Decorating Asterisk: Experiments in Service Creation for a Multi-Protocol Telephony Environment Using Open Source Tools

Submitted in fulfilment
of the requirements of the degree
Masters in Computer Science
of Rhodes University

Jonathan Hitchcock

March, 2006

Abstract

As Voice over IP becomes more prevalent, value-adds to the service will become ubiquitous. Voice over IP (VoIP) is no longer a single service application, but an array of marketable services of increasing depth, which are moving into the non-desktop market. In addition, as the range of devices being generally used increases, it will become necessary for all services, including VoIP services, to be accessible from multiple platforms and through varied interfaces. With the recent introduction and growth of the open source software PBX system named Asterisk, the possibility of achieving these goals has become more concrete. In addition to Asterisk, a number of open source systems are being developed which facilitate the development of systems that interoperate over a wide variety of platforms and through multiple interfaces. This thesis investigates Asterisk in terms of its viability to provide the depth of services that will be required in a VoIP environment, as well as a number of other open source systems in terms of what they can offer such a system. In addition, it investigates whether these services can be made available on different devices. Using various systems built as a proof-of-concept, this thesis shows that Asterisk, in conjunction with various other open source projects, such as the Twisted framework provides a concrete tool which can be used to realise flexible and protocol independent telephony solutions for a small to medium enterprise.

Acknowledgements

I would firstly like to thank my supervisor, Alfredo Terzoli, for the patience and support he has given me over the course of my postgraduate career. Without his energy and enthusiasm and inexplicable ability to make you think that you actually are excited, however much you lacked interest previously, this project may never have been completed. In addition to his help with this project, he has tirelessly given aid in many other aspects of my life, and I am very grateful.

Thanks also to the staff and postgraduate students of the Rhodes Computer Science Department, who have provided a community in which it was a pleasure to work. In particular, Jason Penton put in a lot of time in helping me get started in the area in which I finally placed this project, and with whom I collaborated in the design and implementation of the iLanga web interface. In addition, the members of the Rhodes University Computer Users Society have provided invaluable support and input, not only to my work on this project, but to my development as a whole over my entire university career. I would not have made it to where I am without the constant challenge and aid that they provided.

I would also like to specifically thank Ingrid Brandt and Russell Cloran, who have both provided aid and support, and put in large amounts of time and energy in helping me to finish this project.

This project is dedicated to the Family who always believed I would get here in the end.

Contents

List of Figures

Chapter 1

Introduction and Background

In traditional telephony, calls are made over the Public Switched Telephone Network (PSTN). Both PSTN networks, and Private Branch Exchange (PBX) systems using the same technology, are expensive, both in terms of buying the hardware used for switching, and in requesting maintenance and programming for the proprietary hardware. In recent years, however, telephony utilising Interner Protocol (IP) networks has become popular, coming under the general label of Voice over IP (VoIP).

This thesis will look at service creation on a converged network (with both PSTN and IP technologies), with specific emphasis on voice services in a multiprotocol environment, where a variety of telephony technologies are used. This chapter is an overview of various telephony technologies, with a brief discussion on work that has been done at Rhodes University on investigating service provision in each area. It also introduces Asterisk, the open source PBX system which promises to ease the interoperability of telephony technologies and services.

1.1 The Public Switched Telephone Network

The Public Switched Telephone Network is the collection of all the public circuit switched networks in the world. Circuit switching can be conceptualised as the setting up a continuous electrical circuit between the endpoints. While the network used to be completely analog, it is now almost entirely digitized, with services such as ISDN being widespread. One of the main advantages PSTN has over its packet switched counterpart is its guaranteed quality of service.

PSTN most commonly uses the Signalling System 7 (SS7) [Russell 2000] set of protocols for signalling.

With traditional PSTN telephony, voice services were built into the switches, which made them difficult to modify or upgrade. Specialist engineers were required to reprogram the switches, and upgrade the services. This has, however, changed with the advent of VoIP. With everything being done in software, it is much easier to rewrite a voice service. Anjum et al say:

> In terms of computational resources, the current PSTN assumes that the resources available at the user terminals, or "customer premises equipment" (CPE), are extremely limited, so that all the intelligence required to provide services is centralized in network switches, databases, and operations support systems.[Anjum et al. 2001]

They go on to describe how services on IP networks are application-centric, in that the networks themselves merely passively provide connectivity, and can remain unaware of the applications and services running on the hosts on the network: this allows the applications and services much greater control, and thus increases the ease of service provision.

1.2 Voice over IP

The use of the IP data network to carry real-time communication has come to be known as Voice over IP. Because the network is packet switched, bandwidth is saved, since packets are sent only when necessary, instead of the constantly available (often unused) bandwidth of a circuit switched network. In addition, the data is easier to compress, increasing the efficiency of the medium. However, packet switching has its own disadvantages, especially with latency and buffering. In order to utilize bandwidth to maximum efficiency, voice packets are buffered together before they are sent over the network, which means that the recipient needs to wait for several packets to be produced, packaged together, and sent, before even the first one is received and decoded [Percy 1999]. In addition, the Internet Protocol does not guarantee quality of service - it is a "best effort" system, which means that it does not ensure that packets are delivered in the right order, or whether they are delivered at all. Also, the voice packets might arrive at a variable rate. This is known as "jitter", and needs to be fixed on the receiving side. The most common

method to fix this is to buffer the data received for a brief period before playing it. This buffering increases latency, which decreases the quality of the communication experience of the users.

There are several protocols that are used for VoIP. The next sub-section will briefly cover the most prominent.

1.2.1 The H.323 Protocol

H.323 [ITU-T 1998c] is composed of a number of protocols which together provide audio-visual communication over any packet network. H.323 uses the Real Time Protocol (RTP) and the Real Time Control Protocol (RTCP) to send and receive audio-visual data, and uses H.225 [ITU-T 1998a] and H.245 [ITU-T 1998b] to set up and tear down calls. An H.323 network consists of a number of different components, which include terminals, a Gatekeeper, a gateway and a Multipoint Controller Unit (MCU).

The terminals are any endpoints that provide real-time, two-way multimedia communications with other H.323 terminals. They also encode and decode the audio and video signals.

An H.323 terminal must at least be capable of sending and receiving audio data, and can optionally support video and data transmission. An H.323 network can consist simply of terminals, but these will only be capable of point-to-point calls. For more sophisticated operations, the network needs to have a Gatekeeper.

A Gatekeeper is a component that manages an H.323 zone, providing call-control services such as admission control (allowing terminals to register, so that they can then request permission to make calls), address resolution (keeping track of the different aliases of H.323 terminals), and bandwidth management (keeping track of how much bandwidth has been requested, and allowing or denying further requests based on this information).

A Multipoint Control Unit (MCU) allows a number of terminals (three or more, since two terminals is just a simple call) to engage in multipoint conferences. The MCU needs to mix and transcode the media streams of each participant to maintain the conference.

Finally, a gateway enables an H.323 network to communicate with other types of network, such as a PSTN network. It provides translation to and from H.323 at various layers. For example, an H.323/PSTN gateway will need to turn the packet-switched H.323 media on the IP network into circuit-switched TDM media.

1.2.2 The Session Initiation Protocol

The Session Initiation Protocol (SIP) [Handley et al. 1999, Charter 2003b, Rosenberg and Shockey 2000] is ousting H.323 from its position as the leading signaling protocol for VoIP. It is unique among signaling protocols in having been developed by the IP community (it is being standardized and governed by the IETF), instead of by the telecommunication industry, like all the other VoIP protocols (which are governed by the ITU).

Since SIP is a signalling protocol, all it does is set up, modify, and tear down multimedia sessions between clients. The actual media involved in the call is separated from the call signalling by SIP, to the extent that the media stream and the signalling messages might take a completely different route between the two endpoints.

One important advantage of SIP is its support for mobility. The stream is associated with the user, and not with the specific device currently being used. A SIP user can register his location from a variety of devices, and the media stream is routed to the device he is currently registered on.

SIP most commonly uses RTP to carry the media, although it does not define a specific transport. SIP uses the Session Description Protocol (SDP) [Handley and Jacobson 1998], which describes the media content (the codec being used, IP address and port, etc).

SIP is a client-server protocol, but SIP endpoints contain both User Agent Clients (UAC) and User Agent Servers (UAS): User Agent Clients make SIP requests, and User Agent Servers reply to them. There are other types of SIP servers which can respond to these requests. SIP Proxy servers transparently forward the requests on to other User Agent Servers. SIP Redirect Servers remap SIP addresses to other addresses, and return these new addresses to the client, so that it may remake the request. Finally, SIP Registrar servers keep track of the registration location of SIP users, and maintain a mapping between their SIP addresses and their actual location (i.e. the IP address at which they can be contacted).

Media forking is another powerful feature of SIP, which allows the creation of multiple signalling streams associated with a single call, each sent to a different destination.

1.2.3 Inter-Asterisk Exchange

Inter-Asterisk Exchange (IAX) is a protocol, now in its second version, which was developed for the open source Asterisk PBX. It was originally intended to be used for Asterisk servers to communicate with each other, but is also being used more and more for endpoints. It has several advantages, mostly due to the fact that it uses a single UDP stream for communication. This stream contains both the signaling and the multimedia data. Unlike other protocols, it does not use RTP (or a similar encapsulation protocol) to carry the media. This makes it a lot easier to administer (firewalls and Network Address Translation (NAT) pose less of a problem than for other streaming protocols).

Another advantage of IAX is its support for trunking: it can encapsulate the data for more than one call in a single packet. This reduces the IP overhead without increasing the latency, meaning that less bandwidth is used without losing call quality.

1.2.4 The Media Gateway Control Protocol and Megaco

The Media Gateway Control Protocol (MGCP) [Arango et al. 1999, Greene et al. 2000] is a gateway protocol, initially used to connect other VoIP networks with legacy PSTN networks. An MGCP system consists of a Call Agent and a Media Gateway. The Call Agent acts as a VoIP gateway, and communicates with endpoints from other networks using their native protocols (such as SIP or H.323). These VoIP protocols connect to the Call Agent, and make standard requests to it. It relays these requests to the media gateway, which converts the media signals from their packet-switched system to a circuit-switched system, and relays it onto the PSTN network. The media gateway can connect to various endpoints on the PSTN side, such as a digital channel (as found in ISDN lines), an analog line, or an ATM interface. Of course, the connection goes both ways, with the PSTN endpoints communicating with the Media Gateway, which relays the calls on to the VoIP endpoints via the Call Agent.

The Megaco Protocol [Greene et al. 1999, Charter 2003a, ITU-T 2000] holds a similar position to MGCP, and has a fairly similar architecture, but supports a broader range of networks. For Megaco, the two basic constructs are terminations, and contexts. A termination is a stream that enters or leaves the Media Gateway, and is uniquely identified. They may be ephemeral, being created on demand, or they may be instantiated when the Media Gateway starts up, and remain

in existence while it is active - these are typically for ports on the gateway. A context is the combination of two or more terminations connected together. For example, a normal active call might consist of a permanent termination (an end-device, for example) and an ephemeral termination to the network [Allen 2000].

1.2.5 The Real-time Transport Protocol

The Real-time Transport Protocol (RTP) [Force 1996] is one of the most commonly used protocols for carrying audio and video data. It is often used in conjunction with the Real Time Control Protocol (RTCP) [Huitema 2003], which allows the receiver to control the inflow of the data by issuing commands such as pause and rewind. However, for VoIP systems, it needs to be used in conjunction with a signaling protocol such as SIP or H.323. The signaling protocol is used to establish and maintain the RTP connection, additionally taking care of aspects such as choice of codec (the compression/decompression algorithm used to encode the multimedia data contained in the stream).

1.3 Service Interoperability

With so many different protocols, creating interoperable services presents quite a challenge. Chatzipapadopoulos et al say:

> ... although network integration at transport technologies may be an idealistic assumption, the management of services traversing different networks cannot be done without considering the accurate definition of a unified service framework, taking care of harmonising, from a service designer's viewpoint, differences existing at network transport layers. In other words, if transport integration cannot be achieved, integration at service design and service management level must be pursued in order to cope with emerging and future services. [Chatzipapadopoulos et al. 2000]

Perdikeas and Venieris take a similar view, and present a higher-level solution, using the Parlay Java API [Perdikeas and Venieris 2001].

A different view is taken by Glitho, who says that both H.323 and SIP have problems, and that alternatives to these two service architectures are being sought. The solution, he says, lies in two trends:

> the returns to the old and well-known intelligent network (IN) architectural framework, and the exploration of more recent approaches, such as mobile agent technology, that go beyond IN. [Glitho 2001]

However, C.A. Licciardi et al consider that a hybrid of the Intelligent Network and IP frameworks is necessary for progress to be made [Licciardi et al. 2001]. Rinde attempts to predict the path of internet telephony, and prophesizes the growing convergence of IP network frameworks, along with the PSTN network [Rinde 1999].

Bond et al developed their own system, BoxOS, an IP telecommunication system using Distributed Feature Composition (DFC) to create services [Bond et al. 2004]. This system implements "pseudo-drivers" to allow interaction with different protocols, and also requires address translation between the different systems.

The next sub-section looks at work done at Rhodes University on providing services for the various protocols, and how it affects the problem of service interoperability.

1.3.1 H.323 Services

Some examination of the H.323 protocol has been done by Jason Penton, who categorises the varieties of services that can be offered on an H.323 network. He makes the distinction between basic and enhanced services, signalling and non-signalling services, standard and proprietary services, and between distributed and centralised services. Using this categorisation, several proof-of-concept services are successfully created [Penton 2003, Penton et al. 2001a].

As far as interoperability is concerned, Penton touches on the subject only briefly. He examines communications between an H.323 network and a legacy PSTN network, and attempts to make the H.323 services available to legacy telephones, using an ISDN interface connected with the ISDN4Linux interface [Project 2004]. While he does succeed in this, it is at the cost of a certain degree of functionality. The only method available for PSTN devices to communicate with the

services on the H.323 network is using DTMF tones [Technologies 2004]. Penton modifies the PSTN/H.323 gateway to encapsulate DTMF tones in the H.245 media control channel of H.323 [Force 2000]. This enables PSTN devices to communicate with H.323 services, but only in a very limited manner. To be accessible to PSTN devices, an H.323 service must still be designed to be controlled through DTMF encapsulated in H.245. He successfully designs a system for reading email from legacy devices using DTMF codes for control [Penton et al. 2001b].

While this does mean that H.323 services can be made available to PSTN networks, it is only in a very limited manner. In addition, no other protocols were investigated.

1.3.2 SIP Services

SIP service creation has been investigated by Ming Chi Hsieh. He creates a taxonomy of SIP services, dividing them broadly into basic services (which simply consist of the normal functions of SIP clients), and advanced services. Advanced services are further broken down into call related services, interactive services, internetworking services, and hybrid services, and proof-of-concept services are developed for the first three of these categories in two different SIP environments [Hsieh 2003, Hsieh et al. 2001, Hsieh et al. 2002].

Hsieh examines the internetworking of services between SIP networks and H.323 networks, basing his work on [Agrawal et al. 2001], and between both of these and legacy PSTN networks via an MGCP network. However, his solution is not extensible. His method of internetworking is to provide translators between each protocol. He uses a translator to turn SIP service requests into H.323, and vice-versa, and a translator to turn each of these into MGCP requests.

While this solution does work, and allows the internetworking of services, it does not provide an easy method to add services written for a third protocol (such as IAX) - if this needs to be done, a translator would be required between the new protocol and each of the existing protocols. It is clear that the number of translators required goes up rapidly as new protocols are added.

Jiang et al have also developed a SIP system that replaces a traditional telephony PBX, using PSTN/IP gateways to do the translations [Jiang et al. 2001].

1.3.3 Media Gateway Control Protocol

Ashley Jacobs considers MGCP as a protocol for developing services [Jacobs 2004, Jacobs and Clayton 2003]. Since it allows the interaction of other protocols with each other, it is the "lowest common denominator". Jacobs creates a service which allows H.323 endpoints to send Short Message Service (SMS) messages [Jacobs and Clayton 2002, Halse and Wells 2002], and then extends this service to allow SIP endpoints to do the same. These services use MGCP to communicate their commands to the PSTN network. Using the MGCP gateway, it is possible to develop a service in either H.323 or SIP, and expose this service to endpoints on the other network.

However, this internetworking still requires what Jacobs refers to as an Interworking Function (IWF) - namely MGCP. This IWF is analogous to Hsieh's translators. The benefits of using MGCP as an IWF lie in the fact that adding a new protocol to the system merely requires extending MGCP to understand the new protocol, instead of implementing a translator between the new protocol and every existing protocol. However, the system still has problems. MGCP lies on the edge of the network, and is intended for communication with legacy networks. As Jacobs says, it is the "lowest common denominator", and this means that all services must run at the level of the lowest common denominator. Thus, while this work goes a long way to making services universal, it does not quite succeed.

1.4 Device Interoperability

While the focus up till now has been on the number of protocols available for VoIP networks, there is another problem that is to be faced when creating voice services. As more and more devices are becoming available with VoIP support, the range of capabilities that needs to be catered for widens. Services will need to support everything from small handheld devices to graphically rich desktop machines.

There has been a fair amount of work done in the field of device independent interfaces. Most of the work has been done in a similar manner to UIML:

> the User Interface Markup Language (UIML) to describe user interfaces in an appliance-independent manner. [...] UIML is a declarative language that distinguishes *which*

> user interface elements are present in an interface, *what* the structure of the elements are for a family of similar appliances, *what* natural language text should be used with the interface, *how* the interface is to be presented or rendered using cascading style sheets, and *how* events are to be handled for each user interface element [Abrams et al. 1999].

There are a number of other XML-based languages which profess to describe user-interfaces in a device-independent manner. The problem with these solutions is that all services that conform to the interface language have to drop down to the level of the lowest common denominator [Penton et al. 2001b]. If one service cannot handle a specific method of interaction, and they all have to conform to the same interface, then that method of interaction is ruled out for all the other services, to maintain interoperability. In addition, while these interface languages do solve the interface problem to a certain extent, they also add another layer of complexity, relying all devices to understand whichever schema has been chosen to describe the user-interfaces.

An ideal solution would be a form of "middleware", that is able to negotiate the abilities of the telephony technology, and the requirements of the service, and mediate between the two to achieve compatible communication. This would give rise to "differentiated services", with varying functionality, according to the capabilities of the endpoint. Some services would be rendered inoperative on certain devices (such as a video service on a legacy telephone), but at least the problem that services can only provide functionality on a level with the least powerful endpoint is resolved.

1.5 Asterisk: A possible solution

A lot of attention has recently been given to the Asterisk [Asterisk 2004] system, a complete PBX in software, both standards-based and open source. It is interesting to this thesis because it allows various telephony protocols to inter-communicate with each other, and with voice services, in a transparent manner, making the development of mature VoIP solutions easy. It exposes several protocol- and technology-agnostic APIs that can be used to communicate with it, and with the other applications and telephony technologies connected to Asterisk. In other words, Asterisk acts as middleware between services and the telephony technologies, so that neither side is required to conform to any API other than Asterisk - adding or removing different protocols is something done to Asterisk, and will not affect the way other parts of the system connect.

This agnosticism with respect to transport makes Asterisk an ideal platform for developing services for a converged network. This thesis will investigate Asterisk as a possible solution to the problem of a multi-protocol environment. In addition, Asterisk is worth investigating in itself, purely because it is unique in its position as a versatile open source softswitch (see 2.4).

1.6 Aims of this thesis

As described, the growth of VoIP has resulted in two separate problems for service creation. On the one hand, a voice service needs to be accessible from devices with a wide-range of capabilities, from small hand-held devices to rich graphical interfaces such as desktop computers. On the other hand, there are a number of VoIP protocols in use, which need to inter-operate if a network is to be useful.

Asterisk has been proposed as a solution to these problems, due to its architecture as middleware. This thesis will experiment with the various ways in which Asterisk can be used to develop voice and non-voice services for devices of varying capabilities and connected via various protocols. In addition, the methods by which Asterisk's behaviour can be extended and modified will be investigated. On the one hand, we will extend services to be accessible from a variety of different interfaces. On the other hand, we will expand the depth of these services to allow the user complete control over the PBX, utilising the APIs which Asterisk exposes to communicate with the PBX. This is a service, in that it is an extension to the PBX, not an integral part of it: if this level of interaction is possible, Asterisk's ability to support any type of service will be well demonstrated. For the rest of this thesis, the words "expand" and "extend" will be used to convey the difference in these two aims: services will be *extended* in terms of their accessibility, and *expanded* in terms of their functionality.

It became apparent in the course of investigation into services that one of the main problems facing the development of services was ensuring that the users of the service were who they claimed to be, and were allowed access to the service. Once the service framework has been put in place, the main problem facing the creator of a service is authentication. For this reason, the investigation and development of a standard authentication framework for these services is included in the goals of this thesis.

The tools used for this project are all open source, and of fairly recent development. For example, the Twisted framework, described in chapter 4, and used to create the extended Juke service in

chapter 5, is still under active development, and has very little documentation as yet. Asterisk itself has only recently undergone its first official stable release, and is very lacking in official documentation. Both projects rely very much on user contributed documentation and unofficial "how to" documents. A substantial amount of the work done throughout this thesis was investigating these systems and working out how they worked. Thus, a further, but important, aim of this thesis is to document more systematically the process of developing using these systems.

To summarise, this project has the following goals:

1. To investigate the methods of extending the behaviour of Asterisk, to provide services to users of the PBX which can be accessed from any telephony technology.

1.1 To extend these services to be accessible from a variety of different interfaces.

1.2 To expand these services so that they allow the user complete control over the PBX.

2. To investigate the different methods by which a service can authenticate itself to the framework, and implement them.

3. To investigate and document the various open source tools that are available for use in achieving these goals, and assess their readiness for deployment in a production environment.

1.7 Methodology

This section will describe the methodology which we followed when pursuing the aims and objectives described above.

1.7.1 Initial Installation

Before the investigation of service creation could be conducted, the actual Asterisk PBX needed to be set up. This involved the installation and configuration of the software, and the setting up of the required hardware. Part of the investigation involved ensuring that the addition of services to

the PBX did not result in instability or unusability of the system, so a complete, usable telephone system was required before the investigation could continue.

A number of users of the system needed to be created, and given telephony devices in the form of softphones, legacy telephones, and VoIP hardphones. Additionally, extensions needed to be configured on the PBX so that they could make calls to each other, and outside the system. The system needed to be deemed stable and running properly before investigation could continue. This process is not documented in this thesis, as it forms part of Jason Penton's work [Penton and Terzoli 2004].

1.7.2 Service Creation

Various methods of creating simple services were investigated. Since documentation is sparse, a good deal of experimentation was expected in this step, to work out how different types of service could be created, and to discover the different service capabilities that each method allowed. In the course of this investigation, the findings were documented, to rectify the problem of the lack of documentation, for future developers.

Once this investigation had been performed, services were created, and added to the existing PBX. As each service was added, the stability of the PBX was monitored - a service cannot be considered successfully deployed if it reduces the usability of the entire system. In the first stage, only very simple services were added, to test the ease with which generic services could be added to the PBX.

1.7.3 Service Extension

Having created services within the PBX, a new, external system was developed, outside of the PBX, running independently, but communicating with the PBX in the same way as the basic services did. This removed the dependence of the services on the PBX, and created the possibility that services could run from other devices and interfaces. This external system was effectively a PBX service itself, like those in the previous stage, but had the added ability to communicate with other external services. As such, it was necessary to make sure that this system did not affect the stability or performance of the PBX either.

Once the external system was in place, external services were created that connected to it, and through it, to the PBX. These services were "extended services", in that they were accessible from multiple interfaces, and not merely through the PBX.

1.7.4 Service Expansion

In this stage, the depth of the services' functionality was expanded. A service was created which allowed users complete control over the PBX, and this service was integrated into the existing PBX and its other services. Once again, it could not interrupt the users current experience, and could not degrade the performance of the PBX in any way. It was expected that a number of the internals of the Asterisk PBX would require modification, or at least examination, in order for this service to integrate and control them in the desired manner. All the discoveries in this area were again documented.

1.8 Document Overview

The structure of this thesis is as follows:

Chapter 2. This chapter describes Asterisk, the telephony system used as the basis for the services being developed, and its architecture. It describes how Asterisk has a highly flexible method for directing calls between channels, and introduces the idea of the dialplan. Particular focus is placed on the methods that can be employed to extend the behaviour of Asterisk, since this is how services will be integrated into the telephony network. Four methods of extending Asterisk are described. Finally the iLanga system is introduced as an example of a fully deployed PBX using Asterisk in conjunction with other systems.

Chapter 3. This chapter discusses the creation of basic services for Asterisk. It outlines the Asterisk Gateway Interface, and the Asterisk Application API, the two basic methods for interfacing one's own applications with the PBX. Having discussed their architecture and compared them, it creates example services using each method.

Chapter 4. This chapter describes the Python/Twisted framework, which will be used for the development of the two larger services described in chapters 5 and 6, and how it can be used to write clean, efficient network applications, avoiding network latency and blocking problems. It also covers the credential system of authentication in Twisted, and Perspective Broker, an advanced system for remote procedure calls that is built into the Twisted framework. Finally, it briefly describes Nevow, an advanced webserver that integrates with the rest of the Twisted framework.

Chapter 5. This chapter investigates the design and implementation of an extended voice service to allows users to play music over the telephony network, controlling the playback from their devices. This service will be accessible from multiple interfaces. It looks at the different interfaces which might need to be catered for, and identifies three possible types of interface. The chapter then details how the communication between the interfaces and the central service is carried out. Finally it covers the design and implementation of proof-of-concept examples of each type of interface.

Chapter 6. This chapter describes the development of a more complex service, which allows a user sophisticated control over their telephony environment. It details various components of the iLanga system which need to be accessed from the service, and chooses methods by which this communication can be achieved.

Chapter 7. In this chapter the work done in this thesis is outlined, and conclusions drawn.

Chapter 2

Asterisk

As Voice over IP (VoIP) moves into the mainstream, software to take advantage of the protocol is becoming more prolific. Asterisk is gaining a lot of interest as the first complete free, open source implementation of a PBX in software. This chapter describes Asterisk and its architecture briefly, and details those aspects of the software that are relevant to this thesis.

