P2 Skype Demo: How To Interact With Skype

Martin Hamrle, Tomáš Klačko, Tomáš Plch, Ondřej Šerý, Petr Tůma[+]

Technical Report No. 12/2006
October 2006

Distributed Systems Research Group, Department of Software Engineering
Faculty of Mathematics and Physics, Charles University
Malostranské náměstí 25, 118 00 Praha 1 Malá Strana, Czech Republic

[+]tuma@nenya.ms.mff.cuni.cz

Keywords

Component middleware, IP telephony, voice recognition, peer-to-peer, Skype.

Abstract

This report is part of a research project aimed at investigating the perspective of using autonomic computing principles in a peer-to-peer environment to drive interactive voice applications. It describes a proof-of-concept demo implemented to verify the technical usability of various technologies to be used in future applications.

The demo implements a simple answering machine using Skype as the underlying voice transport medium. The user calls the answering machine Skype ID and leaves a message that is played back immediately. The technical challenges faced by the demo include the ability to place multiple calls on the answering machine Skype ID simultaneously, which requires running and controlling multiple Skype instances with the same Skype ID at the same time and on the same machine.

Table of Contents

1 Introduction....3
2 Design Constraints....3
    2.1 Skype....3
    2.2 DBus....3
3 Demo Architecture....4
    3.1 Call Handling....4
    3.2 The Session Controller Architecture....5
    3.3 The Session Manager Architecture....6
    3.4 The Virtual Audio Device Architecture....6
    3.5 Demo Startup....7
4 Technical Details....7
    4.1 Skype Setup....7
        4.1.1 Automatic Login....7
        4.1.2 Home Directory....7
        4.1.3 Audio Device....7
        4.1.4 Automatic Control 7
    4.2 DBus Setup....8
    4.3 Virtual Audio Device ioctl Commands....8
    4.4 Demo Startup....8
    4.5 Controlling Skype Through DBus....9
        4.5.1 Registering With DBus....9
        4.5.2 Register With Skype Through DBus....9
5 Future Work and Demo Applicability....12
6 Acknowledgements....13
7 References....14

1. Introduction

This report is part of a research project aimed at investigating the perspective of using autonomic computing principles in a peer-to-peer environment to drive interactive voice applications. For more details of the project, see [11].

The report describes a proof-of-concept demo implemented to verify the technical usability of various technologies to be used in future applications. The demo implements a simple answering machine using Skype [1] as the underlying voice transport medium. The user calls the answering machine Skype ID and leaves a message that is played back immediately. The technical challenges faced by the demo include the ability to place multiple calls on the answering machine Skype ID simultaneously. This requires running and controlling multiple Skype instances with the same Skype ID at the same time and on the same machine. The implementation has been tested in a Linux environment, with Linux kernel 2.6, DBus 0.23.4 and statically linked Skype 1.2.0.18.

The rest of the report outlines the architecture and implementation of the demo, focusing only on the technically interesting details. Note that the architecture is strongly influenced by the behavior exhibited by Skype. The information on the relevant behavior of Skype is therefore also covered in the text.

The report is organized as follows. Chapter 2 describes the design constraints that the demo is built around. Chapter 3 gives an overview of the architecture, describing the components and their interaction. Chapter 4 covers some important and interesting technical details related to Chapter 3.

2. Design Constraints

2.1. Skype

Although undocumented, it appears possible to run multiple Skype instances with the same Skype ID. When a call is placed on this Skype ID, all the Skype instances start ringing. The bad news is that on very few occassions, we have experienced this behaviour to be nondeterministic. In spite of the nondeterminism, we have decided to rely on this behavior in the demo (it appears to work well enough under normal circumstances).

When running multiple Skype instances, each instance needs an audio device to read audio data from and write audio data to. Obviously, the demo also needs to intercept the Skype audio data stream so that it can record and replay the voice message. To achieve both, we have implemented a Virtual Audio Device in the kernel (note that similar technologies that run in user space also exist [8]). Each Skype instance is given an instance of this device, which is used to intercept the Skype audio data stream. The implementation of the Virtual Audio Device is hindered by what appears to be an unusual audio device usage pattern exhibited by Skype, which for example can open the same audio device multiple times. The Virtual Audio Device is covered in detail in Section 3.4.

