

Maemo Diablo Reference Manual for maemo 4.1

Using Connectivity Components

December 22, 2008

Contents

1	Using Connectivity Components	2
1.1	Introduction	2
1.2	Maemo Connectivity	3
1.2.1	Connectivity Subsystem	4
1.2.2	Internet Connectivity Daemon	10
1.2.3	LibConIC Library	12
1.2.4	Bluetooth Libraries	20
1.2.5	Connectivity UI	23
1.2.6	Samba Network Shares	32
1.2.7	Location	33
1.3	Implementing Custom Connection Managers	35
1.3.1	Connection Manager Implementation Examples	35
1.3.2	Connection Manager and Connections	35
1.3.3	Channels and Channel Types	36
1.3.4	Additional Connection Interfaces	37
1.4	Using STUN in Applications	38
1.4.1	Network Address Translation (NAT)	39
1.4.2	NAT Problems	40
1.4.3	NAT Peer-to-Peer Circumvention Techniques	40
1.4.4	STUN, TURN and ICE Protocols	41
1.4.5	NAT Transversal API in Maemo	42
1.4.6	Example: P2P Client	42
1.4.7	Signaling Server	50
1.4.8	STUN and Relay Servers	51

Chapter 1

Using Connectivity Components

1.1 Introduction

The following code examples are used in this chapter:

- [example_bluetooth.c](#)
- [example_gps.c](#)

For communications with the outside world, the maemo platform provides frameworks ranging from Bluetooth file transfer to making video calls. This chapter goes through the components important for the maemo application developer.

Acquiring Internet Access

LibConIC is the library that must be used by all applications wanting to use Internet connectivity. It takes care of e.g. scanning of available WLAN networks, and setting up the IP network after the user has selected a connection. *LibConIC* API works together with the maemo connectivity daemon (*ICd*) that handles both WLAN and Bluetooth connections. *LibConIC* and *ICd* are described more deeply in section [1.2](#).

VoIP, Instant Messaging and Presence

The *Telepathy*[\[4\]](#) communications framework provides a unified API for presence, messaging and voice/video calls. The list of supported protocols is long: IRC, ICQ, XMPP (Jabber), SIP, MSN etc. The connection manager of *Telepathy* is expandable, and uses D-BUS for communication. *Telepathy* usage in maemo is described in more detail in section [1.3](#).

Bluetooth

A high level API for Bluetooth is offered as part of the maemo connectivity subsystem. Using its D-BUS API, a program can find remote Bluetooth devices, such as phones, send files over OBEX object push, and create pairings with remote devices. For these tasks, it is recommended for an application to use

this framework, as it not only has a lot simpler API, but makes the applications look and behave consistently.

For Bluetooth operations that are not supported by the maemo connectivity framework, maemo includes a lower level BlueZ D-BUS API, which is also the main Bluetooth interface for all Linux systems. The BlueZ API has features for practically all aspects of Bluetooth systems, and as a consequence is a lot more complex than the higher level Maemo Connectivity subsystem's offerings.

Section 1.2 describes the high level D-BUS API and its use. More information about the BlueZ API can be found at BlueZ web site[1]. The maemo-example package also includes example code about both libraries.

OBEX

Bluetooth devices use OBEX protocol to exchange data objects. Maemo includes libraries to help working with this protocol. For OBEX FTP, the easiest way is to use GnomeVFS's OBEX backend. For more granular control, there is *libgwobex*, which GnomeVFS uses in its implementation, and an even more low level interface can be found in the OpenOBEX library.

More about GnomeVFS is written in the GnomeVFS section of this document. Section 1.2 contains separate subsections for *libgwobex* and *OpenOBEX*. For *libgwobex*API, see Maemo API Reference[2]. Lots of information about OpenOBEX can be found at the project's web site[3].

1.2 Maemo Connectivity

The maemo connectivity subsystem is implemented by using known Linux conventions. It resides in the user mode area of Linux, and relies on the Linux kernel through standard C libraries. Wireless LAN or WiMAX is the main channel to the Internet, but dial-up connections through cellular networks are also supported. The only medium to the phone is Bluetooth. The Bluetooth software of maemo is based on BlueZ, which is known as the de-facto implementation of Bluetooth for Linux. D-Bus is used for internal application level message exchange.

Even though the connectivity device drivers are closely related to this subsystem, they are considered to be outside of the scope.

Components of the maemo connectivity architecture:

Maemo connectivity UI - User Interface parts of the connectivity. This includes Connection manager, Control Panel applets and several different dialogs.

Maemo connectivity daemon (ICd) - LibConIC API works together with ICd, handling all Internet Access Points (IAPs). IC daemon handles both WLAN and Bluetooth connections.

OBEX wrapper - Interface to OBEX services. The primary target user of this library is the OBEX gnome-vfs module.

OpenOBEX - Open source implementation of the Object Exchange (OBEX) protocol. More information on OpenOBEX can be found from <http://triq.net/obex/>

BlueZ Bluetooth stack - The de-facto implementation of Bluetooth for Linux. More information on BlueZ can be found from <http://www.bluez.org>

BlueZ D-Bus API - BlueZ accepts commands via D-Bus.

WLAN connectivity daemon - The daemon controlling WLAN connections.

WLAN device driver - Device driver for Wireless LAN (IEEE 802.11g). Kernel driver is composed of two parts: a binary part (closed source) and an open source wrapper, binding the binary to the current Linux kernel.

WiMAX onnectivity daemon - The daemon controlling WiMAX connections.

WiMAX device driver - Device driver for mobile WiMAX (IEEE 802.16e).

Internet Access Points (IAP)

The central concept regarding Internet connections from maemo is the Internet Access Point (IAP). It represents a logical Internet (IP) connection, which will be defined by the user according to their needs. An IAP has a unique name usually in form of UUID). It defines the radio bearer (e.g. WLAN, CSD, GPRS) to be applied, and usually the data transfer speed, username, password, proxy server, and the corresponding access point in the Internet or the telephone number of the service provider's modem, among other characteristics.

1.2.1 Connectivity Subsystem

This section describes the system decomposition of the Connectivity subsystem. Maemo applications can open Internet connections by using the LibConIC API. The Internet Access subsystem will take care of the connection to the phone, if necessary, by using the services of the Phone Access subsystem. If an application needs to gain access to the phone's files, then the File selector will consult the Phone Access subsystem.

During the offline mode, none of the radios must be active. The Device System Management Entity (DSME) of maemo provides information about the transitions to and from the offline mode.

Name	Connection Manager
Purpose	Provides the UI for managing phone and Internet connections. Available as a Control Panel applet, and from the Status Bar.
Responsibilities and additional requirements	<ul style="list-style-type: none">• Displaying active connections with statistics• Updating status indicators• Providing dialogs for changing and disconnecting connections
Concurrent usage	(N/A)

Name	Phone Access
Purpose	Provides connections to phones with different Bluetooth profiles
Responsibilities and additional requirements	<ul style="list-style-type: none"> • Searching for phones and inquiring their services • Keeping phone register • Providing status of the phone connections for Connection Manager • Binding RFCOMM devices to DUN and FTP services on the phone • Providing easy access to OpenOBEX
Concurrent usage	Number of clients not limited by maemo. However, some phones may not support more than one Bluetooth profile at a time.

Name	Internet Access
Purpose	Provides Internet connections over different bearers.
Responsibilities and additional requirements	<ul style="list-style-type: none"> • Providing means for configuration and management of IAP settings • Providing API for Internet connections over different bearers (e.g. WLAN, WiMAX, Bluetooth dialup) • Providing status of Internet connections for Connection Manager
Concurrent usage	Number of clients not limited, but only one connection to the Internet can exist at any given time

Phone and Internet connections are quite different by nature and behavior. These will be introduced in more detail in the following sections.

Phone Access

Phone Access is the subsystem handling connections to a cellular phone. It has a search utility for finding potential phones and inquiring the services they can offer. This is based on the standard Bluetooth service discovery mechanism. Phone Access also keeps a record of phones having been connected to the device in GConf, and provides a list of them for the user to choose from. Phone Access relies on the Linux Bluetooth implementation called BlueZ. BlueZ offers the Berkeley socket interface to the HCI and to the L2CAP protocol for the user space applications.

In principle, any cellular phone supporting Bluetooth Service Discovery Protocol (SDP), Dial-up Networking profile (DUN) and File Transfer Profile

(FTP) can be connected to maemo. However, there is variation especially in the level of file transfer services and OBEX in different mobile phones. Some products limit the access to the Inbox (Object Push), whereas more sophisticated ones make the Gallery and the memory card available. The recent products support the OBEX Capability request, which can be used to get more specific information about the file system on the phone.

Maemo connects to a phone on an on-demand basis, i.e. when an application requires a connection. For example, when the Internet browser is about to open a URL, it will request the Phone Access to establish a connection to the phone. This makes Phone Access to bind an RFCOMM device to the requested service (in this case DUN) on the phone. In a similar fashion, the File Selector can set up a file transfer connection to the phone using another RFCOMM device. After binding to a service, the application in question can open the local RFCOMM device. Normal file selector access is performed with GnomeVFS layer to get transparent access to phone in the same way as internal flash and MMC are accessed.

The Bluetooth SIM Access (SAP) profile is also needed in maemo to perform WLAN authentication using the EAP-SIM authentication method. In this case, the EAP component will ask the BT sap component to get session keys from a GSM/UMTS phone.

Name	Phone selection UI
Purpose	User interface for managing phone operations. Supports Connection Manager.
Responsibilities and additional requirements	<ul style="list-style-type: none"> • Keeping a list of phones • Storing the Bluetooth device addresses of phones to GConf • Invoking device search and capability query
Concurrent usage	N/A

Name	General Bluetooth UI
Purpose	User interface for managing all paired BT devices
Responsibilities and additional requirements	<ul style="list-style-type: none"> • Keeping a list of all paired devices (not only phones)
Concurrent usage	N/A

Name	BT search
Purpose	Searches for available Bluetooth devices
Responsibilities and additional requirements	<ul style="list-style-type: none"> • Bluetooth inquiry • Checking on offline mode state
Concurrent usage	N/A

Name	BT service discovery
Purpose	Checks if a found Bluetooth device is sufficient
Responsibilities and additional requirements	<ul style="list-style-type: none"> • Bluetooth service discovery (SDP)
Concurrent usage	Not limited (but used by Phone selection UI and Phone connection daemon only)

Name	GW OBEX library
Purpose	Provides access to OpenOBEX library for the File selector (Gnome VFS) on a higher abstraction level than OpenOBEX itself supports
Responsibilities and additional requirements	<ul style="list-style-type: none"> • Handling OBEX requests
Concurrent usage	Not limited

Name	BT SAP
Purpose	Obtains session keys for EAP-SIM authentication from the phone
Responsibilities and additional requirements	<ul style="list-style-type: none"> • SIM/PIN entry • Connecting to the phone with SIM Access Profile • Delivering the session keys to EAP
Concurrent usage	N/A

Maemo Bluetooth also supports HID (keyboard) and OPP (object push file transfer) profiles.

Internet Access

The Internet Access subsystem manages connections to the Internet over different bearers. It is also responsible for the configuration and management of Internet Access Points. The Internet Access provides applications with TCP/IP connections. They can be established with:

- WLAN connection to a wireless access point.
- WiMAX connection to a WiMAX base station.
- Bluetooth connection through phone using Point-to-Point Protocol (PPP) and a cellular modem (in the phone).

In the latter case, AT commands are applied to establish a PPP link to the cellular modem and the connection to the Internet.

