An Investigation into the Provision of Video Capabilities in iLanga

Submitted in partial fulfilment
of the requirements of the degree
Bachelor of Science (Honours)
in Computer Science
at Rhodes University

Fred Otten

Supervisors:
Prof Alfredo Terzoli
Prof Peter Clayton

Abstract

iLanga is a complete, cost effective, computer based voice private branch exchange (PBX). It is capable of connecting different endpoints using different protocols and delivering high quality voice with value-added services. It is not, however, capable of video transport. The channel based architecture of Asterisk, the core of iLanga, should, in principle, allow the transport of video with relative ease. This paper introduces the relevant protocols used in real time multimedia, provides background on Asterisk, iLanga and their relationship and explores the channel architecture, shedding light on the channel API and the source code of the Session Initiation Protocol (SIP) channel. It also shows how video capabilities are made available in iLanga, detailing the configuration of SIP video within Asterisk. It also takes a look at possible H.323 implementations and discusses their lack of video support. The iLanga user interface and the extensions made for call parking and call transfer are also discussed. Through this paper we highlight the provisions made for video calls in iLanga with associated services such as call transfer, call parking and music on hold.

Acknowledgements

I would firstly like to thanks my supervisors, Prof Alfredo Terzoli and Prof Peter Clayton for their constant support and patience through out the whole year. Thanks for proof reading my work, providing me with advice and steering me in the right direction from the beginning to the end. I would like to thank Jason Penton and Bradley Clayton for their advice and patience. Thanks for taking the time to help me with installation problems. Thanks to Dr Hannah Slay for proof reading my paper and assisting me in correcting my writeup. I would like to thank my friend and classmate Justin Zondagh for his assistance and ideas through out the year, especially while we were learning Asterisk, and delving into the code. Lastly I would like to thank the whole Honours class for their support and encouragement in the labs through out the year. Without all the help from these people, this project would not nearly have been what it is.

I must also acknowledge the financial and technical support of this project from Telkom SA, Business Connexion, Comverse SA and Verso Technologies through the Telkom Centre of Excellence at Rhodes University.

Contents

1 Introduction **8**

1.1 Motivation . . . 9

1.2 The Problem . . . 10

1.3 Approach . . . 10

1.4 My Project aims . . . 10

1.5 Writeup Structure . . . 11

2 Relevant Standards **12**

2.1 Protocol stack . . . 12

2.2 Signalling Protocols . . . 13

2.2.1 SIP . . . 13

2.2.2 H.323 . . . 17

2.3 Media Protocols . . . 21

2.3.1 RTP . . . 21

2.4 Summary . . . 23

3 Asterisk and iLanga **24**

3.1 Asterisk . . . 24

3.1.1 Architecture of Asterisk . . . 25

3.1.2 Configuring Asterisk . . . 26

3.1.3 Facilities Provided . . . 26

3.1.4 Extending Asterisk . . . 27

3.2 iLanga . . . 28

3.2.1 Architecture . . . 28

3.2.2 User interface . . . 30

3.2.3 iLanga, Asterisk and Video . . . 31

3.3 Summary 31

4 Channels in Asterisk and their Implementation 33

4.1 Channel Concept 33
4.2 A call in Asterisk 34
4.3 The dial application 34
4.3.1 What is the dial application 34
4.3.2 Calling the dial application 35
4.3.3 Extensions and the dial application 36
4.3.4 How the dial application operates 36
4.3.5 Importance of understanding the dial application 37
4.4 The channel structure 39
4.5 The channel API 40
4.6 Frames in Asterisk 46
4.7 How to design a channel in Asterisk 47
4.7.1 Modules in Asterisk 48
4.7.2 Adding commands to the CLI 48
4.7.3 Outputting to the logs 49
4.7.4 The example channel 49
4.8 Summary 50