2.2. DBus

An application that needs to control Skype under Linux can use the Skype API accessible through DBus [4]. DBus is a middleware that transports messages between Skype and the controlling application.

At runtime, DBus has a form of a process (a DBus message bus) that both the controlling application and Skype are connected to. A major restriction (as of DBus 0.23.4, which is statically linked into Skype 1.2.0.18) is that a process cannot be connected to more than one message bus of a given type (there are two bus types, system message bus and session message bus, the demo uses the session message bus).

This particular restriction comes into play when multiple Skype instances are to be controlled. For each controlled Skype instance, a separate DBus session message bus process must be run. For each instance pair of Skype and DBus, a Session Controller process that connects to the Skype through DBus is run. All the Session Controllers are managed by a single Session Manager process, responsible for coordinating the individual instances of Skype through the associated instances of Session Controller. Most notably, the Session Controllers ask the Session Manager which Skype instance is to pick up each incoming call.

3. Demo Architecture

The architecture of the demo consists of five components, namely Skype, DBus, Session Controller, Session Manager and Virtual Audio Device. A Session Controller instance is associated through an instance of DBus with an instance of Skype. All the Session Controller instances are connected to a single Session Manager instance, which selects the Skype instance that picks up each incoming call. Each instance of Skype is also connected to an instance of the Virtual Audio Device. The Session Controller uses the Virtual Audio Device to pass audio data to and from Skype.

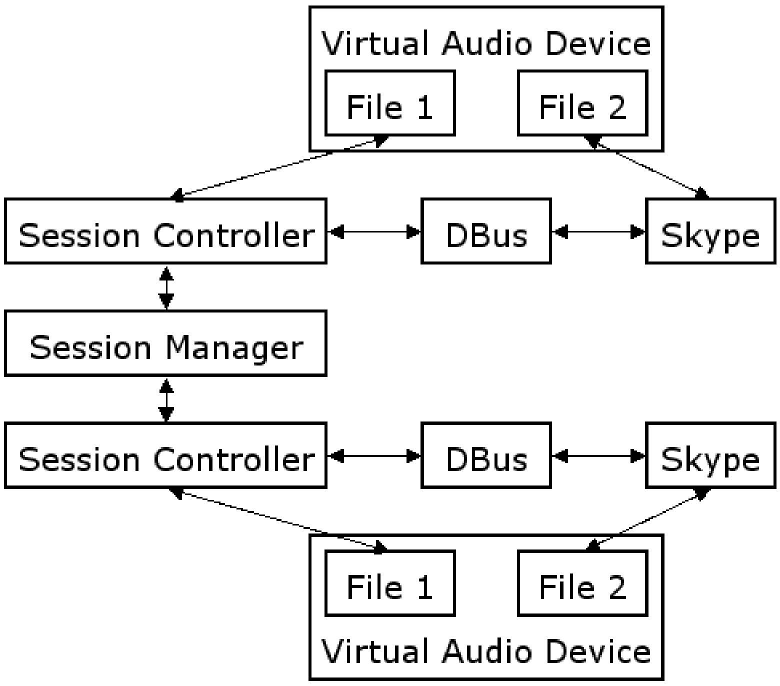

Figure 1: Demo Architecture

The components of the demo run in multiple processes, specifically the Skype processes, the Session Controller processes, the DBus processes and the Session Manager process. Each Virtual Audio Device instance is available as a pair of two special files in the operating system.

Figure 1 shows the Session Manager talking to two Skype instances through two Session Controller instances (boxes are component instances, arrows denote communication paths).

3.1. Call Handling

This chapter describes what happens when a user calls the answering machine demo. The chapter uses an example setup consisting of one Session Manager process (SessionManager), two Session Controller processes (SessionControllerA and SessionControllerB), two Skype processes (SkypeA, SkypeB) and two DBus processes (DBusA, DBusB). SkypeU denotes the instance of Skype used by the user to place the call. The explanation skips the interaction with the Virtual Audio Device.

