IceBT

IceBT is a transport plug-in that enables clients and servers to communicate via Bluetooth RFCOMM connections on Android and Linux platforms.

On this page:

IceBT Overview

IceBT is an Ice plug-in that must be installed in all of the clients and servers in your application that need to communicate over Bluetooth. This section reviews some concepts that will help you as you learn more about IceBT.

Service Discovery

The Bluetooth specification defines a standard mechanism for discovering services called the Service Discovery Protocol (SDP). It's a flexible but complex specification that accommodates a wide range of Bluetooth device functionality and requirements. Fortunately, Ice users only need a passing familiarity with SDP.

The operating system's Bluetooth stack implements an SDP service that provides two basic functions: query and registration. IceBT considers each object adapter endpoint in an Ice server to be a service and adds a corresponding entry for it in the local SDP registry. This entry associates a UUID with a human-friendly name and an RFCOMM channel. For example, an entry might contain:

SDP Entry

```
Name: My Bluetooth Service
UUID: 1c6a142a-aae6-4d58-bef8-33196f531da7
RFCOMM: Channel #8
```

The SDP entry automatically expires when its service terminates.

An Ice client requires two values to connect to a server:

1. The server's Bluetooth device address (such as `01:23:45:67:89:AB`)
2. The UUID of the desired service

To establish a connection, the client first queries the SDP service on the server device for an entry matching the target UUID. If a match is found, the SDP service returns the server's current RFCOMM channel, and the client open a connection to that channel.

IceBT takes care of all of this for you during server initialization and connection establishment.

Back to Top ^

Device Discovery

When developing a client application, you'll normally hard-code the UUIDs of the remote services that your client requires because those UUIDs must match the ones advertised by your servers. However, in addition to a UUID, a client also needs to know the device address on which a service is running. Typically the client will use the system's Bluetooth API to initiate device discovery and present the results to the user. We discuss this further in the "Using IceBT" section below.

Back to Top ^

Installing IceBT

The IceBT plug-in must be installed in every client and server that needs to communicate via Bluetooth.

You can use the `Ice.Plugin` property to load and install the plug-in at run time; the property value depends on the language mapping you're using:

C++

```
# Linux only
Ice.Plugin.IceBT=IceBT:createIceBT
```

Java

```
# Android only
Ice.Plugin.IceBT=com.zeroc.IceBT.PluginFactory
```

Java Compat

```
# Android only
Ice.Plugin.IceBT=IceBT.PluginFactory
```

Python

```
# Linux only
Ice.Plugin.IceBT=IceBT:createIceBT
```

Ruby

```
# Linux only
Ice.Plugin.IceBT=IceBT:createIceBT
```

PHP

```
# Linux only
Ice.Plugin.IceBT=IceBT:createIceBT
```

If using C++, instead of dynamically loading the plug-in at run time your application can explicitly link with and register the plug-in. To register the plug-in, you must call the `Ice::registerIceBT(bool loadOnInitialize = true)` function before the communicator initialization. The `loadOnInitializ` parameter specifies if the plug-in is installed when the communicator is initialized. If set to `false`, you will need to enable the plug-in by setting the `Ice.Plugin.IceBT` property to `1`.

`Ice::registerIceBT` is a simple helper function that calls `Ice::registerPluginFactory`.

Refer to the next section for information on configuring the plug-in.

Back to Top ^

Configuring IceBT

The IceBT plug-in supports some new configuration properties, including settings to modify the size of the send and receive buffers for a connection. The default settings should be sufficient for most applications.

Developers should also be aware of some core Ice properties that can affect Bluetooth connections:

- Default device address – If you omit a device address from an object adapter endpoint or proxy endpoint, the plug-in defaults to the address specified by the property `Ice.Default.Host`.
- Connect timeout – Establishing a Bluetooth connection can take several seconds to complete. Ice's default timeout settings give plenty of time for a connection to succeed, but an application could experience problems if it configures custom connection timeouts that are too small for Bluetooth connections.

Back to Top ^

Using IceBT

This section describes how to incorporate IceBT into your Ice applications.

Object Adapter Endpoints

A Bluetooth "service" corresponds to an Ice endpoint, and each endpoint requires its own UUID.

On Linux, you can use the `uuidgen` command to generate new UUIDs. Web-based UUID generators are also available.

For example, using the syntax for Bluetooth endpoints, you can configure an object adapter named `HelloService` as follows:

```
HelloService.Endpoints=bt -u 4f140cef-d75e-4c93-b4e4-20ac111d36d1 --name "Hello Service"
```

We're associating the UUID `4f140cef-d75e-4c93-b4e4-20ac111d36d1` with our service. At run time, this service will be advertised in the Service Discovery Protocol (SDP) registry along with the descriptive name `Hello Service`. We omitted a device address, so the plug-in will listen on the host's default Bluetooth adapter. We also did not specify a particular RFCOMM channel (using the `-c` option) and therefore the plug-in will automatically select an available channel.

If you omit the `-u UUID` option from the object adapter's endpoint, the plug-in will automatically generate a random UUID for use in the SDP registry. Note however that your clients will still need some way of discovering this UUID. Generally speaking, you should generate and use your own well-known UUIDs instead.

On Linux, use the `sdptool` command to view the contents of the SDP registry on a device:

```
> sdptool browse local
> sdptool browse 01:23:45:67:89:AB
```

The first command displays the active services of the local host, and the second command shows the active services of a remote device.