Name	IC daemon
Purpose	IC daemon establishes Internet connections over different bearers.
Responsibilities and additional requirements	<ul style="list-style-type: none"> • Controlling that only one Internet connection (one active IAP) can exist at any given time • Using Phone Access for getting a character device for a dial-up connection, and using WLAN or WiMAX connection daemon for getting a network device and getting the connection authenticated • Starting IP level services like PPP and DHCP • Providing statistics about the usage of IAPs to any application
Concurrent usage	Has multiple clients and limits the connections to one at a time

Name	Internet Connectivity GUI
Purpose	This package has the GUI applications for configuring Internet Access Points and WLAN settings. N.B. The Connection Manager is a separate application.
Responsibilities and additional requirements	<ul style="list-style-type: none"> • Providing the UI for configuring an IAP • Providing the UI for WLAN settings • Scanning for available WLAN networks (on request) • Saving IAP and WLAN settings (excluding EAP settings) to GConf • Opening up dialogs and displaying notifications
Concurrent usage	N/A

Name	WLAN connection daemon
Purpose	Manages WLAN network connections
Responsibilities and additional requirements	<ul style="list-style-type: none"> • Starting WLAN driver with the settings provided; stopping WLAN • Requesting authentication when necessary • Relaying WLAN IAP settings from IC daemon to WLAN drivers and authentication requests to EAP • Relaying WLAN events from WLAN drivers • Providing WLAN status information
Concurrent usage	Not limited

Name	EAP UI
Purpose	User interface for EAP authentication
Responsibilities and additional requirements	<ul style="list-style-type: none"> • Invoking a dialog for an authentication password • Showing authentication status
Concurrent usage	N/A

Name	EAP
Purpose	Provides WLAN and WiMAX security excluding basic WEP settings, which are in Wireless Extensions
Responsibilities and additional requirements	<ul style="list-style-type: none"> • Taking care of the authentication process for the active IAP • Delivering progress events to IC daemon • As a special case, controlling the EAP-SIM authentication using the SIM Access Profile • Providing authentication status
Concurrent usage	N/A

Name	WiMAX connection daemon
Purpose	Manages WiMAX network connections
Responsibilities and additional requirements	<ul style="list-style-type: none"> • Starting WiMAX driver with the settings provided; stopping WiMAX • Relaying WiMAX IAP settings from IC daemon to WiMAX drivers • Relaying WiMAX events from WiMAX drivers • Providing WiMAX status information • Scanning WiMAX NAPs and NSPs
Concurrent usage	Not limited

1.2.2 Internet Connectivity Daemon

This section describes how the Internet Connectivity daemon works internally. The following subsections explain the behavior and the decomposition of this component in detail, also covering the interfaces that this component realizes.

Decomposition

When the ICd receives a request to activate or deactivate an IAP, the ICd will activate the IAP or show a ui requesting the user to choose one, if no IAP has been selected as the default. Depending on the type of the IAP, the ICd will use appropriate network type plug-in to activate or deactivate specific network interface.

The ICd tracks the applications requesting IAPs by recording their D-Bus base service names. This allows the ICd to detect situations where processes using an IAP have aborted or crashed. The ICd also implements an idle timeout mechanism to shut down the active IAP, if no packets have been sent in a configured amount of time.

Maemo version 3.0 introduced the automatic connection creation feature in the Internet Connectivity Daemon. In other words, the device will try to connect automatically to the saved IAPs, and keep connected as long as possible, unless the idle timeout is set. With this feature, applications like e-mail and RSS reader will always be up to date. The device will also be always ready for online usage, for example, incoming VoIP calls or IM chat. In former versions, the Internet connection was automatically closed when there were no more applications using it, or when the connection was idle for a given period of time defined by the configuration parameter idle timeout.

When not connected, the device scans for saved IAPs and tries to connect automatically, taking into account the value defined by the configuration parameter for search interval, which can be 5, 10, 30 or 60 minutes. All other values will be automatically mapped to "Never". This setup switches off the automatic connection feature. In this case, the device will behave just like the

former versions: Connections will be created only when required by applications.

The ICd is responsible for connection creation only, as it is the responsibility of each application to keep its data updated, and then providing the always-online feature.

While writing an application making use of the ICd system, the following points should be kept in mind:

- The application must always use the existing available connection.
- As it was done in former versions, if device is not connected but a connection is required by user interaction, the application must require connection creation using LibConIC API.
- The user should be kept aware of updates, making visible when the data was last updated.
- The application must register via LibConIC and listen to signals emitted by the ICd (Connection Created, Lost and Changed), and react as follows:
 - **Connection Created:** Use the connection and update all data.
 - **Connection Lost:** Go to an idle state silently and wait until a new connection is created.
 - **Connection Changed:** Use the new connection.
- Automatic data updates must run in background and silently:
 - Alarming the user with unnecessary banners or dialogs should be avoided.
 - Usernames and passwords should be saved, so that automatic updates can be performed without prompts.
 - In this case, no failures are allowed to show error notification dialogs.
- The connectivity infrastructure takes care of error situations in a centralized way.

Automatic connection creation feature can also be switched off using offline mode (i.e. offline mode). While in this mode, the configuration parameter for allowing WLAN in offline mode is checked. Depending on the state of this configuration parameter, WLAN IAPs are either enabled or disabled in the offline mode. Also other radios like Bluetooth are normally disabled in the offline mode.

Bluetooth Dial-up Networking The ICd uses PPP to establish IP connectivity over Bluetooth DUN interfaces. If there already is a different IAP active using Bluetooth DUN, the old IAP is first deactivated. The IAP is activated according the following action sequence:

- The character device used by the Bluetooth DUN device is acquired from btcond. If the device is not available due to gateway not being present, exhaustion of simultaneous Bluetooth connections, or similar, the ICd shows an error message to the user and aborts with a D-BUS error message.

- The ICd starts PPP using the exec family of system calls. It directs PPP to use the acquired Bluetooth DUN device with the dial-up configuration parameters specified for the configured DUN IAP type. If PPP cannot get the connection established, the ICd will show an error message to the user, and abort with a D-BUS error message. When the PPP connection is established, PPP-specific scripts will be run. These scripts will set dynamic IP connection related configuration entries, and send a state change D-BUS message to all the interested applications to indicate that the IAP has been established.

If the previously active IAP was not using Bluetooth DUN, it will be closed down after establishing the PPP connection.

A Bluetooth DUN is closed down by sending the PPP daemon a SIGINT or SIGTERM signal. This will terminate the PPP daemon, and remove all routing entries associated with the PPP dial-up interface. The PPP shutdown scripts will remove the dynamic IP connection related configuration entries, and send a state change D-BUS message announcing deactivation of the IAP. This is described below.

WLAN When connecting to a WLAN, the ICd needs to associate with the network, and enable EAP authentication and the DHCP client as needed. Independently of whether there is an IAP active using WLAN, the requested WLAN network will first be scanned to ensure that it is available. The current IAP will be deactivated, if the requested network is found and the current IAP is using WLAN. WLAN is activated according to the following procedure:

- If the network requires EAP authentication, the EAP authentication procedure is started. While performing the EAP authentication, the EAP software may show GUI dialogs relating to the EAP authentication procedure. When the EAP authentication has been completed, the EAP software will set security keys for the WLAN network, resulting in state change messages from wlancond. The ICd will receive these messages but ignore them, and wait for the reply from EAP authentication instead. If the EAP authentication fails, the ICd aborts with a D-Bus error message.
- After the EAP process has been started, the ICd instructs wlancond to associate with the WLAN network. Any static security settings relating to pre-shared security keys are also supplied at this point. If a connection to the WLAN network cannot be established, the ICd aborts with an error.
- As the DHCP client is a stand-alone program, it is started by using exec when the WLAN IAP requires dynamic IP address acquisition. When the DHCP client has obtained an IP address, it configures IP-related parameters, and sends a D-Bus signal to the ICd. If the IP address lease cannot be obtained, the ICd will timeout, stop the DHCP client and abort with a D-Bus error message.

1.2.3 LibConIC Library

Internet Connectivity API (in shorter form: Libconic) is an API for applications to manage internet connections on Maemo devices. It was introduced in the first

IT OS 2007 release, deprecating the old OSSO IC API (osso-ic-lib). OSSO IC API was conclusively removed in IT OS 2008 release. The interface documentation to libconic can be found at [Internet Connectivity API](#)

Unlike osso-ic-lib, Libconic is high level and stable object-oriented API. It can be used to

- Request Internet connection
- Listen for Internet connection status events
- Receive statistics of Internet connection
- Get proxy settings for the current connection
- Get list of user-saved connections (IAPs)

Application Requirements

Applications have a few requirements prior using the Libconic API. They are:

1. use non-blocking sockets
2. system D-BUS running
3. *g_type_init()* has to be called
4. no threading support in the Libconic API

If the application is a standard Hildon application, almost all of these requirements are already fulfilled. LibOSSO context initialization connects the application to both session and system D-BUS buses, *g_type_init()* is called as a part of *gtk_init()*, and there probably will be no extra threads used.

Non-Blocking Sockets

Blocking sockets cannot be used, because that would also block receiving the Connectivity events. Non-blocking sockets should be used in order to receive the events properly. For example, GLib IO Channels with the *G_IO_FLAG_NONBLOCK* flag provide non-blocking way to use sockets.

With threads, blocking sockets can be used, although Libconic API itself is not thread safe.

System D-BUS

Libconic API uses internally system D-BUS for delivering messages to the Connectivity components. Application needs to have normal D-BUS dispatch, watch and timeout monitoring running before the Libconic API can be used. If the GLib mainloop is used, this can be accomplished with *dbus_connection_setup_with_g_main()*.

N.B. Setting up LibOSSO context connects the application to required D-BUS.

GType

Libconic API is GObject-based. This means that to get API working, GLib's GType needs to be initialized properly. Use *g_type_init()* to do that.

No Multiple Threads

Libconic API is not thread-safe. If the applications have threads, use Libconic API only from the same context where GMainloop is running.

Libconic Usage

Requesting for Connection Libconic is an asynchronous connection API, which heavily relies on GObject signals. Basically this means that GMainloop must be iterated in order to successfully execute connection requests. After the application has been correctly set up (please see the previous section for some requirements), a connection can be requested with *ConIcConnection* object:

```
gboolean success = FALSE;

/* Create connection object */
ConIcConnection *connection = con_ic_connection_new();

/* Connect signal to receive connection events */
g_signal_connect(G_OBJECT(connection), "connection-event",
                 G_CALLBACK(my_connection_handler), NULL);

/* Request connection and check for the result */
success = con_ic_connection_connect(connection,
                                     CON_IC_CONNECT_FLAG_NONE);
if (!success) g_warning("Request for connection failed");
```

At this point, the application does not yet have an Internet connection. A successful return from *con_ic_connection_connect()* means only that the request was successfully dispatched to the Internet Connectivity daemon. When the daemon has a connection ready, the application will receive an event (as an GObject signal) indicating that the device is connected. If the connection attempt fails, the application will receive a disconnected event with an error describing the reason for the failure.