The sequence of actions is as follows:

1. SkypeU calls.

2. SkypeA and SkypeB start ringing.

3. SkypeA notifies SessionControllerA through DBusA about the incoming call.
   SkypeB notifies SessionControllerB through DBusB about the incoming call.

4. SessionControllerA and SessionControllerB ask SessionManager
   for a permission to pick up the incoming call.

5. SessionManager chooses SessionControllerA.
   SessionManager tells SessionControllerA to pick up the call.
   SessionManager tells SessionControllerB to not pick up the call.

6. SessionControllerA tells SkypeA through DBusA to pick up the call.
   SessionControllerB does nothing.

7. SkypeA picks up the call.
   SkypeB receives information from the Skype network that the call is finished
   and sends a notification about this through DBusB to SessionControllerB.

8. When the call is finished, SkypeA sends notification to SessionControllerA through DBusA.
   SessionControllerA sends the notification to SessionManager.

The fact that SessionControllerB does nothing in step 6 is very important. If SessionControllerB were to tell SkypeB to reject the incoming call, the call initiated by SkypeU would be terminated. In effect, this would end the call that is, by that time, being negotiated between SkypeU and SkypeA.

3.2. Session Controller Architecture

The Session Controller is a single-threaded process. It registers itself with the other components, installs the notification handler and then processes Skype notifications in an infinite cycle.

The Session Controller performs the following steps:

1. Register with DBus
2. Install notify handler
3. Register with Skype
4. Register with Session Manager
5. Handle Skype notification messages

Steps one and two are performed using the DBus API. Step three is performed by sending Skype

API messages through DBus to the Skype instance. Step four establishes a socket connection to the Session Manager.

The Session Controller only handles Skype notifications that inform about incoming calls. After the initialization is performed, the controller process is either in a free state or in a busy state. The busy state means that the associated Skype instance is handling a call. The free state means that the associated Skype instance is waiting for a call. The Session Controller starts in the free state.

Here is what happens when a Skype user calls the answering machine, with the Session Controller using Skype API 1.2 [2]:

1. The answering machine Skype instance sends a "CALL X STATUS RINGING" notification message to the Session Controller. The number X uniquely identifies the incoming call for that particular Skype instance. If the controller is in the busy state, it does nothing. If the controller is in the free state, it proceeds with the next step.

2. The controller sends a "GET CALL X PARTNER_HANDLE" to Skype to retrieve the Skype ID of the user that is calling. Skype replies with a "CALL X PARTNER_HANDLE H" message, where H is the name of the Skype user that calls the answering machine.

3. The controller sends an "ANSWER_CALL H" query to the manager. If the manager replies with "YES", the controller enters the busy state and proceeds with the next step. If the manager replies with "NO", the controller does nothing.

4. The controller sends "SET CALL X STATUS INPROGRESS" to Skype. Skype replies with a notification message "CALL X STATUS INPROGRESS".

5. When the call is finished, Skype sends a "CALL X STATUS FINISHED" message. The controller then sends a "FINISHED_CALL" notification to the manager, the manager replies with "OK". The controller returns to the free state.

3.3. Session Manager Architecture

The SessionManager keeps track of which Skype instance is handling a call with what Skype ID. It makes sure that only one Skype instance communicates with the same Skype ID at the same time by keeping a table of <SessionController ID, Skype ID> pairs. The SessionController ID uniquely identifies the Session Controller connected to the Session Manager. The Session Manager assigns the SessionController ID when the Session Controller registers with it.

At runtime, the Session Manager uses two threads. The first thread serves requests for a new connection to the socket at which the Session Controllers register, the second thread serves the already connected Session Controllers.

3.4. Virtual Audio Device Architecture

Every Virtual Audio Device instance consists of two special files, denoted here as file A and file B. The logic behind the device is that what you write into file A, you can read from file B and vice versa.

Eight such file pairs are implemented by the Virtual Audio Device in a single Linux kernel module. Each file pair is internally implemented as two FIFO buffers, with the read and write commands issued on the special files internally implemented as read and write operations on the corresponding FIFO buffer.

