_t`** structure, which is ultimately used to store each element of information in a service record.

- At a high level, it is a node of a linked list that carries a piece of data (the **val** field). As a variable type data structure, it can be used in different ways, depending on the context.

The SDP daemon (sdpd)

- Bluetooth devices can run an SDP server that responds to queries from other Bluetooth devices. In BlueZ, the implementation of the SDP server is called **sdpd**, and is usually started by the system boot scripts.

- sdpd handles all incoming SDP search requests. Applications that need to advertise a Bluetooth service must use inter-process communication (IPC) methods to tell sdpd what to advertise.

- Registering a service with sdpd involves describing the service to advertise, connecting to sdpd, instructing sdpd on what to advertise, and then disconnecting.

Describing a service

- Describing a service is essentially involves creating several lists and populating them with data attributes.

- The code fragment below shows how to describe a service application with UUID 0xABCD that runs on RFCOMM channel 11, is named “Roto-Rooter Data Router”, provided by “Roto-Rooter”, and has the description “An experimental plumbing router”.

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

sdp_session_t *register_service()
{
   uint32_t service_uuid_int[] = { 0, 0, 0, 0xABCD };
   uint8_t rfcomm_channel = 11;
   const char *service_name = "Roto-Rooter Data Router";
   const char *service_dsc = "An experimental plumbing router";
   const char *service_prov = "Roto-Rooter";

   uuid_t root_uuid, l2cap_uuid, rfcomm_uuid, svc_uuid;
   sdp_list_t *l2cap_list = 0,
            *rfcomm_list = 0,
            *root_list = 0,
            *proto_list = 0,
            *access_proto_list = 0;
   sdp_data_t *channel = 0, *psm = 0;

   sdp_record_t *record = sdp_record_alloc();

   // set the general service ID
   sdp_uuid128_create( &svc_uuid, &service_uuid_int );
   sdp_set_service_id( record, svc_uuid );

   // make the service record publicly browsable
   sdp_uuid16_create(&root_uuid, PUBLIC_BROWSE_GROUP);
   root_list = sdp_list_append(0, &root_uuid);
   sdp_set_browse_groups( record, root_list );

   // set l2cap information
   sdp_uuid16_create(&l2cap_uuid, L2CAP_UUID);
   l2cap_list = sdp_list_append( 0, &l2cap_uuid );
   proto_list = sdp_list_append( 0, l2cap_list );

   // set rfcomm information
   sdp_uuid16_create(&rfcomm_uuid, RFCOMM_UUID);
   channel = sdp_data_alloc(SDP_UINT8, &rfcomm_channel);
   rfcomm_list = sdp_list_append( 0, &rfcomm_uuid );
   sdp_list_append( rfcomm_list, channel );
   sdp_list_append( proto_list, rfcomm_list );

   // attach protocol information to service record
   access_proto_list = sdp_list_append( 0, proto_list );
   sdp_set_access_protos( record, access_proto_list );

   // set the name, provider, and description
   sdp_set_info_attr(record, service_name, service_prov, service_dsc);
   .
```

.

Registering a service

- Building the description is quite straightforward, and consists of taking those five fields and packing them into data structures.

- Most of the work is just putting lists together. Once the service record is complete, the application connects to the local SDP server and registers a new service.

```
    .
    .
    int err = 0;
    sdp_session_t *session = 0;

    // connect to the local SDP server, register the service record, and
    // disconnect
    session = sdp_connect( BDADDR_ANY, BDADDR_LOCAL,
SDP_RETRY_IF_BUSY );
    err = sdp_record_register(session, record, 0);

    // cleanup
    sdp_data_free( channel );
    sdp_list_free( l2cap_list, 0 );
    sdp_list_free( rfcomm_list, 0 );
    sdp_list_free( root_list, 0 );
    sdp_list_free( access_proto_list, 0 );

    return session;
}
```

- The special argument **BDADDR_LOCAL** causes sdp_connect to connect to the local SDP server (via the named pipe /var/run/sdp) instead of a remote device.

- Once an active session is established with the local SDP server, sdp_record_register advertises a service record. The service will be advertised for as long as the session with the SDP server is kept open.

- As soon as the SDP server detects that the socket connection is closed, it will stop advertising the service. sdp_close terminates a session with the SDP server.

```
sdp_session_t *sdp_connect( const bdaddr_t *src, const bdaddr_t *dst, uint32_t
      flags );
int sdp_close( sdp_session_t *session );

int sdp_record_register(sdp_session_t *sess, sdp_record_t *rec, uint8_t flags);
```