5 The SIP and H.323 Channels 51

5.1 The SIP Channel 51
5.1.1 The SIP channel driver 51
5.1.2 Channel registration 52
5.1.3 Devicestate 52
5.1.4 The SIP Private Structure 52
5.1.5 Media Handling 53
5.1.6 Signalling 54
5.2 SIP Video 55
5.2.1 Testing Environment 55
5.2.2 Configuration 56
5.2.3 Results 57
5.3 H323 Channel Implementations 58
5.3.1 The OH323 channel 58

5.3.2 The H323 channel 59
5.3.3 The OOH323 channel 59
5.3.4 Channel Woomera 59
5.4 No H323 Video 60
5.5 Inheritance of features 61
5.6 Summary 62

6 iLanga User Interface and Extensions 63
6.1 Architecture 63
6.1.1 Asterisk Manager Interface 64
6.1.2 Python script 64
6.1.3 Extensibility 65
6.2 Extensions 65
6.2.1 Call Transfer 65
6.2.2 Call Parking 67
6.3 Summary 71

7 Conclusion 72
7.1 Document Summary 72
7.2 Video in iLanga 73
7.3 Inheritance of features 73
7.4 Summary of Findings 73
7.5 My project achievements 74
7.6 Further Extensions 74
7.6.1 Video mail and Video on Hold 75
7.6.2 Legacy video channel 75
7.6.3 Video MeetMe application 75
7.6.4 H323 video within a H323 channel 75
7.6.5 H264 codec for cell phone technology 75
7.6.6 Streaming 76
7.7 Final words 76

References 77

A More Channel API Functions 80

B Example Channel **84**

B.1 chan_eg.c 84

C iLanga User Interface Extensions **86**

C.1 directory.fla 86

C.1.1 Action script extracts 86

C.1.2 Graphics and movie clips 87

C.2 nav.fla 88

C.2.1 Action script extracts 88

C.2.2 Graphics and movie clips 91

C.3 ilangaproxy.py 92

List of Figures

2.1 Protocol Stack [29] 12
2.2 SIP Message Structure [29] 14
2.3 SIP Architecture [5] 15
2.4 Establishing a SIP Connection 16
2.5 H.323 network [12] 18
2.6 Protocol relationships in H.323 [12] 19
2.7 The phases of an H.323 call [12] 20
2.8 RTP packet [12] 22

3.1 Asterisk Architecture [33] 25
3.2 iLanga system architecture 29
3.3 iLanga implementation [21] 30
3.4 iLanga graphical frontend [8] 31

4.1 Use of channels between disparate protocols 34
4.2 Dial Application Flow Chart 37
4.3 Parts of the dial application 38
4.4 ast_channel structure 39
4.5 ast_channel_pvt structure 40
4.6 Asterisk frames 46
4.7 ast_frame structure 46

5.1 Test setup for SIP video 55
5.2 Setup of windows messenger 5.0 57
5.3 SIP video in operation 58
5.4 gnomemeeting speaking with netmeeting 60

6.1 Architecture of the web interface 63
6.2 Initiating a call transfer 66
6.3 Confirmation of call transfer 68
6.4 The call transfer tab in the updated iLanga interface 68
6.5 Parking a call in the updated iLanga interface 69
6.6 Indication of having parked a call 70
6.7 The call parking tab in the updated iLanga interface 71

C.1 Call Parking dialog box 87
C.2 Call Parking tab in the directory 87
C.3 Call Transfer dialog box 88
C.4 Call Transfer tab in the directory 88
C.5 Status button when a call is parked 92

List of Tables

2.1 SIP Messages [22, 25, 28, 29] 15
2.2 Descriptions of fields in Figure 2.8 22

3.1 Configuration files in Asterisk 26

4.1 Example arguments for the dial application 35
4.3 Frame Types in Asterisk 47

Chapter 1

Introduction

Internet telephony is a great technology which makes long distance calls possible at the price of a dedicated line or local call (or whatever the fee is for connecting to the internet for the call duration). The integration of data and voice services has been a significant focus in the telecommunications industry. These new networks are called next generation networks and generally operate over IP. Research into sending IP over all sorts of technologies such as ATM, frame relay, and fibre has been significant in the past decade. It is projected that sending voice over IP will save businesses millions, and it is evident that it is changing the way most telecommunications companies world-wide are operating and marketing their services.