2.1 Architecture of Asterisk

Asterisk can be thought of as "middleware": it sits between telephony technologies and telephony applications, providing a generic interface between them, as illustrated in figure 2.1 [Spencer et al. 2003]. In other words, it connects to various telephony technologies, giving a consistent way to receive and send calls to and from various VoIP protocols, as well as to and from such technologies as ISDN and legacy PSTN networks. At the same time, it gives a consistent interface to various telephony applications, such as voicemail, conferencing, IVR scripting, and so on. Thus, creating a telephony application need not take into account the different methods by which somebody might connect to the PBX (using a software SIP client, or using a legacy telephone, or from outside over an ISDN line). In the same way, connecting your PBX to another telephony technology (adding an ISDN card, or new VoIP connections) need not affect any of your telephony applications.

The architecture may be seen in a slightly more detailed way, as well. There are four APIs involved in Asterisk: the Application API, the Codec Translation API, the File Format API, and

Figure 2.1: Asterisk as middleware ([Spencer et al. 2003])

the Channel API, as illustrated in 2.2 [Spencer et al. 2003]. The Application API is responsible for interfacing with the telephony applications that run on the PBX mentioned above: the voicemail, conferencing, etc. The Channel API is responsible for interfacing with the telephony technologies on which the PBX runs: MGCP, SIP, H.323, etc. The File Format API interfaces with various file-formats used by different parts of the PBX. For example, certain applications might play voice prompts that are encoded in the MP3 sound file format - these applications will use the File Format API to decode these files. In this way, if any developer wishes to use a new type of file format for some purpose, he merely needs to create a module for the File Format API, and from then on, all parts of the PBX will be able to use this type of file. The Codec Translation API works in a similar way: it is responsible for encoding and decoding the various types of audio (and, in theory, video, text, etc) stream that pass through the PBX. Examples of codecs are GSM, G.723 and A-Law. Using the Codec Translation API allows channels that are compressed using different codecs to communicate with each other seamlessly.

This architecture is what makes Asterisk particularly attractive. The APIs modularity means that every aspect of the PBX is sealed off from the rest of the system, and modifying one (for example, adding a new telephony technology) does not require any modification of the rest. Asterisk acts as a "lingua franca" for VoIP, allowing the deployment of services without needing to cater to a specific protocol, and without necessitating service rewrites when new protocols are

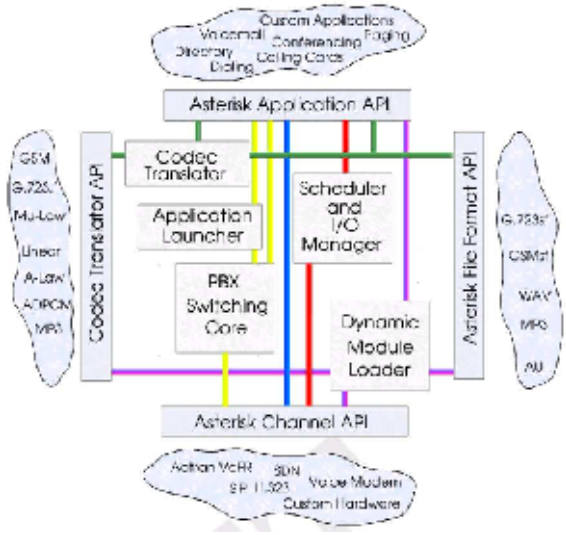

Figure 2.2: Asterisk APIs ([Spencer et al. 2003])

incorporated. In other words, nothing needs to be known about interfacing with SIP or H.323 in developing these services. The Asterisk Channel API interfaces with those technologies, and deals with protocol specifics.

2.2 Extensions

One of the core concepts in Asterisk is that of "extensions". These are a set of numbers which the PBX recognises as valid, and to which it knows how to react. The most common use for extensions, and that from which they get their name, can also be found in legacy PBXs: when a certain extension number is dialled, the call is routed to an associated telephone, which rings, and communication begins. In Asterisk, the concept covers a lot more than that. For a start, extensions are not necessarily "dialled". It is easier to think of an extension simply as a numbered request to Asterisk, which will respond by taking action according to the extension configuration (also known as the 'dial plan'). Secondly, the request will not necessarily result in the routing of the call to an end-device. Asterisk works in terms of channels, which may or may not be end-devices. The call may end up at a certain device in the end, but it could just as easily be configured to remove all voicemail waiting for a certain person, and then instantly terminate. Finally, the extension may not necessarily be a specific number. Asterisk has dialplan extensions

for the initial state: the state a call is in when it first connects to the PBX, without having dialled an extension. This connection might be made because a device attached to the PBX is picked up, or because somebody dials the PBX on an external (e.g. PSTN) interface. There are also dialplan extensions for In addition, Asterisk can recognise patterns of numbers, and the behaviour for unrecognised extensions, and even the initial behaviour (before any extension is dialled), is configurable. The easiest way to illustrate this is with an example of extension configuration.

```
exten => 75, 1, Dial(Zap/1,20)
```

This configuration simply redirects all calls made to extension 75 to the `Zap/1` device (the first Zaptel telephone) - much as a legacy PBX would. The `20` indicates that Asterisk should attempt to dial this device only for 20 seconds. The `1` after the `75` simply indicates that this instruction is the first (and, in this case, only) instruction in this extension.

```
exten => _8/1000, 1, Dial(IAX/nihil/${EXTEN:1})
```

This configuration matches a number pattern - in this case, simply any number that begins with an '8'. There is a further restriction, however - the call has to be placed by the person with the caller ID of '1000', otherwise the directives will not be followed. This is known by Asterisk developers as an "Anti Ex-Girlfriend" extension. The directives in this case simply consist of forwarding the call to another extension on another Asterisk PBX (identified by the name 'nihil'). The extension in this case is derived from the extension that was actually dialled (stored in the variable "${EXTEN}"). The ":1" after the variable name indicates that the first character should be removed from the extension. To summarise, the configuration directive above states that if the person with caller ID '1000' makes a call that begins with an '8', then whatever he dials after the '8' must be taken as the extension on the 'nihil' PBX to which his call must be forwarded. If he dials '875', then it would be as if he had dialled '75' on the 'nihil' PBX.

```
exten => _95, 1, Wait(5)
exten => _95, 2, CheckTime(900,1700)
exten => _95, 3, Playback(workhours)
exten => _95, 4, HangUp
exten => _95, 1003, Dial(IAX/overseas/2634${EXTEN:2})
```

This configuration is somewhat more complicated. As before, it will match a pattern - in this case, any number that begins with '95'. However, this time, it has multiple directives, which will be followed in order. The first one simply tells Asterisk to wait for five seconds - the caller will hear a ringing tone. Asterisk will then move onto the second directive, which makes a call to an "application" called `CheckTime`. This application is given two parameters ('900' and '1700' in this example), which it interprets as times. The application will return successfully if the current time is between the two limits passed as parameters. A successful return, in Asterisk, means that execution of the dial plan continues with the next directive. An unsuccessful return means that the number of the current directive is increased by 1000, and then execution continues as before. In the above example, the next directive (number 3) is to play a pre-recorded sound file (probably something along the lines of "please do not make international calls during work hours"). After that, directive number 4 will hang up the call. Since the next directive (number 5) does not exist, execution of the dial plan for this call is terminated.

If, however, the `CheckTime` application had found that the system time was not between the two limits, the directive number is increased to '1002'. After this, execution continues with the next directive, which is number 1003. This directive instructs the PBX to remove the first two digits ('95') from the extension number, add the result onto the end of the number '2634', and use this final number as an extension request to the 'overseas' PBX. It so happens that '263' is the international dialling code for Zimbabwe, and '4' is the Zimbabwean code for the capital city, Harare. Thus, in plain English, the above configuration snippet will allow anybody to phone a Harare telephone, by dialling its number prefixed with '95', but it will only allow this if it is not during work hours.

I must stress that the `CheckTime` application does not actually exist in a default Asterisk installation. It is a separate module, written in C, which is dynamically found and loaded by Asterisk at runtime. Any number of similar applications may be written, thus allowing one to perform all sorts of tasks and checks during execution of the dialplan. For example, one could write an application which checks whether the extension currently being dialled is today's "lucky number" - this would enable the creation of a sort of lottery service, where one can dial a random extension, and if the number chosen is the correct one, the user wins a prize - maybe a free call to Harare. It would also be easy to write an application that stores the caller ID of the user in a database, so that one may not attempt to guess the lucky number twice in one day. While this sounds like a trivial, fairly frivolous use of a PBX, it is simply an example of one of the things possible with the Asterisk dialplan configuration, with minimal work. By contrast, legacy PBXs

require trained engineers to reprogram their hardware for anything like this.

Extensions define the basic functionality of the PBX - everything else is configuration and external applications.

2.3 Extending Asterisk

This section details the various methods by which one can alter the behaviour of an Asterisk PBX. These methods will be used later on in the thesis by services which need to perform some action within the PBX. While Asterisk has a lot of basic functionality, which is highly configurable, one needs a way to extend it to do something completely new. There are four main methods to do this, each of which I will briefly describe. They are the Dial Plan, the External Interfaces, the Asterisk Gateway Interface, and Application Modules.

2.3.1 The Dial Plan

As described above, the dialplan can contain some fairly sophisticated instructions. It has scripting commands like `Goto` and `GotoIf`, and has commands to manipulate variables like `SubString`, `SetVar`, and `Math`. While these commands are fairly basic compared to most modern programming languages, they are still sufficient to be able to create some powerful applications using Asterisk.

2.3.2 External Interfaces

There are two external interfaces to Asterisk, to which other applications can connect, query the PBX, and issue commands. These are the Manager API and the Command Line Interface.

2.3.2.1 The Manager API

When the Asterisk PBX starts up, it opens a TCP port, on which it listens for connections. External applications may connect to this port, and communicate with the PBX by writing (and

reading) plain-text commands (and responses) over this TCP connection. This is the Manager API, by which various actions can be performed remotely. It allows querying about various aspects of the PBX (such as which channels are open, what calls are in progress, and so on), and allows actions to be performed (such as originating a call, closing a channel, or redirecting a call). It has an authentication system, so that the PBX is not open to influence from unauthorized users.

2.3.2.2 The Asterisk Command Line Interface

By default, Asterisk starts up as a daemon - a program running in the background, with no interaction with the user. However, it is possible to start it up in command-line mode, where it presents a prompt to the user, at which commands can be issued. If Asterisk is already started as a daemon, it is possible to start another instance of Asterisk which simply presents the command-line prompt, and relays the commands on to the already-running, daemonized Asterisk process. It is this latter case which is of interest to us, because it enables a program to open up a command-line prompt to the running PBX, and issue it with various instructions to query, or alter the behaviour of the system. This prompt is very powerful: it has commands to change most aspects of the PBX, start and end calls, reload configuration files, and even shut down the PBX completely.

2.3.3 Asterisk Gateway Interface

As described in section 2.2, the Asterisk dial plan can be configured to perform a number of different actions. One of the most useful actions is specified with the "AGI()" directive. This instructs Asterisk to execute the specified external executable, in the Asterisk Gateway Interface environment. While this will be discussed in greater detail later, the point here is that Asterisk pauses its execution of the dial plan commands while the external application runs. The executable can use the Asterisk Gateway Interface to send any of a limited set of commands back to the PBX, allowing it to manipulate the current call to a certain degree.

2.3.4 Applications

In section 2.2 above, I mention that the "CheckTime" application does not exist in the core Asterisk installation, but can be plugged in as a module. This is the concept of an "application"

- a program that conforms to a certain API, compiled with, and linked to, the Asterisk libraries, that can be loaded or unloaded dynamically, and is called from the dial plan. While applications are not strictly external programs, they are included in this section because they offer the greatest flexibility and power over the behaviour of the PBX. They will be discussed in greater detail later: the main point here is that an application can be written to do anything one wants (as shown in the example of the "CheckTime" application in section 2.2). Applications have full access to the inner workings of the Asterisk PBX, via library calls, and are only limited in terms of the actual implementation - they must be written in C, and conform to the Asterisk C API.

2.4 The Alternatives to Asterisk

There are various alternatives to using Asterisk. Asterisk itself consists of the PBX software itself, as well as a SIP proxy, a built in IVR (Interactive Voice Response) server, and support for the four major VoIP protocols (SIP, MGCP, H.323 and IAX). It is commonly run with SER (SIP Express Router), as well. Examples of other PBX software systems are OpenPBX, PBX4Linux, SIPexchange, and YATE (Yet Another Telephone Engine). However, none of these support all the features that Asterisk has built in. For example, only YATE has an IVR server built-in, although there are several other stand-alone IVR servers that can be installed together with PBX software to make up for this. Rash investigates both SIPexchange and Asterisk. He says Asterisk is

> a lot more than a private branch exchange. It also takes on the functions of a media server, a protocol gateway and a conference bridge. It goes beyond Voice Over Internet Protocol, too, supporting other types of digital communication [Rash 2005].

He speaks well of SIPexchange, too, but purely as a SIP server, and says it has a “more commercial flavour”.

In the end, Asterisk has the largest feature set, and, probably more importantly, the largest developer support-base: there is such interest in Asterisk currently that many large companies are sponsoring development in various aspects of it, and the open source community is putting a lot of effort into improving it as well.

2.5 iLanga

iLanga was developed at Rhodes University. The system was designed to be a "complete, cost-effective, computer-based PBX" [Penton and Terzoli 2004], using Asterisk [Penton and Terzoli 2003]. It is an example of a fully-deployed working Asterisk PBX [Asterisk 2004], but it also has several components in addition to Asterisk. It runs an instance of SER (the SIP Express Router), and an instance of the OpenGK implementation of the H.323 Gatekeeper [OpenH323 2003].

Asterisk has its own SIP proxy, but it is very limited. A better solution is to use Asterisk as a SIP gateway to other protocols (such as PSTN), and to use SER as a SIP proxy and registrar. Examples of the advantages are that SER is capable of performing SIP forking, and can forward a SIP call straight to another SIP agent, without going through the Asterisk PBX, if necessary.

2.5.1 The concept of the User

One of the main advantages of iLanga is that it has the idea of a user. Asterisk itself merely deals with channels: it forwards a call from one channel to another, or provides a service to an incoming channel. On the other hand, iLanga has specific users, for which details can be registered, and to whom devices can be allocated. Thus, instead of making a call to a certain softphone, I will call a specific person - iLanga has records of all of that persons devices, and will ring on any (or all, depending on the configuration) of the devices. In this way, the system could be configured to ring devices in order of priority, to ring different devices at different times of day, and to ring the same device for different users (for a shared phone, for example). Regardless of the configuration, the users of the system will simply phone a *person*, and do not need to be exposed to the non-intuitive idea of a specific device.

2.5.2 Web interface

Because iLanga is a voice-over-IP system, it has several inherent advantages over traditional PBX systems. One which stands out in particular is the web interface. This interface allows users to log in to the system, change their settings (such as which devices will ring when they are phoned), check their voicemail and account credit, check their call-records, and do directory searches. It uses a variety of methods to communicate with the Asterisk PBX to achieve this

[Hitchcock et al. 2004]. This system will be discussed in much greater detail in chapter 6, as an example of a complex service added to Asterisk.

2.6 Summary

This chapter described the Asterisk PBX system, and gave an overview of the various parts of the system, and the ways in which its behaviour could be modified. In addition, alternatives to Asterisk were discussed, and the iLanga system introduced.

Chapter 3

Basic Services

In this chapter, I will discuss the basic creation of a voice service. As we have seen, the advantage of Asterisk is that services are protocol-agnostic: they interface with the Asterisk API only, and do not use any specific telephony technology. In this chapter I will describe the two main methods for communicating with the Asterisk Application API, namely the Asterisk Gateway Interface (AGI), and an Application module. After discussing the architectures of each method, I will implement a simple service in each as an example.

3.1 An AGI service

The Asterisk Gateway Interface provides a standard API for communication between a script or executable, and the Asterisk PBX. As the name states, it is a "gateway interface": a standard for external programs to interface with the PBX. The external program can perform whatever tasks are necessary, and can send and receive messages from the PBX by simply reading from its standard input or writing to its standard output. The program can be written in any language, and may perform any task that is possible in that language. As far as security concerns go, they are no more a problem than any other system which executes external scripts - the scripts to be executed are explicitly specified by the administrator, and should only be used if he is sure that they are secure.

When the script is ready, it can send a command back to the PBX such as "`say_number(10)`", which instructs the PBX to dictate the number "ten" to the user who has dialled in. While the

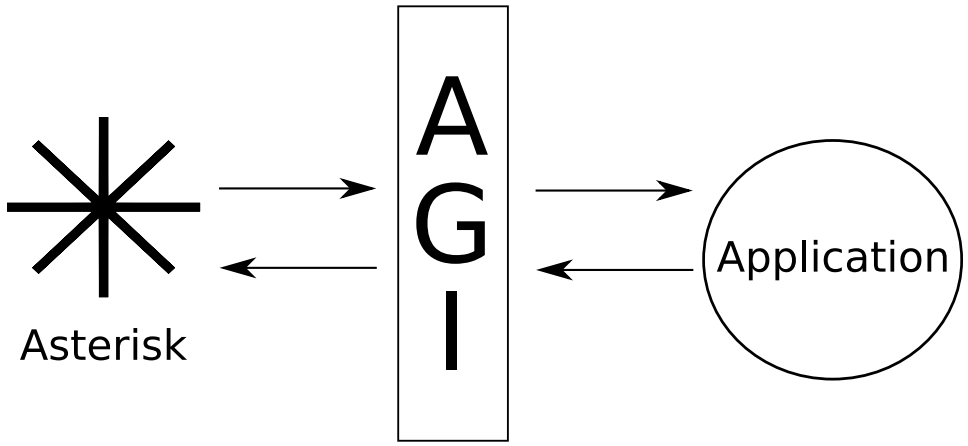

Figure 3.1: AGI Architecture

influence that AGI scripts have over the PBX is limited by the number of commands available in the AGI command set (even though this number is fairly high), they are highly flexible in terms of their actual functionality - they can be tailored for the developer's preference in language and environment, and can take advantage of any feature of the language.

3.1.1 Architecture of AGI

Figure 3.1 illustrates the simplicity and flexibility of the AGI architecture.

The external script can be implemented in any language, and perform any task. The only restrictions that are placed on it are concerning what it writes to its standard output stream.

When the script starts up, Asterisk writes a number of lines to its standard input stream, containing information about the current environment. An example of this is:

```
agi_request: testscript.agi
agi_channel: Zap/2
agi_language: en
agi_type: Zap
agi_callerid: 7510
```

```
agi_context: default
agi_extension: 5000
```

The AGI script can read these in, and determine, for example, who has placed the call that has caused the script to run - this information is in the "agi_callerid" header, and is 7510 in this case.

After this, the script can do anything it wants to - create a connection to a remote webserver and download information, or open a file and read in the contents, for example. When it needs to, it can send a command to the PBX to instruct it to perform an action. It does this by writing the command to its standard output stream, for example:

```
HANGUP Zap/2
```

Asterisk will perform this command (in this case, terminating the connection on the Zap/2 channel), and then return the status to the script on its standard input stream:

```
200 result=1
```

In this case, the result was successful.

As one can see, since no restrictions are placed on the implementation or execution of the AGI script, some very complex tasks are possible. As mentioned before, security is not an issue unless the administrator uses AGI scripts which he has not checked, or does not trust. The only real restriction is on the interaction between the script and the PBX: the command set is slightly limited (just over twenty commands currently), although the "Exec" command does allow an AGI script to execute an Application (discussed later in this chapter), which means that it does have limited access to the full power of Application modules.

3.1.2 Example of AGI

The code for the weather.agi script can be found in Appendix A. The transcript of the AGI session is as follows:

```
agi_request: weather.agi
agi_channel: IAX2[7510@146.231.121.171:4569]/4
agi_language: en
agi_type: IAX2
agi_uniqueid: 1102682204.384
agi_callerid: "7510" <7510>
agi_dnid: 3450
agi_rdnis: unknown
agi_context: local
agi_extension: 3450
agi_priority: 1
agi_enhanced: 0.0
agi_accountcode: iax

 STREAM FILE weather/currently ""
200 Result=0
 STREAM FILE weather/temperature ""
200 Result=0
 SAY NUMBER 24 ""
200 Result=0
 STREAM FILE weather/point ""
200 Result=0
 SAY DIGITS 44 ""
200 Result=0
 STREAM FILE weather/degrees ""
200 Result=0
 STREAM FILE weather/windchill ""
200 Result=0
 SAY NUMBER 24 ""
200 Result=0
 STREAM FILE weather/point ""
200 Result=0
 SAY DIGITS 44 ""
200 ResultA.2 The =0
```

```
 STREAM FILE weather/degrees ""
200 Result=0
 STREAM FILE weather/windspeed ""
200 Result=0
 SAY NUMBER 4 ""
200 Result=0
 STREAM FILE weather/kph ""
200 Result=0
```

As can be seen, the actual AGI transaction is very simple, consisting of nothing but a sequence of requests that the PBX stream an audio file, or use its text-to-speech application to pronounce a number.

In the above transaction, the headers are received, and read in by the AGI script, which then immediately starts making AGI requests back to the PBX. The `STREAM FILE` command requests that a pre-recorded message be played. The `SAY NUMBER` command requests that a number be read out, and the `SAY DIGITS` command requests that digits be read out in order. Thus, the first six AGI commands in the above transaction result in the user hearing the following message: "In Grahamstown currently: The temperature is twenty-four point four four degrees." The rest of the AGI transaction produces similar results.

The actual work is done of this service is done behind the AGI, by the code that downloads the weather data from the internet, parses it, and turns it into the numbers that it then passes to the PBX via AGI. An excerpt of this code is given below, to illustrate the comparative complexity:

```
my $ua = new LWP::UserAgent;
my $datapage='http://www.ru.ac.za/weather/ARCHIVE/CURRE'.
               'NT5MIN/current-000';
my $req = GET $datapage;
my $res = $ua->request($req);
my $retval = "";
my $html = $res->as_string;
$html =~ m/D,(.*)\n/;
$html = $1;
my ($date, $time, $temp, $hum, $baro, $wdir, $wspd, $wshi,
```

```
    $rf_day, $srad, $batt, $chill) = split /,/, $html;
my $T = ($temp-32)*(5/9);
my $W = ($chill-32)*(5/9);
my $R = $hum;
```

This code fetches a webpage of raw weather data, parses it, and extracts the relevant information. This information is then usA.2 The A.2 The ed to construct the AGI commands that are sent in the AGI transaction above.

The important point is that the code in the AGI script could do anything from making an XML-RPC request to examining the current system status of the machine it is running on. The AGI command set may be limited, but there is no limit to what can be done behind the AGI.

3.2 An Application Service

The commands used to construct the dialplan described in chapter 2 are known as *Application Commands*. While Asterisk comes with a large number of these commands "built in", each one is actually a stand-alone module, written in C, and dynamically loaded at runtime. As such, it is possible to write new Applications and plug-them into Asterisk, calling them directly from the dialplan. Because they are loaded as dynamic libraries, they get linked with the other Asterisk runtime libraries, and can thus make calls to the functions in those libraries. This gives them a high degree of low-level access to all aspects of Asterisk, and makes them very powerful.

Unfortunately, there is no documentation available for the Application API as yet. The source code for all of the existing Asterisk Applications is available, along with the rest of Asterisk, of course, and much can be learned of the API through inspection of this code, but there is no formal, easy-to-study API. As such, a lot of the work when developing in this area involves reading the source code for other applications, and using trial and error to work functions out. However, the source code is very clearly written and easy to read, and this task is not as hard as it could be.

3.2.1 Architecture of an Application

An Application is not a standalone program - it is written in C, but is compiled to object code, and linked to other libraries, and then loaded dynamically by Asterisk when necessary. The source code for each Application has to have certain parts in order to work. To be more precise, there are six functions which must be defined by each Application, so that Asterisk can execute them when necessary. These functions are called `unload_module()`, `load_module()`, `description()`, `usecount()`, `key()`, and another function which can be named anything, and which holds the bulk of the Application's functionality. I will refer to this last function as the `exec()` function.

The first two functions are similar to the constructors and deconstructors in object oriented programming. Any tasks that need to be done when the Application is loaded or unloaded can be placed in here. Typically, these functions make calls to the Asterisk library functions `ast_register_application()` and `ast_unregister_application()`, to insert themselves into (or remove themselves from) the list of available Applications that can be used in a dialplan. The former function also gives Asterisk the name of the command which will be used to invoke it from the dialplan, a brief description of the function, and, importantly, a pointer to the last function mentioned above, which contains the actual functionality of the Application. This is how Asterisk knows what to call when the Application is invoked.

The next three functions simply return, respectively, the description of the module, the number of times it is in use, and a key which is used to ensure that the code is licensed under the GNU General Public License.

The `exec`() function is where the actual work is done. When a dialplan entry invokes the Application, this function is executed, and passed the current channel that the dialplan is working on, and any parameters passed to the Application from the dialplan. It can then perform any tasks it wants to, including calling various Asterisk library functions. When this function returns, the dialplan will continue with the next command.

Having written an Application in C, the source code is compiled to object code, and the file containing this code is placed in a special directory where Asterisk looks for loadable modules. When Asterisk is started, it scans this directory, and makes a list of the Applications available, for use in the dialplan.

3.2.2 Example of an Application

In this section, I will illustrate the development of a voice service using the Application API. To some extent, this service will serve as an introduction to chapter 5, since it will be the method by which Asterisk communicates with the service framework which it discusses.

The service developed here will be called the `Juke` service: it will act as a sort of music player. When a user dials a certain number on the Asterisk PBX, music will be streamed back to her. The music is controlled using the DTMF (Dual Tone Multi Frequency) keys (that is, by pressing certain numbers on the phone's keypad). For example, pressing `5` will pause or un-pause the music, pressing `3` will advance to the next song, and so on. The service will spawn an external application, a standalone music player, which will take care of actually decoding the music files - the service will act as a proxy between Asterisk and the music player.

The basic structure of the `exec()` function is as follows:

```
juke_exec(chan, data) {
    initialisation
    spawn player
    loop {
       test for music data from player:
          stream music to channel
       test for DTMF data from the user:
          send appropriate command to player
    } while not hungup
    deinitialisation
}
```

In the initialisation, pipes are created for communication with the music player, and the channel is configured so that audio data can be written to it. Because this is being written in C, it is necessary to make sure that attempting to read data from the player does not block if there is no data to be read. This is done by setting the `NON_BLOCK` status flag on the file descriptor from which we will read:

```
res = fcntl(fromappfd, F_GETFL);
res |= O_NONBLOCK;
fcntl(fromappfd, F_SETFL, res);
```

Once this has been done, any attempt to read from the file descriptor will return zero straight away, if there are no bytes to be read - the Application can then continue testing for DTMF keypresses, without needing to wait until more sound data is available.