The audio device in Linux is usually accessed as a special file /dev/dsp. The Open Sound System (OSS) specification [7] states that an audio device can not be owned by more than a one process at a time and discourages (but permits) a single process opening the audio device for reading and writing multiple times. The Virtual Audio Device implementation expects multiple readers and multiple writers because Skype opens the audio device up to twice. This makes the implementation more difficult because a number of synchronization issues arises. To solve those, exclusive locks are used to guard opening and closing of a device as well as reading and writing of the FIFO buffers. (Note that this can lead to a problem when flushing the FIFO buffers, as all three exclusive locks need to be acquired, yielding a possibility of a deadlock.)

The Virtual Audio Device implements several ioctl commands. These only exist to make Skype believe that it is using a real audio device, and have no function otherwise. For a list of the supported ioctl commands, see Section 4.3.

The audio data format that Skype by default uses with a real audio device is 16 bit signed little endian, 48000 Hz, mono. For simplicity, the Virtual Audio Device acknowledges this format and rejects any other. The record and playback speed depends on the audio data format. It turned out to be necessary to control the playback speed in the Virtual Audio Device as Skype relies on the audio device timing.

3.5. Demo Startup

This chapter shows the steps taken during the demo startup. In the example, we consider two Skype instances. The architecture thus consists of two triples of a Skype process, a Session Controller process and a DBus process, and of one Session Manager process. The two triplets, as well as the processes that make up the triplets, are here denoted as A and B.

1. Start the Session Manager.
2. Start the DBus session message bus A.
3. Start the Skype instance A. Let it connect to bus A.
4. Start the Session Controller A. Let it connect to bus A and to the Session Manager.
5. Start the DBus session message bus B.
6. Start the Skype instance B. Let it connect to bus B.
7. Start the Session Controller B. Let it connect to bus B and to the Session Manager.

Care must be taken not to run the Session Controller too early. When the Session Controller is run while Skype is still initializing, all the processes will appear to run correctly but Skype will not respond to the commands of the Session Controller.

A set of shell commands that perform these steps is in Section 4.4.

4. Technical Details

4.1. Skype Setup

In order to run multiple Skype instances on the same machine, few additional details need to be solved. Skype uses a configuration file $HOME/.Skype/<login>/config.xml. Some things can be configured by editing this file.

4.1.1 Automatic Login

The demo requires that Skype does not ask for user login at startup and logs into the Skype network automatically instead. This can be configured by checking a checkbox in the Skype login dialog.

4.1.2 Home Directory

Each Skype instance must have its own home directory. The home directory for a particular Skype instance can be set by setting the HOME environment variable.

4.1.3 Audio Device

The audio device that Skype is to use can be set from the configuration file using the SoundDevice node (for example, <SoundDevice>0</SoundDevice> tells Skype to use the /dev/dsp0 audio device).

4.1.4 Automatic Control

When an application tries to control Skype, the Skype instance asks the user for confirmation. This can be avoided as follows. When the Session Controller identifies itself to Skype using an identifier X in the initial "NAME X" message, setting the Authorizations node to X will prevent Skype from asking for user confirmation (for example, <Authorizations>X</Authorizations>).

4.2. DBus Setup

The communication between the application process and the DBus message bus process is done using the BSD Unix sockets. The address of the DBus session message bus socket is stored in the environment variable $DBUS_SESSION_BUS_ADDRESS.

An application process connects to the DBus message bus process using the DBus library function dbus_connect(). This function retrieves the $DBUS_SESSION_BUS_ADDRESS value from the environment and then connects to the session message bus socket.

The command dbus-launch starts up a new session message bus and prints the value of $DBUS_SESSION_BUS_ADDRESS to the standard output. The export command is then executed to export this value so that future processes run from this shell will have the variable set in their environment.