There is a significant interest in Voice over IP (VoIP) and next generation networks in both research and industry. Video over IP receives less attention, however is an emerging field. In the past we have seen the use of video telephony for conferences, and recently we have seen the advent of streaming services over broadband internet, third generation (3G) cell phone networks, and intranet local area network (LAN) environments. Video telephony over IP has also been emerging, using the same protocols as VoIP.

Biologists have noted that human beings are more prone to visual cognition, and rely most predominantly on their sight other and above their over sense. Video telephony offers a visual alternative to the current auditory telephony for businesses and home users. Most of the protocols that have been developed for real-time multimedia focus on the use of IP for generic real time transmission, which makes video a possibility using the current frameworks. The only question is whether the larger packets are able to be transferred end to end in an acceptable time. This requires an acceptable amount of bandwidth and a low latency. Henning Schulzrinne, one of the

fathers, and significant contributors to the development of real-time transmission over IP networks says that it "offers the opportunity to design a global multimedia communication system that may eventually replace the existing telephony infrastructure, without being encumbered by the legacy of a century-old technology" [28]. This makes it an exciting field to investigate. The high speed LAN environment provides sufficient bandwidth for video telephony, which makes this investigation viable, as this is the environment in which our PBX runs.

This investigation assumes that the iLanga PBX is in place and aims to find out whether the voice functionality provided can be easily extended to video. The remainder of this chapter defines the research problem, outlining the motivation and approach taken. It also gives a layout of the thesis being presented.

1.1 Motivation

Next generation networks are having a profound impact on the telecommunications industry. These next generation networks need to be capable of high quality voice and video transmission with relevant value-added services. VoIP and Video over IP are vital components of next generation networks. There is a commercial and a practical drive from the industry, which makes these topics very applicable for research.

Next generation PBXs are emerging with facilities for voice and video analogous with the concept of next generation networks. These PBXs are bundled with services such as conferencing, call forwarding and call parking for users of voice and video.

iLanga is a full featured PBX developed at Rhodes University. It currently only provides high quality voice over multiple protocols with services such as voicemail, call forwarding and call parking. In order for iLanga to be a true next generation PBX, it needs to also provide facilities for video. The visual nature of cognition, described earlier, is also a major motivation for this work. Video features along with call parking, call transfer and the audio features provided by iLanga would thus be a useful extension for is users.

1.2 The Problem

The channel-based architecture of Asterisk, which is the main component of iLanga, offers the potential for this extension. Channel modules may use the services provided in Asterisk. These include conferencing, call transfer and call parking. If a new channel module is created and registered with Asterisk, it also inherits the services available in Asterisk. This has been shown to be effective for voice channels, however the extent of this inheritance for video and even the support for video in Asterisk remains unanswered.

The basic problem addressed in this thesis in the investigation of the possibility of including video into the iLanga framework, and determining if the existing features available for voice can be extended and applied to video.

1.3 Approach

The approach taken in this project was largely an incremental one. The initial stages involved reading literature on the various protocols and installing and configuring Asterisk. After that, we began looking into the source code of Asterisk, for which no documentation was available, and developed an understanding of the channel implementation, and the structures involved. It was initially thought that a channel would need to be implemented in Asterisk to provide the possibility of video conferencing, but further investigation revealed that video support was already present in Asterisk for some channels. We produced results for the channels that do support video, and investigated the channels such as H.323 that don't support video. We lastly checked the availability of the features present for voice channels within video channels. For this, it was necessary to implement a facility in the iLanga user interface for call parking and call transfer.

1.4 My Project aims

My project aims include:

- Produce a document which explains channels, and the channel API available.
- Implement an example channel in Asterisk from the knowledge gained.
- Check for the availability of video through testing in Asterisk.