3.2.3 The Music Player

The Application described above connects to what has been referred to as "the music player". The music is stored on disk in the MP3 format, and will need to be decoded into raw PCM (Pulse Coded Modulation) audio data. In other parts of Asterisk (such as the music-on-hold system, and an example Asterisk application called `mp3`), the external application `mpg123` is spawned to do this. This program can be invoked with certain options that cause it to write the audio data it decodes directly to the standard output stream, instead of writing to the sound device of the machine. The `mp3` example application calls `mpg123` with this option, and writes the audio data it reads from `mpg123`'s standard output to the current Asterisk channel, thus playing the sound back to the user. This is similar to the function of the `Juke` application, but `Juke` needs more control. For example, it needs to know when the music file has been completely decoded, so that it can tell `mpg123` to load the next song. It also needs a fine grain of control over `mpg123`, such as the ability to pause the playback, and so on. These features are possible to implement purely within the C code of the Application, but they get very difficult to maintain. In addition, it is not really necessary to solve the problem this way. The only events that ever occur, from the `Juke` perspective, are the starting of the service, and when the user presses a DTMF key. The first is already taken care of, so all that `Juke` should need to do from then on is pass on the commands received from the user (via DTMF keys) to the music player.

The solution is to place a Perl script between the `Juke` application and `mpg123`, so that it can mediate between them - accepting the simple commands from `Juke`, and passing them on to `mpg123`, but also monitoring `mpg123`, and dealing with events such as the end of a song.

There was, however, a complication. The `mpg123` application can be told to write its audio data either to standard output, or to a file, instead of to the sound card as usual. This is necessary

for this script, so that it can capture the audio data and pass it on to Asterisk. However, when `mpg123` is also being run in "remote control" mode (i.e. so that other applications, such as this very script under discussion, can issue it commands to pause playback, and so on), it writes the current status of the song (what frame it is currently decoding, whether it is playing or paused, and so on) to its standard output. This was a bad design decision on the part of the `mpg123` developers, because it means that the status is dumped right into the middle of raw audio data which is also being sent to standard output. (The correct way of dealing with status information in such a situation is to print it to standard error.) I attempted to get around this problem by telling `mpg123` to write to a FIFO (First In First Out) special file, but there were buffering problems, and this seemed an inelegant solution in any case. The solution, in the end, was to patch the source code of `mpg123` to make it write status information to standard error. This proved to be simple and effective.

The basic structure of this script is as follows:

```
spawn mpg123
initialisation
repeat:
    if data read from stdin:

        respond to command (possibly by passing on to mpg123)
    if data read from mpg123 stdout:
        write audio data to stdout
    if data read from mpg123 stderr:
        interpret status, and react if necessary
```

`Initialisation` in this case involves setting up the playlist, and other internal variables, but most importantly, the script needs to ensure that it will not block if it cannot read data at any stage. It does this in the same way as the Application does, but, of course, in Perl, and not C:

```
autoflush RDF 1;
fcntl(RDF, F_GETFL, $flags)
    or die "Couldn't get flags for rdf: $!\n";
```

```
$flags |= O_NONBLOCK;
fcntl(RDF, F_SETFL, $flags)
    or die "Couldn't set flags for rdf: $!\n";
```

It needs to do this for each of the three file handles from which it needs to read (two for the standard output and standard error of the `mpg123` process, and one for its own standard input). Additionally, when it reads from the file handles, the following code is necessary:

```
if($errb !~ m/\n/) {
    $rv = sysread(ERRF, $errb, 1024, length($errb));
    undef $errd;
}
else {
    ($errd, $errb) = split /\n/, $errb, 2;
}
```

The `$errd` variable will now contain the latest whole line of output, if there is one, and will otherwise be undefined, making for easy checking. When reading the audio data from the standard output of `mpg123`, things are simpler, since this just needs to be read and passed straight on (without being split into whole lines for analysis):

```
if($rv = sysread(RDF, $rdb, 1024, 0)) {
    print STDOUT $rdb;
}
```

In the pseudo-code above, `respond to command` generally means passing the command straight on to `mpg123`, such as the `pause`, `stop`, and `play` commands. However, some commands, such as the `next song` command, or a playlist-editing command, would need the script to alter its own state, and issue some extra commands to `mpg123`, to tell it to load a new song, for example.

Similarly, `interpret status` generally requires no action on the scripts part. However, occasionally, the status received from `mpg123` will indicate that the song is finished, and the script should then instruct `mpg123` to load and play the next song.

This script, combined with the C code of the Application, form the `Juke` service, which will form the basis of the next two chapters. An important thing to note about this script is the lengths taken to ensure that it did not block on input - later on, when the Twisted framework is used, this will cease to be an issue.

3.3 AGI vs Applications

It should be noted at this point the difference between AGI and an Application. In order to play sound, the AGI can simply send the `STREAM FILE` command. However, this command is limited in that it will play the sound from the file until keys are pressed, and then will return. There is no way for AGI to interact with this process in detail. If, for example, it wanted to start streaming the file from ten seconds in, there would be no way to achieve this. This is because, as described before, AGI is limited by the commands available. It can use the `EXEC` command to call an Application to perform a task for it, of course, but this is just deferring the problem. AGI sacrifices control for ease of use.

By contrast, the only way for an Application to play sound is as follows. First, it reads a chunk of raw sound data from a source (which, additionally, needs to decode the sound from the file first - another thing AGI does not need to do):

```
res = read(fromappfd, myf.frdata, sizeof(myf.frdata));
```

Then it has to construct a voice frame, and write this frame to the channel:

```
myf.f.frametype = AST_FRAME_VOICE;
myf.f.subclass = AST_FORMAT_SLINEAR;
myf.f.datalen = res;
myf.f.samples = res / 2;
myf.f.mallocd = 0;
```

```
myf.f.offset = AST_FRIENDLY_OFFSET;
myf.f.src = __PRETTY_FUNCTION__;
myf.f.delivery.tv_sec = 0;
myf.f.delivery.tv_usec = 0;
myf.f.data = myf.frdata;
ast_write(chan, &myf.f);
```

And that is just one frame. The advantage of this, of course, is that in between writing frames, it can check for any DTMF keypresses:

```
ms = ast_waitfor(chan, 1000);
if (ms) {
    f = ast_read(chan);
    if (f->frametype == AST_FRAME_DTMF) {

       // the ASCII value of the DTMF key pressed
       // is stored in f->subclass
    }
    ast_frfree(f);
```

Clearly, on this issue, the AGI versus Application distinction is the same as any ease-of-use versus flexibility, power and control debate. However, another issue to take into account is process overhead. AGI services are spawned as separate processes, which communicate back to Asterisk via AGI. Applications, however, are loaded dynamically as modules of Asterisk itself, running directly within the process. This makes them much more efficient in terms of processing overhead, and memory. Additionally, because they are written in low-level C, they are going to be fairly efficient and quick, although this is not such an important point, since AGI scripts can be written in any language, from C to Assembly to COBOL. The example given above also indicates that sometimes an Application will spawn separate processes itself, eliminating this difference. However, for high-use services that might need to run as efficiently as they can, Applications are definitely more desirable.

Clearly, though, the best solution in most situations will be some combination of AGI and Applications: AGI scripts using `EXEC` to call custom-made Applications to do certain tasks, and the rest being done easily with the AGI interface.

3.4 Summary

In this chapter, two sample services were created for the Asterisk PBX, using the Asterisk Gateway Interface (AGI), and the Application API. These services are examples of the possible ways for services to plug into Asterisk, and their advantages and disadvantages are discussed, and compared.

Chapter 4

Twisted and Nevow

Before continuing to report on the work done for this thesis, a brief detour is necessary to describe the Python/Twisted framework [Labs 2004]. As with Asterisk, this chapter is the result of active experimentation with this sytem, and investigation into its architecture and inner workings.

The Python scripting language [Python 2004] has recently gained popularity as a powerful, yet easy-to-use language for nearly all aspects of computer science, from embedded systems to web scripts to large high-end graphical applications. The Twisted framework is

> a framework, written in Python, for writing networked applications. It includes implementations of a number of commonly used network services such as a web server, an IRC chat server, a mail server, a relational database interface and an object broker. Developers can build applications using all of these services as well as custom services that they write themselves. Twisted also includes a user authentication system that controls access to services and provides services with user context information to implement their own security models.

The Twisted framework is still under active development, and is an advanced type of system. There is little documentation on the system apart from some user-contributed "how to" documents, and a few brief tutorials. For this reason, a good deal of the work done consisted of investigating the Twisted framework and finding out how it worked.

This chapter outlines how Twisted applications work, and how the framework can be used to simplify the networking side of an application, with special emphasis on Perspective Broker - the remote-procedure call system built on top of Twisted. It will also briefly discuss Nevow - a web templating system written using Twisted.

4.1 The Twisted Framework

At face value, Python is a straight-forward scripting language, albeit one with a good object orientation model and a powerful set of libraries. While Twisted can be thought of as a library, containing a number of functions which the programmer's application can call, this is not how it was intended to be used. The Twisted framework turns Python into a full event-driven execution environment. When a Twisted application runs, it is the Twisted code that does most of the work, only occasionally calling the programmer's functions.

The unit that takes care of much of this work and makes the calls to the programmer's functions is the `Reactor`.

4.1.1 The Reactor

The Reactor is a class which contains an event loop. This loop is where a Twisted application spends most of its time. Prior to telling the reactor to "run" (i.e. to enter the loop), however, a lot of setting up and initialising needs to be done. This involves creating various services within the reactor, and setting their parameters. A service is generally something that uses a specific transport, and either connects to somewhere else via that transport (as a client), or listens for connections (as a server). There are generic transport classes in the Twisted framework which can be overridden to create new types of service, but there are already complete implementations of all the usual types - clients and servers for TCP, UDP, Unix domain sockets, inter-process communication, and so on.

In Twisted, everything runs asynchronously. Whenever an action is requested (for example, creating a TCP connection to a remote server as described above), the action is simply added to a list of actions that need to happen. There are also methods for scheduling an action for a later time, or with a delay. The reactor's event loop keeps track of this list, and performs the

actions as their time arrives, or as soon as possible if no time has been specified. In this way, there is never a situation where the entire application is blocked by one action - if the action cannot be completed at the present time, it is simply rescheduled for later, or pushed further down the list. This is often achieved by using *deferreds*, described below. What is actually being done during the service initialisation described above is the scheduling of several actions to be performed immediately: for example, if a TCP client service is created, then a request to connect to a remote server is added to the list of actions. In addition, there are regularly scheduled actions which the programmer never sees - these continually watch for activity on the various connections that have been made, and when something does occur, the required behaviour (also specified during initialisation) is added to the event list, to be performed immediately (or as soon as possible).

Once all of the initialisation has been performed, all that remains is to tell the reactor to start the event loop. The basic event loop simply fetches the next scheduled action and performs it. Twisted has several different reactors which can be used according to the type of application. For example, there is the `GTKReactor`, which is used when writing GUI applications using the GTK graphical toolkit. It is necessary to have a separate reactor for these because the GTK toolkit works on a very similar basis to the Twisted framework - the graphical widgets are created, and a set of callbacks are specified, and then the GTK event loop runs, checking for events such as button presses and mouse movements. Since it is not possible to have two event loops, the *GTKReactor* includes all the functionality of the GTK event loop (which is very simple in itself), along with the standard Twisted reactor functionality. In addition to this, the reactor needs to check for activity on the various files, sockets, and so on, that have been opened. There is an implementation of a *"PollReactor"* and a *"SelectReactor"*, if the programmer wishes to use a specific method for checking for activity on the handles, otherwise Twisted will use the system default.

Because Twisted takes care of creating sockets, opening files, connecting and disconnecting streams, and so on, and continually checks for activity without blocking, the programmer is able to focus on the actual functionality that he wishes to create, without worrying about complicated socket code, or the need to ensure that his application will not block. His focus is entirely on the behaviour he desires his application to exhibit in response to certain events, or as the result of activity in other parts of the Twisted framework.

4.1.2 Configuration

As explained at the Tenth International Python Conference,

> The formats of configuration files have shown two visible trends over the years. On the one hand, more and more programmability has been added, until sometimes they become a new language. The extreme end of this trend is using a regular programming language, such as Python, as the configuration language. On the other hand, some configuration files became more and more machine editable, until they become a miniature database formates. The extreme end of that trend is using a generic database tool.
>
> Both trends stem from the same rationale – the need to use a powerful general purpose tool instead of hacking domain specific languages. Domain specific languages are usually ad-hoc and not well designed, having neither the power of general purpose languages nor the predictable machine editable format of generic databases [Zadka and Lefkowitz 2002].

To illustrate, take the following excerpt of code, which has the general form of nearly all Twisted applications:

```
import twisted.internet import reactor
reactor.connectTCP("remotehost", 8800, ClientFactory())
reactor.listenTCP(8801, ServerFactory())
reactor.run()
```

The Twisted modules are imported, the reactor is instructed to connect using TCP to a remote host on port 8800, using a certain client factory (factories will be covered in the next section), and to listen for TCP connections on port 8801, using a certain server factory. Then the reactor simply runs.

This excerpt of Python code describes the configuration of the Twisted application: it sets it up and defines its behaviour. The same goal could be achieved by describing the services required in an XML file, or in some binary format. This configuration, combined with the current contents of the event loop, will define the state of any Twisted application at a given point in time.

Twisted combines the two approaches to configuration described above, being able to read application configuration either from a Python file, or from a "pickled" serialization such as an XML document. ("Pickling" is the term used to describe the encoding of the data structures involved in configuration into a "flat" format such as binary or XML.)

The `twistd` utility uses these configurations to start and restart applications. The Twisted applications can be constructed and initialised, and then serialised onto disk in whatever form fits the requirements. `twistd` will load the application and start or continue its event loop. If the application gets interrupted, twistd will serialise its current state back onto disk, and resume where it left off later. Furthermore, twistd allows a high level control over how the application runs. For example, it has the ability to daemonize it to run in the background, or keep logs of the output.

4.1.3 Factories and Protocols

One of the most important things required in setting up a service is the "protocol" - this dictates what the behaviour of the application will be when the service runs. When a service is created, as above, it is configured to use a certain factory. These are so called because they manufacture protocols when connections are made. When a client service is created, it first connects to the remote server, and then spawns an instance of its associated protocol to handle all the events generated by the connection. When a server is created, it starts listening for remote connections, and whenever one is made, it creates a new instance of its protocol to handle the events. A protocol is basically a set of event handlers: it is a class that defines a method to handle, for example, the "connection made" event, one to handle the "data received" event, and one to handle the "connection lost" event. Obviously different types of service will generate different events: there is a protocol written to act as a file transfer protocol (FTP) client, and this protocol defines the `retr` (file retrieval request), `pwd` (present working directory request), `quit` (termination request) events, and so on.

When a service is created, it gets initialised with a factory, which has an associated protocol. This protocol defines various callback methods for various events, and these callbacks are where the programmer puts all his work. In the "connection made" event, the programmer needs to put only code which does what he wants to happen when a connection is made. Thus, he doesn't need to worry about actually making the connection, handling concurrent connections, or anything else

- he merely defines precisely that behaviour that needs to occur on certain events. If events don't require any extra behaviour, the programmer can just ignore the callbacks.

Here is an example of a client factory and protocol which implements a simple IRC client:

```
class TestIRCClient(twisted.protocols.irc.IRCClient):
    def privmsg(self, user, channel, msg):
        self.say(channel, "The person '%s' said '%s'" % (user, msg))
class TestIRCClientFactory(twisted.internet.protocol.ClientFactory):
    protocol = TestIRCClient
```

The `TestIRCClientFactory` can be specified as the factory to be used for a normal TCP connection to an Internet Relay Chat (IRC) server:

```
reactor.connectTCP("irc.freenode.net", 6667, TestIRCClientFactory())
```

When the reactor runs, the service will connect to that server, on that port, and create the client factory. When the connection is created, an instance of the `TestIRCClient` protocol will be created. This inherits all the behaviour of Twisted's `IRCClient` protocol, which handles events such as server pings and status messages automatically. The `IRCClient` protocol overrides the `LineReceiver` protocol, which in turn overrides the `TCPClient` protocol. At each level, some simple event handling is done. For example, TCP clients know how to deal with connections and raw data reception; the `LineReceiver` knows how to parse this raw data for complete lines of text (very useful for handling text-based protocols, such as IRC, SMTP or HTTP). The `IRCClient` protocol then uses these complete lines of text to communicate with the IRC server using the fairly basic IRC protocol. Thus, when it receives a line that says, for example:

```
:George!george@test.com PRIVMSG TestUser :Hello there
```

It interprets this as a private message from George (address "george@test.com") saying "Hello there". The `IRCClient` protocol then generates its own event *"*`privmsg`*"*, which is given a default empty event handler. A programmer may override this event handler when he sub-classes

the `IRCClient` protocol. In the example above, the `TestIRCClient` protocol did override it, and defined the resulting behaviour, to say what the message was and who said it. Thus, with only one line of non-boiler-plate code, a fully functional IRC client has been created (although, admittedly, the only thing it does is echo back everything said to it).

4.1.4 Deferreds

As noted above, Twisted is asynchronous: no part of the application will block while waiting for another part. This is difficult to achieve, since some functions need data which is not immediately available. The solution to this problem is achieved by using an object called a `deferred`. A `deferred` is simply an object that maintains a list of things that need to be done when certain results have been obtained.

For example, if one has a function called `getTemperature()`, which queries a website about the current temperature, the result will clearly not be immediately available. However, we do not want the program to block until it is. We use `deferreds` to solve the problem as follows:

```
d = getTemperature()
d.addCallback(printTemperature)
```

The return value from `getTemperature()` is a deferred object. When the data arrives back from the remote web-server, the deferred will "fire" with the results. That is, its callbacks will be called, in order, and passed its results. As one can see above, the `printTemperature()` function is added as the first callback, so when the results do arrive, they will be passed as arguments to this function, which can print them. Thus, the application can get on with everything else it has to do, without blocking on this web request. *Deferreds* are used often in Twisted, since most activity is network-based, and one cannot be sure when it takes place.

A more detailed example follows:

```
class AdditionProtocol(twisted.protocols.basic.LineReceiver):
    self.deferreds = []
```

```
        def addNumbers(x,y):
            d = defer.Deferred()
            self.deferreds.append(d)
            self.transport.write("calc %s: %s+%s" %
                        (self.deferreds.len(),x,y))
            return d
        def lineReceived(data):
            m = re.match("result (\d+): (\d+)")
            if m:
          self.deferreds[int(m.group(1))].callback(int(m.group(2)))

      ...

    d = a.addNumbers(3,5)
    d.addCallback(printResult)
```

The `AdditionProtocol` class creates a deferred objects, and adds it to its list. It then sends a network request off for two numbers to be added together (the request is numbered so that it can be linked back to the correct deferred object). It then simply returns the deferred, and everything else continues as normal. When it receives the result from the network, it fires the correct deferred object with the result received from the network. As shown at the end of the excerpt, this result will get passed to the `printResult()` function when it is available.

Using this mechanism, the application can be protected from any activity that might block - most commonly, communication over the network.

4.1.5 Authentication in Twisted

In a normal situation, a voice service will be provided to a user who connects to a PBX using some telephony technology. These technologies have implemented methods to identify and authorize the caller, and it is presumed that if the caller has managed to connect to the service, he is sufficiently authorized (he has enough credit to use the service, for example, or has provided means of payment). However, with the extended services being developed in this thesis, a user

can connect to a service from multiple sources, using multiple methods, and will not necessarily come via the standard pre-authorized telephony channels.

For this reason, once the core service has been created, the main problem in developing these extended services is simply ensuring that the user connecting to a service by whatever means, is who she says she is. Thus, authorization methods have a vital role in the system.

Twisted has a versatile authentication system which allows many different protocols to connect, and to authenticate using many different methods. Once authentication is successful, the protocol will have only those aspects of the system which are relevant exposed to it. Take a bulletin board system as an example. If somebody connects using anonymous authentication, they will only have access to the methods involving public messages. If somebody connects using normal authentication, they will have access to a much wider range of methods, including methods to post their own messages. Finally, if somebody connects using a secure protocol **and** supplies the authentication token of the administrator, she will have access to the full range of administrative methods.

The authentication system is broken into six different objects: the *Portal*, the *Credential Checkers*, the *Credentials*, the *Realm*, the *Avatar*, and the *Mind*.

4.1.5.1 The Portal

There is only one *Portal* in an authentication system, and it contains the core of the login. This is the object to which everything else authenticates itself. Its main job is to associate the *Realm* with the *Credential Checkers*. After the *Portal* is created, various *Credential Checkers* are registered with it, and their interfaces are described. Subsequently, when authentication is attempted, the interface being used for authentication is used to find the best *Credential Checker*, and this checker is used to validate the user. Once the *Portal* has validated the user, it passes the identification token returned from the checker to the *Realm*, so that the *Realm* can create an *Avatar*, which the *Portal* will then return to the user.

4.1.5.2 The Credential Checkers and Credentials

Several *Checkers* can be associated with a *Portal*. This makes it possible to use different methods of authentication: there can be a *Checker* whose interface is a public key (to be checked against

a private key), one whose interface is a username and password, or one whose interface is a Kerberos token. Once the *Portal* has chosen the best *Checker* (that is, the one best suited to handle the *Credentials* passed to the login system), the *Checker* performs the actual authentication, using those *Credentials*. Note that the *Credentials* do not need to simply be a set of data (like a username and password). It could easily be an object which connects somewhere else, and performs some task. A good example of this is a challenge/response server.

If the authentication is successful, the *Checker* needs to determine what the correct identification token is. This will often be the username of the user, if they used a username/password pair, or it could be a generated number. Whatever the case, it will uniquely identify the user who has authenticated himself.

4.1.5.3 The Realm

The *Realm* is linked to the *Portal*, and is not involved with any of the actual authentication. Once the *Portal* has done the authentication, it passes the identification token to the *Realm*, which simply creates and returns an *Avatar*, using that token.

4.1.5.4 The Avatar

The *Avatar* is the object which represents the user who has logged in, in all regards. Everything the user needs to do is done through the *Avatar*. The *Avatar* is also tailored to the specific needs of the user, depending on how she logged in. For example, if she logged in using email authentication tokens, then the *Avatar* will contain a representation of her mailbox, with all the methods that she will need to manipulate it. If she logged in using anonymous credentials, the *Avatar* might represent a much smaller set of capabilities.

4.1.5.5 The Mind

If the *Avatar* is the representative of the user on the server, the *Mind* is the representative of the user on the client side of things: any notifications from the server which need to return to the client can be relayed through the mind. When the user logs in, the server can keep a record of the *Mind* object, and use it to send messages to the client, just as the client will use its *Avatar* to

send messages to the server. The *Mind* is often not used at all, since most systems simply use a request-response model, where the server will only respond when the client contacts it. However, Twisted does allow the use of a *Mind* if the programmer desires two-way communication.

This authentication system will be used extensively later in this thesis, to develop a method for various interfaces to authenticate to a single service.

4.2 Perspective Broker

Since this thesis is developing diverse frontends which all need to communicate with a service that is possibly running on a remote machine, remote procedure calls are of great interest to us. Given that Twisted is a framework for creating network applications, it is predictable that there is a method for remote procedure calls. This system is called Perspective Broker, and will be examined in greater detail below. First, however, the alternatives need to be examined.

4.2.1 Traditional RPC

Two common methods for remote procedure calls are XML-RPC and SOAP [Consortium 2003].

XML-RPC uses HTTP requests to a listening web-server, encapsulating the procedure arguments in an XML document, and receiving the results back in the same form. While this is a very simple method, because it uses common technologies available in a web server, this is also its main drawback. It is limited to sending and receiving datatypes defined by the XML schema, and it is restricted to the HTTP transport. Another problem is that traditional XML-RPC implementations will block until the HTTP response is received: XML-RPC is supposed to be completely transparent, and behave exactly like local procedure calls.

SOAP stands for Simple Object Access Protocol, and it is anything but simple. It is a complicated protocol that is traditionally carried over HTTP, although it is not restricted to it.

CORBA is another common method for remote object access, but it is known for being slow and bloated.

While none of these methods are ruled out - there are clients for all three systems written in most major languages, including Python - the Twisted Framework has a much more comprehensive

solution, which not only fits the Twisted model better, but also has direct support for transporting complex data types (as well as simple structures), a sophisticated authentication system, and, most importantly, two-way communication.

4.2.2 Translucent Remote Method Calls

Other remote procedure call methods claim to be *"transparent"* methods. However, Perspective Broker calls itself *"translucent"*. There is no illusion that a remote object whose methods are being called is a local object: it is always clear on which side of the network the object exists. This is, of course, done using *deferreds.*

Perspective Broker uses a client-server model to initiate communication, although once this has been done, the model becomes much more balanced, with communication flowing just as easily in both directions. The Perspective Broker server is initialised with a server factory, as is usual with Twisted (one benefit of the Twisted model is that the factory can be attached to any transport - TCP, Unix domain socket, and so on). This factory takes what is known as the *Root* object as a parameter: this is the object to which any connecting client will first get access. The *Root* object is a special type of *Referenceable* object: that is, an object which can be accessed remotely, and whose methods can be called across the network. To illustrate how *deferreds* are used for this, the following code first connects the Perspective Broker client to the server, and then gets the root object.

```
factory = pb.PBClientFactory()
reactor.connectTCP("localhost", 8789, factory)
d = factory.getRootObject()
d.addCallback(setMyRootObject)
```

The factory's *getRootObject* method returns a *deferred* which will, at some stage (when the reference to the root object is returned from the network), fire, and call the callback which we have to it, namely *setMyRootObject*, which can simply store the result if it wishes:

```
def setMyRootObject(r):
   rootobj = r
```

Later on, a remote method can be called on this root object. As usual, the call returns a deferred, to which callbacks can be added that specify what to do with the results when they are obtained:

```
d = rootobj.callRemote("add",4,5)
d.addCallback(displayResults)
 ...
def displayResults(n):
   util.println("The remote method returned the result: %d"
                                                        % n)
```

However, to demonstrate the power of *deferreds*, consider how the above code can be condensed, by chaining the callbacks together. If more than one callback is added to a *deferred*, the result of the first callback will be passed to the second callback, and so on. Thus, the above is equivalent to the following:

```
factory = pb.PBClientFactory()
reactor.connectTCP("localhost", 8789, factory)
d = factory.getRootObject()
d.addCallback(lambda r: r.callRemote("add",4,5))
d.addCallback(lambda n: "The remote method returned "+
                              "the result: %d" % n)
d.addCallback(util.println)
```

Note that, at this stage, the reactor is not even running. No network connections have been made at all. The behaviour has been set up, so that when the reactor does start running, it will perform the actions required, and when results are obtained from the network, certain things will be done. This illustrates the power of *deferreds*: they allow a programmer to specify exactly what she wants to happen *when possible*. After this, she can simply let the application run:

```
reactor.run()
```

In due course, connections will be made, requests will be sent, and results will be returned from the network. When this happens, the actions she has specified will be performed.