The connection handler (*my_connection_handler()* function registered in the previous snippet) could look like this:

```
static void my_connection_handler(ConIcConnection *connection,
                                 ConIcConnectionEvent *event,
                                 gpointer user_data)
{
    ConIcConnectionStatus status = con_ic_connection_event_get_status(
        event);
    ConIcConnectionError error;

    const gchar *iap_id = con_ic_event_get_iap_id(CON_IC_EVENT(event));
    const gchar *bearer = con_ic_event_get_bearer_type(CON_IC_EVENT(
        event));

    switch(status) {
        case CON_IC_STATUS_CONNECTED:
            g_debug("Hey, we are connected to IAP %s with bearer %s!",
                    iap_id, bearer);
            break;

        case CON_IC_STATUS_DISCONNECTING:
            g_debug("We are disconnecting...");
            break;
```

```

    case CON_IC_STATUS_DISCONNECTED:
        g_debug("And we are disconnected. Let's see what went wrong
            ...");
        error = con_ic_connection_event_get_error(event);

        switch(error) {

            case CON_IC_CONNECTION_ERROR_NONE:
                g_debug("Libconic thinks there was nothing wrong.");
                ;
                break;

            case CON_IC_CONNECTION_ERROR_INVALID_IAP:
                g_debug("Invalid (non-existing?) IAP was requested.
                    ");
                break;

            case CON_IC_CONNECTION_ERROR_CONNECTION_FAILED:
                g_debug("Connection just failed.");
                break;

            case CON_IC_CONNECTION_ERROR_USER_CANCELED:
                g_debug("User canceled the connection attempt");
                break;

        }
        break;

    default:
        g_debug("Unknown connection status received");
    }
}

```

Listening for Connection Events Sometimes the application does not want to actively start connections, but it is still interested in knowing, when the device is online. It is possible to achieve this with Libconic "automatic events" feature, which is enabled with "automatic-events" GObject property:

```

/* Create connection object */
ConIcConnection *connection = con_ic_connection_new();

/* Connect signal to receive connection events */
g_signal_connect(G_OBJECT(connection), "connection-event",
    G_CALLBACK(my_connection_handler), NULL);

/* Set automatic events */
g_object_set(G_OBJECT(connection), "automatic-events", TRUE, NULL);

```

When automatic events are turned on, the application will receive connected and disconnected events for all Internet connection changes. In addition to this, the application will receive an event for the initial connection status. If the device is disconnected, *ConIcConnectionEvent* with status *CON_IC_STATUS_DISCONNECTED* will be emitted. This event will have NULL IAP ID and bearer, as there is no IAP getting disconnected, but the event just indicates that the device is offline.

N.B. The main loop has to be reiterated in order to receive the event. If the connection status information is needed synchronously, one can always iterate the main loop oneself:


```

static void connection_info(ConIcConnection *connection,
                           ConIcConnectionEvent *event,
                           gpointer user_data)
{
    ConIcConnectionStatus status = con_ic_connection_event_get_status(
        event);
    ConIcConnectionStatus *status_ptr = (ConIcConnectionStatus*)
        user_data;
    *status_ptr = status;
}

/* ... */

/* Create connection object and set on automatic events (see
previous snippet) ... */
static ConIcConnectionStatus status = 0xFFFF;
ConIcConnection *connection = con_ic_connection_new();
g_signal_connect(G_OBJECT(connection), "connection-event",
                G_CALLBACK(connection_info), &status);
g_object_set(G_OBJECT(connection), "automatic-events", TRUE, NULL);

/* Iterate main loop for the first connection event */
while (status == 0xFFFF) g_main_context_iteration(NULL, TRUE);

if (status == CON_IC_STATUS_CONNECTED)
    g_debug("We are connected!");
else
    g_debug("We are not connected!");

```

Receiving Statistics of Connection Receiving statistics of the current Internet connection is achieved with the `con_ic_connection_statistics()` function and the corresponding event handler. If wanted, it is possible to get statistics for a specified IAP, or just for the current default connection. N.B. Currently the Internet Connectivity daemon provides only one Internet connection at the time, so leaving IAP ID NULL is the best option.

```

static void connection_statistics(ConIcConnection *connection,
                                 ConIcStatisticsEvent *event,
                                 gpointer user_data)
{
    g_debug("Here are all kind of nice statistics about the connection:");
    g_debug("Time active: %u, signal strength: %u, received packets: %llu, "
            "sent packets: %llu, received bytes: %llu, sent bytes: %llu",
            con_ic_statistics_event_get_time_active(event),
            con_ic_statistics_event_get_signal_strength(event),
            con_ic_statistics_event_get_rx_packets(event),
            con_ic_statistics_event_get_tx_packets(event),
            con_ic_statistics_event_get_rx_bytes(event),
            con_ic_statistics_event_get_tx_bytes(event));
}

/* ... */

/* ConIcConnection object named "connection" has already been
created */
g_signal_connect(G_OBJECT(connection), "statistics",

```

```
G_CALLBACK(connection_statistics), NULL);
if (!con_ic_connection_statistics(connection, NULL))
    g_warning("Requesting connection statistics failed!");
```

Getting Proxy Settings With Libconic, it is possible to get Internet connection proxy settings for various protocols. The first step is to query, what kind of proxy mode is in use. This is achieved with the `con_ic_connection_get_proxy_mode()` function. After getting the proxy mode, use the following functions to get the actual proxy settings:

- If proxy mode is `CON_IC_PROXY_MODE_NONE`, do not use any proxies.
- If proxy mode is `CON_IC_PROXY_MODE_MANUAL`, use the following functions to query proxy settings:
 - `con_ic_connection_get_proxy_host()` to get the proxy host
 - `con_ic_connection_get_proxy_port()` to get the proxy port
 - `con_ic_connection_get_proxy_ignore_hosts()` to get a list of hosts, for which the proxy should not be used.
- If proxy mode is `CON_IC_PROXY_MODE_AUTO`, use `con_ic_connection_get_proxy_autoconfig_url()` to get a proxy auto configuration URL.
 - Use of auto configuration URL is explained in [Wikipedia](#)

In this example, the "connection-event" handler is modified to print HTTP proxy settings when establishing connection:

```
static void my_connection_handler(ConicConnection *connection,
                                ConicConnectionEvent *event,
                                gpointer user_data)
{
    ConicConnectionStatus status = con_ic_connection_event_get_status(
        event);
    GSList *ignore_hosts;

    if (status == CON_IC_STATUS_CONNECTED) {
        g_debug("We are connected! \
Let's see what kind of settings we have for HTTP proxy...");

        /* Do things based on specified proxy mode */
        switch (con_ic_connection_get_proxy_mode(connection)) {

            case CON_IC_PROXY_MODE_NONE:
                g_debug("No proxies defined, it is direct connection");
                break;

            case CON_IC_PROXY_MODE_MANUAL:
                g_debug("HTTP proxy %s:%d in use",
                    con_ic_connection_get_proxy_host(connection,
                        CON_IC_PROXY_PROTOCOL_HTTP),
                    con_ic_connection_get_proxy_port(connection,
                        CON_IC_PROXY_PROTOCOL_HTTP));
```

```

        g_debug("List of hosts, for which proxy should not be
            used:");
        ignore_hosts = con_ic_connection_proxy_ignore_hosts(
            connection);
        for (GSLIST *iter = ignore_hosts; iter != NULL; iter =
            g_slist_next(iter))
        {
            g_debug("%s", (gchar *)iter->data);
            g_free(iter->data);
        }
        g_slist_free(ignore_hosts);
        break;

    case CON_IC_PROXY_MODE_AUTO:
        g_debug("Proxy auto-config URL %s should be used",
            con_ic_connection_get_proxy_autoconfig_url(
                connection));
        break;
    }
}
}

```

There are also proxy functions for each individual protocol (like `con_ic_connection_get_proxy_ftp_host()`), but these functions are deprecated and should be avoided in newly-written code.

Getting List of User-Saved Connections All user-saved connections (IAPs) can be queried with the `con_ic_connection_get_all_iaps()` function. The function returns simply a singly linked list of `ConIcIap` objects:

```

/* ConIcConnection object named "connection" has already been
   created */
GSLIST *saved_iaps = con_ic_connection_get_all_iaps(connection);

g_debug("The following connections have been saved by the user:");
for (GSLIST *iter = saved_iaps; iter != NULL; iter = g_slist_next(
    iter)) {

    /* Get IAP object and print some information about it */
    ConIcIap *iap = (ConIcIap *)iter->data;
    g_debug("Connection %s called '%s' using bearer %s",
        con_ic_iap_get_id(iap), con_ic_iap_get_name(iap),
        con_ic_iap_get_bearer_type(iap));

    /* We unref the IAP object as we are not going to use it
       anymore */
    g_object_unref(iap);
}

g_slist_free(saved_iaps);

```

Porting Application from OSSO IC API to Libconic

- If the application has not configured GType, GLib or D-BUS, then set them up:

```

DBusConnection *system_dbus;
GMainloop *main_loop;

```

```
g_type_init();
main_loop = g_main_loop_new(NULL, FALSE);

system_dbus = dbus_bus_get(DBUS_BUS_SYSTEM, NULL);
dbus_connection_setup_with_g_main(system_dbus, NULL);
```

- Include the correct header file:

```
#include <osso-ic.h>

==>

#include <conic.h>
```

- Set up *ConIcConnection* object and "connection-event" handler instead of *osso_iap_cb_t* callback:

```
static void my_connection_cb(struct iap_event_t *event, void
    *arg)
{
    /* ... */
}

/* ... */

osso_iap_cb(my_connection_cb);

==>

static void my_connection_cb(ConIcConnection *connection,
    ConIcConnectionEvent *event,
    gpointer user_data)
{
    /* ... */
}

/* ... */

ConIcConnection *connection = con_ic_connection_new();
g_signal_connect(G_OBJECT(connection), "connection-event",
    "",
    G_CALLBACK(my_connection_cb), app_data)
    ;
```

- Manage connections through *ConIcConnection* API instead of *osso_iap_connect()* and *osso_iap_disconnect()*:

```
osso_iap_connect(OSSO_IAP_ANY, OSSO_IAP_REQUESTED_CONNECT,
    app_data);
osso_iap_disconnect(iap_name, app_data);

==>

con_ic_connection_connect(connection,
    CON_IC_CONNECT_FLAG_NONE);
con_ic_connection_disconnect(connection);
```

- Request statistics with *con_ic_connection_statistics()* instead of *osso_iap_get_statistics()*.

- List all available IAPs with `con_ic_connection_get_all_iaps()` instead of `osso_iap_get_configured_iaps()`.
- Configure autoconf to use Libconic instead of OSSO IC API:

```
PKG_CHECK_MODULES(OSSOIC, osso-ic)
AC_SUBST(OSSOIC_CFLAGS)
AC_SUBST(OSSOIC_LIBS)

==>

PKG_CHECK_MODULES(CONIC, conic)
AC_SUBST(CONIC_CFLAGS)
AC_SUBST(CONIC_LIBS)
```

- In debian/control file "Build-Depends" section, depend on libconic0-dev instead of osso-ic-dev.

1.2.4 Bluetooth Libraries

This section explains how maemo Bluetooth libraries work internally. The following subsections explain the behavior and the decomposition of the Bluetooth library components in detail.

Libgwobex

Libgwobex provides access to libopenobex functionality by providing a helper/wrapper interface for it. Libopenobex is explained in detail in the following section.

The interface to libgwobex can be found at [GW OBEX Library Documentation](#).

Creating Connection

The connection with libgwobex is established using the `gw_obex_setup_dev` function, setting up the connection.

```
#define OBEX_FTP_UUID \
    "\xF9\xEC\x7B\xC4\x95\x3C\x11\xD2\x98\x4E\x52\x54\x00\xDC\x9E\x09"
#define OBEX_FTP_UUID_LEN 16

/* ... */

GwObex* gw_obex_setup_dev (const gchar * device, const gchar * uuid,
                           gint uuid_len,
                           GMainContext * context, gint * error )
```

Listing 1.1: gw-obex.h

The following code snippet illustrates how to open a handle using `gw_obex_setup_dev`.

```
if (ctx->rftcomm_dev) {
    if (ctx->use_ftp)
        ctx->obex = gw_obex_setup_dev(ctx->rftcomm_dev,
                                       OBEX_FTP_UUID, OBEX_FTP_UUID_LEN,
                                       NULL, &err);
}
```

```

else
    ctx->obex = gw_obex_setup_dev(ctx->rftcomm_dev, NULL,
                                  0, NULL, &err);

    if (ctx->obex == NULL)
        printf("OBEX setup failed: %s\n", response_to_string(
            err));
}

```

In this example, `ctx->rftcomm_dev` points to a string containing the device node name (e.g. `/dev/rftcomm0`). `ctx->use_ftp` dictates whether standard folder browsing services should be set up. If `use_ftp` is untrue, then INBOX is connected.