- Explain how video is provided in Asterisk.
- Investigate the inheritance of the features available for voice channels to video channels and report the results.
- Extend the iLanga user interface to provide support for transferring and parking calls.

1.5 Writeup Structure

This thesis begins in Chapter 2: Relevant Standards, by taking a look at the relevant standards applicable to the areas of VoIP and Video over IP. It then moves on and in Chapter 3: Asterisk and iLanga, takes a look at Asterisk and iLanga, discussing the individual systems, their architectures and the relationships between them. Chapter 3 also expands on the problem statement and the relevance of channels. This sets the scene for Chapter 4: Channels in Asterisk and their Implementations, which takes a closer look at channels and the channel API detailing the example channel we have created. Chapter 5: The SIP and H.323 video channels, then expands on the SIP channel driver provided with Asterisk and presents SIP video, detailing its configuration and showing results. It explains the lack of H.323 video in the four H.323 channels presented, and explains why. It finally concludes with a summary of the features that the video channels inherit from voice channels. Chapter 6: iLanga User Interface and Extensions, explains the architecture of the iLanga User Interface that has been developed, and details the extensions made to add call transfer and call parking to this interface. This thesis concludes in Chapter 7: Conclusion, which summarises this thesis and provides details on future work which could be done in this area. Appendixes are also attached containing extracts from the source code, and a few explanations.

Chapter 2

Relevant Standards

This chapter provides an analysis of the different signalling and media protocols applicable to this project. We will begin by taking a look at their positions within in a protocol stack, and then provide more detail on the signalling and media protocols that are applicable to this project. We will expand on the SIP and H.323 signalling protocols and the RTP media protocol.

2.1 Protocol stack

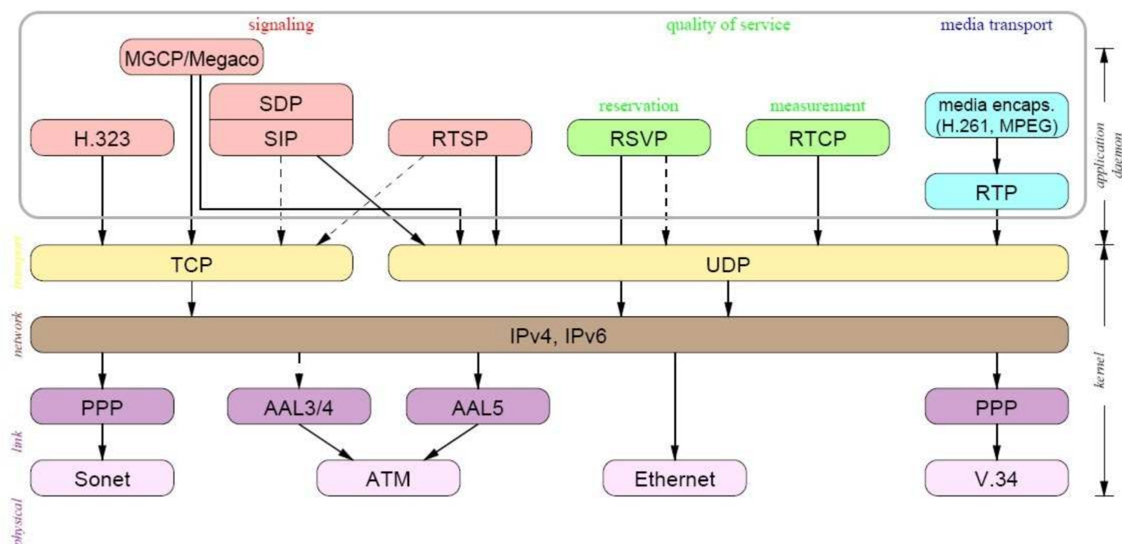


Figure 2.1: Protocol Stack [29]

There are many different protocols which may be used for VoIP or Video over IP. Figure 2.1 illustrates the relationships between the various protocols and where they fit into the Open System Interconnection (OSI) protocol stack. The focus of this research is only from the transport layer up, running over IP. Figure 2.1 illustrates this focus by enclosing the region of interest in a rectangle. The protocols of interest can be broken down into two categories: Signalling Protocols and Media Protocols. SIP and H.323 are examples of signalling protocols, and RTP is the protocol used to encapsulate the media frames for these protocols. It is important to have a good understanding of these protocols when taking a look at their channel drivers and understanding its operation. We will take a brief look at the SIP, H.323 and RTP protocols.