4.2.3 Different types of remote call

The above example shows a very simple remote method call, which could easily have been done using XML-RPC (and, in fact, there is a robust XML-RPC server implementation in the Twisted framework). However, Perspective Broker allows more than just these simple one-way calls. For a start, communication is two way: if the client passes a *Referenceable* object to the server as a parameter in a remote call, the server can then start making remote calls back to this object on the client side.

This illustrates another feature of Perspective Broker: the ability to pass complex types such as objects. Any object may be passed as a parameter, or returned from a remote call, provided there is a "mirror object" set up on the other side to receive it: it would be insecure to be able to send absolutely any object across the network, so Perspective Broker has a method by which each end can specify which types of object they are willing to receive, and what sort of class they should be instantiated as when they are received.

There are two other types of remote method call: those made on a *view*, and those made on a *perspective.* As described above, once a Perspective Broker client has authorized itself to the server, an *Avatar* is created for it on the server side. The client then communicates with the *Avatar*, requesting that various tasks be performed.

A *perspective* is a method called on the *Avatar* directly. It is so called because the *Avatar* is unique for its client - it represents that clients own unique perspective on the Perspective Broker's server. Thus, *perspective* calls can assume that they object on which they work will not be affected by any other clients, and that they have its "full attention", so to speak. (This, incidentally, is how Perspective Broker gets its name: it is most commonly used as the means to broker communication between a client and its *Avatar*, through the *perspective* method calls.)

A *view* is a method called (via the *Avatar*, since all communication is done through the *Avatar*) on another object, one that is not unique to Perspective Broker client. This is very similar to the basic remote method calls described in the previous section. The difference is that a *view* method is aware of who is calling it. Since the call comes through an *Avatar*, the *view* method receives the identity of that *Avatar*, as well as the usual method arguments. By contrast, the basic remote method calls are anonymous - they receive nothing but the method arguments.

As can be seen, Perspective Broker is extremely versatile, and has several features not normally

found in remote method calls: features that are useful when creating complex network applications that need complex communications with each other.

4.3 Nevow

Nevow is a webserver written using the Twisted framework. It is based on the Woven package that is currently a part of Twisted. Woven was a method for templating webpages, and was intended to be used as part of the Twisted webserver. However, the package was substantially reworked, and separated out into Nevow (which is "Woven" backwards, but also has the advantage of being able to be pronounced like "nuevo", which is Spanish for "new").

Nevow has many very intricate features and capabilities, which are beyond the scope of this discussion. This section serves as a brief introduction to the webserver, because it will be used later in the thesis to create an interface to a service.

4.3.1 Separation of Form and Logic

One of the fundamental aspects of Nevow is the way it separates the form of its webpages from the logic involved in generating their content. The template webpages it uses are simple XHTML documents, which contain certain nodes in the "nevow" XML namespace, which are given XML attributes which tag them with meaning for Nevow. However, because these tags are in a different namespace to the standard XHTML tags, they will be ignored by standard web-authoring tools, and web-browsers. Thus, a web designer can focus entirely on the form of the page, altering the XHTML tags to her satisfaction, while leaving the functional Nevow tags intact. Once a good-looking page has been created, it is used as a template: the Nevow tags will be replaced with proper content by Nevow later on.

To illustrate with an example, consider the XHTML:

```
<body>
  <h1>Important Page</h1>
  <strong><em>This is the <span nevow:render="title" /> page</em></strong>
</body>
```

This page can be designed and re-designed as much as is necessary, to make it look the way the web designer wants. The only important thing as far as Nevow is concerned is the empty *span* tag which contans the *nevow:render* attribute labelled *"title"*. When the page gets displayed, Nevow will reproduce it as it is, with one important difference: it will replace the contents of the tag labelled *"title"* with the output of its member function *"render_title"*. As can be seen, any number of *nevow* attributes can be included in a page, and as long as each one corresponds to a function in the source code of the Nevow object rendering it, it will be filled with the relevant content.

4.3.2 Integration with Twisted

Because Nevow is written using the Twisted framework, it integrates tightly with the other features described above. A Nevow web server can communicate with other Twisted services using Perspective Broker, and can be run under *twistd*. The same Twisted authentication can be used in a Nevow web server as in other Twisted services, for consistency.

4.4 Summary

This chapter introduced the Twisted framework: a highly versatile, easy-to-use framework for developing network applications. This framework proved very useful in developing the extended services for Asterisk, which are inherently network-based, and this chapter documents the findings of the experimentation that was performed using the framework. In addition, Perspective Broker and the Nevow system are introduced: two subsidiary projects of the Twisted framework, respectively for performing remote method execution, and providing a versatile webbased system.

Chapter 5

The Extended Juke Service

In chapter 3, two voice services were developed for an Asterisk PBX. The one on which I shall focus in this chapter is the `Juke` service, developed as an Application. In this chapter, the `Juke` service is going to be extended, so that it can be controlled from any interface, anywhere, instead of just from the telephony device of the user. In other words, while the service is running, instead of only accepting input from the user via DTMF keypresses on her handset, she will be able to interact with the service via a webpage, or a standalone application running on another machine connected to the internet.

5.1 Architecture of a remotely accessible service

This service is not distributed in the commonly understood sense of having no central point (such as in peer-to-peer systems). There is one application running centrally, to which the remote interfaces connect and communicate. This will run the bulk of the service, and the remote applications will simply present various interfaces to their users, passing the user's requests on to the central application, and receiving, in return, status information or responses. This system is analogous to the traditional server and thin client model, and I will use the terminology "server" and "client" for convenience sake.

In order for the server to be accessible from the remote clients, two things are needed: a transport method, and an identification method. In other words, a route verified as belonging to a specific user, for information to flow to and from the client is needed.

5.2 Choice of Transport and Language

There are a number of possible choices for the implementation of the communication system between the clients and the server. A common choice would be to use a simple TCP/IP socket with a simple plain-text protocol to send and receive commands. This would make it easy to develop clients in basically any language, as long as it was simply capable of creating a TCP/IP connection. However, this requires the development of a whole new protocol, including authentication methods, and this seems foolish when there are many systems already available which can be used. Another option would be to use XML-RPC, since most languages have libraries that allow XML-RPC calls. This has several problems, however. XML-RPC allows one-way communication only, so both sides would need to run XML-RPC servers, and communication becomes kludgy. In addition, in most implementations, XML-RPC calls block while they wait for a response, and this is highly undesirable for the purpose of this system. Most of the activity in this system involves remote procedure calls between the clients and the server. If the entire system had to pause while it waits for the result of these remote procedure calls, the system would be virtually unusable. A proper solution requires the ability for remote calls to be made in the background, while the rest of the system continues to run without blocking.

In the end, Perspective Broker was chosen. The Python/Twisted framework provides an extremely easy way to write network applications, and all four applications (the three clients and the server) proved fairly easy to write using this framework. Given that the applications will be written using the Twisted framework anyway, it makes sense to take full advantage of the Perspective Broker system. The most important advantage of Perspective Broker and the Twisted framework is `pb.cred`: their credential/authorization system, as described in 4.1.5.

5.3 Identification and Authentication

The second aspect of remote access is the identification and authentication of the client attempting to gain access. Clearly, an authentication system needs to be centralized, and belongs, architecturally, with the server. However, different types of client have their own specific constraints, and these will have repercussions with regard to authentication. In general, the use of different clients might require authentication at different points in the route from the client to the server, and this needs to be taken into count.

There are three different possible configurations for these clients, which can be called “indirect authentication”, “direct authentication” and “no authentication”. Some clients will not require authentication because the server knows that they are to be trusted - these clients could be thought of as “zero steps away” from the server. Secondly, some clients will need to identify directly to the server - they can be considered to be “one step away”. Finally, other clients might need to identify to some intermediary agent (the best example of this is a web application which needs to authenticate with the web server, which will then authenticate with the extended Juke server). It is not relevant how many intermediary agents there are, since the architecture and authentication system will be the same. These clients, therefore, can be thought of as “two (or more) steps away”.

Implementing three clients, one of each of the types discussed above, will serve as a sufficient proof of concept for any further clients. The details of actual client implementation may differ, but as far as communicating with the server is concerned, it will always fall into one of these three categories, and will follow a similar system.

5.4 Architecture

The final architecture of this system is shown in figure 5.1. The Player application is the “final goal” for the service frontends - they need to establish an authenticated channel to this application to send and receive commands and data.

As can be seen, Asterisk connects directly to the Player, with no authentication necessary: it is assumed that once the connection is made with a telephony technology, the user has already been authorized. This “No Authentication” method is described below in 5.5.

Secondly, the GTK Application does need to authenticate itself to the system, since it is a standalone application that cannot rely on being carried over a pre-authenticated telephony channel. It uses the `iLangaAuth` method to directly authenticate itself to the central Perspective Broker service, which then connects it to the Player application. This “Direct Authentication” method is described below in 5.6.

Finally, the Nevow page also need to authenticate itself. However, it runs inside its own server: a Nevow web server, which has its own authentication system. The solution is for the Nevow server to authenticate itself (directly) to the Perspective Broker service, using the credentials it receives

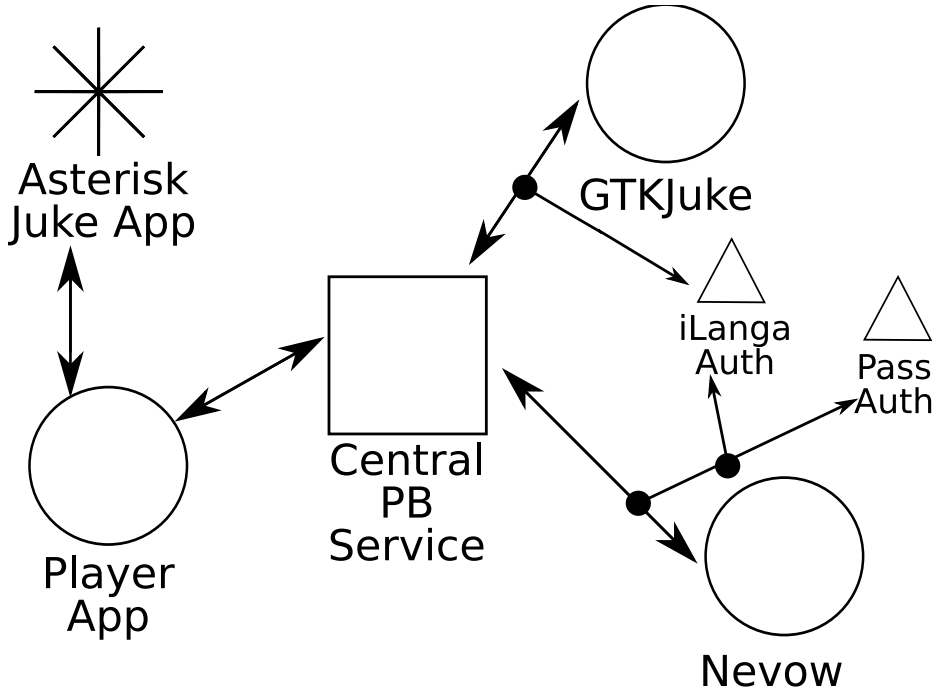

Figure 5.1: Architecture of the Juke Service

from the Nevow page. The Perspective Broker service connects the Nevow server to the Player app, which then connects the Nevow page to it. Thus, the Nevow page uses the "Pass Through" authentication method to indirectly authenticate itself to the Perspective Broker service, and gain access to the Player application. This "Indirect Authentication" method is described below in 5.7.

5.5 No Authentication: The Player App

The `Juke` service developed in the previous chapter is a good example of a client that does not require authentication. The `Juke` service is a service within the Asterisk PBX itself, and if a user is connecting through the PBX, she may be considered to be trustworthy: Asterisk has done any authentication that it considers necessary, even if all that is required to gain access is the picking up of a telephone. The service will, of course, need to be extended to incorporate the new architecture, but apart from that, the authentication details remain the same.

The development of this client in Perl has already been discussed in section 3.2.3. However, since

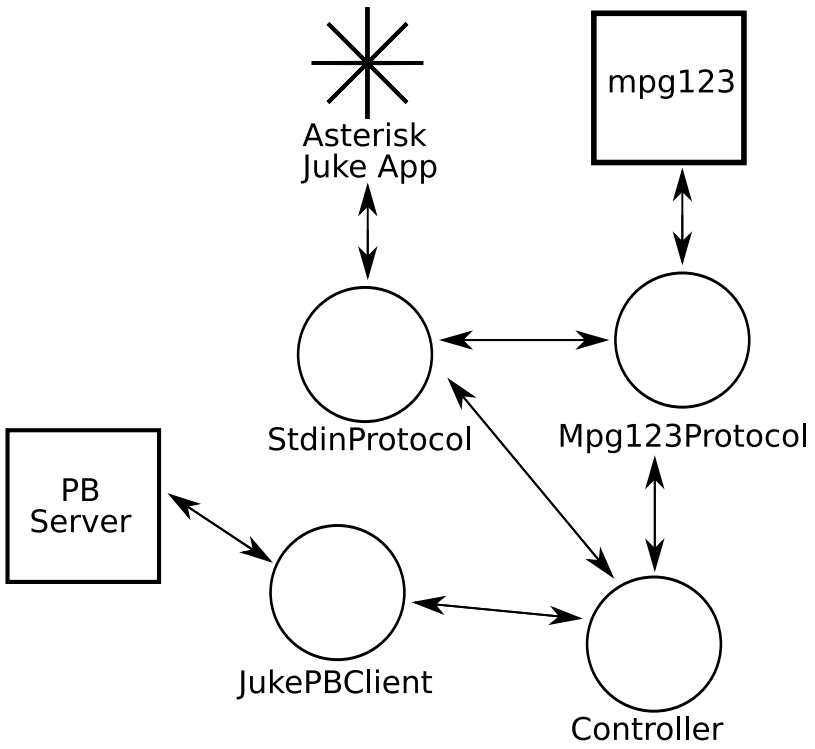

Figure 5.2: Architecture of the Player script

we want to use Perspective Broker, this script will need to be re-implemented in Python/Twisted. This is not actually too much of an issue, since Twisted's built-in asynchronicity will remove the necessity for the non-blocking reads that took up much of the Perl script.

The new implementation will set up three services: one for communicating on the standard input and output streams (to send audio data, and receive commands from the `Juke` Application), one for communicating with the `mpg123` sub-process, and one to connect to and communicate with the Perspective Broker service running within the main server component of the extended `Juke` service. These services are governed, respectively, by the `StdinProtocol`, the `Mpg123Protocol`, and the `JukePBClient`. There is a fourth class in the script, called the `Controller`, which is used to link the three protocols together. Furthermore, the `Mpg123-Protocol` is linked to the `StdinProtocol`, so that it can write the audio data it receives to the standard output stream, for the `Juke` Application to read. The architecture of the script is shown in figure 5.2.

5.5.1 The Controller

The `Controller` class's purpose is to make sure that all three protocol classes are in the same state. When it is told to *stop*, it needs to send the *stop* command to both the Perspective Broker server (via the `JukePBClient`) and the `mpg123` program (via the `Mpg123Protocol`). The class basically provides a standard way for any of the three protocols to control the playback, and ensures that all relevant classes are notified of any change in status.

When the *play* command is issued, the `Controller` uses the `JukePBClient` to make a request to the Perspective Broker server for the filename of the current song. When the response arrives, it uses the `Mpg123Protocol` to load that file into the `mpg123` program.

5.5.2 The StdinProtocol

As described in 4.1.3, protocols in Twisted are classes which dictate how interaction happens over a connection. They dictate what should happen when certain input is received on the connection to which the protocol is attached. The `StdinProtocol` is connected to the standard input and output streams via the following Python code:

```
s = StdinProtocol()
stdio.StandardIO(s)
```

The protocol itself is fairly simple. It is a subclass of the `LineReceiver` protocol, so it simply overrides the `LineReceived` method, to react when it receives the commands which the `Juke` Application will write to its standard input. All it does when it receives these commands is to call the relevant method of the `Controller` class:

```
if line == "stop": self.ctrl.stop()
if line == "play": self.ctrl.play()
```

5.5.3 The Mpg123Protocol

The `Mpg123Protocol` deals with three streams: it writes commands to the standard input stream of the child `mpg123` process, it receives audio data from the standard output stream of the process, and (thanks to the patch described in section 3.2.3) it receives status information from the standard error stream of the process.

The audio data received is sent straight to the `StdioProtocol` class, which sends it on to the `Juke` Application via the standard output stream.

The status information is parsed and interpreted. The only important information needed from `mpg123` is the message that states that the current song has finished playing, and is now stopped. Since `mpg123` can only be told to load one song at a time when in remote-control mode, this event needs to be identified, so that `mpg123` can be instructed to load and play the next song. This is achieved by identifying the end-of-song state, and then calling the `nextsong()` function of the `Controller` class.

There is, however, one complication. The `mpg123` program decodes and outputs audio data as fast as it can. When it is writing to a sound device, this is not a problem, as the sound device will only accept audio data at the speed it plays it. The perl script did not have a problem, either, since a large delay in the poll loop meant that it consumed audio data much slower than `mpg123` produced it. There are no delays in the Twisted reactor, and it consumes the audio data and writes it to the `Juke` Application as fast as it can. The result is that `mpg123` can decode and write the entire song in a matter of seconds. The problem with this is that, if the user attempts to pause the song, and the "*pause*" command gets written to `mpg123`, it has no effect, since `mpg123` has decoded and streamed the entire song already.

The solution is to keep track of how long the `Juke` Application has been playing, and how far the `mpg123` process is in decoding the file. The latter is easy enough to do: `mpg123` writes a status line every couple of milliseconds that states what frame it is currently decoding, and how far (in seconds) it is into the song. To achieve the former, a `LoopingCall` is set up in the Twisted reactor: this is simply a request to call a function every so often (every second in the case of this script). This function increases a counter (if the song is not paused or stopped), thus keeping track of how long the current song has been playing. Having obtained the two times, it is a simple matter to compare them - if the `mpg123` decoding time is getting too far ahead of the Application playing time, the `Mpg123Protocol` calls the `pauseProducing()` function inherited from

its parent `ProcessProtocol`. This stops reading data from the `mpg123` program, which thus has to stop producing it. The Application continues playing the audio data it has already received until it catches up with the decoding time, at which stage the `resumeProducing()` function can be called, and decoding continues.

5.5.4 The JukePBClient

The `JukePBClient` is connected to the remote Perspective Broker server, and passes any commands on to it. If the `Juke` Application requests that the music be paused, the *pause* command is passed on the Perspective Broker via the `JukePBClient`. Since it sometimes takes a while for the connection to be established to the Perspective Broker server, the `JukePBClient` maintains a list of enqueued commands which it sends off once the connection is made. The class also accepts remote procedure calls from the Perspective Broker server: if one of the other clients issues the *stop*, *play*, or *pause* commands, these are sent to this `JukePBClient`, and it passes them on to the `Controller`. Because the playlist is maintained on the Perspective Broker server side (and the current song is obtained using the mechanism described in the `Controller` section above), the *stop*, *play* and *pause* commands are the only ones which the `JukePBClient` needs to respond to (because the *next* command has no relevance: it is the equivalent of a *stop*, a server-side filename change, and a *play*).

5.5.5 Auto Authentication

One of the fundamental ways in which this service client differs from the others is in its method of authentication: because it comes directly from Asterisk, it is already identified, and its user ID is verified. As explained in section 4.2.3, one needs to obtain an *Avatar* in order to communicate with the Perspective Broker server, unless simple anonymous calls are sufficient. In this case, anonymous calls are not sufficient for the entire service: the server needs to be able to identify which client is making the requests for commands to be carried out. However, Perspective Broker does not actually have a way to request an *Avatar* without providing a set of credentials. The only option available was the anonymous calls. Using these, I implemented a simple anonymous remote procedure call that takes a user ID as a parameter, and returns an authenticated perspective, boot-strapping past the normal login procedure. This is discussed in detail in section 5.8.1, where the portal is overridden to accept this procedure call. As far as this client is concerned,

however, it is a matter of calling the `loginRemote()` function, and passing it the user ID received from Asterisk - the returned result of this function is a fully authenticated perspective onto the `Juke` system, which can be used to pass the commands described above.

5.6 Direct Authentication: The GTK Application

Direct authentication will be by far the most common method for communication with the service. Most direct clients will have very similar implementations: the client-side details of the interface will, of course, be different, but as far as communicating with the server is concerned, they will be identical.

The simplest example of direct authentication is a standalone application, running on a desktop machine, which connects directly to the service. It will provide a graphical interface to the user to allow the same functionality he would have if he had phoned in: that is, the interface allows the user to pause the song, advance to the next song, and so on. It would be possible to stream the audio data across the network to the standalone client, to be played on the local sound card, thus allowing an identical experience. However, this is unnecessary for the purposes of this demonstration: if the user's instructions can get to the server, and the status can be sent back, then it is clear that communication is possible.

5.6.1 GTK

GTK is the GIMP Toolkit, a toolkit for creating graphical user interfaces. It actually works in a similar way to Twisted (and, of course, every other event-driven system). *Widgets* (graphical interface entities such as text boxes, buttons, and listboxes) are created, and callback functions are specified for events that they might generate. Once this is done, the GTK event loop is started, and the application runs until callbacks are called.

The `GTKJuke` program uses GTK to create a simple interface, allowing a user to specify a username and password for identification to the `Juke` service. Once she has been identified, she is presented with buttons that allow the control of music playback, much as the DTMF keypad allows control in the `Juke` Asterisk Application.

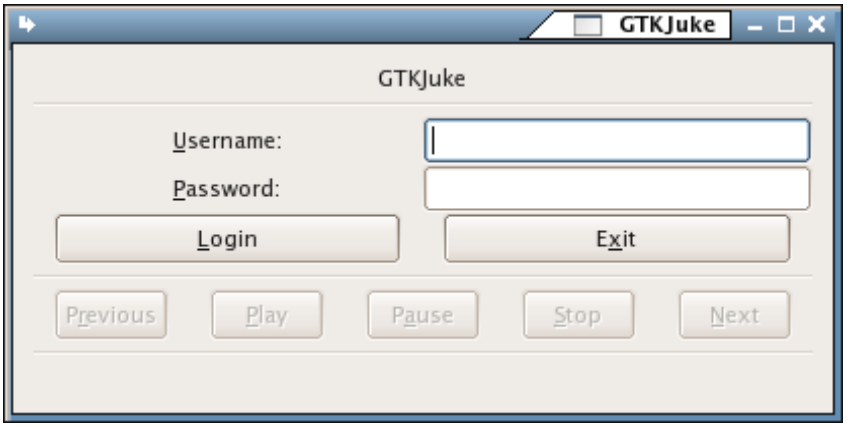

Figure 5.3: GTK Application on startup

Figure 5.4: GTK Application in use

5.6.2 The GTKJukeClient

The `GTKJukeClient` class is almost trivially simple. It inherits Perspective Broker's `Referenceable` class, which means it can be used as a Perspective Broker client, for connecting to a Perspective Broker service. When the *login* button is pressed, the identification details are passed to the `login()` function of the Perspective Broker server. If identification succeeds, the music control and logout buttons are enabled. When, for example, the *pause* button is pressed, the *pause* command is simply sent to the Perspective Broker server. When another client orders the Perspective Broker to *pause*, the `GTKJukeClient` class receives this command, and it simply updates its status box.

5.7 Indirect Authentication: The Nevow Service

As mentioned earlier, the best example of a client that requires indirect authentication is a web application. This allows an easy way to present an interface similar to the standalone graphical application (i.e. with an HTML form), which submits the user's instructions to the web server. The web server needs to authenticate the user before it will accept any commands from her. However, it needs to use the `Juke` service's authentication system. To do this, it passes the authentication credentials on to `Juke` as soon as it receives them - if they are accepted, then the user of the web application is granted access. In the same way, after successful authentication, any commands the user issues to the web application are passed on to the `Juke` service, and the response relayed back.

5.7.1 The NevowJukeClient

The `NevowJukeClient` class is a descendant of Nevow's `rend.Page` class, which means it can act as a resource which the Nevow web server serves to users. However, it also descended from Perspective Broker's `Referenceable` class, meaning that it can be used as a Perspective Broker client.

As a Nevow `Page` resource, it renders the file *`nevowjuke.html`*. As described in section 4.3.1, this file gets rendered directly, except for tags which are labelled with the `nevow:render`

attribute. There are three of these tags in the file - one for the username, one for the action which has just been performed, and one for the logout link. These tags invoke functions within the `NevowJukeClient` class which print out the username identified with, the actions which have been performed, and a link to the resource which will log the user out, respectively. Apart from those few simple dynamic elements, the page is just a static resource, which contains links for the various actions which the user can perform (*play*, *pause*, *next*, and so on). These links actually point straight back to the resource handled by this class, passing it the relevant action as a parameter. When the class is asked to render a resource with an action parameter, it uses Perspective Broker to send the associated command to the Perspective Broker server.

As a Perspective Broker client, it receives notifications of commands that other clients have executed, and adds them to a list it maintains. The next time the resource is rendered, it lists each of the commands in this list, to indicate to the user what actions have been taken by other clients.