Closing Connection

For closing a gwobex connection, it is possible to use the function

```
void gw_obex_close ( GwObex * ctx );
```

Listing 1.2: gw-obex.h

The following code demonstrates this usage.

```

if (ctx->obex) {
    gw_obex_close(ctx->obex);
    ctx->obex = NULL;
}

```

If `ctx->obex` is not NULL, it will simply be passed as an argument to `gw_obex_close()`.

Using Connection

The libgwobex library provides general file handling functionality, including reading directory structure, browsing in different folders and getting files.

For reading entries from an opened directory, it is possible to use the function

```

gboolean gw_obex_read_dir (GwObex * ctx,
                           const gchar * dir,
                           gchar ** buf,
                           gint * buf_size,
                           gint * error );

```

Listing 1.3: gw-obex.h

`gw_obex_read_dir` reads an entry from the selected folder and returns the result in the `buf` argument given to the function.

```

gboolean ret;

/* ... */

ret = gw_obex_read_dir(ctx->obex, dir, buf, buf_size, err);

```

This reads an entry from the directory `dir` (`char *`) and returns it in `buf` (`char **`).

For changing the current directory, it is possible to use the function:

```
gboolean gw_obex_chdir (GwObex * ctx, const gchar * dir, gint * error )
;
```

Listing 1.4: gw-obex.h

which changes the directory of the FTP connection. Below is a code example using this function.

```
/* Ignore parent dir pointers */
if (g_str_equal(name, ".."))
    return TRUE;

if (!gw_obex_chdir(ctx->obex, name, err)) {
    printf("Could not chdir to %s\n", name);
    return FALSE;
}
```

To retrieve files over the OBEX connection, the `gw_obex_get_file` function can be used.

```
gboolean gw_obex_get_file (GwObex * ctx,
                           const gchar * local,
                           const gchar * remote,
                           const gchar * type,
                           gint * error);
```

Listing 1.5: gw-obex.h

`gw_obex_get_file` uses the `ctx` context for retrieving the remote file to local file.

```
gboolean ret;

ret = gw_obex_get_file(ctx->obex, name, name, err);
```

There are a lot more functions that can be performed by using libgwobex wrapper directly, for full list of functions and their usage, see the [API document](#).

Libopenobex

The LibOpenOBEX library implements a generic OBEX Session Protocol. It does not implement the OBEX Application Framework. OBEX is a protocol designed to allow data interchanging between different kinds of connections (e.g. Bluetooth, IrDA). Specific information about the OBEX protocol can be found at <http://www.irda.org>, by selecting the Developer->Specifications category. OBEX is similar to HTTP protocol, expect for a few differences:

- Transports: While HTTP is normally layered above a TCP/IP connection, OBEX is usually transported over IrLAP/IrLMP/Tiny TP (on IrDA) or over Baseband/Link Manager/L2CAP/RFCOMM (on Bluetooth).
- Binary transmissions: OBEX communicates using binary transmissions, as HTTP is transmitted in a human-readable XML-based format.
- Session support: HTTP is stateless, while OBEX maintains the connection.

A fairly good overlook of OBEX can be found at <http://en.wikipedia.org/wiki/OBEX>.

Code examples for libopenobex can be obtained from <http://openobex.triq.net/downloads>, from the example apps package.

Using BlueZ D-Bus API

The BlueZ system exports a D-Bus API that can be employed instead of OSSO Bluetooth tools. See the following documents:

- [BlueZ D-Bus API documentation](#)
- [BlueZ Wiki](#)

1.2.5 Connectivity UI

UI Components

Connectivity UI contains various dialogs and other components used to control the connectivity. The different UI parts are:

- Connection manager
- Connectivity dialogs
- Status bar applets
- Control panel applet
- Bluetooth UIs

The connectivity dialogs are invoked by D-Bus method calls, so for example the ICd is using these D-Bus method calls for showing dialogs when they are needed. The next section specifies the D-Bus API of maemo connectivity UI.

D-Bus Connectivity UI Interface

If some information is needed from the user about the IAP that is about to be connected to, the following can be used.

```
Service:      com.nokia.icd_ui
Interfaces:   com.nokia.icd_ui
Object paths: /com/nokia/icd_ui
```

The Internet Connectivity UIs implement the following D-Bus API used by the ICd and EAP.

```
Method:      show_conn_dlg
Parameters:  none
Return parameters: none
Errors:      com.nokia.icd_ui.error.flight_mode:
             Flight mode enabled, dialog not shown
Description: Shows the Connect Dialog where the user can choose an IAP.
```

```
Method:      show_disconnect_dlg
Parameters:  none

Return Parameters: none
Errors:      com.nokia.icd_ui.error.flight_mode:
             Flight mode enabled, dialog not shown
Description: Shows the disconnect dialog.
```


Method:	show_retry_dlg
Parameters:	1. string Bluetooth address of the device used with SAP 2. string Name of the connection attempt error which selects the retry dialog type.
Return Parameters:	none
Errors:	com.nokia.icd_ui.error.flight_mode: Flight mode enabled, dialog not shown
Description:	Shows the retry dialog.

Method:	show_change_dlg
Parameters:	1. string Name of the currently active IAP 2. string Name of the IAP to be activated
Return Parameters:	none
Errors:	com.nokia.icd_ui.error.flight_mode: Flight mode enabled, dialog not shown
Description:	Shows the Change IAP Dialog

Method:	show_passwd_dlg
Parameters:	1. string Username supplied by ICD 2. string Password supplied by ICD 3. string Name of the IAP
Return Parameters:	none
Errors:	com.nokia.icd_ui.error.flight_mode: Flight mode enabled, dialog not shown
Description:	Shows the username/password dialog.

Method:	show_gtc_dlg
Parameters:	1. string GTC challenge string
Return Parameters:	none
Errors:	com.nokia.icd_ui.error.flight_mode: Flight mode enabled, dialog not shown
Description:	Shows EAP GTC challenge dialog.

Method:	show_mschap_change_dlg
Parameters:	1. string Supplied username 2. string Old password that is to be changed 3. string Name of the IAP
Return Parameters:	none
Errors:	com.nokia.icd_ui.error.flight_mode: Flight mode enabled, dialog not shown
Description:	Shows EAP MSCHAPv2 change password dialog.

Method:	show_private_key_passwd_dlg
Parameters:	1. uint32 The private key ID
Return Parameters:	none
Errors:	com.nokia.icd_ui.error.flight_mode: Flight mode enabled, dialog not shown
Description:	Shows EAP private key password dialog

Method: show_server_cert_dlg
Parameters: 1. string Certificate name
2. string Certificate serial
3. boolean TRUE if certificate is expired, FALSE otherwise
4. boolean TRUE if root CA is unknown or self-signed certificate, FALSE otherwise

Return Parameters: none
Errors: com.nokia.icd_ui.error.flight_mode:
Flight mode enabled, dialog not shown

Description: Shows server certificate error and expiration dialogs. If both boolean arguments are false, the error dialog is shown. If either or both boolean arguments are TRUE, the expiration dialog is shown instead.

Method: strong_bt_req
Parameters: 1. string Bluetooth address of the device to pair with
2. boolean TRUE if strong authentication enabled, FALSE if strong authentication is disabled

Return Parameters: none
Errors: com.nokia.icd_ui.error.flight_mode:
Flight mode enabled, dialog not shown

Description: Requests strong (16 digit) BT PIN dialog for a BT device

Method: show_sim_pin_dlg
Parameters: 1. string Bluetooth address of the device used with SAP
2. boolean TRUE if PIN was incorrect and retry dialog should be displayed before asking PIN. FALSE if this is the first PIN request.

Return Parameters: none
Errors: com.nokia.icd_ui.error.flight_mode:
Flight mode enabled, dialog not shown

Description: Shows SIM PIN dialog

The code example for the application to show the connect dialog using show_conn_dlg is following. Please note the use of macro for doing this.

```
#include <osso-ic-ui-dbus.h>

/* ... */

/* in our code somewhere, where we need the Connect Dialog*/
DBusMessage *uimsg;

/* construct the message for Connect Dialog request*/
uimsg =
    dbus_message_new_method_call(ICD_UI_DBUS_SERVICE,
                                ICD_UI_DBUS_PATH,
                                ICD_UI_DBUS_INTERFACE,
                                /*macro for show_conn_dlg */
                                ICD_UI_SHOW_CONNDLG_REQ);

/* send the message */
reply =
    dbus_connection_send_with_reply_and_block(connection,
                                              uimsg,
                                              reply_timeout,
                                              &error);

if (reply == NULL) {
    DLOG_ERR("Failed to show connect dialog: %s", uierror.message);
}
```

```

        dbus_error_free(&uierror);
    }

    dbus_message_unref(uimsg);
    dbus_message_unref(reply);

    /* ... */

```

The signals emitted from com.nokia.icd_ui interface are listed below.

Signal:	disconnect
Parameters:	1. boolean TRUE if "disconnect" pressed, FALSE if "cancel"
Description:	Signal emitted from UI when disconnect dialog has been closed.

Signal:	retry
Parameters:	1. string The IAP that is to be retried 2. boolean TRUE if "retry" pressed, FALSE if "cancel"
Description:	Signal emitted from UI when the retry dialog has been closed.

Signal:	change
Parameters:	1. string Old IAP to change from 2. string New IAP to change to 3. boolean Change to the new IAP If TRUE, keep old if FALSE
Description:	Signal emitted from UI when change connection dialog has been closed.

Signal:	passwd
Parameters:	1. string Username supplied or modified by the user 2. string Password supplied or modified by the user 3. string IAP name 4. boolean TRUE if "ok" pressed, FALSE if "cancel"
Description:	Signal emitted from UI when the username/password dialog has been closed

Signal:	gtc_response
Parameters:	1. string Response to the given challenge or empty string if cancelled 2. boolean TRUE if "ok" pressed, FALSE if "cancel"
Description:	Signal emitted from UI when the EAP GTC challenge dialog has been closed.

Signal:	mschap_change
Parameters:	1. string Supplied username 2. string The new password or empty string if cancelled 3. string IAP name 4. boolean TRUE if "ok" pressed, FALSE if "cancel"
Description:	Signal emitted from UI when the MSCHAPv2 password has been changed

Signal:	private_key_passwd
Parameters:	1. uint32 The id of the private key 2. string Password for the private key or empty string if none 3. boolean TRUE if "ok" pressed, FALSE if "cancel"
Description:	Signal emitted from UI when the private key password dialog has been closed

Signal:	server_cert
Parameters:	1. boolean TRUE if strong PIN entered, FALSE if strong PIN dialog was canceled
Description:	Signal emitted from UI when the server certificate error dialog has been closed

Signal:	strong_bt
Parameters:	1. boolean TRUE if strong PIN entered, FALSE if strong PIN dialog was cancelled
Description:	Signal emitted from UI when the strong (16 digit) BT PIN has been entered

Signal:	sim_pin
Parameters:	1. string SIM PIN code or empty string if cancelled 2. boolean TRUE if "ok" pressed, FALSE if "cancel"
Description:	Signal emitted from UI when the SIM PIN has been entered.