2.2 Signalling Protocols

This section describes two of the standards established for signalling in real time multimedia sessions. We take a brief look at the SIP and H.323 protocols, providing a bit of history, some important details about their architectures and dealing with how a basic session is set up using these protocols.

2.2.1 SIP

SIP [23], as its name suggests, is a client-server protocol designed to establish, modify and terminate sessions. It was designed by the Multiparty Multimedia Session Control (MMUSIC) working group, who have designed a family of protocols for the setup and teardown of realtime multimedia session over the internet. [12]. After many revisions, it was finally approved by the IETF as a proposed standard in 1999 [7]. It has since been updated to take into account pressing needs such as security [1], locating SIP servers [24] and compression [4].

The remainder of this section describes 6 key features of SIP: protocol description, message structure, architecture, operation, the attraction and the applicable concepts to the project.

SIP is a text based protocol, similar in both syntax and semantics to the HTTP (Hypertext transfer protocol) and SMTP (Simple mail transfer protocol) protocols [22]. It makes minimal assumptions about the underlying transport protocol, but usually runs on top of UDP. Both requests and responses are textual [28]. This makes it easy to use text processing languages such as Perl, and textual interfaces such as CGI for developing services [22]. New tags such as language could be easily added to the header and be identified intuitively by programmers [28].

This makes SIP easily expandable.

The SIP protocol is a clean request-response model which makes simple programming possible. Jonathan Rossenberg [22] chooses SIP as the preferred platform for programming Internet telephony services.

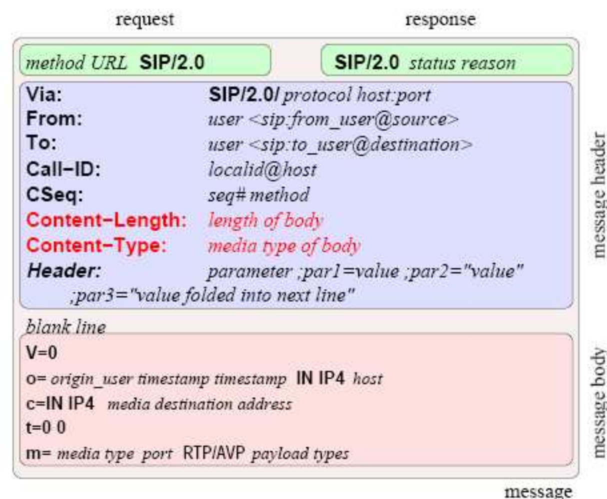


Figure 2.2: SIP Message Structure [29]

Figure 2.2 illustrates the structure of a SIP Message for both requests and responses. The message body contains data of the type specified in the Content-Type field. Another protocol, Session Description Protocol (SDP), is used to send information about the session. In figure 2.2 the message body contains SDP information. The value for the Content-Type field would be `application/sdp` in this case. This is a common value for this field, as SDP information is often sent with a SIP packet.

Message	Function
INVITE	Request to establish a session
BYE	Terminate a session between two end points
OPTIONS	Deal with information related to capabilities of an end point
STATUS	Informs another server about progress of signalling actions requested
ACK	Used for reliable message exchange
PRACK	Provisional acknowledgement
REGISTER	Convey information to server about end point
CANCEL	Terminate search for an end point
INFO	Mid call information
SUBSCRIBE	Subscribe to an event
NOTIFY	Notify subscribers about an event
REFER	Request the recipient to issue a SIP request

Table 2.1: SIP Messages [22, 25, 28, 29]

Table 2.1 summarises the SIP requests that may be sent. Responses are issued to these requests in a similar manner to that of an HTTP request using codes 1xx-6xx.