4.3. Virtual Audio Device ioctl Commands

The virtual audio device implements the following ioctl system call commands:

- SNDCTL_DSP_GETBLKSIZE
- SNDCTL_DSP_GETISPACE
- SNDCTL_DSP_GETOSPACE
- SNDCTL_DSP_GETCAPS
- SNDCTL_DSP_SETFRAGMENT
- SNDCTL_DSP_STEREO
- SNDCTL_DSP_SETFMT

- SNDCTL_DSP_SPEED
- SNDCTL_DSP_CHANNELS

The ioctl commands have no special meaning to the virtual audio device and in fact do nothing. They are only implemented so that the audio device initialization performed by Skype is not interrupted by an ioctl error.

4.4. Demo Startup

The following commands launch the demo.

```
> session_manager --ip=127.0.0.1 --port=7007 &
> dbus-launch
DBUS_SESSION_BUS_ADDRESS='unix:abstract=/tmp/dbus-XXXX'
DBUS_SESSION_BUS_PID=1234
> export DBUS_SESSION_BUS_ADDRESS='unix:abstract=/tmp/dbus-XXXX'
> Skype --use-session-dbus &
> session_controller --ip=127.0.0.1 --port=7007 &
> dbus-launch
DBUS_SESSION_BUS_ADDRESS='unix:abstract=/tmp/dbus-YYYY'
DBUS_SESSION_BUS_PID=5678
> export DBUS_SESSION_BUS_ADDRESS='unix:abstract=/tmp/dbus-YYYY'
> Skype --use-session-dbus &
> session_controller --ip=127.0.0.1 --port=7007 &
```

First, the Session Manager is started and directed to listen for Session Controllers on IP address 127.0.0.1 and port 7007. Next, a DBus session message bus is started and prints its address to the standard output. The address is exported and becomes part of the shell environment. After that, Skype is started and told to connect to the previously started DBus session message bus (without the command line options, it would try to connect to the system message bus). Finally, the Session Controller is launched. It connects to the Session Manager on IP address 127.0.0.1 and port 7007. Another triplet of Skype, DBus and Session Controller is started in the same way afterwards.

4.5. Controlling Skype Through DBus

This chapter contains an example C source code that shows how to control Skype through DBus. The code illustrates how to connect to the DBus session message bus, how to register with Skype, and finally how to process Skype notifications. In order to understand the source code, familiarity with DBus is required.

DBus assigns specific meaning to the terms object, object path, interface and message.

From the Skype developer pages, the name of the Skype DBus service is com.Skype.API. The client-to-Skype object path is /com/Skype. The Skype-to-client object path is /com/Skype/Client. The Invoke method with one string parameter is used for client-to-Skype commands (the method has one additional argument where Skype stores its response). The Notify method is used for Skype-to-client commands and responses.

4.5.1 Registering With DBus

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

int main(int argc, char **argv)
```

```
{
  GMainLoop       *loop;
  DBusConnection  *bus;
  DBusError       error;

  // Create a new event loop to run in
  loop=g_main_loop_new (NULL, FALSE);

  // Get a connection to the session bus
  dbus_error_init (&error);
  bus=dbus_bus_get (DBUS_BUS_SESSION, &error);
  if (!bus) {
    g_warning ("Failed to connect to D-BUS: %s", error.message);
    dbus_error_free (&error);
    return 1;
  }

  // Set up this connection to work in a GLib event loop
  dbus_connection_setup_with_g_main (bus, NULL);

  // Start the event loop
  g_main_loop_run (loop);

  return 0;
}
```

The demo uses this approach. The use of the GLib event loop is the main reason why the demo depends on GLib. Also, note that the address of the DBus session bus is not passed explicitly, only the type of the message bus is specified.

4.5.2 Register With Skype Through DBus

First, the Session Controller connects to DBus and installs a notify handler with a "/com/Skype/Client" object path. Next, the Session Controller sends "NAME X" message to Skype. Skype replies with "OK". Last, the Session Controller sends "PROTOCOL 1" message to Skype. Skype replies with "PROTOCOL 1". Now, the Skype instance knows about the Session Controller and it will send notifications to it. The notifications are sent to the "/com/Skype/Client" object path to the Notify method.