Bluetooth DBUS UI dialogs

```

/**
 * Example of use (command line):
 *
 * dbus-send --system --print-reply \
 *   --dest='com.nokia.icd_ui' /com/nokia/bt_ui \
 *   com.nokia.bt_ui.show_send_file_dlg \
 *   array:string:file:///home/user/MyDocs/.documents/testing.txt
 *
 * dbus-send --system --print-reply \
 *   --dest=com.nokia.bt_ui /com/nokia/bt_ui
 *   com.nokia.bt_ui.show_search_dlg \
 *   string: string: array:string: boolean:true
 */

#ifndef CONBTDIALOGS_DBUS_H
#define CONBTDIALOGS_DBUS_H

#ifdef __cplusplus
extern "C" {
#endif

/** Conbtdialogs service, resides in system dbus */
#define CONBTDIALOGS_DBUS_SERVICE "com.nokia.bt_ui"
/** Conbtdialogs interface */
#define CONBTDIALOGS_DBUS_INTERFACE "com.nokia.bt_ui"
/** Conbtdialogs path */
#define CONBTDIALOGS_DBUS_PATH "/com/nokia/bt_ui"

/**
 * Show send file dialog
 *
 * Arguments:
 *
 * uris: DBUS_TYPE_ARRAY          Array of strings representing
 *       the URIs of the          the files to send.
 *
 * Returns:

```

```

    DBUS_TYPE_BOOLEAN          TRUE, if dialog was shown successfully.
*/
#define CONBTDIALOGS_SEND_FILE_REQ          "show_send_file_dlg"

/**
    File sending result signal

    Arguments:

    success: DBUS_TYPE_BOOLEAN          TRUE, if all files were sent
            successfully or
            FALSE, if error occurred or sending was
            canceled.
*/
#define CONBTDIALOGS_SEND_FILE_SIG          "send_file"

/**
    Show BT device search dialog

    Arguments:

    major_class: DBUS_TYPE_STRING To set filtering based on major_class
            or
            "". Possible major class values are:
            "miscellaneous", "computer", "phone",
            "access point", "audio/video",
            "peripheral", "imaging", "wearable",
            "toy" and "uncategorized".

    minor_class: DBUS_TYPE_STRING To set filtering based on minor_class
            or "".
            Possible minor class values are:
            - Minor classes for "computer":
              "uncategorized", "desktop", "server",
              "laptop", "handheld", "palm", "
              wearable"
            - Minor classes for "phone": "
              uncategorized",
              "cellular", "cordless", "smart phone
              ",
              "modem", "isdn"

    service_classes: DBUS_TYPE_ARRAY To set filtering based on service
            classes.
            Supported classes include "
            positioning",
            "networking", "rendering", "
            capturing",
            "object transfer", "audio", "
            telephony",
            "information". Can be empty list,
            when no
            service class filtering is performed
            .

    bonding: DBUS_TYPE_STRING          Bonding mode for found and selected
            device:

```

```

        "require" for requiring a bonding from
        a
        selected device (i.e. bond device if it
        has not
        been bonded before).

        "force" to always bond (i.e. device
        will be
        bonded even if bonded before).

        Any other string will allow to search
        and
        select device without bonding it.

Returns:

    DBUS_TYPE_BOOLEAN        TRUE, if dialog was shown successfully.
*/
#define CONBTDIALOGS_SEARCH_REQ        "show_search_dlg"

/**
Bluetooth search result signal

Arguments:

    address: DBUS_TYPE_STRING        Bluetooth address of the selected
        device, or ""                if search dialog was cancelled.

    name: DBUS_TYPE_STRING        Name of the device.

    icon: DBUS_TYPE_STRING        Logical name for the icon describing
        the                        device.

    major_class: DBUS_TYPE_STRING Major class of the device.

    minor_class: DBUS_TYPE_STRING Minor class of the device.

    trusted: DBUS_TYPE_BOOLEAN    Defines whether the device is marked as
        a trusted device.

    services: DBUS_TYPE_ARRAY    List of strings describing the service
        classes                    and SDP-based services provided by the
                                    device.

*/
#define CONBTDIALOGS_SEARCH_SIG        "search_result"

```

Listing 1.6: conbtdialogs-dbus.h

```

/**
Bluetooth UI Library for maemo

Copyright (C) 2006 Nokia. All rights reserved.

This sample demonstrates the use of conbtdialogs API and especially
send_file function. Compile the program with conbtdialogs and dbus:

gcc -Wall 'pkg-config --libs --cflags dbus-glib-1 conbtdialogs' \
-o send_file conbtdialogs_send_file.c

```

```

Run with list of URLs:

./send_file file:///home/user/MyDocs/.sounds/Everyday.mp3
*/

#define DBUS_API_SUBJECT_TO_CHANGE

#include <glib.h>
#include <conbtdialogs-dbus.h>
#include <dbus/dbus.h>
#include <dbus/dbus-glib.h>

DBusGConnection *connection = NULL;
GMainLoop *mainloop = NULL;

static gboolean initialize(void)
{
    GError *error = NULL;

    g_type_init ();

    /* Create main loop */
    mainloop = g_main_loop_new(NULL, TRUE);
    if ( mainloop == NULL ) return FALSE;

    /* Create DBUS connection */
    connection = dbus_g_bus_get(DBUS_BUS_SYSTEM, &error);

    if (connection == NULL )
    {
        g_print ("Error: %s\n", error->message);
        g_clear_error (&error);
        return FALSE;
    }

    return TRUE;
}

static gboolean uninitialize(void)
{
    /* Quit main loop and unref it */
    if (mainloop != NULL)
    {
        g_main_loop_quit(mainloop);
        g_main_loop_unref(mainloop);
    }

    return TRUE;
}

static DBusHandlerResult file_sent_signal ( DBusConnection *connection,
                                           DBusMessage *message,
                                           void *data )
{
    gboolean success = FALSE;

    /* check signal */
    if (!dbus_message_is_signal(message,

```

```

CONBTDIALOGS_DBUS_INTERFACE,
CONBTDIALOGS_SEND_FILE_SIG))

return DBUS_HANDLER_RESULT_NOT_YET_HANDLED;

/* get args */
if ( !dbus_message_get_args ( message, NULL,
                                DBUS_TYPE_BOOLEAN, &success,
                                DBUS_TYPE_INVALID ) )
    return DBUS_HANDLER_RESULT_NOT_YET_HANDLED;

/* print if file sending was success or failure */
g_print ( "File sending was a " );

if (success) g_print("success\n"); else g_print("failure\n");
dbus_connection_close(connection);
uninitialize();

return DBUS_HANDLER_RESULT_HANDLED;
}

gint main(gint argc, gchar **argv)
{
    GError *error = NULL;
    gchar **files = NULL;
    gint idx = 0;
    DBusGProxy *proxy;
    DBusConnection *sys_conn;
    gchar *filter_string = NULL;

    if (argc < 2) return 1;

    if (initialize() == FALSE) {
        uninitialize();
        return 1;
    }

    /* Copy urls to GLib compatible char array */
    files = g_new0(gchar*, argc);

    for (idx = 1; idx < argc; idx++)
        files[idx-1] = g_strdup(argv[idx]);

    files[argc-1] = NULL;

    /* Open connection for btdialogs service */
    proxy = dbus_g_proxy_new_for_name(connection,
                                       CONBTDIALOGS_DBUS_SERVICE,
                                       CONBTDIALOGS_DBUS_PATH,
                                       CONBTDIALOGS_DBUS_INTERFACE);

    /* Send send file request to btdialogs service */
    if (!dbus_g_proxy_call(proxy, CONBTDIALOGS_SEND_FILE_REQ,
                           &error,
                           G_TYPE_STRV, files, G_TYPE_INVALID,
                           G_TYPE_INVALID))
    {
        g_print("Error: %s\n", error->message);
        g_clear_error(&error);
        g_strfreev (files);
        g_object_unref(G_OBJECT(proxy));
        uninitialize();
    }
}

```



```

        return 1;
    }
    g_strfreev (files);
    files = NULL;

    g_object_unref(G_OBJECT(proxy));

    /* Now wait for file sent signal, use low level bindings as glib
       bindings require signal marshaller registered */
    sys_conn = dbus_bus_get(DBUS_BUS_SYSTEM, NULL);
    g_assert(dbus_connection_add_filter(sys_conn,
                                       file_sent_signal,
                                       NULL,
                                       NULL ));

    filter_string =
        g_strdup_printf ("type='signal',interface='%s'",
                        CONBTDIALOGS_DBUS_INTERFACE);

    dbus_bus_add_match(sys_conn, filter_string, NULL);
    dbus_connection_unref(sys_conn);

    /* Run mainloop */
    g_main_loop_run(mainloop);

    return 0;
}

```

1.2.6 Samba Network Shares

The device has support for connecting Samba network shares. To access Samba network shares with GnomeVFS, a Samba module for the GnomeVFS (libossgnomevfs2-samba) needs to be installed.

Here is a simple example on how to read a file via Samba:

```

#include <libgnomevfs/gnome-vfs.h>
#include <stdio.h>

int main(void)
{
    GnomeVFSHandle *handle = NULL;
    GnomeVFSResult res;
    GnomeVFSFileSize bytes_read;
    gchar buf[1024];

    // URI to open for reading
    gchar *uri = "smb://host/file.txt";

    if (!gnome_vfs_init()) {
        fprintf(stderr, "GnomeVFS initialization failed.\n");
        return 1;
    }

    if (GNOME_VFS_OK != (res = gnome_vfs_open(&handle, uri,
        GNOME_VFS_OPEN_READ))) {
        fprintf(stderr, "GnomeVFS open failed.\n");
        return 2;
    }

    while (GNOME_VFS_OK == res) {
        res = gnome_vfs_read(handle, buf, sizeof(buf)-1, &bytes_read);
    }
}

```

```

        buf[bytes_read] = 0;

        // Write buffer to stdout
        write(1, buf, bytes_read);

        if(bytes_read == 0)
            break;
    }

    if (GNOME_VFS_OK != gnome_vfs_close(handle))
        fprintf(stderr, "GnomeVFS close failed\n");

    gnome_vfs_shutdown();
    return 0;
}

```

Example can be compiled including GnomeVFS:

```
gcc 'pkg-config gnome-vfs-2.0 --libs --cflags' vfscat.c -o vfscat
```

1.2.7 Location

Location framework has one library; liblocation. The liblocation is made up of functions for parsing GPSD's output, for controlling GPSD and for other helper functions.

Using Liblocation

The headers for liblocation are stored in the location subdirectory, and so they should be included as follows.

```
#include <location/location-gps-device.h>
```

Listening to GPSD

GPSD is used in maemo to talk to GPS devices and report position data. Liblocation contains an object that listens to GPSD and converts the output into GObject signals. Creating the object goes as follows

```
LocationGPSDevice *device;

device = g_object_new (LOCATION_TYPE_GPS_DEVICE, NULL);
```

It is then possible to connect to the changed signal to hear about gps fix changes.

```
g_signal_connect (device, "changed", G_CALLBACK (location_changed),
    NULL);
```

And the location changed callback looks like this

```
static void
location_changed (LocationGPSDevice *device, gpointer userdata)
{
    g_print ("Latitude: %.2f\nLongitude: %.2f\nAltitude: %.2f\n",
        device->fix->latitude, device->fix->longitude, device
        ->fix->altitude);
}
```

That is all that is required for a simple GPS client that listens for GPSD. N.B. The LocationGPSTDevice object has some public fields. They are as follows:

- device->online: Whether GPSD is connected to a GPS
- device->status: The GPS status
- device->fix: The GPS fix information
- device->satellites_in_view: The number of satellites that the GPS can see
- device->satellites_in_use: The number of satellites used in fix calculation
- device->satellites: An array of satellite details

The most important of these is device->fix, as this contains the fix location data from GPSD. The fields of the LocationGPSTDeviceFix structure are fairly self-explanatory except for the fields field. This field is a bitmask of what other fields in the structure have valid content. To check whether the latitude and longitude fields are valid, you would do

```
if (fix->fields & LOCATION_GPS_DEVICE_LATLONG_SET)
```

and so on. The bitmask flags are defined in location-gps-device.h

Controlling GPSD

Sometimes a program may need to start or stop GPSD, or find out when GPSD has been started or stopped. This is accomplished using the LocationGPSTDeviceControl object. This object can only be created once in a program, so it is obtained with the location_gpsd_control_get_default() function (see location-gpsd-control.h).

```
LocationGPSTDeviceControl *control;  
  
control = location_gpsd_control_get_default ();
```

There are three signals on this object: error, gpsd_running and gpsd_stopped. Error is emitted when there is an error starting GPSD, gpsd_running is emitted whenever GPSD starts up, and gpsd_stopped is emitted when GPSD stops. Only one application is able to control GPSD at a time, and this will be the application that initially starts it. The control->can_control field will be TRUE if the application can control it, or FALSE if it cannot.