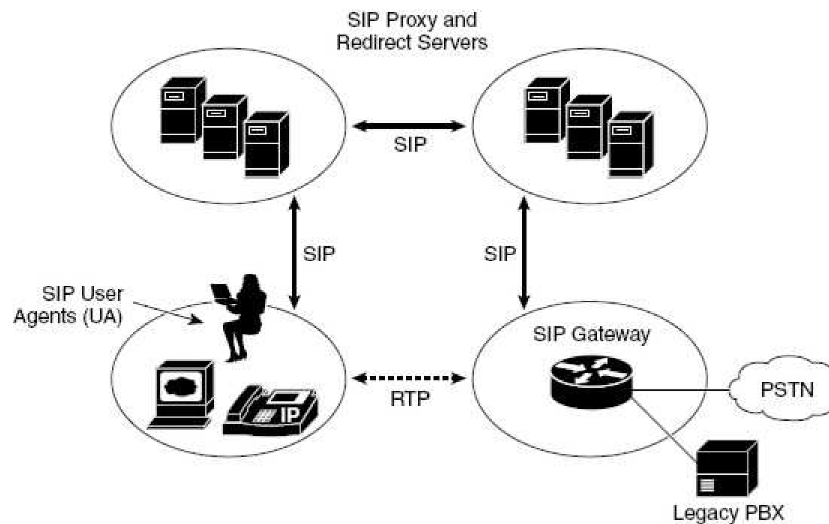


Figure 2.3: SIP Architecture [5]

Figure 2.3 shows a graphical representation of the SIP architecture, as shown in this figure, a SIP network contains two main architectural elements, the user agent (UA) (basically the phone you have on your desk or the soft-phone running on your PC) and the network server (examples would be Asterisk or SER). UA end stations may be further divided into two types, the User Agent Client (UAC) (client who is being sent the request) and the User Agent Server (UAS) (server who is sending the request). There are also three different types of network servers: redirect, proxy and registrar servers. It must be emphasised, however, that a basic call does not need servers, but more powerful features rely on them. We could just have two endpoints, a UAC and a UAS. [6]. Redirect servers process an INVITE message by sending back the SIP-URL where the callee is reachable. Proxy servers perform application layer routing of the SIP requests and responses. They can either be stateless (deals on a message to message basis forgetting about the call until another message arrives) or stateful (holds info about the call for entire duration), forking (ring several phones at once till someone takes the call) or non-forking. Registrar servers are used to record the SIP address (called a SIP URL) and the associated IP address. Note that a SIP network server implements a combination of different types of servers [6]. It must be noted that SIP requests can traverse many proxy servers each of which receives a request and forwards it towards a next hop server, which may be another proxy server or the final user agent server (which responds to the requests) [28]. Further expansion of these concepts and architecture is beyond the scope of this project. For more information refer to [7, 29, 28, 25, 6, 5].

The SIP address used to identify clients is an email like identifier of the form “user@domain” (or “phonenumber@gateway” for external phones) eg. `fred@sip.phone.ac.za`. This is great because we can put it on a web page and create a link `sip:fred@sip.phone.ac.za`, similar to the `mailto:` URL used today [3, 25].

Figure 2.4: Establishing a SIP Connection

So let us look at an example of setting up a session for voice or video over IP. Each of the end points have a SIP address. Figure 2.4 illustrates the process of setting up a session and the messages which are sent. The SIP addresses are translated from domain to an IP address using

DNS Service records, Canonical Name (CNAME) and finally address records. The media stream will most likely be transported by RTP point-to-point, as in this example, however any session may be set up using SIP.