```
#include <glib.h>
#include <dbus/dbus-glib.h>
#include <assert.h>
#include <stdio.h>

static void invoke (DBusConnection *bus,
                    const char *cmd,
                    const char *check);

static DBusMessage* create_message (void);

static void set_arg (DBusMessage *msg, const char *arg);

static void install_notify_handler (
  DBusConnection *bus, DBusObjectPathVTable *vtable);

static DBusHandlerResult notify_handler (
  DBusConnection *bus, DBusMessage *msg, void *user_data);


int main (int argc, char **argv)
{
  GMainLoop       *loop;
  DBusConnection  *bus;
  DBusError       error;
  DBusObjectPathVTable  vtable;

  // Create a new event loop to run in.
  loop=g_main_loop_new(NULL, FALSE);

  // Get a connection to the session bus.
  dbus_error_init(&error);
  bus=dbus_bus_get(DBUS_BUS_SESSION, &error);
  if (!bus) {
    g_warning ("Failed to connect to D-BUS: %s", error.message);
    dbus_error_free(&error);
    return 1;
  }

  // Set up this connection to work in a GLib event loop.
  dbus_connection_setup_with_g_main(bus, NULL);

  // Install notify handler to process Skype's notifications.
  // The Skype-to-client method call.
  install_notify_handler(bus, &vtable, notify_handler);

  // Register with Skype. Use the Skype API 1.2 protocol.
  invoke(bus, "NAME abc", "OK");
  invoke(bus, "PROTOCOL 1", "PROTOCOL 1");

  // Start the event loop.
  g_main_loop_run(loop);

  return 0;
}


static void invoke (DBusConnection *bus,
                    const char *cmd, const char *check)
{
  DBusMessage  *send;
  DBusMessage  *reply;
  DbusError    error;
  const char   *ack;

  // Create a message to be sent to Skype
  send=create_message();

  // Set the argument of the Invoke method
  set_arg(send, cmd);

  // Send the message and wait for reply
  reply=dbus_connection_send_with_reply_and_block (bus, send, TIMEOUT, &error);
  assert(reply);

  // Get Skype's reply string
  dbus_message_get_args (reply, 0, DBUS_TYPE_STRING, &ack, DBUS_TYPE_INVALID);

  if (check) {
    // Check that Skype's reply is that what expected to be
    if(strcmp(check, ack)!=0) assert(0);
  }
  else {
    // Check for error from Skype itself
    if(strncmp("ERROR", ack, 5)==0) assert(0);
  }
  // Show no interest the previously created messages.
  // DBus will delete a message if reference count drops to zero.
  dbus_message_unref(send);
  dbus_message_unref(reply);
}


static DBusMessage* create_message (void)
{
  DBusMessage *msg=0;

  // Constructs a new message to invoke a method on a remote object.
  // Sets the service the message should be sent to "com.Skype.API"
  // Sets the object path the message should be sent to "/com/Skype"
  // Sets the interface to invoke method on to "com.Skype.API"
  // Sets the method to invoke to "Invoke"
  msg=dbus_message_new_method_call(
    "com.Skype.API",
    "/com/Skype",
    "com.Skype.API",
    "Invoke");
  assert (msg);

  dbus_message_set_sender (msg, "com.Skype.Client");

  return msg;
}


static void set_arg (DBusMessage *msg, const char *arg)
{
  // Sets arg to be an argument to be passed to the Invoke method.
  // It is an input argument. It has a type string.
  // There are no output arguments.
  dbus_message_append_args (
    msg,
    DBUS_TYPE_STRING,
    arg,
    DBUS_TYPE_INVALID,
    DBUS_TYPE_INVALID);
}


static void install_notify_handler (
  DBusConnection *bus, DBusObjectPathVTable *vtable)
{
  dbus_bool_t check;

  vtable->vtb.message_function=notify_handler;

  // We will process messages with the object path "/com/Skype/Client".
  check=dbus_connection_register_object_path (
    bus,
    "/com/Skype/Client",
    vtable,
    0);
  assert(check);
}


static DBusHandlerResult notify_handler (
  DBusConnection *bus, DBusMessage *msg, void *user_data)
{
  char *notify_argument=0;