Other Liblocation Functions

Liblocation also comes with a function for getting the distance between two points. This is called the great-circle distance (see [Wikipedia](#) for more details on it). The function location_distance_between() (see location-distance-utils.h) takes the latitude and longitude of two locations and returns the distance between them in kilometers.

```
g_print ("distance between LAX and BNA is %fkm\n",  
        location_distance_between (36.12, -86.67, 33.94, -118.40));
```

Compiling Programs With LibLocation Support

Liblocation comes with a pkgconfig file, so adding support to a program is just a case of adding liblocation to the configure scripts PKG_CHECK_MODULES macro, as follows:

```
PKG_CHECK_MODULES (LOCATION)
AC_SUBST (LOCATION_CFLAGS)
AC_SUBST (LOCATION_LIBS)
```

Then in the Makefile.am, these variables need to be added to the program's CFLAGS and LDADD flags.

1.3 Implementing Custom Connection Managers

This section introduces briefly how to implement a custom connection manager plug-in for the Internet Tablet OS. The messaging framework in Internet Tablet OS is based on [Telepathy framework architecture](#).

Telepathy provides [D-Bus](#)-based framework that unifies all forms of real-time communication, such as instant messaging, IRC, voice and video over Internet. The framework provides an interface for plug-ins to extend the protocol support by implementing new connection managers. These new supported protocols can then be used by all client applications that communicate via [Telepathy framework architecture](#).

For more detailed information and source code examples, see the Telepathy framework [specification](#) by [Freedesktop.org](#).

Since the connection manager uses D-Bus for communication, it can be implemented using any language supporting D-Bus communication, even an interpreted language such as Python. However, in order to enable running on Internet tablets natively, C or C++ is currently preferred.

1.3.1 Connection Manager Implementation Examples

Several open source Telepathy connection manager implementations already exist. See, for instance, [Gabble](#) and [Idle](#) projects. In maemo 4.x, there is a source package of telepathy-gabble that has been modified for maemo platform and packaged as a DEB package.

The package can be downloaded from the 4.x repository. The package can be built the usual way with dpkg-buildpackage.

1.3.2 Connection Manager and Connections

Telepathy connection managers establish the actual connections to IM or VoIP servers. A connection manager provides support for one or more connection protocols, e.g. SIP. Connection managers are started using D-Bus service activation, and they present a connection manager object to the bus. A connection can be established by sending a D-Bus message request to the connection manager object. A new D-Bus object is then created to handle the new connection. The D-Bus interface of a connection manager is:

```

org.freedesktop.Telepathy.ConnectionManager
Methods:
GetParameters ( s: proto ) -> a(susv)
ListProtocols ( ) -> as
The following well-known values should be used when applicable:

    * aim - AOL Instant Messenger
    * gadugadu - Gadu-Gadu
    * groupwise - Novell Groupwise
    * icq - ICQ
    * irc - Internet Relay Chat
    * jabber - Jabber (XMPP)
    * msn - MSN Messenger
    * napster - Napster
    * silc - SILC
    * sip - Session Initiation Protocol (SIP)
    * trepia - Trepia
    * yahoo - Yahoo! Messenger
    * zephyr - Zephyr

RequestConnection ( s: proto, a{sv}: parameters ) -> so
Signals:
NewConnection ( s: bus_name, o: object_path, s: proto )
Sets of flags:
Conn_Mgr_Param_Flags

```

The connection object provides the interfaces for basic connection signaling as well as for requesting communication channels. The D-Bus interface of a connection is:

```

org.freedesktop.Telepathy.Connection
Methods:
Connect ( ) -> None
Disconnect ( ) -> None
GetInterfaces ( ) -> as
GetProtocol ( ) -> s
GetSelfHandle ( ) -> u
GetStatus ( ) -> u
HoldHandles ( u: handle_type, au: handles ) -> None
InspectHandles ( u: handle_type, au: handles ) -> as
ListChannels ( ) -> a(osuu)
ReleaseHandles ( u: handle_type, au: handles ) -> None
RequestChannel ( s: type, u: handle_type, u: handle, b: suppress_handler ) -> o
RequestHandles ( u: handle_type, as: names ) -> au
Signals:
NewChannel ( o:object_path, s:channel_type, u:handle_type, u:handle, b:suppress_handler)
StatusChanged ( u: status, u: reason )

```

1.3.3 Channels and Channel Types

Communication with the server and other contacts is carried out with instances of various channel types. The interface for creating, closing and handling channels is the following:

```

org.freedesktop.Telepathy.Channel
Methods:
Close ( ) -> None
GetChannelType ( ) -> s
GetHandle ( ) -> uu
GetInterfaces ( ) -> as
Signals:
Closed ( )

```

Various channel types are supported in the Telepathy framework to match the needs of the real-time communication protocol. For example, a text channel provides methods to send messages, and is able to send signals to indicate that

messages have been sent and received.

```
org.freedesktop.Telepathy.Channel.Type.ContactList

org.freedesktop.Telepathy.Channel.Type.ContactSearch
Methods:
GetSearchKeys ( ) -> sa{s(bg)}
GetSearchState ( ) -> u
Search ( a{sv}: terms ) -> None
Signals:
SearchResultReceived ( u: contact, a{sv}: values )
SearchStateChanged ( u: state )

org.freedesktop.Telepathy.Channel.Type.StreamedMedia
Methods:
ListStreams ( ) -> a(uuuuuu)
RemoveStreams ( au: streams ) -> None
RequestStreamDirection ( u: stream_id, u: stream_direction ) -> None
RequestStreams ( u: contact_handle, au: types ) -> a(uuuuuu)
Signals:
StreamAdded ( u: stream_id, u: contact_handle, u: stream_type )
StreamDirectionChanged ( u: stream_id, u: stream_direction, u: pending_flags )
StreamError ( u: stream_id, u: errno, s: message )
StreamRemoved ( u: stream_id )
StreamStateChanged ( u: stream_id, u: stream_state )

org.freedesktop.Telepathy.Channel.Type.RoomList
Methods:
GetListingRooms ( ) -> b
ListRooms ( ) -> None
Signals:
GotRooms ( a(usa{sv}): rooms )
ListingRooms ( b: listing )

org.freedesktop.Telepathy.Channel.Type.Text
Methods:
AcknowledgePendingMessages ( au: ids ) -> None
GetMessageTypes ( ) -> au
ListPendingMessages ( b: clear ) -> a(uuuuus)
Send ( u: type, s: text ) -> None
Signals:
LostMessage ( )
Received ( u: id, u: timestamp, u: sender, u: type, u: flags, s: text )
SendError ( u: error, u: timestamp, u: type, s: text )
Sent ( u: timestamp, u: type, s: text )
```

1.3.4 Additional Connection Interfaces

The connection interfaces can handle special needs of the connection protocol, such as aliasing, which means that contacts can have an alias that they can change via the server. The D-Bus interfaces are briefly presented below. More details can be found in the [specification](#).

```

org.freedesktop.Telepathy.Connection.Interface.Aliasing
Methods:
GetAliasFlags ( ) -> u
RequestAliases ( au: contacts ) -> as
SetAliases ( a{us}: aliases ) -> None
Signals:
AliasesChanged ( a(us): aliases )

org.freedesktop.Telepathy.Connection.Interface.Avatars
Methods:
GetAvatarRequirements ( ) -> asqqqqq
GetAvatarTokens ( au: contacts ) -> as
RequestAvatar ( u: contact ) -> ays
SetAvatar ( ay: avatar, s: mime_type ) -> s
Signals:
AvatarUpdated ( u: contact, s: new_avatar_token )

org.freedesktop.Telepathy.Connection.Interface.Capabilities
Methods:
AdvertiseCapabilities ( a(su): add, as: remove ) -> a(su)
GetCapabilities ( au: handles ) -> a(usuu)
Signals:
CapabilitiesChanged ( a(usuuuu): caps )

org.freedesktop.Telepathy.Connection.Interface.ContactInfo
Methods:
RequestContactInfo ( u: contact ) -> None
Signals:
GotContactInfo ( u: contact, s: vcard )

org.freedesktop.Telepathy.Connection.Interface.Forwarding
Methods:
GetForwardingHandle ( ) -> u
SetForwardingHandle ( u: forward_to ) -> None
Signals:
ForwardingChanged ( u: forward_to )

org.freedesktop.Telepathy.Connection.Interface.Presence
Methods:
AddStatus ( s: status, a{sv}: parms ) -> None
ClearStatus ( ) -> None
GetStatuses ( ) -> a{s(ubba{ss})}
RemoveStatus ( s: status ) -> None
RequestPresence ( au: contacts ) -> None
SetLastActivityTime ( u: time ) -> None
SetStatus ( a{sa{sv}}: statuses ) -> None
Signals:
PresenceUpdate ( a{u(ua{sa{sv}})}: presence )

org.freedesktop.Telepathy.Connection.Interface.Privacy
Methods:
GetPrivacyMode ( ) -> s
GetPrivacyModes ( ) -> as
SetPrivacyMode ( s: mode ) -> None
Signals:
PrivacyModeChanged ( s: mode )

org.freedesktop.Telepathy.Connection.Interface.Renaming
Methods:
RequestRename ( s: name ) -> None
Signals:
Renamed ( u: original, u: new )

```

1.4 Using STUN in Applications

This section shows how to use the *libjingle* API to create peer-to-peer connections between parties that are behind NAT routers.

1.4.1 Network Address Translation (NAT)

NAT routers were born because of the imminent IP address exhaustion. The currently most used Internet address family, the Internet Protocol version 4, is 32 bits wide, meaning roughly four billion available addresses.

Even though four billion addresses may seem like an abundant resource, not all addresses can be assigned to hosts, just as not all telephone number permutations can be assigned to users. IP addresses also carry routing information, analogous to the prefix of a telephone number that identifies country, region, city, etc.

The definitive solution for the problem is to increase the address length. For this, IPv6 was designed, introducing 128-bit addresses. Adopting new address families cannot be done over night, so an intermediate solution was found: NAT - Network Address Translation.

A NAT router acts like a switchboard between the public Internet and a private network. The NAT router needs to have at least one valid Internet address in order to be "seen" by the rest of the Internet. The computers in the private network have private address in the ranges 10.0.0.0/8, 192.168.0.0/16 or 172.16.0.0/12, which are not routable in the Internet.

When a private computer, e.g. address 10.0.0.1, tries to connect to a public server 200.215.89.79, the network packet must pass through the NAT router. The NAT router knows that the source address 10.0.0.1 is not routable and replaces 10.0.0.1 with its own valid address (e.g. 64.1.2.3), and sends the packet forward.

The remote server 200.215.89.79 will receive a connection from the host 64.1.2.3, and will reply to that host.