5.7.2 Pass-through Authentication and the ReAuthChecker

The Nevow web server uses `pb.cred` for authentication and identification, in the same way that Perspective Broker does. As such, it has the same system of `Portals`, `Avatars`, `Realms` and `Checkers`. The only thing that really needs to be done to provide a specific type of identification is to create a personalised Checker. In this case, the `reAuthChecker` was created, and registered with the portal (which was an instance of the `JukePortal` class, described in section 5.8.1). This checker overrides the `requestAvatarID()` function. This function receives a set of credentials, and is supposed to return the ID of an *Avatar* that has been created for that set of credentials. The `reAuthChecker`'s implementation of this function uses the credentials that it receives as a parameter to its own login request to the Perspective Broker server. Typically, the login request returns a deferred (as described in 4.1.4). The `requestAvatarID()` function adds callbacks to this function which connect the returned perspective with the Perspective Broker client (that is, the `NevowJukeClient` class), and returns the *Avatar* ID. In this way, it performs the tasks it is supposed to do (take a set of credentials and return an *Avatar* ID), as well as identifying to the Perspective Broker server, and connecting the perspective which it receives with the Perspective Broker client.

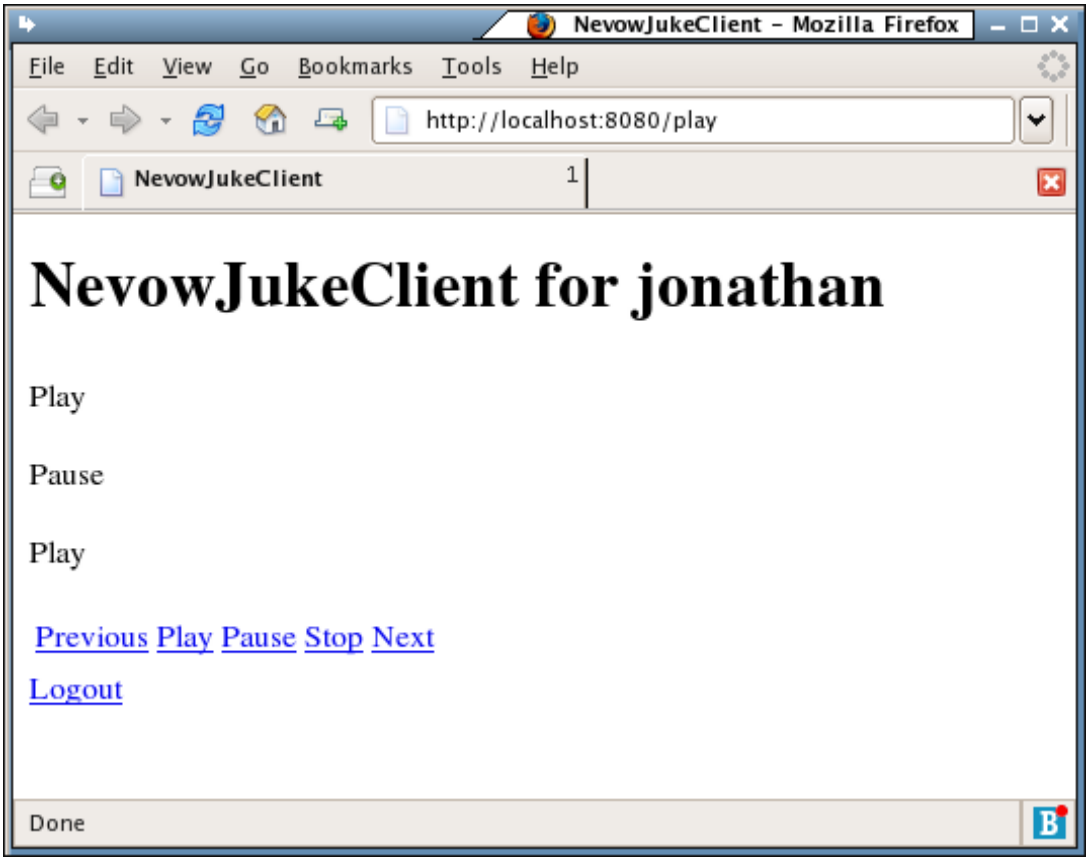

Figure 5.5: The Nevow Juke Client

5.7.3 Reverse Proxying

While the Nevow web server is a very powerful and versatile web server, it is not in common use at all. More than 70% of web servers on the internet run the Apache web server [Survey 2005]. To force a system to use Nevow simply so that the web client of the `Juke` system is available is unrealistic. To solve this problem, a system called "reverse proxying" is used. This means that when a specific resource is requested from the Apache web server, the web server relays the request on to another web server, somewhere else, and returns the response as if it were coming from the resource on the Apache web server itself. In other words, it hides access to another web server behind a specific resource served by itself.

Achieving this is simply a matter of adding the following declarations to the server configuration of the Apache web server:

```
ProxyPass /juke/ http://nevowserver:8080/
ProxyPassReverse /juke/ http://nevowserver:8080/
```

There are two declarations, because traffic flows in both directions. Each states that the "`/juke/`" resource on the Apache web server should redirect to the web server running on "`nevowserver`" on port 8080. This makes access to the Nevow web server completely transparent, and allows the system to continue running Apache as before.

5.8 The Perspective Broker Server

Having described the various clients that will be connecting to the service, all that remains for discussion is the central server itself. This server takes the form of a Perspective Broker service listening on port 8800. Normally, creating a Perspective Broker server involves subclassing the `Realm` class to provide your own `requestAvatar()` function, and often creating your own `CredentialsChecker` class to verify a username/password pair. However, this server has to be able to handle the `AutoAuth` system used in the Player application. To achieve this, the Portal class needs to be subclassed, in spite of statements in the documentation that say "You will never need to subclass Portal".

5.8.1 The JukePortal

Under normal circumstances, a Perspective Broker service is initialised either to point to a `Root` subclass, or a `Portal` subclass. In the former case, the `Root` class will provide anonymous `remote_*` methods for clients. In the latter case, the `Portal` class allows clients to authenticate, and receive a perspective and an *Avatar*, and proceed from there.

The Perspective Broker server being implemented in our case needs to have both capabilities. Without a doubt, clients will identify to the server, and require an *Avatar*. However, the Portal has no system for authenticating without a set of credentials, which is what we require. In other words, since the Player application cannot identify to the Perspective Broker server, all it can do is make anonymous calls to `remote_*` methods. The solution was to create a Portal that exposed a remotely, anonymously callable method called `loginAuto()` which took a username, and returned an *Avatar* as if the normal login process had been used. That is, under normal circumstances, a set of credentials is passed to the Portal, which authenticates the user, creates an *Avatar* associated with the user, and returns this back to the remote client. The new anonymous `loginAuto()` function must return such an *Avatar*, but without having the set of credentials which the normal Portal takes. The normal Portal login process has several hidden classes (such as `_PortalRoot`, `_PortalWrapper`, and `_PortalAuthChallenger`), the functionality of which needed to be incorporated into the `JukePortal loginAuto()` process. Once this was done, the Player application was able to acquire the Avatar as if it had logged in in the normal way.

There are a few security issues involved in this process: anybody can write a Perspective Broker client that can connect to the server without authorization. This would be avoidable by forcing some sort of authentication from the Player, but this is unnecessary. The emphasis is on identification, rather than authentication - should the service start to deal with more sensitive data (such as bank account details), some more sophisticated security can be put into place.

5.8.2 The iLangaMySQLDB checker

The credentials checker which the Portal uses for normal Perspective Broker logins is fairly simple. It connects to the MySQL database [MySQL 2004] running on the PBX, and extracts the password for the user who is attempting to log in, and then compares it with the received password.

5.8.3 The Playlist

One of the fundamental aspects of this system is that the same user will connect to it from multiple clients - she might phone in on the PBX, and open up the webpage client in her browser, or run the standalone GUI application. Normally, when Twisted applications make use of the `cred` system, they create a new *Avatar* every time a user tries to log in. However, with this system, the different *Avatar*s will need to share data about the current state, what songs are in the playlist, and so on. The solution is to use a `Playlist` class which holds all the common data, and common functionality. Every *Avatar* with the same ID is linked to one instance of a `Playlist` class, and they all make calls to the `Playlist` when they need common actions done.

5.8.4 The JukeRealm

The `Realm` is responsible for handling requests for *Avatar*s with a certain ID. The `JukeRealm` class creates a new *Avatar* for the requested ID, and then does a check for the existence of a `Playlist` associated with the *Avatar* ID. If none exists, one is created. The *Avatar* is then linked to the `Playlist` responsible for its ID and returned to the `Portal`.

5.8.5 The JukeService

The `JukeService` class acts as the *Avatar* for connecting clients. As described above, it is associated with a `Playlist` class. It exposes various services for remote procedure calls, and when it receives requests for actions to be performed from the client connected to it, it passes them on to the `Playlist`. The playlist deals with the action, and then notifies all the other `JukeService` *Avatars* that are associated with it that this action has been performed. In other words, if four clients with the same user ID are connected, their *Avatars* will all be linked to the same `Playlist`. If the first client requests that the music be paused, his *Avatar* will pass the request on to the `Playlist` class, which will notify the other three clients that the music has been paused. If any of them are Player clients, they will pause the music. The other types of client merely print out the new, paused, status of the music.

5.8.6 The Music Player

Ideally, the music player for the `Juke` service should be a part of the server - it is as much a central part of the system as the authentication. However, the Asterisk Application service (which is being used as an example for a client without authentication) is, in this case, the only client which is receiving sound. The other clients in this example are simply interfaces to control the music, but the music is only audible when the service is dialled from the Asterisk PBX. As such, it is much easier and more efficient to keep the music player in the script which is spawned by the Application - this saves having to stream sound from the server to the client.

This does not make any difference to the other clients. When, for example, the "pause" button is pressed on one client, this information is passed on to the server, which then broadcasts the "pause" command to all the clients that are connected, in case they wish to display this in their status information, or take some other form of action. If the music player was a part of the central server, the server would issue it with the "pause" command before broadcasting the command to the connected clients. However, with the music player as a part of the Asterisk client, this client will receive the broadcast commands, and pass the relevant instructions on to it. It is simply a relocation of the music player, to save the trouble of streaming audio data, and has no other repercussions on the system.

5.9 Summary

This chapter described the extended Juke service for the Asterisk PBX. Using a number of different interfaces, communicating with Perspective Broker on the Twisted framework, a service was created that allowed the user to control the playback of music from a variety of different clients. The biggest problem in the creation of these clients was creating an authenticated path of communication to the PBX, and much of this chapter covers the taxonomy of authentication methods used.

Chapter 6

The iLanga web interface

In the previous chapter, some simple Asterisk services were extended to be accessible from a variety of different devices and interfaces. Having solved this problem, the next step is to expand the very idea of a service, creating a system that allows much more powerful functionality. The services demonstrated so far have been simple and somewhat shallow in terms of their control, supplying the user with weather information and music. This chapter describes the development of a service that allows the user complete control over their telephony environment, using the various methods made available by Asterisk to communicate with the PBX. The service provides an easy-to-use, user-friendly frontend, which allows the user access to his environment independently of the telephony endpoint actually used. The general design of the service and parts of the actual implementation were done in collaboration with Jason Penton, a fellow student at Rhodes University.

6.1 iLanga and its Components

As described in 2.5, the iLanga system was built to be a “complete, cost-effective, computer based PBX”. The service developed in this chapter will plug into the iLanga system and its components. iLanga has various components which work together to form the system as a whole, but those of interest to the front-end service are the database backend, the voicemail storage system, and the Manager API.

6.1.1 The Database Backend

The PBX stores its persistent information in a MySQL database. This includes user information, device information, call details, user prepaid account balance, and routing information. The frontend will access this database to retrieve information about the user's personal profile, details of their calls, and the configuration of their telephones and other communication devices.

6.1.2 The Voicemail Storage System

The PBX includes a voicemail service, which allows callers to leave a message for users who do not answer their phones. The system records these messages and stores them in a variety of formats in files directly on the server filesystem, in a directory tree.

This voicemail will be accessed by several services of the user frontend, in order to retrieve lists of messages, to play the messages, to delete them, and to move them between folders.

6.1.3 The Manager API

As described in chapter 2, the Asterisk system can be managed through the use of a client/server protocol, implemented over TCP. This Manager can be used to obtain information about the state of the PBX, and broadcasts events that occur, such as a user hanging up the phone, or a new voicemail. The frontend will communicate with this Manager API to find out the status of the system, and to perform actions such as initiating a call with another user.

6.2 System Requirements

We determined that the frontend system would require the following features:

1. A status bar at the bottom, with the current status of the system: how many voicemails the user currently had, the current prepaid balance of the user, and how many calls had been made (Figure 6.1)

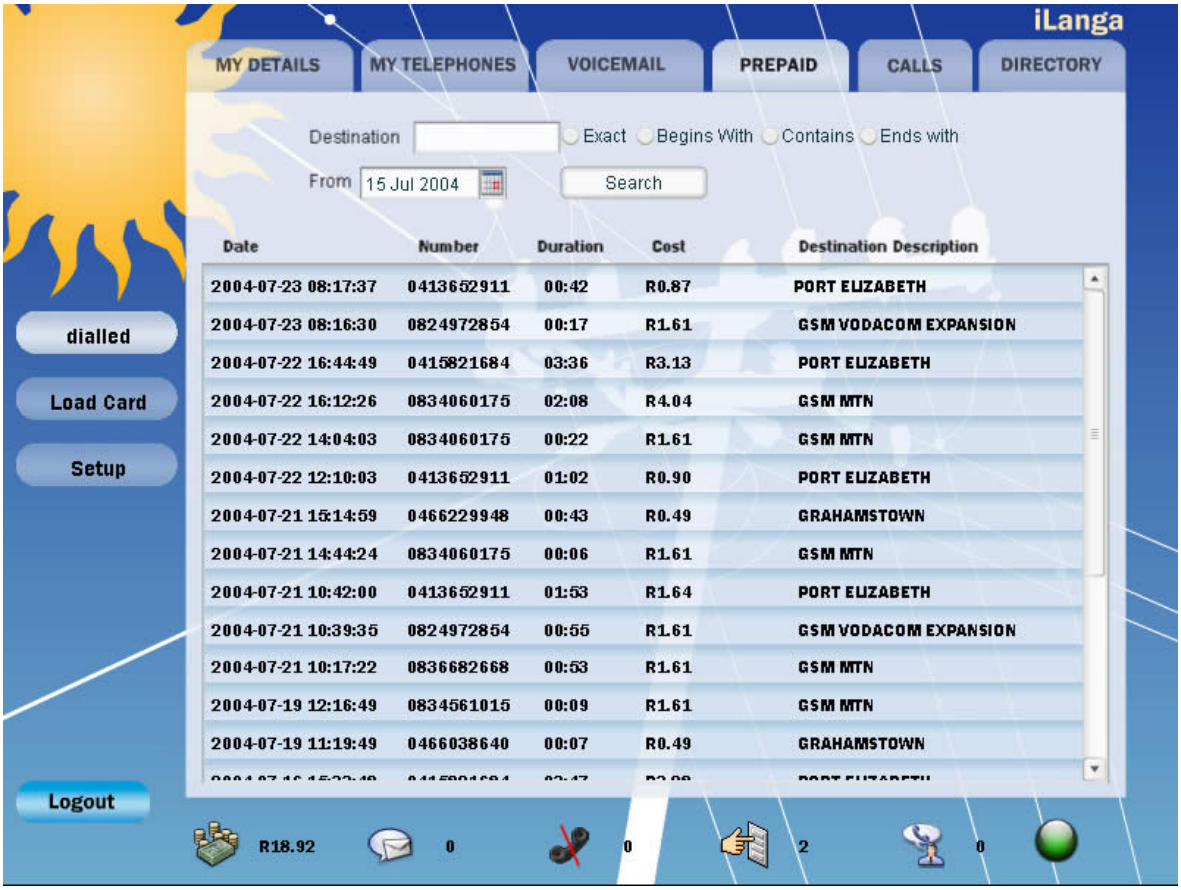

Figure 6.1: the iLanga system in use, viewing the call register

Figure 6.2: Making a call from the user directory

2. The ability to initiate calls to other users via the interface.

3. Interfaces to various services:

   a. A searchable user directory, for finding the extensions of other users (Figure6.2)

   b. A searchable list of calls made and received, including those that were charged for (Figure 6.1)

   c. Access to the voicemails the user has received, including facililites to listen to them, delete them, or move them between mailboxes (Figure 6.3)

   d. Simple personal profile configuration, including personal details (email

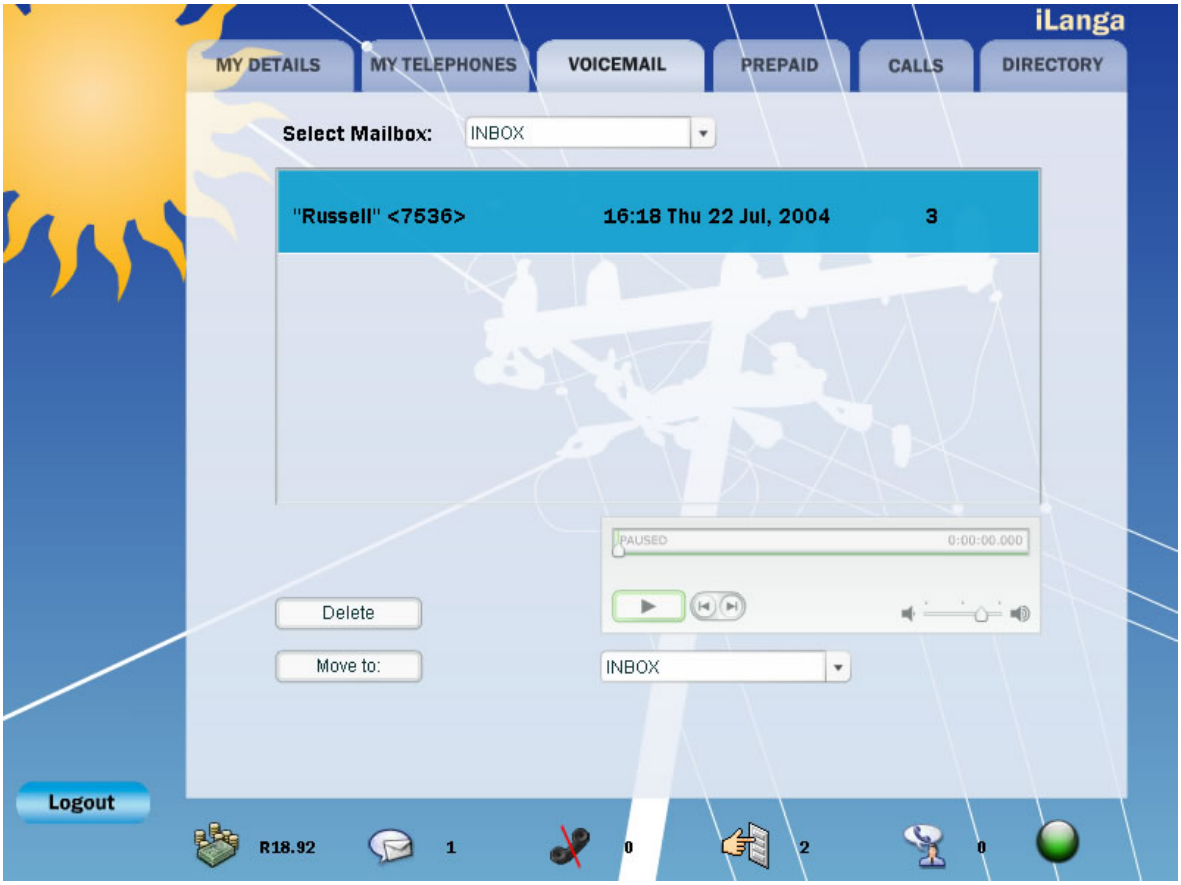

Figure 6.3: Voicemail

address, timezone, etc, Figure 6.4), and configuration of personal devices (specifying which devices should ring when one is called, for example - Figure 6.5)

From this list of features, we determined that there would be five basic communication methods required:

1. A method to be in constant communication with the server, to receive up-to-date information about the state of the system, numbers of voicemails, current balance, etc.

2. A method to make function calls directly to the system, in order, for example, to initiate a phonecall.

3. A method to retrieve once-off information about the system, required for:

    a. authentication when logging in (Figure 6.6)

    b. receiving lists of users for the directory

    c. receiving the lists of available devices for device configuration

    d. receiving the call register information

    e. receiving the personal details for the profile

4. A method to alter information stored in the system, for:

    a. updating one's personal profile

    b. updating the configuration of one's personal devices

5. A method for performing privileged tasks on the server. The voicemail storage system has restricted permissions, for security purposes. In addition, there are restrictions on the configuration files for the PBX, again for security reasons. However, certain services will need to access the voicemail, and the configuration files, in order to perform properly. There must, therefore, be some method of performing privileged tasks. This will not be a security risk if it is designed properly: if authentication is built into the method under discussion, then privileges may safely be elevated without compromising the security of the server. This method will be used for:

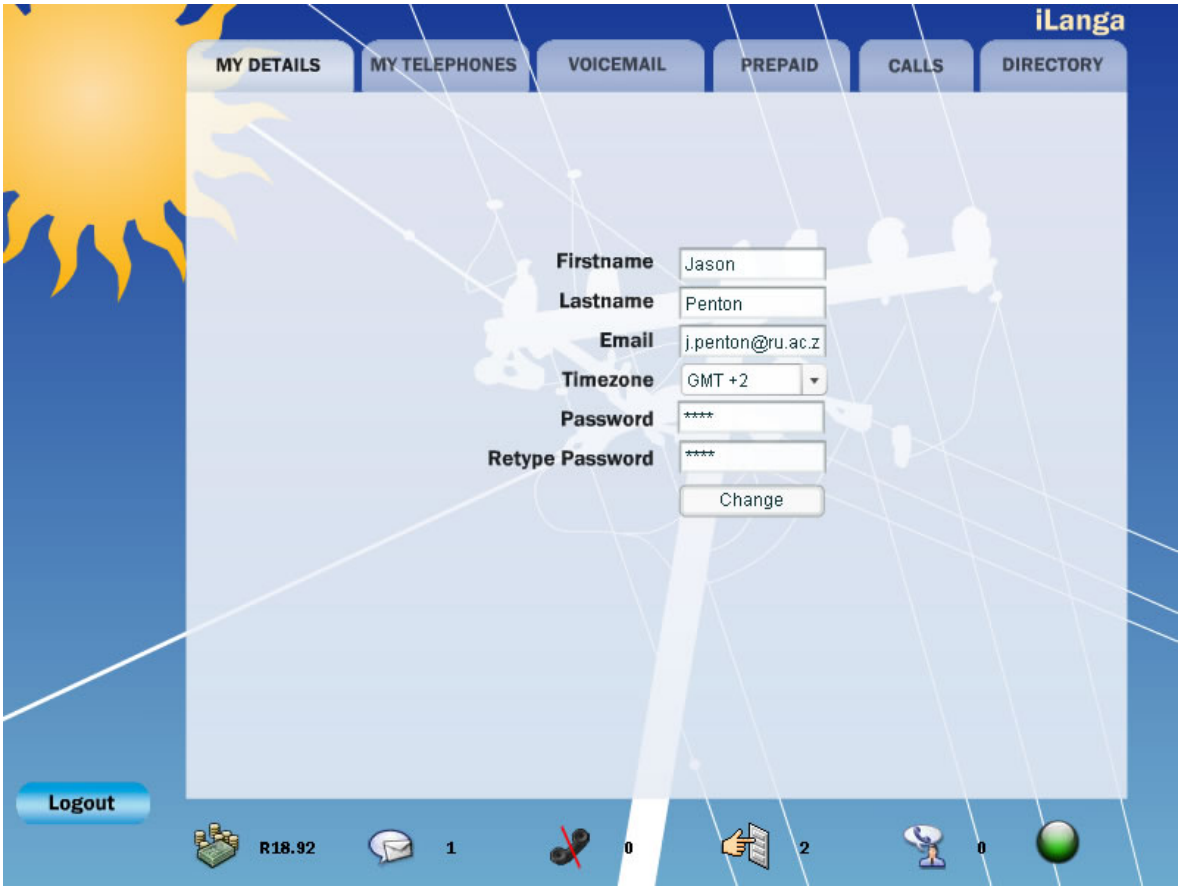

Figure 6.4: Altering a user profile

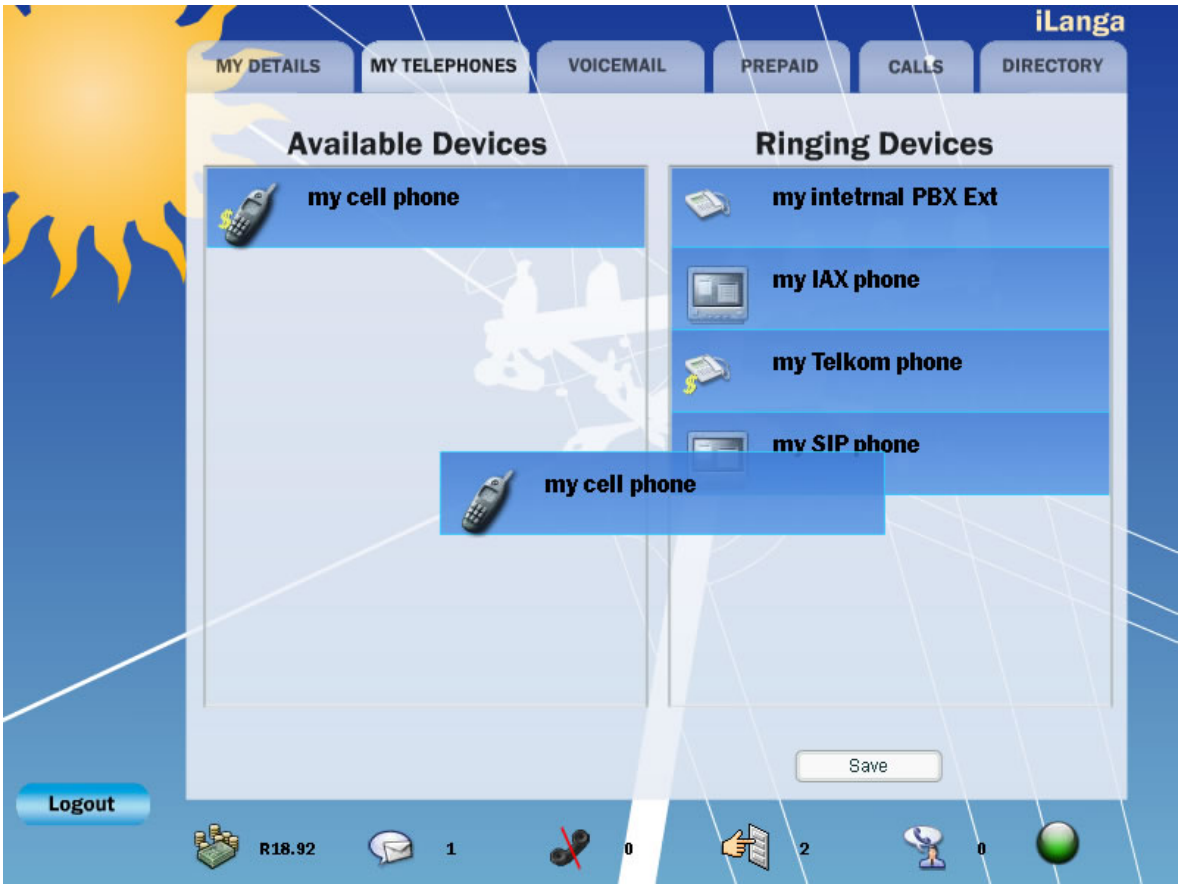

Figure 6.5: Configuring personal devices

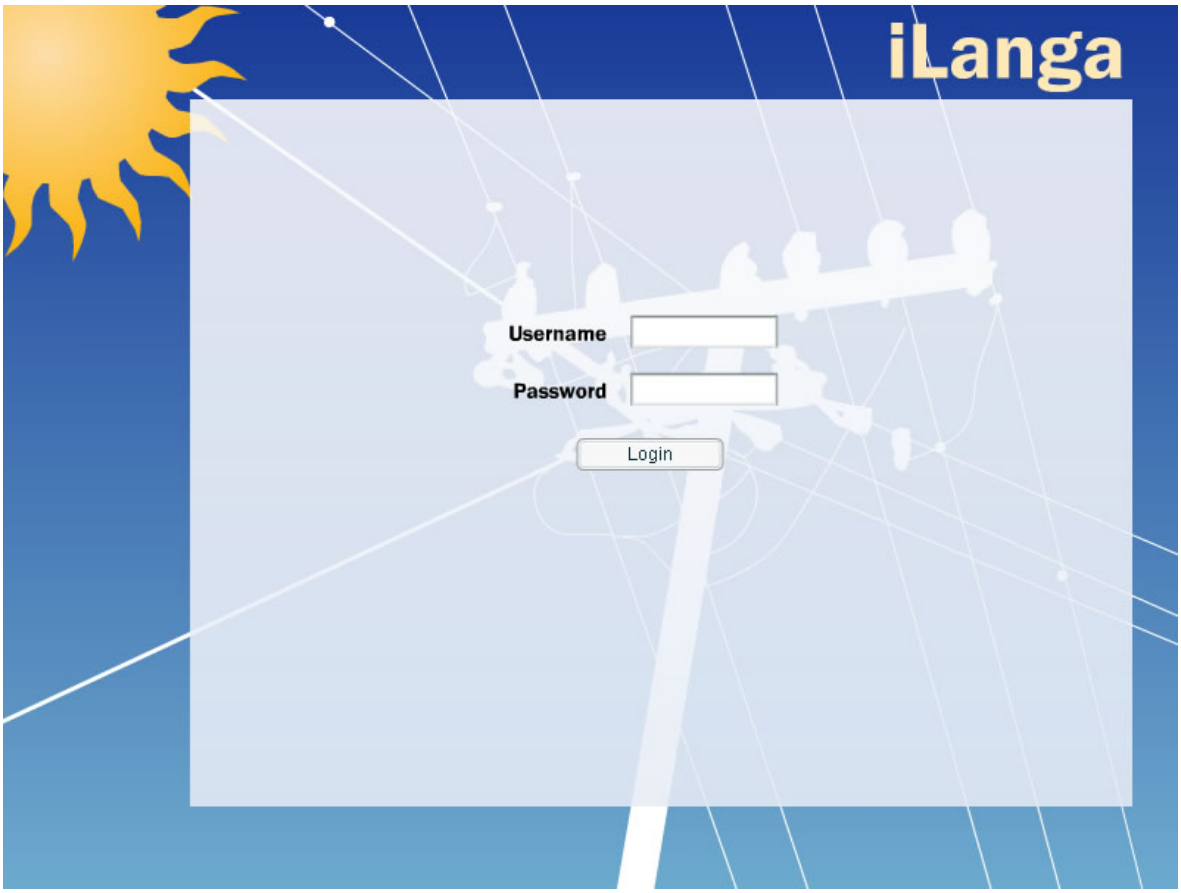

Figure 6.6: The Login screen

a. accessing one's voicemail, playing it back, deleting and moving it

b. changing the configuration of one's personal devices - this will require an alteration of the configuration files of the system and an instruction to the system to re-read these files

6.3 Implementation Choices

Having identified the system requirements, it was necessary to make implementation choices. Macromedia Flash [Inc 2004] was chosen to implement the user interface, due to its graphical richness, and the ease with which good looking, functional interfaces can be rapidly developed

and deployed. Given this decision, it was necessary to make choices that would interface well with the Actionscript language in which Flash backends are written.

The communication methods chosen were as follows:

1. For the constant communication medium, a TCP Socket between the Flash frontend and the server was chosen. Actionscript has an *XMLSocket* class which can be used to send and receive raw data across a TCP connection. However, for reasons that will be discussed later, a multiplexer was placed in between the Flash Actionscript, and the Manager API. This multiplexer will connect to the Manager API once, and accept connections from multiple instances of the frontend user interface. It will then mediate the data flowing between the Manager API and the frontend, making sure that only data relevant to the user currently logged into the frontend is passed along.

2. In order to make function calls directly to the system, it was decided to use the TCP Socket mentioned above, and add a facility for the clients to make requests to the Manager API, which would be passed directly along.

3. In order to retrieve once off information from the system, specifically from the database, the simplest method was to write web-based scripts that received queries, and returned data retrieved from the database. HTTP GET requests are well- documented and were easy to implement.

4. Altering data stored in the system was simply a matter of using the web based scripts described above, but supplying the data instead of requesting it. HTTP POST requests are best suited for this purpose.

5. In order to achieve elevated privileges, it was necessary to use the *SUID* mechanism, which allows programs to gain superuser privileges even when run by normal users. These programs will be called from the web, in the same manner as the scripts in points 3 and 4, but will gain superuser privileges when executed.

6.3.1 The XMLSocket

The Actionscript language in Flash has an *XMLSocket* class, which can be used to communicate over TCP/IP connections. The class is entirely event-driven: callback functions are specified to

be called when a connection is made, when one is lost, and when data is received. For this system, a socket is opened to the multiplexer (discussed later) running on the server, and the connection remains open for the lifetime of the frontend. The *dataReceived* callback is called whenever a chunk of data is received. When this happens, the data chunk is split up into its separate lines, the contents examined, and the correct action taken. For example, if the data chunk is a confirmation of authentication, then the interface moves into the connected state, and shows the logout button. If the data chunk shows that the users extension has changed state to, for example, busy, then the extension indicator is changed to show a red (busy) icon. In this way, as events happen in the system, they are sent to the user interface, which can display the appropriate information to the user immediately.

In the same way as information is received from the system, function calls can be made to the system using this socket. This is simply a matter of constructing the correct data chunk, and calling the send method to send it over the connection to the system. The PBX interprets the various data chunks, and takes the correct action. Thus, when the user indicates that she wishes to make a call to another user, a data chunk is constructed that requests the initiation of a call between the current user and the target user, and this is passed to the PBX.

The event-driven (callback) nature of the *XMLSocket* class makes this implementation very efficient and easy to construct: the interface does not need to repeatedly poll the connection to see if more data has been received - the code will be executed only if and when a full data chunk is available.

6.3.2 The Multiplexer

There is empirical evidence to suggest that the Asterisk Manager API can become somewhat unstable if there are more than just a few connections made to it. In addition, the Manager broadcasts all events and all information to all the clients connected to it. This is undesirable, both in terms of privacy, and in terms of excessive data-flow. The solution was to write a script that ran on the server, connected to the Manager API, and parsed all data received from it. In addition, the script would listen on a port for connections from the various user frontends running on users machines. These frontends would connect to this multiplexer as if it were the Manager API, and the multiplexer would proxy the data flowing in both directions. Data received from the frontends is passed on to the Manager API, allowing the frontends to communicate with it in

an apparently direct way. Data received from the Manager API is collected into full chunks, and the contents are examined. The multiplexer keeps track of which users are connected, and only passes those data chunks that are relevant to each user on to them.

The multiplexer script was written in Python, using the Twisted framework. The Multiplexer extends the LineReceiver protocol from the Twisted library for both its client side (the side that connects to the Manager API), and the server side (the side to which the frontend connects). This is a protocol that simply collects all data until an entire line has been received and then passes this line to a callback function defined by the Multiplexer script. This means that the Multiplexer's server and client sides can deal with data a line at a time, building it up into full chunks (delimited by a blank line), before dealing with it. In addition, the client side of the Multiplexer extends the ReconnectingClientFactory Twisted class, which means that if it loses its connection to the Manager API, it will automatically attempt to reconnect at increasingly delayed intervals.

The client and server sides of the Multiplexer are both implemented as finite state-machines: they keep track of what state they are currently in (authenticated, receiving data, idle), and examine the lines they receive to determine which state to proceed to. This way, if only part of a data chunk has been received, the Multiplexer does not block while it waits for the rest of it, it just stores its current state and continues with execution, until more data is received, and the callback function is triggered again.

On the server side of the Multiplexer, it was necessary to decide which of the connected clients received the data chunks being broadcast from the Manager API. Accordingly, we implemented a filter mechanism. When an entire data chunk has been received, and the state machine identifies this, the data is passed to a message handler function, the state is re-initialized, and data-reception begins again. The message handler is the function that examines the data and determines whether or not to pass it to each client. This is where the filters come in: for each user connected, the data is passed through a number of filters, and each one examines the contents. If a filter determines that the data is relevant to the user currently in focus, it will return a true value. If any one of the filters return true, the data will be sent to the user, and otherwise discarded. In this way, it is easy to extend the behaviour of the Multiplexer, to allow it to send new information to a user, by adding another filter.

6.3.3 The PHP scripts

PHP [PHP 2004] was chosen to implement most of the web-based scripts. PHP is a general purpose scripting language that is most useful for web development, being easily embedded within HTML pages. It was designed expressly for use in web scripts, and handles a lot of the complexities of HTTP automatically, such as the automatic interpretation of the GET and POST request variables that HTTP uses to send data between pages [Fielding et al. 1999]. For pure simplicity, PHP is perfect for writing web scripts - one can achieve a lot without writing too much code, which minimizes the chance that bugs will creep in. PHP is most often used in the *"mod_php"* form, where it acts as a plugin to the Apache web server. The benefit of this is that it is integral to the web-servers processing of the HTTP request, and does not incur any overhead. Normal CGI scripts (such as the perl scripts also used in this system) must be executed by the Apache web server, requiring the processing overhead of spawning an entirely separate process. With large systems, if the use of CGI scripts is very heavy, the web server can be slowed down significantly. PHP does not have this problem, running as it does as a part of the web server.

Database access is also substantially more easy in PHP, which has builtin functions to access MySQL, MSSQL and Postgres databases. Other languages rely on DBI (Data Base Interface) mechanisms to connect to databases - these are often unwieldy and difficult to use, a price not reasonable to pay considering the ease with which PHP can perform the same tasks.

Flash Actionscript has a *LoadVars* class which can send and receive variables to and from a webpage, using the HTTP protocol's GET and POST methods. The frontend uses this class to send and receive the information described in methods 3 and 4 above. This is done simply by setting the variables to be sent as properties of the class, and calling the *sendAndLoad* function, passing it the address of the PHP web script. This script can access the variables using PHP's built-in *$_REQUEST* array, and may store them in the system's MySQL database if necessary, using PHPs built-in MySQL functions. It may also retrieve data from the database and return it to the system, where it will be set as properties of the *LoadVars* class that called it.

In this manner, Flash Actionscript can send and receive as much information as is needed to and from the system. The database's security is not compromised, as authentication to the database server is done from the PHP script, which is never seen outside the web server.

6.3.4 The SUID wrapper and perl scripts

Performing privileged tasks on the server is more complicated. Normally, when an executable file is run, it runs with the privileges of the user who has requested its execution. In the case of this system, the frontend is communicating over a network connection with an unprivileged web server. However, Unix systems have a mechanism to get around this, called *SUID*, which stands for "Set User ID". If an executable file has the *SUID* bit set, it will run with the privileges of the user that owns the file, instead of those of the calling process. For example, suppose the superuser creates an executable file that tries to create a file in a restricted area of the filesystem. When the superuser runs this program, it will work fine, since the superuser has permission to write to this area of the filesystem. If a normal user runs it, however, permission will be denied. If the executable is then given the *SUID* bit, and the normal user attempts to run it again, it will work fine: since the file is owned by the superuser, and has the *SUID* bit set, it will temporarily gain superuser privileges, for the duration of its execution. This can, of course, be a severe security risk, but if the program is written properly, checking all input from both the user and the environment, and performs only very limited tasks (i.e. only those it absolutely has to), then SUID can be a useful way of allowing users to perform useful tasks that they would not normally have permission to do (reading their voicemail, in this case, is an excellent example).

There is a further complication: Unix programs are often scripts. This means that they are written in interpreted languages, and are run directly, without being compiled. The interpreter is called at run-time, and it executes the program as it reads it. Programmers often attempt to set the *SUID* bit of a script, and are surprised that it does not work. The reason for this is that the script itself is not what is actually being executed: it is the interpreter that is being executed, not the script. In order for the script to run with elevated privileges, the interpreter itself would have to have the *SUID* bit set, and this would be a terrible security risk - any script on the system would run as the superuser. The solution to this problem is to create what is called a wrapper. This is a very simple C program, that merely makes a system call to execute a certain script. The benefit of this is that the C program can be compiled to a binary executable, which can, in turn be given the *SUID* bit. Thus, when the compiled executable is run, it gains superuser privileges, and then makes the system call to execute the script. Since the script is now being executed by something with superuser privileges, it runs with those privileges itself.

While this explanation is somewhat intricate, the final solution is fairly straightforward. Wrapper programs are written for each service that needs to run with elevated permissions, and they are

compiled and given the *SUID* bit. The services themselves are then written. These services will be called in the same way as the PHP scripts - that is, via HTTP, as web scripts. Thus, they can use the same HTTP GET and POST methods to send and receive data, and no extra interface needs to be designed. The benefit of this is that exactly the same Flash Actionscript can be used - the *LoadVars* class can be used in exactly the same way.

Even having illustrated how useful and practical PHP is, the scripting language Perl [Perl 2004] was chosen for the implementation of these scripts, for various reasons. Perl does not have the same level of integration with HTTP that PHP does - manual interpretation of GET and POST variables needs to be performed, for example - but it has other benefits that make it much more useful in this situation.

Because PHP runs as a module of the Apache web server, it cannot take advantage of the *SUID* mechanism described earlier. Only separately spawned processes, or processes that are already running as the superuser, may change their execution ID. For this reason, we use a normal CGI script, executed by the web server, with the *SUID* bit set, to elevate the permissions. As described previously, a small C program is required to gain the superuser privileges. One may ask why the entire script is not written in C if this is the case, and why one goes to the trouble (and creates the system overhead) of spawning a separate process to run the perl script. The reason for this is simply security. The script will be running with elevated privileges, and C is very difficult to write securely - one must constantly check all pointer references, and all memory allocations, and so on. Even were this not the case, Perl is an excellent choice for writing scripts that need to be secure. It has a feature called "taint checking". This means that every time data is received from an "external source" (such as user input, or an environment variable), it is marked as "tainted". This means that it cannot be trusted. If data is received from user input, the user could supply any data he wanted, and one cannot rely on this data for security. So, Perl marks this data as tainted, and it is necessary to "untaint" all data before it can be used for anything that needs to be secure (for example, calling a system function, or executing another process). Data can be untainted by applying checks to it, and making sure it is valid data.

Perl is unique in its taint checking, and is also fairly easy to write code in. This makes it both secure, and easily maintainable, and a good choice for the implementation of the SUID scripts.

The implementation of these scripts, once the *SUID* has been dealt with, is fairly simple. Most of the tasks that need to be performed with elevated privileges involved file manipulation: deleting voicemails, moving them between folders, or playing them back. All of these tasks simply

involve making basic system calls (delete file, move file, write to file, and so on), and there are Perl functions that perform all of them. There was one exception to this: when a user changed the preferred order that they wanted their devices to ring, it was necessary to change the `extensions.conf` file to reflect the new dialplan, and to request that Asterisk reload the file. This task was accomplished easily enough as well.

6.4 Summary

By utilising a number of commonly available open source tools and languages, a service was created which plugs into the iLanga PBX, and allows users complete control over their telephony experience. The various services available at any one endpoint are presented in a consistent manner, regardless of the capabilities of the endpoint. The service interface is intuitive and easy to use, and is easily extendible.

Chapter 7

Conclusion

In this chapter, we summarise the work done for this entire thesis, and then revisit the aims and objectives of the work as described in the first chapter, and show that the goals of the project have been achieved. Finally, the possibilities for further research and work in this area are discussed.

7.1 Summary of work

In this thesis, we investigated Asterisk in terms of its viability to provide the depth of services that will be required of a VoIP solution. As well as examining the PBX itself as a whole, and investigating how it worked, we examined the various ways in which the functionality of the PBX could be altered, and the methods that could be used to create services that could run on top of the PBX.

After studying Asterisk, its architecture, configuration, and operation, four methods of altering the behaviour of the PBX were identified and investigated: the dial plan, the external interfaces, the Asterisk Gateway Interface, and the Application API. Two of these methods in particular were isolated as being ideal for creating services: the AGI interface, and the application API. These methods were investigated in detail, and the advantages and disadvantages of each were determined. We then developed two simple example services that ran within the PBX, utilising these two methods: an automated AGI weather service to provide local temperature information on request, and a service application to play music on demand to users of the PBX.

Upon investigating service creation in Asterisk, we realised that there were two areas that needed work: those of accessibility, and functionality. The terms *extend* and *expand*, introduced earlier, were used respectively to distinguish these two areas. Firstly, services needed to be able to be extended to be accessible from any interface or device, so that they were no longer bound solely to the PBX. Secondly, they needed to be expanded so that their functionality reached deep into the core of the PBX, and their capabilities became much greater.

In order to investigate how to extend Asterisk services, the music application service was chosen as an example, and it was re-engineered so that it could be controlled from other interfaces. In the process of this re-engineering, it became clear that there were two major areas that required thought and consideration: remote procedure calls, and authentication. Three proof-of-concept interfaces to the service were created to demonstrate the different scenarios that investigation into these areas isolated: an interface through the PBX itself, a stand-alone application running on a desktop machine, and a web-based client running within a webserver. The extended service worked well, and transparently to the PBX.

The next step was to investigate the expansion of Asterisk services. We used the iLanga PBX system as a base for this investigation - it was a full Asterisk phone system that had been deployed at Rhodes University, and thus provided an excellent environment in which to test the expanded services. The other two methods of altering the behaviour of Asterisk identified earlier (the dial plan, and the external interfaces) were utilised to create a system that allowed users of the PBX to configure and control almost every aspect of their telephony experience. The system was unified into one integrated web-based system that presented the various services available in a consistent manner, regardless of the capabilities of the endpoint being used. The interface was intuitive and easy to use, and easily extendible.

In the course of investigating the extension and expansion of Asterisk, several open source systems were investigated and evaluated for their usefulness and relevancy to the project at hand. Because of the nature of the open source methodology, these tools and systems were not always production-ready, nor were they always adequately documented and explained. As such, part of the aims of this thesis were to investigate and document any aspects of the open source systems being used that were not already well explained. Asterisk itself, of course, is a prime example of this - there is still much in the PBX that does not have sufficient documentation, although the open source community is contributing more in this area every day. In addition, in particular, the Twisted Matrix framework was thoroughly investigated. This is a powerful framework for creating networked asynchronous applications, but it suffers from quite a dearth of adequate ex-

planation. We investigated this framework thoroughly, and provided a good introduction to the components of the framework and their usage.

7.2 Aims revisited

In this section, I will revisit the goals of this thesis, as stated in the introduction, and assess whether or not they have been achieved.

7.2.1 Creating Basic Asterisk Services

While there are several well documented and researched methods for creating services for various telephony protocols, notably SIP and H.323, until recently there was no unified method for creating a general telephony service. This thesis proposed that Asterisk would be ideal for such a purpose, because its architecture is such that it acts as "middleware" between the telephony networks and the applications which will run on them. The API by which services can communicate with the PBX is protocol-agnostic, so any Asterisk service will be accessible from any telephony technology which Asterisk communicates with.

This aim was easily achieved, and two proof-of-concept services were created to illustrate how Asterisk services worked. The AGI weather service used an external data source to provide weather information, showing how versatile the services could become. The music player application allowed the user to interact with the service, controlling the playback of the music being provided, showing that these services were fully interactive, and not just passive providers.

7.2.2 A More Advanced Service

One goal of this thesis was to investigate how to extend services from being merely accessible through telephony endpoints, and make them available through a variety of different interfaces, and on a variety of different platforms.

For this, the music player application service was extended, providing it with an architecture in which the central point was outside of the PBX itself. This meant that other interfaces could be,

and were, attached to it, without requiring them to communicate directly with the PBX itself. A web-based interface and a standalone application were created, allowing the user to control the playback of the music as easily as the telephone-based interface did.

In the course of extending this service, an important point was realised. An important aspect of creating a new interface for a service was ensuring that the communication channel between the interface and the service was fully authorised. Once the basic architecture of the service was created, allowing different interfaces to communicate with it was trivial, but making sure that the interface was authorised to perform that communication was not. The various interfaces which were created highlighted different aspects of authorisation which needed to be taken into account, and a taxonomy of possible authentication methods was created that should prove useful for all future types of interface. The interfaces created for the music player service were each given their own method of authentication, which tied into the central authentication of the Asterisk PBX, ensuring that each interface, while using different methods of communication, still exposed themselves to the user in a similar way.

7.2.3 A Complex, Extensible Service

The next goal of this thesis was to create a service with much deeper functionality. With this goal in mind, we decided to create a service that would provide users of the iLanga PBX with in-depth control of their telephony environment. This PBX was widely deployed and used at Rhodes University, and therefore provided an excellent environment in which to test this, as it had a large user base, and was a fully-fledged functional system which would provide practical examples of the demands of real users.

A number of services for the iLanga PBX were created, which allowed its users extensive control over their telephony environment. It granted them the ability to customize their telephony devices, access their voicemail from a number of interfaces, request their call history, and so on. In addition, these services were tied together in a web-based system which presented the users with a consistent interface that exposed a unified set of services, which was easy to use and intuitive.

In the course of creating these services, a number of challenges were encountered and dealt with. The problem of a large number of users attempting to access the system and causing instability was dealt with by creating a multiplexer wrapper around the system that lessened the load. In addition, the problem of maintaining security on a system which needed to expose a number of

services to the outside world was considered and dealt with in various ways, including the use of SUID wrappers around taint-checked perl scripts.

7.2.4 Documenting the systems used

A large part of this thesis consisted of an investigation into the readiness of open source telephony software for production system deployments. By the nature of open source software, there is often a lack of good documentation in certain areas - the software is created by volunteers in their spare time, and documentation is often neither a priority, nor an enjoyable task. As such, one of the goals of this project was to record the investigations into the software and the discoveries made, and in so doing provide some basic documentation for the systems.

In the course of this work, both Asterisk and the Python Twisted framework were used. Both of these systems suffered from a serious lack of good documentation in their early stages, and while both projects are improving in this area, there are still a number of aspects of each software system which is inadequately or badly documented. A large amount of the work done for this thesis involved finding out how these systems worked, and experimenting with them in order to become acquainted with the intricacies of using them for real systems. Having thoroughly examined them, this thesis includes two chapters which explain the basic architecture of each system, and gives a broad introduction to the use of the two software packages.

7.2.5 Investigating the readiness of Open Source

As previously stated, one of the broad goals of this thesis was to investigate whether open source telephony software is ready for use in production system deployments. There has been a lot of skepticism about open source software, since by its nature it has a sort of amateurish feel about it. It rarely has official or corporate backing, and one does not require any qualifications for working on an open source project other than an eagerness to give ones time and effort for free. As such, people are reluctant to commit to using this software in systems on which a company may rely, or to invest any amount of money in systems which depend on this free software that comes with no guarantees.

This thesis has helped to put together a stable telephony system, which has been deployed and used at Rhodes University. This system uses entirely open source software, and provides high

quality service to its users. While there have been times in the development of this system where bugs in the software have been found, or certain aspects of the software have been noticed to be slightly unstable, these bugs have been fixed and the instabilities overcome. Indeed, this is an indication of one of the strengths of open source software: when a bug is found, it can be immediately reported and fixed. On the whole, the open source software used in this thesis has proven itself sturdy and of high quality, and it definitely seems ready for deployment in a production environment.

7.3 Further Work

The work done in this thesis has highlighted a few points which might prove productive if they were investigated further.

7.3.1 Language Independence

When extending the Juke Application, I used Perspective Broker and the Twisted Framework for communication between the different elements. While there were good reasons for choosing these two systems, their use does limit the choice of language for writing other services and service interfaces. There are, in fact, implementations of Perspective Brokers for Elisp and for Java, but they are still in development, and no other languages support Perspective Broker as yet. It may also be true that Perspective Broker does not limit the choice any more than another system would - CORBA, for example. However, ideally, a completely language-agnostic communication system should be used, to allow complete versatility for the services and service interfaces. Work could be done on designing a custom protocol that allows these extended services to communicate with each other, and with the PBX.

7.3.2 Authentication Taxonomy

As discussed, we realized that one of the most important aspects of creating extended services was ensuring that there was a valid authenticated communication channel between the interface and the central service. We developed a taxonomy of authentication methods that we consider

to be exhaustive. However, there is still room for investigation into whether this taxonomy is correct, and whether it can be simplified, or should be extended. In addition, it may be possible to design a generic authentication method that will work for all types of service and interface, without the need to customize it per interface.

7.3.3 Service Discovery

The expanded services discussed in chapter 6 are well integrated into a consistent, user-friendly interface, but this is done manually at design time. If a new service were added to the system, or one of the existing services were changed, it would require a rewrite of the interface in order to incorporate this new functionality. Ideally, there should be some sort of service discovery system, possibly with a central registry of services and service capabilities, which the interface can query, before building up the view it exposes to the user of the system. Work can be done as well on categorizing the possible types of services in order to create this registry.