  // Stores the argument passed to the Notify method
  // into notify_argument.
  dbus_message_get_args (
    msg,
    0,
    DBUS_TYPE_STRING,
    &notify_argument,
    DBUS_TYPE_INVALID);

  printf("Notify: %s\n", notify_argument);

  // Here notify_argument would be parsed.
}
```

5. Future Work and Demo Applicability

Future work of technical nature includes developing a client for automated testing of the demo, as well as modifications to the code that would remove the need for the GLib library and the VAD kernel module.

This work is a part of a project aimed at developing peer-to-peer network overlay for semi-automatic distribution and load-balancing of applications, which is our main goal for the future research. The basic idea is in developing framework that would allow distribution of applications that would require only minor changes in their actual code. The peer-to-peer framework would monitor compatibility attributes (operating system, versions of installed software, etc.) and load of connected nodes and provide search of suitable nodes for expansion of the distributed applications.

This demo constitutes a main case study for application of the peer-to-peer framework. Being developed as a local application but properly divided into functional parts, it is a perfect adept for semi-automatic distribution. With minor changes, the Session Controllers (with VAD, DBus and Skype components) can be distributed over the network using the peer-to-peer framework for suitable node allocation.

We have also other ideas about application of the technology in the demo; however, since our main goal is the peer-to-peer framework, we are probably not going to realize them in a near future due to the lack of man power. Among the applications is a bridge between the Skype network and the GoogleTalk network. The bridge would allow forwarding calls between users in the two networks, as outlined on Figure 2.

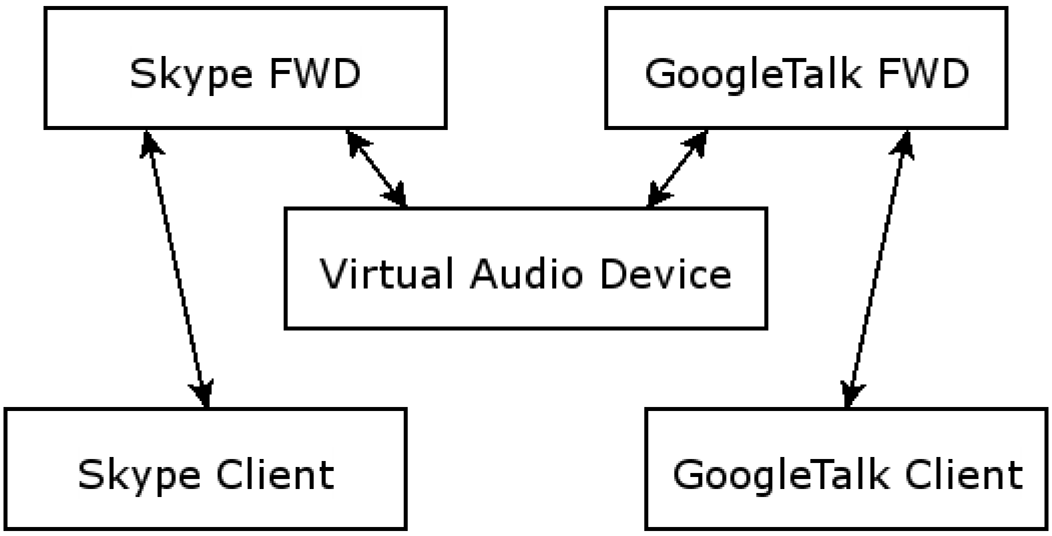

Figure 2: Call Forwarding Application

Another use of the technology used in the demo is securing Skype. Skype is a closed source application that gives rise to numerous security issues [10]. It is possible to use the Virtual Audio Device mechanism to isolate the local network from Skype while still using Skype, simply by positioning Skype outside the local network and forwarding the audio data into the local network using a secure channel, as outlined in Figure 3.

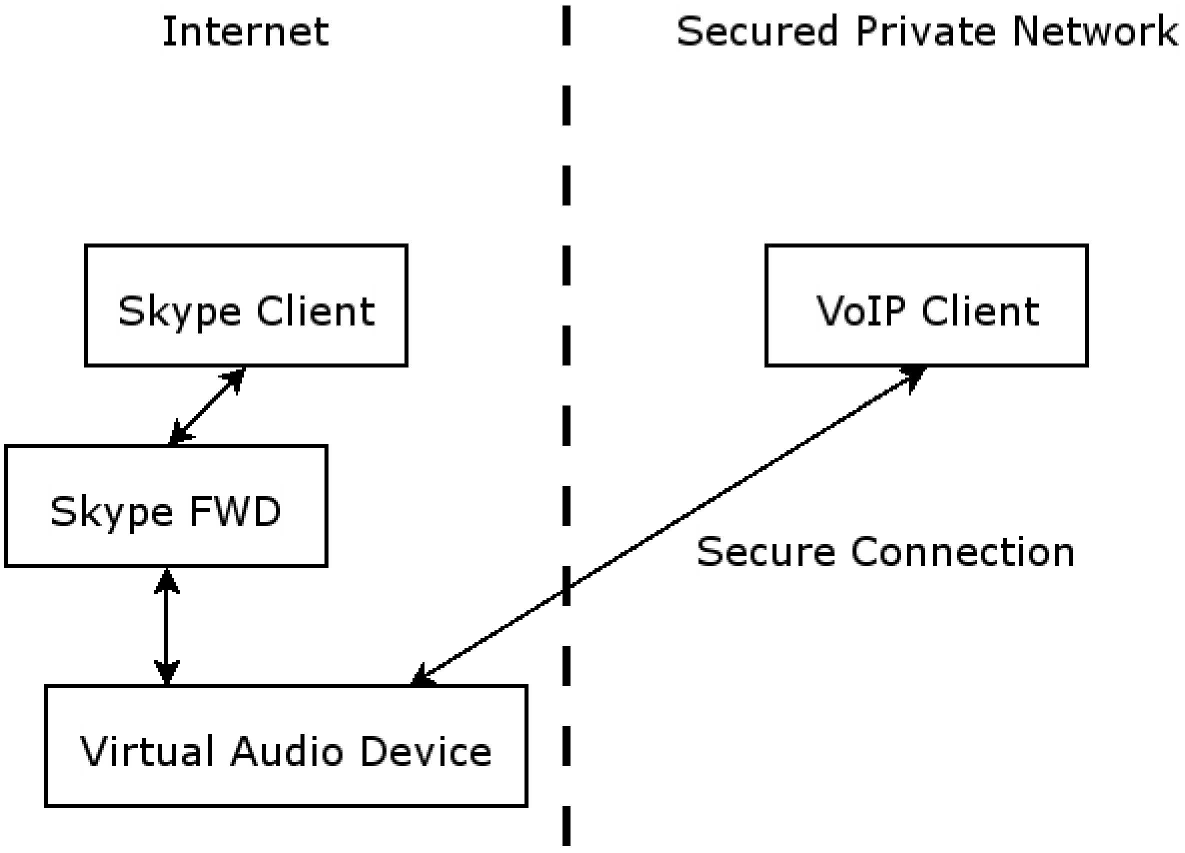

Figure 3: Network Security Application

6. Acknowledgements

This project is partially sponsored under the IBM Faculty Awards program.

7. References

[1] Skype Home Page, http://www.skype.com
[2] Skype API 1.2, http://share.skype.com/media/1.2_api_doc_en.pdf
[3] An introduction to DBus. http://www-128.ibm.com/developerworks/linux/library/l-dbus.html
[4] DBus Home Page, http://dbus.freedesktop.org/wiki
[5] DBus Specification, http://dbus.freedesktop.org/doc/dbus-specification.html
[6] OSS Home Page, http://www.opensound.com
[7] OSS API, http://www.opensound.com/pguide/oss.pdf
[8] User Space Skype Audio Stream Sniffer, http://sourceforge.net/projects/skype-rec
[9] How to Write a Linux Kernel Module (And More), http://lwn.net/Kernel/LDD3
[10] Silver Needle in the Skype, http://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-biondi/bh-eu-06-biondi-up.pdf
[11] P2 Peer-To-Peer Component Middleware Home Page, http://nenya.ms.mff.cuni.cz/p2