SIP has attracted a lot of attention because of its simplicity and ability to support rapid introduction of new services. All this means that inexpensive terminals based on SIP protocol may be developed [12]. HTTP header fields similarity leads to easy integration with web services [28], and it also provides rich support for personal mobility services [28], making it applicable in this day and age. It must be emphasised that it is not just limited to internet telephony (though it is its main application) and can be used to initiate and manage any type of session including video, interactive games and text chat [34]. SIP is a very attractive support tool for IP telephony because it can operate as stateless or stateful. Once a call is in progress the servers do not have to maintain information about the call state [3].

It is important to understand the various types of SIP messages, their uses and in what situations they are applicable. It is also important to understand how sessions are established, and where the media information is stored. This information is important when analysing how the SIP channel operates within Asterisk. The architecture of a SIP network is also important to gain perspective on where, how and why the components of iLanga fit together to accommodate SIP users and provide them with services.

2.2.2 H.323

H.323 is the International Telecommunications Union (ITU) specified standard for real time sessions. H.323 is essentially an umbrella for a number of standards and protocols for setting up, controlling and performing realtime multimedia sessions. The remainder of this section will look at 6 aspects of the H.323 protocol: Its history, the architecture of a H.323 network, the protocols it specifies, the call process, the services provided and the applicable concepts to this project.

In 1996 the ITU decided on H.323 v1, referred to as a standard for real time videoconferencing over non-guaranteed Quality of Service (QoS) LAN. Interworking with the Public Switched Telephone Network (PSTN) was the focus from the very beginning. This H.323 standard assumed that a gateway handled signalling conversion, call control, and media transcoding in one box. This poses serious scalability problems [12]. It has since been developed and adapted and is currently sitting at version 4 [15]. It is a very complicated protocol, essentially an umbrella for

many sub protocols referring to real time communication over packet switched networks [30]. H.323 embraces the more traditional circuit switched approach to signalling. It started out as a protocol for multimedia communication on a LAN segment without QoS guarantees but has evolved to try and fit the more complex needs of Internet telephony.

The architecture of an H.323 consists of the following 3 possible components, with perhaps more than one present on a single node: The terminal end (TE), the gateway (GW) and the gatekeeper (GK). The TE is the endpoint from which you are placing or receiving the call, the GK provides address translation and controls access (admission control, bandwidth control, and zone management) to the network. It also provides other services (such as call control signalling, call authorization, bandwidth management and call management), but is an optional component. The Multipoint Control Unit (MCU) provides conferencing facilities and handles the mixing of audio or video for these conferences.

A typical H.323 network is composed of a number of zones connected by a Wide Area Network (WAN) each zone consists of a GK, a number of TEs, a number of GWs and a number of MCUs interconnected by a LAN. Each zone must contain exactly one GK which acts as an administrator of the zone [12], this may be seen in Figure 2.5. It must be emphasised that a

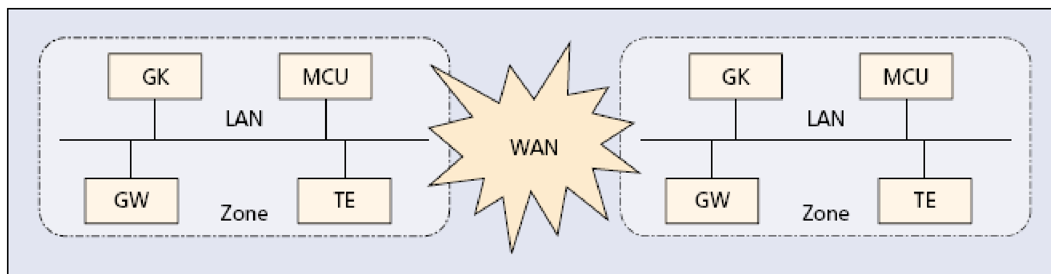


Figure 2.5: H.323 network [12]

gatekeeper is not explicitly necessary to make a call, a terminal end may set up a call directly, but a gatekeeper is necessary for added functionality and scalability [12].