7.4 Conclusion

This thesis has shown how easy it is for services to be created in Asterisk. It has also illustrated that these services can be extended so that they are accessible from any interface or device, and that they can be expanded so that their functionality reaches deep into the system, allowing the users total control over their environment. In addition, it has investigated and documented various open source systems that were used in achieving these goals, and shown that they are mature enough and stable enough to be used in the deployment of a production telephony system.

References

[Abrams et al. 1999] Abrams, Marc, C. Phanouriou, A. L. Batongbacal, S. M. Williams and J. E. Shuster (1999). Utml: an appliance-independent xml user interface language. In *WWW '99: Proceeding of the eighth international conference on World Wide Web* (1999), Elsevier North-Holland, Inc., pp. 1695–1708.

[Agrawal et al. 2001] Agrawal, H., R. Roy, V. Palawat, A. Johnston, C. Agboh, D. Wang, H. Schulzrinne, K. Singh and J. Maeng (2001). Sip-h.323 interworking. Internet-draft, IETF, 2001.

[Allen 2000] Allen, D. (2000). Megaco and mgcp, 2000, Last accessed Mar 2004, <http://www.commweb.com/shared/article/showArticle.jhtml?articleID=8702913>.

[Anjum et al. 2001] Anjum, Farooq, F. Caruso, R. Jain, P. Missier and A. Zordan (2001). Cititime: a system for rapid creation of portable next-generation telephony services. *Computer Networks 35* (2001).

[Arango et al. 1999] Arango, M., A. Dugan, I. Elliott, C. Huitema and S. Pickett (1999). RFC 2705: Media Gateway Control Protocol (MGCP), Oct. 1999.

[Asterisk 2004] Asterisk (2004). The asterisk pbx, 2004, <http://www.asterisk.org>.

[Bond et al. 2004] Bond, Gregory W., E. Cheung, K. H. Purdy and P. Zave (2004). An open architecture for next-generation telecommunication services. *ACM Transactions on Internet Technology, Vol. 4, No. 1* (2004).

[Charter 2003a] Charter, Media Gateway Control (2003a). The media gateway control charter working group at the ietf, 2003, <http://www.ietf.org/html.charters/megaco-charter.html>.

[Charter 2003b] Charter, SIP (2003b). The sip-charter working group at the ietf, 2003, <http://www.ietf.org/html.charters/sip-charter.html>.

[Chatzipapadopoulos et al. 2000] Chatzipapadopoulos, F., G. D. Zen, T. Magendanz, I. S. Venieris and F. Zizza (2000). Harmonised internet and pstn service provisioning. *Computer Communications 23* (2000).

[Consortium 2003] Consortium, W3 (2003). Soap: Simple object access protocol, 2003, <http://www.w3.org/TR/SOAP>.

[Fielding et al. 1999] Fielding, R., J. Gettys, J. Mogul, H. Frystyk and T. Berners-Lee (1999). RFC 2616: Hypertext Transfer Protocol — HTTP/1.1, June 1999, <ftp://ftp.internic.net/rfc/rfc2068.txt, ftp://ftp.math.utah.edu/pub/rfc/rfc2068.txt>.

[Force 1996] Force, Internet Engineering Task (1996). Rtp: A transport protocol for real-time applications, Jan. 1996, <http://www.ietf.org/rfc/rfc1899.txt>.

[Force 2000] Force, Internet Engineering Task (2000). Rtp payload for dtmf digits, telephony tones and telephony signals, May 2000, <http://www.ietf.org/rfc/rfc2833.txt>.

[Glitho 2001] Glitho, Roch H. (2001). Emerging alternatives to today's advanced service architectures for internet telephony: In and beyond. *Computer Networks 35* (2001).

[Greene et al. 2000] Greene, N., M. Ramalho and B. Rosen (2000). IETF RFC 2805: Media Gateway Control Protocol Architecture and Requirements, 2000.

[Greene et al. 1999] Greene, N., A. Rayhan, C. Huitema, B. Rosen and J. Segers (1999). RFC 3015: Megaco Protocol Version 1.0, Oct. 1999.

[Halse and Wells 2002] Halse, G. and G. Wells (2002). A bi-directional soap/sms gateway service. In *SATNAC Conference Proceedings* (2002).

[Handley and Jacobson 1998] Handley, M. and V. Jacobson (1998). RFC 2327: SDP: Session description protocol, Apr. 1998, Status: PROPOSED STANDARD., <ftp://ftp.internic.net/rfc/rfc2327.txt, ftp://ftp.math.utah.edu/pub/rfc/rfc2327.txt>.

[Handley et al. 1999] Handley, M., H. Schulzrinne, E. Schooler and J. Rosenberg (1999). IETF RFC 2327: SIP: Session initiation protocol, Mar. 1999.

[Hitchcock et al. 2004] Hitchcock, J., J. B. Penton and A. Terzoli (2004). A multilanguage, open source approach to connecting user interfaces to a next-generation pbx. In *SATNAC Conference Proceedings* (2004).

[Hsieh et al. 2001] Hsieh, M., J. Okuthe and A. Terzoli (2001). Deploying a sip environment in which to study service creation. In *SATNAC Conference Proceedings* (2001).

[Hsieh et al. 2002] Hsieh, M., J. Okuthe and A. Terzoli (2002). An investigation into multimedia service creation using sip. In *SATNAC Conference Proceedings* (2002).

[Hsieh 2003] Hsieh, Ming Chih (2003). Service provisioning in two open-source sip implementations, cinema and vocal. Master's project, Rhodes University, Computer Science Department, Dec. 2003.

[Huitema 2003] Huitema, C. (2003). IETF RFC 3605: Real Time Control Protocol (RTCP) attribute in Session Description Protocol (SDP), Oct. 2003.

[Inc 2004] Inc, Macromedia (2004). Macromedia flash, 2004, <http://www.macromedia.com/software/flash>.

[ITU-T 1998a] ITU-T (1998a). Itu-t recommendation h.225: Call signalling protocols and media stream packetization for packet-based multimedia communication systems. Tech. rep., Telecommunications Standardization Sector, 1998.

[ITU-T 1998b] ITU-T (1998b). Itu-t recommendation h.245: Control protocol for multimedia communications. Tech. rep., Telecommunications Standardization Sector, 1998.

[ITU-T 1998c] ITU-T (1998c). Itu-t recommendation h.323: Packet-based multimedia communications systems. Tech. rep., Telecommunications Standardization Sector, 1998.

[ITU-T 2000] ITU-T (2000). Itu-t recommendation h.248: Gateway control protocol. Tech. rep., Telecommunications Standardization Sector, 2000.

[Jacobs and Clayton 2002] Jacobs, A. and P. Clayton (2002). Utilizing mgcp to design an h.323 endpoint sms service. In *SATNAC Conference Proceedings* (2002).

[Jacobs and Clayton 2003] Jacobs, A. and P. Clayton (2003). Investigating call control using mgcp. In *SATNAC Conference Proceedings* (2003).

[Jacobs 2004] Jacobs, Ashley (2004). Investigating call control using mgcp in conjunction with sip and h.323. Master's project, Rhodes University, Computer Science Department, Nov. 2004.

[Jiang et al. 2001] Jiang, Wenyu, J. Lennox, H. Schulzrinne and K. Singh (2001). Towards junking the pbx: Deploying ip telephony. *NOSSDAV'01, June 25-27* (2001).

[Labs 2004] Labs, Twisted Matrix (2004). The twisted framework, 2004, <http://www.twistedmatrix.com/products/twisted>.

[Licciardi et al. 2001] Licciardi, C.A., G. Canal, A. Andreetto and P. Lago (2001). An architecture for in-internet hybrid services. *Computer Networks 35* (2001).

[MySQL 2004] MySQL (2004). Mysql, 2004, <http://www.mysql.com>.

[OpenH323 2003] OpenH323 (2003). The openh323 project, 2003, <http://www.openh323.org/>.

[Penton and Terzoli 2003] Penton, J. B. and A. Terzoli (2003). Asterisk: A converged tdm and packet-based communications system. In *SATNAC Conference Proceedings* (2003).

[Penton and Terzoli 2004] Penton, J. B. and A. Terzoli (2004). ilanga: A next generation voip-based, tdm-enabled pbx. In *SATNAC Conference Proceedings* (2004).

[Penton et al. 2001a] Penton, J. B., A. Terzoli and P. Wentworth (2001a). Deploying a feature-rich h.323 environment in which to practice the creation of services. In *SATNAC Conference Proceedings* (2001).

[Penton et al. 2001b] Penton, J. B., A. Terzoli and P. Wentworth (2001b). Retrieving emails via traditional pstn telephones and h.323 services. In *SATNAC Conference Proceedings* (2001).

[Penton 2003] Penton, Jason Barry (2003). An empirical, in-depth investigation into service creation in h.323 version 4 networks. Master's project, Rhodes University, Computer Science Department, Aug. 2003.

[Percy 1999] Percy, Alan (1999). Understanding latency in ip telephony, Feb. 1999.

[Perdikeas and Venieris 2001] Perdikeas, Menelaos K. and I. S. Venieris (2001). Parlay-based service engineering in a converged internet-pstn environment. *Computer Networks 35* (2001).

[Perl 2004] Perl (2004). Perl, 2004, <http://www.perl.org>.

[PHP 2004] PHP (2004). Php, 2004, <http://www.php.net>.

[Project 2004] Project, ISDN4Linux (2004). Isdn protocol api, 2004, <http://www.isdn4linux.de>.

[Python 2004] Python (2004). Python, 2004, <http://www.python.org>.

[Rash 2005] Rash, Wayne (2005). Open source pbxes: Free flexibility. *InfoWorld; 1/31/2005, Vol. 27 Issue 5* (2005).

[Rinde 1999] Rinde, Joseph (1999). Telephony in the year 2005. *Computer Networks 31* (1999).

[Rosenberg and Shockey 2000] Rosenberg, J. and R. Shockey (2000). The session initiation protocol: A key component for internet telephony, 2000, <http://www.cconvergence.com/shared/article/showArticle.jhtml?articleID=8700868>.

[Russell 2000] Russell, T. (2000). Signalling system 7, 2000.

[Spencer et al. 2003] Spencer, Mark, M. Allison and C. Rhodes (2003). The asterisk handbook, Mar. 2003.

[Survey 2005] Survey, Netcraft (2005). The netcraft webserver survey, 2005, <http://news.netcraft.com/archives/>.

[Technologies 2004] Technologies, Agilent (2004). Dtmf tone analysis, 2004, <http://onenetworks.comms.agilent.com/VQT/VQT

[Zadka and Lefkowitz 2002] Zadka, Moshe and G. Lefkowitz (2002). The twisted network framework. In *Proceedings of the Tenth International Python Conference* (2002).

Appendix A

Selected Source Code

A.1 weather.agi

This script was used in developing the AGI service in section 3.1:

```
#!/usr/bin/perl -w
use asterisk::agi;
use lwp::useragent;
use http::request::common qw(post get);
use strict;
my $ua = new lwp::useragent;
my $datapage='http://www.ru.ac.za/weather/archive/current5min/current-000';
my $req = get $datapage;
my $res = $ua->request($req);
my $retval = "";
my $html = $res->as_string;
$html =~ m/d,(.*)\n/;
$html = $1;
#date,time,temp,hum,baro,wdir,wspd,wshi,rf_day,srad,batt,chill
my ($date, $time, $temp, $hum, $baro, $wdir, $wspd, $wshi, $rf_day, $srad,
      $batt, $chill) = split /,/, $html;
my $t = ($temp-32)*(5/9); # temperature in 'c
my $w = ($chill-32)*(5/9); # wind chill in 'c (compensated actual temp, not
            #         wind chill factor)
my $r = $hum;
```

```
my $agi = new asterisk::agi;
sub sayfloat() {
   my $n = shift;
   $n =~ m/^\s*(\d+)\.?(\d{0,2})/;

   my $whole = $1;
   my $frac = $2;

   $agi->say_number($whole);
   if($frac) {
      $agi->stream_file("weather/point");
      $agi->say_digits($frac);
   }
}
my %input = $agi->readparse();
$agi->stream_file("weather/currently");
$agi->stream_file("weather/temperature");
&sayfloat($t);
$agi->stream_file("weather/degrees");
$agi->stream_file("weather/windchill");
&sayfloat($w);
$agi->stream_file("weather/degrees");
$agi->stream_file("weather/windspeed");
&sayfloat($wspd);
$agi->stream_file("weather/kph");
$agi->stream_file("weather/relhumid");
&sayfloat($r);
$agi->stream_file("weather/percent");
$agi->hangup();
exit(0);
```

A.2 app_juke.c

This is the source code for the Juke application that is compiled into asterisk to provide the service in section 3.2.

```
/*
 * This program is free software, distributed under the terms of
```

```
 * the GNU General Public License
 */
#include <asterisk/lock.h>
#include <asterisk/file.h>
#include <asterisk/logger.h>
#include <asterisk/channel.h>
#include <asterisk/frame.h>
#include <asterisk/pbx.h>
#include <asterisk/module.h>
#include <asterisk/translate.h>
#include <string.h>
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/time.h>

static char *cmd_next = "next\n";
static char *cmd_prev = "prev\n";
static char *cmd_play = "play\n";
static char *cmd_stop = "stop\n";
static char *cmd_pause = "pause\n";
static char *cmd_quit = "quit\n";

static char *tdesc = "Juke Box";
static char *app = "juke";
static char *synopsis = "Streams music, with controls\n";

STANDARD_LOCAL_USER;
LOCAL_USER_DECL;

static int spawnplayer(int infd, int outfd)
{
   int res;
   int x;
   res = fork();
   if(res < 0) { ast_log(LOG_WARNING, "Fork failed\n"); }
   if(res) { return res; }
   res = dup2(infd, STDIN_FILENO);
   if(res == -1) { perror("dup2"); }
```

```
    res = dup2(outfd, STDOUT_FILENO);
    if(res == -1) { perror("dup2"); }

    for (x=0;x<256;x++) {
       if ((x != STDOUT_FILENO) && (x != STDIN_FILENO))
         close(x);
    }
    execl("/home/vhata/ctrlmp3.pl", "/home/vhata/ctrlmp3.pl", (char *)NULL);
    exit(1);
}

static int timed_read(int fd, void *data, int datalen, int timeout)
{
    int res;
    struct pollfd fds[1];
    fds[0].fd = fd;
    fds[0].events = POLLIN;
    res = poll(fds, 1, timeout);
    if (res < 1) {
       ast_log(LOG_NOTICE, "Poll timed out/errored out with %d\n", res);
       return -1;
    }
    return read(fd, data, datalen);
}

static int juke_exec(struct ast_channel *chan, void *data)
{
    int res=0;
    struct localuser *u;
    int toapp[2];    int toappfd;
    int fromapp[2];  int fromappfd;
    int ms;
    int pid = -1;
    int owriteformat;
    char c[100];

    struct ast_frame *f;
    struct myframe {
       struct ast_frame f;
       char offset[AST_FRIENDLY_OFFSET];
       short frdata[160];
```

```
} myf;

if (!data) {
   ast_log(LOG_WARNING, "didn't get a param");
   return -1;
}
if (pipe(toapp)) {
   ast_log(LOG_WARNING, "Unable to create pipe to app\n");
   return -1;
}
toappfd = toapp[1];
if (pipe(fromapp)) {
   ast_log(LOG_WARNING, "Unable to create pipe from app\n");
   return -1;
}
fromappfd = fromapp[0];

LOCAL_USER_ADD(u);

ast_log(LOG_WARNING, "Start\n");
ast_stopstream(chan);

owriteformat = chan->writeformat;
res = ast_set_write_format(chan, AST_FORMAT_SLINEAR);
if (res < 0) {
   ast_log(LOG_WARNING, "Unable to set write format to signed linear\n");
   return -1;
}

res = fcntl(fromappfd, F_GETFL);
if(res == -1) {
   perror("fcntl");
}
res |= O_NONBLOCK;
fcntl(fromappfd, F_SETFL, res);

res = spawnplayer(toapp[0],fromapp[1]);

if (res >= 0) {
   pid = res;
```

```
  for (;;) {
res = read(fromappfd, myf.frdata, sizeof(myf.frdata));
if (res > 0) {
   myf.f.frametype = AST_FRAME_VOICE;
   myf.f.subclass = AST_FORMAT_SLINEAR;
   myf.f.datalen = res;
   myf.f.samples = res / 2;
   myf.f.mallocd = 0;
   myf.f.offset = AST_FRIENDLY_OFFSET;
   myf.f.src = __PRETTY_FUNCTION__;
   myf.f.delivery.tv_sec = 0;
   myf.f.delivery.tv_usec = 0;
   myf.f.data = myf.frdata;
   if (ast_write(chan, &myf.f) < 0) {
      res = -1;
      break;
   }
}
ms = ast_waitfor(chan, 1000);
if (ms < 0) {
   ast_log(LOG_WARNING, "Hangup detected\n");
   res = -1;
   break;
}
if (ms) {
   f = ast_read(chan);
   if (!f) {
      ast_log(LOG_WARNING, "Null frame == hangup() detected\n");
      res = -1;
      break;
   }
   if (f->frametype == AST_FRAME_DTMF) {
      sprintf(c, "User pressed key: %d\n", f->subclass);
      ast_log(LOG_WARNING, c);
      switch(f->subclass) {
    case '0':
       ast_log(LOG_WARNING, "quit\n");
       write(toappfd, cmd_quit, strlen(cmd_quit));
       break;
    case '1':
       ast_log(LOG_WARNING, "prev\n");
```

```
            write(toappfd, cmd_prev, strlen(cmd_prev));
            break;
         case '3':
            ast_log(LOG_WARNING, "next\n");
            write(toappfd, cmd_next, strlen(cmd_next));
            break;
         case '4':
            ast_log(LOG_WARNING, "stop\n");
            write(toappfd, cmd_stop, strlen(cmd_stop));
            break;
         case '6':
            ast_log(LOG_WARNING, "play\n");
            write(toappfd, cmd_play, strlen(cmd_play));
            break;
         case '5':
            ast_log(LOG_WARNING, "pause\n");
            write(toappfd, cmd_pause, strlen(cmd_pause));
            break;
           }
        }
        ast_frfree(f);
     }
       }
   }
   close(fromapp[0]); close(fromapp[1]);
   close(toapp[0]); close(toapp[1]);
   if (pid > -1)
      kill(pid, SIGKILL);
   if (!res && owriteformat)
      ast_set_write_format(chan, owriteformat);
   ast_log(LOG_WARNING, "End\n");
   LOCAL_USER_REMOVE(u);
   return res;
}
int unload_module(void) {
   STANDARD_HANGUP_LOCALUSERS;
   return ast_unregister_application(app);
}
int load_module(void) {
   return ast_register_application(app, juke_exec, synopsis, tdesc);
}
```

```
char *description(void) {
   return tdesc;
}
int usecount(void) {
   int res;
   STANDARD_USECOUNT(res);
   return res;
}
char *key() { return ASTERISK_GPL_KEY; }
```

A.3 ctrlmp3.pl

This script was used in developing an Application service in section 3.2:

```
#!/usr/bin/perl -w

use Fcntl qw(F_GETFL F_SETFL O_NONBLOCK);
use POSIX qw(:errno_h mkfifo);
use IPC::Open3;
use FileHandle;
use Time::HiRes qw(usleep);
use strict;

my $DEBUG = 0;

my $pid;
my $fifopath;

if($DEBUG) {
   $pid = open3(*WRF, *RDF, *ERRF, "/usr/bin/mpg123 -f 8192 --mono -r \
                8000 -R");
} else {
   $pid = open3(*WRF, *RDF, *ERRF, "/usr/bin/mpg123 -f 8192 --mono -r \
                8000 -R -s");
}
my @playlist = ("Syntax - Pride.mp3",
      "Vast - Touched.mp3",
      "The Dandy Warhols - We Used To Be Friends.mp3",
```

```
      "Jeff Buckley - Hallelujah.mp3");
my $cursong = -1;
my $random = 0;

autoflush WRF 1;
autoflush RDF 1;
autoflush ERRF 1;

my $flags = 0;
for my $f in (RDF, ERRF, STDIN) {
for fcntl($f, F_GETFL, $flags)
   or die "Couldn't get flags for $f: $!\n";
$flags |= O_NONBLOCK;
fcntl($f, F_SETFL, $flags)
   or die "Couldn't set flags for $f: $!\n";
}

&nextsong();
my $done = 0;

my ($rdb, $stdinb, $errb, $pcmb, $rdd, $stdind, $errd, $rv);

my $playing = 2;
$rdb = $errb = $stdinb = "";

$SIG{TERM} = $SIG{INT} = sub { $done = 1; };

while(!$done) {
   my $rv;
   usleep(100);
   if($rv = sysread(RDF, $rdb, 1024, 0)) {
      print STDOUT $rdb;
   }
   if($errb !~ m/\n/) {
      $rv = sysread(ERRF, $errb, 1024, length($errb));
      undef $errd;
   }
   else {
      ($errd, $errb) = split /\n/, $errb, 2;
   }
```

```
if($stdinb !~ m/\n/) {
   $rv = sysread(STDIN, $stdinb, 1024, length($stdinb));
   undef $stdind;
}
else {
   ($stdind, $stdinb) = split /\n/, $stdinb, 2;
}

if($errd) {
   if($errd =~ m/^\@P\s+(\S+)(?:\s+(\S+))?/) {
      $playing = $1;
      if($playing == 0) { print STDERR "Playing stopped.".
                  ($2?"(End of song)\n":"\n"); }
      if($playing == 1) { print STDERR "Paused.\n"; }
      if($playing == 2) { print STDERR "Unpaused.\n"; }
      if(!$playing and $2) {
         &nextsong();
      }
   }
}
if($stdind) {
   if($stdind =~ m/^\s*quit\s*$/i) {
      print WRF "quit\n";
      $done = 1;
   }
   if($stdind =~ m/^\s*add\s+(.*)\s*$/i) {
      push @playlist, $1;
   }
   if($stdind =~ m/^\s*next\s*$/i) {
      &nextsong();
   }
   if($stdind =~ m/^\s*prev(ious)?\s*$/i) {
      &prevsong();
   }
   if($stdind =~ m/^\s*random\s*$/i) {
      $random = 1-$random;
      if($random) { print STDERR "Random on.\n"; }
      else { print STDERR "Random off.\n"; }
   }
}
```

```
}

kill 1, $pid;
print STDERR "Waiting for children to die...\n";
wait();

sub nextsong {
   if($random) {
      $cursong = int(rand(@playlist));
   }
   else {
      $cursong++;
      if($cursong >= @playlist) {
         $cursong = 0;
      }
   }
   print WRF "load $playlist[$cursong]\n";
}

sub prevsong {
   if(!$random) {
      $cursong--;
      if($cursong < 0) {
         $cursong = $#playlist;
      }
   }
   print WRF "load $playlist[$cursong]\n";
}
```

A.4 CtrlMP3.py

This script is used as the music player for the extended Juke service in section 5.5:

```
#!/usr/bin/python

from twisted.internet import protocol, reactor, stdio, task
from twisted.protocols import basic
from twisted.cred import credentials
```

```
from twisted.spread import pb

import re, os, sys, random, juke

class Mpg123Protocol(protocol.ProcessProtocol):
   _errbuffer = ”
   delimiter = ’\n’
   runtime = pcmtime = 0

   def connectionMade(self):
      self.ctrl.play()
      self.playing = 1
      self.loopcall = task.LoopingCall(self.checktime)
      self.loopcall.start(1.0)

   def checktime(self):
      if self.playing: self.runtime += 1
      if self.pcmtime-self.runtime <= 0.5:
            self.transport.resumeProducing()

   def write(self, text):
      self.transport.write("%s\n" % text)

   def errReceived(self, data):
      lines  = (self._errbuffer+data).split(self.delimiter)
      self._errbuffer = lines[-1]
      for line in lines[:-1]:
         self.lineReceived(line)

   def lineReceived(self, data):
      m = re.match(r"^\@P\s+(\S+)(?:\s+(\S+))?", data)
      if m:
         playing = int(m.group(1))
         if playing == 0:
            if m.group(2):
               self.ctrl.nextsong()
      else:
         m = re.match(r"^\@F\s+(\d+)\s+(\d+)\s+(\d+\.\d+)\s+(\d+\.\d+)\
                                                      \s+(\d+)", data)
         if m:
            self.pcmtime = float(m.group(3))
```

```
                if self.pcmtime-self.runtime > 0.5:
                    self.transport.pauseProducing()

        def outReceived(self, data):
            self.stdinout.write(data)

    ##########################################
    class StdinProtocol(basic.LineReceiver):
        from os import linesep as delimiter

        def lineReceived(self, line):
            if line == "quit": self.ctrl.done()
            if line == "stop": self.ctrl.stop()
            if line == "play": self.ctrl.play()
            if line == "pause":
                self.ctrl.pause()
            if line == "next": self.ctrl.nextsong()
            if line == "prev": self.ctrl.prevsong()

        def write(self, data):
            self.transport.write("%s" % data)
    ##########################################
    class JukePBClient(pb.Referenceable):
        __implements__ = (pb.Referenceable.__implements__,)

        def __init__(self):
            self.connected = False
            self.queuedcmds = []

        def sourceConnected(self, perspective):
            self.service = perspective
            self.connected = True
            if len(self.queuedcmds):
                for z in self.queuedcmds:
                    if type(z) == list:
                        self.service.callRemote(z[0]).addCallback(z[1])
                    else:
                        self.service.callRemote(z)

        def prev(self):
            if self.connected: self.service.callRemote("prev")
```

```
            else:
               self.queuedcmds.append("prev")

        #identical methods for play, pause, stop etc
         def getSong(self, callback):
            if self.connected:
               self.service.callRemote("getCurr").addCallback(callback)
            else: self.queuedcmds.append(["getCurr",callback])
         def remote_stop(self): self.ctrl.stop(False)
         def remote_play(self): self.ctrl.play(False)
         def remote_pause(self): self.ctrl.pause(False)

      ##########################################
      class Controller:
         def __init__(self, m, j):
            self.mpg123p = m
            self.jpbc = j
         def done(self):
            self.mpg123p.write("quit\n");
            reactor.stop()
         def nextsong(self):
            self.jpbc.next()
         def prevsong(self):
            self.jpbc.prev()
         def stop(self, pbc=True):
            self.mpg123p.playing = 0
            self.mpg123p.write("stop\n")
            if pbc: self.jpbc.stop()
         def play(self, pbc=True):
            if pbc: self.jpbc.play()
            self.jpbc.getSong(self.playthis)
         def playthis(self, song):
            self.mpg123p.playing = 1
            self.mpg123p.runtime = self.mpg123p.pcmtime = 0
            self.mpg123p.write("load %s\n" % song)
         def pause(self, pbc=True):
            self.mpg123p.playing = 1 - self.mpg123p.playing
            self.mpg123p.write("pause\n")
            if pbc: self.jpbc.pause()
         def setRoot(self, rootobj):
            self.service = rootobj
```

```
        self.service.callRemote("loginAuto", self.id,
           self.jpbc).addCallback(self.jpbc.sourceConnected)

m = Mpg123Protocol()
s = StdinProtocol()
j = JukePBClient()
c = Controller(m, j)

j.ctrl = m.ctrl = s.ctrl = c
m.stdinout = s

procargs = ["/usr/bin/mpg123", "-f", "8192", "--mono", "-r", "8000",
                               "-R", "-s" ]
if __name__ == "__main__":
   if len(sys.argv) > 1:
      id = sys.argv[1]
   else:
      id = "default"
   c.id = id
   stdio.StandardIO(s)

   pbfactory = pb.PBClientFactory()
   reactor.connectTCP("localhost", 8800, pbfactory)
   pbfactory.getRootObject().addCallback(c.setRoot)

   reactor.spawnProcess(m, '/usr/bin/mpg123', procargs)

   reactor.run()
```