Back to Top ^

Proxy Endpoints

A Bluetooth endpoint in a proxy must include a UUID and a device address:

C++11

```
auto proxy = communicator->stringToProxy("hello:bt -u 4f140cef-d75e-4c93-b4e4-20ac111d36d1 -a \"01:23:45:67:89:
AB\"");
```

C++98

```
Ice::ObjectPrx proxy = communicator->stringToProxy("hello:bt -u 4f140cef-d75e-4c93-b4e4-20ac111d36d1 -a \"01:23:
45:67:89:AB\"");
```

The UUID specified with the `-u` option must match the one you assigned to your object adapter endpoint.

Notice that the device address given by the `-a` option is enclosed in quotes; this is necessary because colon (`:`) characters are used as separators in stringified proxies.

You can omit a device address if you define `Ice.Default.Host`.

Refer to Proxy and Endpoint Syntax for complete details on the format of a Bluetooth endpoint.

Applications are responsible for determining the Bluetooth address of the device hosting the target service, as described in the next section.

Back to Top ^

Implementing Discovery

Device discovery is a platform-specific activity that applications are responsible for implementing. On Android, an app can use the APIs in `android.bluetooth` to initiate discovery and receive intent notifications about nearby devices. The Android sample program `talk` (see below) shows how to implement discovery.

On Linux, the IceBT plug-in provides a C++ API for device discovery:

C++11

```
namespace IceBT
{
    using PropertyMap = std::map<std::string, std::string>;

    class Plugin : public Ice::Plugin
    {
    public:

         virtual void startDiscovery(const std::string& address,
                                     std::function<void(const std::string& addr, const PropertyMap& props)>) =
0;
         virtual void stopDiscovery(const std::string& address) = 0;
    };
}
```

C++98

```
namespace IceBT
{
    typedef std::map<std::string, std::string> PropertyMap;

    class DiscoveryCallback : public IceUtil::Shared
    {
    public:
        virtual void discovered(const std::string& address, const PropertyMap& props) = 0;
    };
    typedef IceUtil::Handle<DiscoveryCallback> DiscoveryCallbackPtr;

    class Plugin : public Ice::Plugin
    {
    public:

        virtual void startDiscovery(const std::string& address, const DiscoveryCallbackPtr& cb) = 0;
        virtual void stopDiscovery(const std::string& address) = 0;
    };
    typedef IceUtil::Handle<Plugin> PluginPtr;
}
```

An application must implement a callback function or object and pass it to `startDiscovery`:

C++11

```
#include <IceBT/IceBT.h>
...

auto communicator = ...
auto plugin = communicator->getPluginManager()->getPlugin("IceBT");
auto btplugin = dynamic_pointer_cast<IceBT::Plugin>(plugin);
btplugin->startDiscovery("", [](const std::string& addr, const PropertyMap& props) { ... });
```

C++98

```
#include <IceBT/IceBT.h>
...
class DiscoveryCallbackI : public IceBT::DiscoveryCallback
{
public:

    virtual void discovered(const std::string& address, const IceBT::PropertyMap& props)
    {
        ...
    }
};
...
Ice::CommunicatorPtr communicator = ...
Ice::PluginPtr plugin = communicator->getPluginManager()->getPlugin("IceBT");
IceBT::PluginPtr btplugin = IceBT::PluginPtr::dynamicCast(plugin);
btplugin->startDiscovery("", new DiscoveryCallbackI);
```

For each nearby device discovered by the Bluetooth stack, the plug-in will invoke the callback. The arguments to the callback are the Bluetooth address of the nearby device and a string map of properties containing metadata about that device. As shown in the example above, the application can pass an empty string to `startDiscovery` and the plug-in will use the default Bluetooth adapter, otherwise the application can pass the device address of the desired adapter.

Discovery will continue until `stopDiscovery` is called or a Bluetooth connection is initiated.

Back to Top ^

Connection Limitations

Be aware of the following limitations when using IceBT:

- An application cannot open multiple Bluetooth connections to the same remote endpoint. This is not a limitation in Ice but rather in the Bluetooth stack. Normally this limitation won't impact your application because Ice's default behavior is to reuse an existing connection to an endpoint whenever possible in preference to opening a new connection. However, some application designs may attach additional semantics to a connection, using Ice APIs to override the default behavior and force the establishment of new connections to the same endpoint. This strategy will not work when using Bluetooth.

- If a Linux server forcefully closes a Bluetooth connection, the connection loss may not be detected by the client. It's important to configure sensible connection timeouts and invocation timeouts to avoid lengthy delays.

Back to Top ^

IceBT Sample Programs

The ice-demos repository includes a C++ command-line program for Linux (in `cpp/IceBT/talk`) and an app for Android (in `java/Android/talk`). These programs allow two devices to talk to one another via Bluetooth.

Back to Top ^

Security Notes for IceBT

The Bluetooth stack performs its own encryption of transmitted data using keys generated during the pairing process. Two devices must already be paired before Ice applications on those devices can communicate with one another. IceBT does not implement or provide an API for pairing; rather, this is something that is normally done at the user level. However, it's possible that an Ice connection attempt will *initiate* a pairing process that the user must then complete.

Android provides two APIs for establishing a connection: a secure version and an insecure version. The difference between the two lies in the pairing behavior, where the secure version causes the system to prompt the user (if necessary) and the insecure version does not. IceBT always uses the secure API.

For added security, you can use SSL over Bluetooth by installing the IceSSL plug-in. During its initialization, the IceBT plug-in checks for the presence of IceSSL and, if it's found, IceBT adds a second transport protocol named `bts`. No additional changes are necessary to your endpoints, and you can use the IceSSL configuration properties as usual to define your security settings.

Back to Top ^

See Also

- IceBT.*
- Plug-in Facility
- Proxy and Endpoint Syntax