When the response packet comes to the NAT router, it must have a way of telling whether the packet is meant for the router itself or a host in the private network. Once the NAT router resolves the packet to the active connection, it changes the destination address from 64.1.2.3 to 10.0.0.1, and delivers the packet to the correct recipient in the private network.

In more technical depth, NAT is possible because absolutely every network connection on the Internet has a unique tuple consisting of the following values:

- Client address (the host that initiates the connection)
- Server address (the passive side that receives the initiation packet)
- Client port number
- Server port number
- The transport protocol e.g. TCP, UDP, SCTP.

If the NAT router replaces the client address with its own, but also replaces the client port number when necessary to avoid a clash with another active connection, the uniqueness of each connection is retained. NAT allows for a virtually unlimited number of computers in a private network to access the Internet via only one NAT router with one public IP address.

1.4.2 NAT Problems

There are some disadvantages in the NAT system. Firstly, the NAT router needs to keep connection states in memory, which partially breaks a cornerstone in Internet philosophy ("dumb routers, smart hosts"). If a NAT router is reset, all connections will be terminated, while a regular router could be reset without breaking any connection.

Secondly, the hosts in a private network cannot easily provide a service to the public Internet, i.e. these hosts cannot be the "passive" side in connections, since the initiation packets will come to the NAT router and it will have no related connection in its memory.

A partial solution for that problem is to open a "hole" in the NAT for specific ports, for example any connection to the port 8000 of the NAT router should be redirected to the machine 10.0.0.2 port 80. It works, but it does not scale up. The number of ports is limited, some protocols work only on a very specific port numbers, and each port requires manual configuration of the NAT router. Manual configuration is a solution only when there are few protocols and users.

Since the bulk of the Internet traffic is HTTP and initiated from the private network, NATs are adequate for most needs.

1.4.3 NAT Peer-to-Peer Circumvention Techniques

The most problematic services to deploy in presence of NATs are peer-to-peer (P2P) applications, where two clients of the service make direct connections to each other without sending data through an intermediate server. This is a problem for SIP-based VoIP and most P2P file sharing networks.

If one of the P2P parties has a routable IP address, the problem is easily solved: the party behind the NAT router must initiate the connection. Unfortunately, the most common case is where both the P2P parties are behind NAT routers.

Several techniques have been proposed to solve this issue. The most elegant solutions require both software updates in the NAT router itself, and explicit support in the client software. One of these solutions is the UPnP IGD (Internet Gateway Device) protocol. This protocol gives a host means to request to open a hole in the NAT router to make a service accessible from the Internet.

IGD is easy to use and has been enjoying good support from NAT router vendors, but it is still far from ubiquitous, and does not work behind two or more NAT routers, which is a common situation.

It is important to remember that none of the existing NAT circumvention techniques, not even the most elegant, can completely solve the problem. There still has to be a signaling protocol for the parties to exchange connection parameters. In other words, it does not suffice to be able to open a hole in the NAT router; there must be a way to *communicate* the address and the port of the hole to the remote party. A pure P2P service can only be achieved when all parties have public addresses. This is expected to happen when IPv6 is widely deployed.

1.4.4 STUN, TURN and ICE Protocols

There are NAT circumvention techniques that do not require router software updates. STUN (Simple Transversal of UDP through NATs) is the most common one. STUN exploits the connectionless nature of UDP, as well as a security weakness of some NAT implementations. The technique is bound to UDP features; hence STUN cannot be used for example for TCP connections.

The STUN protocol requires a STUN server with a well-known public IP address on the Internet. STUN sends the private address and port in the payload of a UDP datagram to the STUN server. If the packet goes through a NAT router, the address and port are changed in the IP headers, while the ones in the payload remain the same.

The STUN server returns the actual address and port to the sender and the client can resolve the type of NAT, if any, from the response. Some NATs have poor implementations which, in conjunction with STUN, allow for incoming UDP flow. These are called full cone NATs. They always translate the same source address and port tuple to the same NAT router port. This relieves the router from storing full connection state. Any packet coming from the Internet to a router port will be delivered to the private host, despite its source.

This allows for the private host to easily punch a hole in the NAT router. The first packet going to the STUN server opens the hole. In turn, the private host learns from the STUN server the IP address and port number of the hole. These parameters are then sent to the remote peer via a signaling protocol. The remote peer can then send UDP data directly.

Unfortunately, most NAT routers are not that naive; they are "symmetric", i.e. they store full connection state, and do not allow incoming packets from anyone but the party that was first contacted (in this case, the STUN server). In this scenario, STUN is not enough to allow P2P communication in the presence of NAT.

TURN (Traversal Using Relay NAT) comes to rescue. TURN employs the same protocol as STUN, but the TURN server acts as a relay to which both parties behind NATs can connect. Since the relay server adds latency and needs to be maintained (obviously for a cost), TURN should be used only as the last resort.

ICE (Interactive Connectivity Establishment), a draft specification that is employed by Google Talk, is not a protocol in itself. It is a collection of techniques like STUN and TURN. It finds all the possible ways to establish a P2P connection, and picks the best one.

ICE works by finding all possible P2P connection candidates, and sending this data to the remote peer via a signaling protocol. The signaling protocol is not specified by ICE; hence the mechanism can be used with any signaling protocol. The parties agree on the best way to initiate the connection, analogous to a modem handshake. The signaling server works as a proxy until the clients can start to exchange data directly.

The biggest advantage of STUN, TURN and ICE is that they do not depend on support by the NAT routers. ICE will work even if there are several routers in the network path. The downside is the need of public STUN and TURN (relay) servers. This is not a great problem, due to the fact that every P2P service requires a signaling server anyway. The same entity that provides the signaling will certainly provide the auxiliary STUN/TURN services.

Another minor disadvantage of ICE is that the signaling protocol will have to accommodate the "connection candidate" message, either by an explicit provision in the protocol (e.g. XMPP) or by a hack.

1.4.5 NAT Transversal API in Maemo

On maemo platform, the developer does not need to worry about these protocols. Maemo includes the libjingle library (used by Google Talk) that offers an API for P2P connections.

The best way to learn to use the API is by example. The example P2P client here is very simple: it requests a service at random times, and processes requests from other P2P clients. The service in this example is nothing more than adding two byte values.

As already stated, every P2P server must have a signaling protocol, over which the parties can exchange initial P2P connection parameters. What is needed for this is a very simple server and the signaling protocol. Since this is just an example, the server architecture does not attribute IDs for the clients, and therefore can handle only two simultaneous clients.

The signaling protocol is very simple and has only one message: the connection candidate that one peer sends to the other. The message is simply forwarded to the other peer. Apart from encoding and decoding, these messages are completely handled by *libjingle*, so it is not necessary to understand them in depth.

Once the connection parameters have been exchanged via the signaling server, the P2P connection is opened, and the parties will communicate directly without any further signaling. Albeit very simple, this protocol simulates all the basic steps of every real-world P2P service.

If interested in using XMPP/Google Talk as the signaling service, Libjingle source also contains examples of P2P communication that employ XMPP/-Google Talk accounts and servers.

1.4.6 Example: P2P Client

The following C example contains some C++, because Libjingle is written in C++. The example is a P2P client based on Libjingle APIs. It is the smallest possible demonstration of NAT piercing capability, so it does not handle network errors and overflows very well. A production implementation must improve in these directions.

First, there is some boilerplate code: includes and prototypes.

```
#define POSIX
#define SIGSLOT_USE_POSIX_THREADS

#include <libjingle/talk/base/thread.h>
#include <libjingle/talk/base/network.h>
#include <libjingle/talk/base/socketaddress.h>
#include <libjingle/talk/base/physicalsocketserver.h>
#include <libjingle/talk/p2p/base/sessionmanager.h>
#include <libjingle/talk/p2p/base/helpers.h>
#include <libjingle/talk/p2p/client/basicportallocator.h>
#include <libjingle/talk/p2p/client/socketclient.h>

#include <string>
```

```

#include <vector>

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/select.h>
#include <sys/time.h>
#include <time.h>
#include <signal.h>

void signaling_init(const char* ip, unsigned int port);
int signaling_wait(unsigned int timeout);
void signaling_sendall(const char* buffer, unsigned int len);

SocketClient* socketclient_init(const char *stun_ip, unsigned int
    stun_port,
                                const char *turn_ip, unsigned int
                                turn_port);

char* socketclient_add_remote_candidates(SocketClient *sc, char* buffer
);
void socketclient_add_remote_candidate(SocketClient *sc, const char *
candidate);

bool socketclient_is_writable (SocketClient *sc);

void socketclient_send(SocketClient *sc, const char *data, unsigned int
len);

void randomize();

```

For the sake of simplicity, some data is kept in global variables. A production implementation would probably move that data into objects.

The *p2p_state* shows, whether the P2P connection is up. *signaling_socket* is a TCP socket, allowing data exchange via the signaling server before the P2P connection is up. *main_thread* contains a libjingle *Thread* object; libjingle is itself multithreaded, and employs one thread per P2P connection.

```

bool p2p_state = false;
int signaling_socket = -1;

cricket::Thread *main_thread = 0;

```

This is the main program loop. It sets up the signaling connection and forwards the signaling data to Libjingle until the P2P connection is up. The P2P connection is simply used to send bytes at random intervals. If the P2P connection breaks, the loop returns to signaling phase. The program only stops when killed or when an unexpected error occurs.

N.B. *main_thread->Loop(10)* is called from time to time. In a "real" application, this method would be called on idle time (e.g. via GLib's *g_idle_add()*).

There are three IP addresses hardcoded: signaling server, STUN server (if any) and TURN server (if any). These addresses should be dated for the particular environment in question.

```

int main(int argc, char* argv[])
{
    signal(SIGPIPE, SIG_IGN);

```

```

randomize();

// P2P signaling server
const char* signaling_ip = "200.184.118.140";
int signaling_port = 14141;

// STUN server, NULL if none
const char* stun_ip = "200.184.118.140";
// const char* stun_ip = 0;
unsigned int stun_port = 7000;

// TURN server, NULL if none
const char* turn_ip = 0;
// const char* turn_ip = 0;
unsigned int turn_port = 5000;

signaling_init(signaling_ip, signaling_port);

SocketClient* sc = socketclient_init(stun_ip, stun_port,
    turn_ip, turn_port);

sc->getSocketManager()->StartProcessingCandidates();

while (1) {
    char buffer[10000];
    char *buffer_p = buffer;
    char *buffer_interpreted = buffer;

    while (! p2p_state) {
        main_thread->Loop(10);

        if (! signaling_wait(1)) {
            printf("-- tick --\n");
            continue;
        }

        int n = recv(signaling_socket, buffer_p, sizeof
            (buffer)
            - (buffer_p - buffer), 0);
        if (n < 0) {
            printf("Signaling socket closed with
                error\n");
            exit(1);
        } else if (n == 0) {
            printf("Signaling socket closed\n");
            exit(1);
        }
        buffer_p += n;
        buffer_interpreted =
            socketclient_add_remote_candidates(sc,
                buffer_interpreted);
    }

    // P2P connection is up by now.

    while (p2p_state) {
        // sends a byte via P2P connection
        unsigned char data = random() % 256;
        socketclient_send(sc, (char*) &data, 1);
        sleep(random() % 15 + 1);
        main_thread->Loop(10);
    }
}

```

```

        // P2P connection is broken, restart handling
        // connection candidates
    }
}

```

The next function seeds random, otherwise the two peers may end up with exactly the same bytes and time intervals during P2P data exchange.

```

void randomize()
{
    struct timeval tv;
    struct timezone tz;
    gettimeofday(&tv, &tz);
    srand(tv.tv_usec);
}

```

This function creates the signaling socket - just a boring and ordinary TCP connection to the P2P signaling server.

```

void signaling_init(const char* ip, unsigned int port)
{
    struct sockaddr_in sa;

    signaling_socket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

    bzero(&sa, sizeof(sa));
    sa.sin_family = AF_INET;
    sa.sin_port = htons(port);
    inet_aton(ip, &(sa.sin_addr));

    if (connect(signaling_socket, (struct sockaddr*) &sa, sizeof(sa)) < 0) {
        printf("Error in signaling connect()\n");
        exit(1);
    }
}

```

This function waits for something to happen with the signaling socket, until the specified timeout. It is important that the timeout is not too long, since the P2P connection may have been brought up meanwhile.

```

int signaling_wait(unsigned int timeout) {
    fd_set rfd;
    struct timeval tv;
    int retval;

    FD_ZERO(&rfd);
    FD_SET(signaling_socket, &rfd);

    tv.tv_sec = timeout;
    tv.tv_usec = 0;

    retval = select(signaling_socket+1, &rfd, NULL, NULL, &tv);
    if (retval == -1)
        printf("error in select()");

    return (retval > 0);
}

```

Libjingle is C++ and employs a signal architecture to notify the client application about changes in P2P connection status. These two classes accommodate the methods that will be called back when something happens.