A.5 GTKJuke.py

This class is used to create the standalone client for the Juke service in section 5.6.

```
#!/usr/bin/env python
if __name__ == "__main__":
   from twisted.internet import gtk2reactor
   gtk2reactor.install()
from twisted.internet import reactor
```

```
from twisted.application import internet, service
from twisted.cred import credentials
from twisted.spread import pb
import gtk, sys
def debug(s):
   print s
class GTKJukeClient(pb.Referenceable):
   __implements__ = (pb.Referenceable.__implements__,)

   def __init__(self, pbfactory):
      self.setup()
      self.pbfactory = pbfactory
      self.loggedin = False

   def sourceConnected(self, perspective, username):
      debug("Connected with %s" % perspective)
      self.service = perspective
      self.username = username
      self.dologin()

   def dologin(self):
      self.label.set_text("GTKJuke for %s" % self.username)
      self.loggedin = True
      self.loginbutton.set_label("_Logout")
      self.status.set_text("Logged in")

      for z in [self.playbutton, self.pausebutton, self.prevbutton,
            self.nextbutton, self.stopbutton]:
         z.set_sensitive(gtk.TRUE)

   def dologout(self):
#     self.service.callRemote("logout")
      self.pbfactory.disconnect()
      reactor.connectTCP("localhost", 8800, self.pbfactory)

      self.service = None
      self.username = None
      self.label.set_text("GTKJuke")
      self.loggedin = False
      self.loginbutton.set_label("_Login")
      self.status.set_text("Logged out")
```

```
        for z in [self.playbutton, self.pausebutton, self.prevbutton,
              self.nextbutton, self.stopbutton]:
           z.set_sensitive(gtk.FALSE)

    def loginFailed(self, err, username):
        self.passwordent.set_text("")
        self.status.set_text("Login failed")

    def setup(self):
        self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)

        self.window.set_title("GTKJuke")

        self.window.connect("destroy", lambda wid: reactor.stop())
        self.window.connect("delete_event", lambda a1,a2: reactor.stop())

        #standard GTK setup code
        entry = gtk.Entry()
        entry.connect("activate", self.login, entry)
        entry.select_region(0, len(entry.get_text()))
        subhbox.pack_start(entry, gtk.TRUE, gtk.TRUE, 0)
        entry.show()

        #more GTK code

        button = gtk.Button("_Login")
        subhbox.pack_start(button, gtk.TRUE, gtk.TRUE, 10)
        button.set_sensitive(gtk.TRUE)
        button.show()
        button.connect("clicked", self.login, self)
        self.loginbutton = button
        button = gtk.Button("E_xit")
        subhbox.pack_start(button, gtk.TRUE, gtk.TRUE, 10)
        button.set_sensitive(gtk.TRUE)
        button.show()
        button.connect("clicked", self.exit, self)
        vbox.pack_start(subhbox, gtk.TRUE, gtk.TRUE, 0)
        subhbox.show()

        button = gtk.Button("P_revious")
        hbox.pack_start(button, gtk.TRUE, gtk.TRUE, 10)
```

```
            button.set_sensitive(gtk.FALSE)
            button.show()
            button.connect("clicked", self.prev, self)
            self.prevbutton = button
            #more code for other buttons

            vbox.show()
            self.window.show()
        def exit(self, widget, entry):
            gtk.main_quit()

        def login(self, widget, entry):
            if self.loggedin:
                self.dologout()
            else:
                u, p = (self.usernameent.get_text(), self.passwordent.get_text())
                d = self.pbfactory.login(credentials.UsernamePassword(u,p),
                             client=self)
                d.addCallback(self.sourceConnected, u)
                d.addErrback(self.loginFailed, u)

        def showret(self, s):
            pass

        def prev(self, widget, data):
            self.status.set_text("Previous")
            d = self.service.callRemote("prev")
            d.addCallback(self.showret)
        def remote_prev(self):
            self.status.set_text("[Previous]")
        #duplicate code for other buttons
    pbfactory = pb.PBClientFactory()
    gjc = GTKJukeClient(pbfactory)
    if __name__ == "__main__":
        reactor.connectTCP("localhost", 8800, pbfactory)
        reactor.run()
    else:
        application = service.Application("GTKJukeClient")
        internet.TCPClient("localhost", 8800,
              pbfactory).setServiceParent(application)
```

A.6 NevowJuke.py

This class is used to create the web Juke client in section 5.7.

```
from twisted.application import internet, service
from twisted.spread import pb
from twisted.cred import credentials, checkers, error
from twisted.python import failure

from nevow import appserver
from nevow import loaders
from nevow import rend
from nevow import tags
from nevow import guard

from nevow import vhost

import juke

class NotLoggedIn(rend.Page):
   docFactory = loaders.stan(
      tags.html[
         tags.head[tags.title["Not Logged In"]],
         tags.body[
            tags.form(action=guard.LOGIN_AVATAR)[
               tags.table[
                  tags.tr[
                     tags.td[ "Username:" ],
                     tags.td[ tags.input(type='text',name='username') ],
                  ],
                  tags.tr[
                     tags.td[ "Password:" ],
                     tags.td[ tags.input(type='password',name='password') ],
                  ]
               ],
               tags.input(type='submit'),
               tags.p,
            ]
         ]
      ])
```

```
    def locateChild(self, request, cseg):
        return self, ()

class NevowJukeClient(rend.Page, pb.Referenceable):
    __implements__ = (pb.Referenceable.__implements__,rend.Page.__implements__)
    docFactory = loaders.htmlfile('nevowjuke.html')

    def sourceConnected(self, perspective, username):
        self.service = perspective
        self.username = username
        self.remoteactions = []

    def render_action(self, context, data):
        ret = loaders.stan([])
        for z in self.remoteactions:
            ret = loaders.stan([ret,tags.p["%s" % z]])
        self.remoteactions = []
        try:
            m = getattr(self, self.action)
        except AttributeError:
            return ""
        return loaders.stan([ret,tags.p[m()]]).load()

    def locateChild(self, request, cseg):
        if cseg[0] != '':
            self.action = cseg[0]
        return self, ()

    def prev(self):
        d = self.service.callRemote("prev")
        return "Previous"
    def remote_prev(self):
        self.remoteactions.append("[Previous]")
    #identical code for other buttons

    def render_logout(self, context, data):
        return loaders.stan(tags.a(href=guard.LOGOUT_AVATAR)["Logout"])

    def render_username(self, context, data):
        return self.username
```

```
    def logout(self):
        print "%s logged out" % self.original

class reAuthChecker:
    __implements__ = (checkers.ICredentialsChecker,)
    credentialInterfaces = (credentials.IUsernamePassword,
                            credentials.IUsernameHashedPassword)

    def __init__(self, pbclient, pbfactory):
        self.pbclient = pbclient
        self.pbfactory = pbfactory

    def connectSource(self, persp, uname):
        self.pbclient.sourceConnected(persp, uname)
        return uname

    def loginFailed(self, exc):
        return failure.Failure(error.UnauthorizedLogin())

    def requestAvatarId(self, c):
        d = pbfactory.login(c, self.pbclient)
        d.addCallback(self.connectSource, c.username)
        d.addErrback(self.loginFailed)
        return d

application = service.Application('NevowJukeClient')

pbfactory = pb.PBClientFactory()
internet.TCPClient("localhost", 8800,
            pbfactory).setServiceParent(application)

####
pbrealm = juke.JukeRealm(juke.JukeService)
pbp = juke.JukePortal(pbrealm, [juke.iLangaMySQLDB()])
internet.TCPServer(8800,
            pb.PBServerFactory(pbp)).setServiceParent(application)
####

njc = NevowJukeClient()
```

```
realm = juke.JukeRealm(njc,NotLoggedIn)
p = juke.JukePortal(realm, [reAuthChecker(njc, pbfactory)])
p.registerChecker(checkers.AllowAnonymousAccess(),
            credentials.IAnonymous)

site = appserver.NevowSite(resource = guard.SessionWrapper(p))
webServer = internet.TCPServer(8080, site)
webServer.setServiceParent(application)
```

A.7 the Juke class

This class provides common functionality for all the services created in chapter 5.

```
#!/usr/bin/python

from twisted.cred import portal, checkers, credentials, error
from twisted.spread import pb
from twisted.internet import defer
from twisted.python import failure
import dbase, juke, md5

from nevow import inevow, rend, tags, appserver, guard, loaders

class Playlist:
   def __init__(self, username):
      self.username = username
      self.db = dbase.dbase("localhost","root","qbasic","juke")
      self.db.query("SELECT * FROM playlists WHERE username='%s'" %
                        dbase.esc(self.username))
      f = self.db.fetchall()
      if not f:
         self.db.query("INSERT INTO playlists SELECT '%s', entry, "+
                   "number FROM playlists WHERE username='default'" %
                    dbase.esc(self.username))
         self.db.query("SELECT * FROM playlists WHERE username='%s'" %
                    dbase.esc(self.username))
         f = self.db.fetchall()
```

```
        self.db.query("SELECT * FROM settings WHERE username='%s'" %
                      dbase.esc(self.username))
        s = self.db.fetch()
        if not s:
           self.db.query("INSERT INTO settings SELECT '%s', current, "+
                      "random, repeat FROM settings WHERE username='default'" %
                       dbase.esc(self.username))
           self.db.query("SELECT * FROM settings WHERE username='%s'" %
                       dbase.esc(self.username))
           s = self.db.fetch()
        t = [(d["number"], d["entry"]) for d in f]
        t.sort(lambda x,y: cmp(x[0], y[0]))
        self.playlist = zip(*t)[1]
        self.settings = s

        if self.settings["current"] >= len(self.playlist):
             self.settings["current"] = 0

        self.avatars = []

    def addAvatar(self, avatar):
        self.avatars.append(avatar)

    def removeAvatar(self, avatar):
        self.avatars.remove(avatar)

    def getCurr(self):
        return self.playlist[self.settings["current"]]

    def remove(self, entnum):
        if entnum < len(self.playlist): del(self.playlist[entnum])
        if self.settings["current"] > entnum: self.settings["current"] -= 1
        if self.settings["current"] == entnum: self.next()

    def add(self, entry):
        self.playlist.append(entry)

    def prev(self):
        if self.settings["random"]:
           self.settings["current"] = random.randint(0,len(self.playlist)-1)
        else:
```

```
            self.settings["current"] -= 1
            if self.settings["current"] < 0:
                self.settings["current"] = len(self.playlist)-1
         self.play()
      #similar code for other buttons
      def jump(self, entnum):
         if entnum < len(self.playlist):
            self.settings["current"] = entnum
            self.play()

   class JukeRealm:
      __implements__ = portal.IRealm

      def __init__(self, avatar, anonavatar = None):
         self.avatar = avatar
         self.anonavatar = anonavatar
         self.playlists = {}

      def requestAvatar(self, avatarID, mind, *interfaces):
         for iface in interfaces:
            if iface is pb.IPerspective:
               juke.debug("Avatar for %s requested by %s" % (avatarID, mind))
               assert pb.IPerspective in interfaces
               avatar = self.avatar(avatarID)
               if avatarID not in self.playlists:
                  self.playlists[avatarID] = Playlist(avatarID)
               avatar.playlist = self.playlists[avatarID]
               avatar.attached(mind)
               return pb.IPerspective, avatar, lambda a=avatar:a.detached(mind)
            if iface is inevow.IResource:
               if avatarID is checkers.ANONYMOUS:
                  resc = self.anonavatar()
                  resc.realm = self
                  return (inevow.IResource, resc, lambda _=None:None)
               else:
                  resc = self.avatar
                  resc.realm = self
                  return (inevow.IResource, resc, resc.logout)
         raise NotImplementedError("Can't support that interface.")

   class iLangaMySQLDB:
```

```
    __implements__ = (checkers.ICredentialsChecker,)
    credentialInterfaces = (credentials.IUsernamePassword,
                            credentials.IUsernameHashedPassword)

    def __init__(self):
        self.db = dbase.dbase("pbx.ict.ru.ac.za","vhata","qbasic","asterisk")

    def _cbPasswordMatch(self, matched, username):
        if matched:
            return username
        else:
            return failure.Failure(error.UnauthorizedLogin())

    def getUser(self, uname):
        self.db.query("select mailbox, password from users where mailbox='%s'"
                      % dbase.esc(uname))
        f = self.db.fetch()
        if not f: raise KeyError
        return (f["mailbox"], f["password"])

    def requestAvatarId(self, c):
        try:
            u, p = self.getUser(c.username)
        except KeyError:
            return failure.Failure(error.UnauthorizedLogin())
        else:
            up = credentials.IUsernamePassword(c, default=None)
            return defer.maybeDeferred(c.checkPassword, p
                     ).addCallback(self._cbPasswordMatch, u)

class JukePortal(pb.Root, portal.Portal):
    def rootObject(self, broker):
        self.broker = broker
        return self
    def remote_login(self, username):
        c = pb.challenge()
        self.portal = self
        return (c,pb._PortalAuthChallenger(self, username, c))
    def remote_loginAuto(self, avatarID, mind):
        interface, perspective, logout = self.realm.requestAvatar(avatarID,
                      mind, pb.IPerspective)
```

```
            self.broker.notifyOnDisconnect(logout)
            return pb.AsReferenceable(perspective, "perspective")

    class JukeService(pb.Avatar):
        __implements__ = (pb.Avatar.__implements__,)

        def __init__(self, username):
            pass

        def attached(self, mind):
            self.remote = mind
            self.playlist.addAvatar(self)

        def detached(self, mind):
            self.remote = None
            self.playlist.removeAvatar(self)

        def perspective_getCurr(self):
            return self.playlist.getCurr()
        #identical code for other functions

        def perspective_logout(self):
            self.detached(self.remote)

        def pause(self):
            self.remote.callRemote("pause")
        def play(self):
            self.remote.callRemote("play")
        def stop(self):
            self.remote.callRemote("stop")
```

A.8 Flash Actionscript: Navigation bar

This actionscript sits behind the navigation bar at the bottom of the screen, and connects to the Asterisk server using the XML Socket to get the current status of the connection.

```
//================================
```

```
//nav statusbar
//================================
statusData = new LoadVars();
statusData.username = _global.username;//"7000";
today_date = new Date();
diff = today_date.getTimezoneOffset();
milli_date = today_date.setMinutes((today_date.getMinutes() + diff));
today_date = new Date(milli_date);

date_str = (today_date.getFullYear() + "-" + (today_date.getMonth() + 1)
   + "-" + today_date.getDate());

statusData.datestring = date_str;
statusData.onLoad = function(success) {
   if (success){
      status_prepaidbalance = "R" + statusData.prepaid_balance/100;
      status_diallednums=statusData.diallednums;
      status_receivedcalls=statusData.receivedcalls;
      status_missedcalls=statusData.missedcalls;
   }
}
var l;
l = this._parent.mc_statusbar.mc_jason.leds;

leds.setstate(leds.DOWN);
devicebusycounter = 0;
state = "init";

myXMLSocket = new XMLSocket();
myXMLSocket.onConnect = myOnConnect;
myXMLSocket.onClose = myOnClose;

myXMLSocket.onXML = myOnXML;
myXMLSocket.parent = this;
if (!myXMLSocket.connect('pbx.ict.ru.ac.za',8305)) {
   connect_status = "Connect failed";
}
function myOnConnect(success) {
   if (success) {
      connect_status = "";//"Connected";
```

```
        myAuth = "Action: Login\r\nUsername: " + _global.username +
                    "\r\nSecret: " + _global.passwd + "\r\n\r\n";
        myXMLSocket.send(myAuth);   //"Action: Login\r\nUsername: 7000"+
                     "\r\nSecret: 7000\r\n\r\n");
        myStr = "Action: MailboxCount,Mailbox: " + _global.username +
                     ",ActionID: 1;\r\n";
        myXMLSocket.send(myStr);

        statusData.sendAndLoad("http://pbx.ict.ru.ac.za/iLanga/statuscdr.php",
                      statusData,"POST");
    } else {
        connect_status = "not connected";
        gotoAndPlay(2);
    }
}
function myOnClose() {
    connect_status = "Disconnected";
    gotoAndPlay(2);
}
function myOnXML(doc) {

    receivedmessage = doc.toString();

    if (state eq "init") {
      onemessage = receivedmessage.split("\r\n\r\n");
      connect_status = onemessage.length;
      messageportions=(onemessage[0]).split("\r\n");
      connect_status = messageportions.length;
      if (messageportions[0] eq "Response: Success"){
       if (messageportions[1] eq "Message: Authorization successful"){
        connect_status = "";//"Authenticated";
        state = "authd";
       }
      }
    }
    else {
        onemessage = receivedmessage.split("\r\n\r\n");
        messageportions=(onemessage[0]).split("\r\n");

        if (messageportions[0] eq "Event: ExtState"){
```

```
      exten = messageportions[1].substring(11);
      extenstate = messageportions[2].substring(7);
      if (extenstate eq "busy") {
        this.parent._parent._parent.nav.container.holder.directory.
                        setbutstate(exten,"up");
        if (exten == _global.username) {
          leds.setstate(leds.UP);
        }
    }
    else {
        this.parent._parent._parent.nav.container.holder.directory.
                        setbutstate(exten,"down");
        if (exten == _global.username) {
          leds.setstate(leds.DOWN);
        }
    }
  }

  if (messageportions[0] eq "Event: Newstate"){
    if (messageportions[1].indexOf("Channel") >= 0){
     if (messageportions[2].indexOf("State: Ringing") >= 0) {
        leds.setstate(leds.RINGING);
     }
     if (messageportions[2].indexOf("State: Up") >= 0) {
        leds.setstate(leds.UP);
        devicebusycounter += 1;
     }
   }
  }

  if (messageportions[0] eq "Event: Newchannel"){
    if (messageportions[1].indexOf("Channel") >= 0){
     if (messageportions[2].indexOf("State: Ringing") >= 0) {
        leds.setstate(leds.RINGING);
     }
     if (messageportions[2].indexOf("State: Up") >= 0) {
        leds.setstate(leds.UP);
        devicebusycounter += 1;
     }
   }
  }
```

```
        if (messageportions[0] eq "Event: Hangup"){
     if (messageportions[1].indexOf("Channel") >= 0){
           devicebusycounter -= 1;
           if (devicebusycounter <= 0){
         leds.setstate(leds.DOWN);//green(true);
         statusData.sendAndLoad("http://pbx.ict.ru.ac.za/iLanga/statuscdr.php",
                     statusData,"POST");
           }

     }
       }
    if (messageportions[0] eq "Event: MessageWaiting"){
     myStr = "Action: MailboxCount,Mailbox: " + _global.username +
                          ",ActionID: 1;\r\n";
     myXMLSocket.send(myStr);
    }
    if (messageportions[0] eq "Event: Status"){
     if (messageportions[1].indexOf("Prepaid_balance") >= 0){
        status_prepaidbalance = "R" + messageportions[1].substr(17);
     }
    }

    if (messageportions[0] eq "Response: Success"){
     if (messageportions[2].indexOf("Mailbox Message Count") >= 0){
        if (messageportions[4].indexOf("NewMessages") >= 0){
           status_voicemail = messageportions[4].substr(13);
        }
     }
    }
   }
}
status_prepaidbalance="-";
status_voicemail="-";
status_missedcalls="-";
status_diallednums="-";
status_receivedcalls="-";
function sendMgrCmd(theMessage) {
   myXMLSocket.send(theMessage);
}
stop();
```

A.9 Flash Actionscript: User directory

This actionscript is used to connect to the webserver to query the user directory.

```
//======================
//directory
//======================
personaluserdata = new LoadVars();
personaluserdata.fname = "";
personaluserdata.lname = "";

var personalfnameasc = true;
var personallnameasc = true;
var personalextensionasc = true;
personaluserdata.onLoad = function (success)
{
   if (success) {
      myPersonalNames = personaluserdata.names.split(",");
      myPersonalExtensions = personaluserdata.extensions.split(",");
      mydirSP.contentPath="personalusersclip";
   }
}

function personal_dosearch() {
   this.personaluserdata.lname = this.mydir_lname.text;
   this.personaluserdata.fname = this.mydir_fname.text;
   this.personaluserdata.order = "order by lastname asc";
   this.personaluserdata.username = _global.username;
   this.personaluserdata.password = _global.passwd;

   this.personaluserdata.sendAndLoad(
     "http://pbx.ict.ru.ac.za/iLanga/search_personal_dir.php",
              this.personaluserdata,"POST");
}
personal_dosearch();
stop();

//============================
//personalusersclip
//============================
```

```
for(n = 0; n < _parent._parent.myPersonalNames.length; n++) {
   theName = _parent._parent.myPersonalNames[n].split(":");
   var ext = _parent._parent.myPersonalExtensions[n];
   this.attachMovie("userclip","user"+ext,100+n);
   this["user"+ext].num = n;
   this["user"+ext]._y = this["user"+ext]._height*n;//+2 + n*5;
   this["user"+ext].lname.text = theName[0];//_parent._parent.myNames[n];
   this["user"+ext].fname.text = theName[1];
   this["user"+ext].number.text = ext;
}
function disableself(onoff) {
   for(n = 0; n < _parent._parent.myNames.length; n++) {
      var ext = _parent._parent.myExtensions[n];
      this["user"+ext].enabled = not onoff;
   }
   if(not onoff) {
      this.himc.userunselected._visible=true;
      this.himc.userselected._visible=false;
   }
}
stop();
```

A.10 Flash Actionscript: Call Records

This actionscript is used to access the call records of the user from the server.

```
//=======================================
//calls
//=======================================
function doSearch() {
   this.cdrdata = new LoadVars();
   this.cdrdata.parent = this;
   this.cdrdata.username =   _global.username;
   this.cdrdata.password = _global.passwd;

   var srcordst = {dialled: "dst", received: "src"};

   this.cdrdata[srcordst[this.which]] = this.srcdst.text;
```

```
    this.cdrdata.type = this.which;

    var selDate = this.date_from.selectedDate;
    var theDate = selDate.getFullYear() + "-" + (selDate.getMonth()+1)
                      + "-" + selDate.getDate();
    this.cdrdata.fromdate = theDate;
    this.cdrdata.matchmethod = this.matchmethod.selectedRadio.__data;
    this.cdrdata.onLoad = function(success) {
     if(success){
      this.mydatetimes = this.datetimes.split(",");
      this.mydispositions = this.dispositions.split(",");
      this.mydurations = this.durations.split(",");
      this.mytype = this.type;

      if (this.mytype == "dialled") {
        this.mybillsecs = this.billsecs.split(",");
        this.mydsts = this.dsts.split(",");

        this.parent.cdrSP.contentPath="cdrsclip";
      }
      else {
        this.mysrcs = this.srcs.split(",");
        this.parent.cdrSP.contentPath="cdrsclip";
      }
     }
    }
    this.cdrdata.sendAndLoad("http://pbx.ict.ru.ac.za/iLanga/getcdr"+
                      this.which+".php",this.cdrdata,"POST");
}
stop();

//=============================================
//prepaid
//=============================================
pp_details = new LoadVars();
pp_details.onLoad = function(success){
    if(success){

       if (this.result == "error") {
     Alert.show("An error has occurred. Please contact admin",
            "Prepaid Setup", Alert.OK, this.parent, undefined, "", Alert.OK);
```

```
        }
        else {
     req_auth.selected = this.prepaid_req_auth;
     play_balance.selected = this.prepaid_play_balance;
     play_timeout.selected = this.prepaid_play_timeout;

        }

    }
    else {
        trace("Error");
        Alert.show("An error has occurred. Please contact admin",
             "Prepaid Setup", Alert.OK, this.parent, undefined, "", Alert.OK);
    }
}
pp_details.username = _global.username;
pp_details.password = _global.passwd;
pp_details.sendAndLoad("http://pbx.ict.ru.ac.za/iLanga/getppdetails.php",
               pp_details,"POST");
req_authListener = new Object();
req_authListener.click = function (evt){
    if (evt.target.selected) {
        pp_details.prepaid_req_auth = 1;
    }
    else {
        pp_details.prepaid_req_auth = 0;
    }

    pp_details.sendAndLoad("http://pbx.ict.ru.ac.za/iLanga/updateppdetails.php",
                  pp_details,"POST");
}
req_auth.addEventListener("click", req_authListener);

play_timeoutListener = new Object();
play_timeoutListener.click = function (evt){
    trace("sending");
    if (evt.target.selected){
        pp_details.prepaid_play_timeout = 1;
    }
    else {
        pp_details.prepaid_play_timeout = 0;
```

```
    }

    pp_details.sendAndLoad("http://pbx.ict.ru.ac.za/iLanga/updateppdetails.php",
                pp_details,"POST");
}
play_timeout.addEventListener("click", play_timeoutListener);

play_balanceListener = new Object();
play_balanceListener.click = function (evt){
    if (evt.target.selected) {
        pp_details.prepaid_play_balance = 1;
    }
    else {
        pp_details.prepaid_play_balance = 0;
    }

    pp_details.sendAndLoad("http://pbx.ict.ru.ac.za/iLanga/updateppdetails.php",
                pp_details,"POST");
}
play_balance.addEventListener("click", play_balanceListener);
stop();
```