Since XMPP is not used as the signaling protocol, the signaling protocol handling needs to be provided, so all signaling handling is carried out separately from those signals.

```
class SignalListener1 : public sigslot::has_slots<>
{
private:
    SocketClient *sc;

public:
    SignalListener1(SocketClient *psc);
    void OnCandidatesReady(const std::vector<Candidate>& candidates);
    void OnNetworkError();
    void OnSocketState(bool state);
};

class SignalListener2 : public sigslot::has_slots<>
{
private:
    SocketClient *sc;

public:
    SignalListener2(SocketClient *psc);
    void OnSocketRead(P2PSocket *socket, const char *data, size_t len);
};

SignalListener1::SignalListener1(SocketClient* psc)
{
    sc = psc;
}

SignalListener2::SignalListener2(SocketClient* psc)
{
    sc = psc;
}

void SignalListener1::OnNetworkError()
{
    printf("Network error encountered at SocketManager");
    exit(1);
}
```

The first signal callback method. It is called when the P2P socket changes state. The *p2p_state* global variable will be updated with the reported state, and this will drive the main loop behavior.

```
void SignalListener1::OnSocketState(bool state)
{
    printf("Socket state changed to %d\n", state);
    p2p_state = state;
    if (state) {
        printf("Writable from %s:%d to %s:%d\n",
            sc->getSocket()->best_connection()->local_candidate().address().
                IPAsString().c_str(),
            sc->getSocket()->best_connection()->local_candidate().address().
                port(),
```

```

        sc->getSocket()->best_connection()->remote_candidate().address().
            IPAsString().c_str(),
        sc->getSocket()->best_connection()->remote_candidate().address().
            port());
    }
}

```

This function packages all P2P socket creation bureaucracy. It creates the socket object, the socket listeners (whose classes have been defined above), and connects the signal callbacks.

```

SocketClient* socketclient_init(const char *stun_ip, unsigned int
    stun_port,
                                const char *turn_ip, unsigned int
                                turn_port)
{
    cricket::SocketAddress *stun_addr = NULL;
    if (stun_ip) {
        stun_addr = new cricket::SocketAddress(std::string(
            stun_ip), stun_port);
    }

    cricket::SocketAddress *turn_addr = NULL;
    if (turn_ip) {
        turn_addr = new cricket::SocketAddress(std::string(
            turn_ip), turn_port);
    }

    cricket::PhysicalSocketServer *ss = new PhysicalSocketServer();
    main_thread = new Thread(ss);
    cricket::ThreadManager::SetCurrent(main_thread);

    SocketClient *sc = new SocketClient (stun_addr, turn_addr);

    // Note that signal connections pass the SignalListener1 object
    as well as the
    // method. Since a new SocketListener1 is created for every new
    SocketClient,
    // we have the guarantee that each SocketListener1 will be
    called back only
    // in behalf of its related SocketClient.

    sc->sigl1 = new SignalListener1(sc);
    sc->sigl2 = new SignalListener2(sc);
    sc->getSocketManager()->SignalNetworkError.connect(sc->sigl1,
        &SignalListener1::OnNetworkError);
    sc->getSocketManager()->SignalState_s.connect(sc->sigl1,
        &SignalListener1::OnSocketState);
    sc->getSocketManager()->SignalCandidatesReady.connect(sc->sigl1
        ,
        &SignalListener1::OnCandidatesReady);
    sc->CreateSocket(std::string("foobar"));
    sc->getSocket()->SignalReadPacket.connect(sc->sigl2,
        &SignalListener2::
            OnSocketRead);

    return sc;
}

```

The method below is called back when LibJingle has some local candidates for connection that should be sent to the remote site via the signaling protocol.

The beauty of ICE protocol is that both parties will be able to agree on a P2P connection, without having to exchange request and response messages. They just send connection candidates to each other. Each side selects the best way to send data, based on the received candidates. With both sides able to send data directly to the remote party, a bi-directional P2P channel is enabled.

```
void Signallistener1::OnCandidatesReady(const std::vector<Candidate>&
candidates)
{
    printf("OnCandidatesReady called with %d candidates in list\n",
        candidates.size());

    for(std::vector<Candidate>::const_iterator it = candidates.
        begin();
        it != candidates.end(); ++it) {
        char *marshaled_candidate;

        asprintf(&marshaled_candidate,
            "%s %d %s %f %s %s %s\n",
            (*it).address().IPAsString().c_str(),
            (*it).address().port(),
            (*it).protocol().c_str(),
            (*it).preference(),
            (*it).type().c_str(),
            (*it).username().c_str(),
            (*it).password().c_str() );

        printf("Candidate being sent: %s", marshaled_candidate)
            ;

        signaling_sendall(marshaled_candidate, strlen(
            marshaled_candidate));

        free(marshaled_candidate);
    }
}
```

An auxiliary function to send a data buffer through the signaling TCP channel. It does not return until all data has been sent.

```
void signaling_sendall(const char* buffer, unsigned int len)
{
    unsigned int sent = 0;
    while (sent < len) {
        int just_sent;
        just_sent = send(signaling_socket, buffer+sent, len-
            sent, 0);
        if (just_sent < 0) {
            printf("Signaling socket closed with error.\n")
                ;
            exit(1);
        } else if (just_sent == 0) {
            printf("Signaling socket closed.\n");
            exit(1);
        }
        sent += just_sent;
    }
}
```

This function is called by the main loop, when some signaling data arrives.

It will find out if there is a complete P2P connection candidate in the buffer. If there is one, it is decoded.

```
// extracts remote candidates from a buffer, returns a pointer to the  
rest of the buffer  
  
char* socketclient_add_remote_candidates(SocketClient *sc, char* buffer  
    )  
{  
    char *n;  
    char candidate[1024];  
  
    while (1) {  
        n = strchr(buffer, '\n');  
        if (! n) {  
            return buffer;  
        }  
        strncpy(candidate, buffer, n-buffer+1);  
        socketclient_add_remote_candidate(sc, candidate);  
        buffer = n+1;  
    }  
}
```

Here, the P2P connection candidate is decoded and made known to LibJingle. Since LibJingle has its own thread and sockets, all further processing of P2P candidates is fortunately outside the scope of our code.

```
// Inform candidates received from the signaling network to LibJingle  
  
void socketclient_add_remote_candidate(SocketClient *sc, const char*  
    remote_candidate)  
{  
    std::vector<Candidate> candidates;  
  
    char ip[100];  
    unsigned int port;  
    char protocol[100];  
    float preference;  
    char type[100];  
    char username[100];  
    char password[100];  
  
    // WARNING: using fixed-size buffers and sscanf is utterly  
unsafe.  
    // Real implementations must be more robust about data coming  
from the network!  
  
    sscanf(remote_candidate,  
        "%s %d %s %f %s %s %s\n",  
        ip,  
        &port,  
        protocol,  
        &preference,  
        type,  
        username,  
        password);  
  
    printf("Received new candidate: %s:%d pref %f\n", ip, port,  
        preference);  
  
    Candidate candidate;  
    candidate.set_name("rtp");
```

```

        candidate.set_address(SocketAddress(std::string(ip), port));
        candidate.set_username(std::string(username));
        candidate.set_password(std::string(password));
        candidate.set_preference(preference);
        candidate.set_protocol(protocol);
        candidate.set_type(type);
        candidate.set_generation(0);

        candidates.push_back(candidate);

        sc->getSocketManager()->AddRemoteCandidates(candidates);
    }

```

Simple helper function, showing whether a P2P socket is writable (which means that the P2P connection is up).

```

bool socketclient_is_writable(SocketClient *sc)
{
    return sc->getSocketManager()->writable();
}

```

Method that is called back when data arrives from the P2P connection. In this example, it is just a byte of data.

```

void SignalListener2::OnSocketRead(P2PSocket *socket, const char *data,
    size_t len)
{
    printf("Received byte %d from remote P2P\n", data[0]);
}

```

Auxiliary function that sends data via P2P connection. Not really difficult.

```

void socketclient_send(SocketClient* sc, const char *data, unsigned int
    len)
{
    sc->getSocket()->Send(data, len);
    printf("Sent byte %d to remote P2P\n", data[0]);
}

```

In order to compile this, you need jinglebase-0.3 and jinglep2p-0.3 packages:

```

gcc client.cpp -o client `pkg-config --cflags --libs jinglebase-0.3 jinglep2p-0.3`

```

1.4.7 Signaling Server

As already mentioned, this signaling server is incredibly simple, and works more like a network pipe, forwarding data from one side to another. The P2P parties agree on a P2P connection through this channel.

```

from select import select
import socket
import time

read_socks = []
port = 14141

server_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_sock.bind("", port)
server_sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server_sock.listen(5)

read_socks.append(server_sock)

# We accept two parties only
buffer = ""

while True:
    rd, wr_dummy, ex_dummy = select(read_socks, [], [], 10)

    if not rd:
        print "-- tick --"
        continue

    rd = rd[0]

    if rd is server_sock:
        # incoming new connection
        newsock, address = rd.accept()
        print "New connection from %s" % str(address)
        if len(read_socks) > 3:
            # we only accept two parties at the most
            newsock.close()
            continue
        read_socks.append(newsock)
        if buffer:
            # we already have data to be sent to the new party
            newsock.sendall(buffer)
            print "    sent buffered data"
            buffer = ""
        continue

    data = rd.recv(999999)

    if not data:
        # socket closed, remove from list
        print "Connection closed"
        del read_socks[read_socks.index(rd)]
        buffer = ""
        continue

    if len(read_socks) < 3:
        print "Buffering data"
        # the other party has not connected; bufferize
        buffer += data
        continue

    print "Forwarding data"
    for wr in read_socks:
        if wr is not rd and wr is not server_sock:
            wr.sendall(data)

```

1.4.8 STUN and Relay Servers

The maemo libjingle-utils package includes both a STUN server and a relay server for testing purposes. To run a test server, do the following:

```

$ stunserv &
$ relayserver &

```

The relay server will print console messages, when a P2P connection is

flowing through it. If more detailed feedback is needed (e.g. when debugging a P2P application), a network sniffing tool like tcpdump or Ethereal should be used to monitor UDP packets.

Bibliography

- [1] BlueZ projects home page. <http://www.bluez.org/>.
- [2] Maemo API reference. http://maemo.org/api_refs/.
- [3] OpenOBEX home page. <http://dev.zuckschwerdt.org/openobex/>.
- [4] Telepathy. <http://telepathy.freedesktop.org/>.