H.323 is the umbrella for many protocols. The 4 major protocols we are concerned with are: RAS, Q.931 (both specified in H.225.0 [6]), H.245 and RTP. Let us take a look at what these four protocols are used for.

1. RAS (Registration Admission and Status) is a transaction orientated protocol between end-

points and the GK. It is used to discover, register and unregister with the GK. It can also be used for requesting call allocation, bandwidth allocation, and clearing a call. The GK uses it for inquiring the status of end points.

2. Q.931 is a signalling protocol for call setup and teardown between two H.323 TEs. It is a variant of one defined for the Public Services Telephone Network (PSTN). H.323 adopted it so interworking with PSTN/ISDN would be simplified.

3. H.245 is used for connection control, negotiating media processing capabilities such as audio or video codecs. It is also used to exchange terminal capabilities and opening and closing logical channels. RTP used as transport protocol [12].

4. RTP is used for the transport of media. This is detailed in section 2.3.1.

Figure 2.6 illustrates the protocol relationships in H.323.

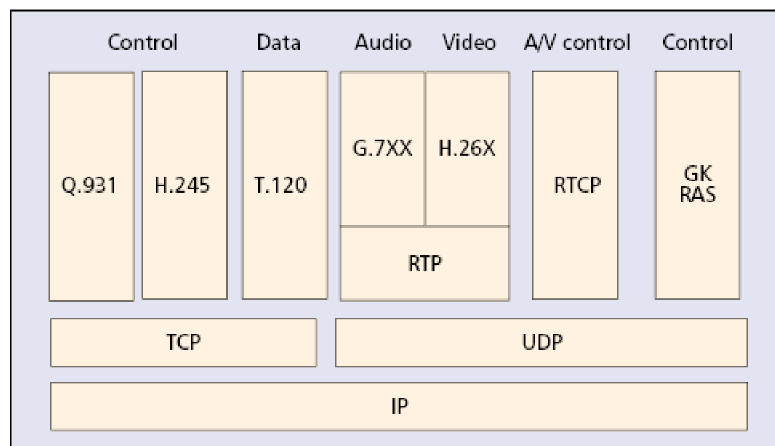


Figure 2.6: Protocol relationships in H.323 [12]

Let us take a look at the basic call process. Lui, et al. [12] illustrates the call well with this diagram seen below in Figure 2.7.

This is when we have a GK involved in the call, when this isn't the case phase 1 and 7 are omitted [12]. Between five and seven phases is quite a lot of phases for setting up and closing a session, especially considering that connections are mostly TCP based [30]. H.323 thus adapted, and fast connect was developed which reduces the phases by combining the Q.931 and H.245 phases [12]. In H.323 v3, TCP and UDP may be used [6] for establishing connections.

Phase		Protocol	Intended functions
1	Call admission	RAS	Request permission from GK to make/receive a call. At the end of this phase, the calling endpoint receives the Q.931 transport address of the called endpoint.
2	Call setup	Q.931	Set up a call between the two endpoints. At the end of this phase, the calling endpoint receives the H.245 transport address of the called endpoint.
3	Endpoint capability negotiation and logical channel setup	H.245	Negotiate capabilities between two endpoints. Determine master-slave relationship. Open logical channels between two endpoints. At the end of this phase, both endpoints know the RTP/RTCP addresses of each other.
4	Stable call	RTP	Two parties in conversation.
5	Channel closing	H.245	Close down the logical channels.
6	Call teardown	Q.931	Tear down the call.
7	Call disengage	RAS	Release the resources used for this call.

Figure 2.7: The phases of an H.323 call [12]

H.323 has adapted a lot as a protocol, which is part of the reason it is quite complex. It uses several protocol components, which have no clean separation. The major bonus is that full backwards compatibility has been maintained [12].

Services such as call transfer, call diversion, call forwarding, call hold, call park and pickup, message waiting indication and call waiting may use the H.450 protocol included under the umbrella of the H.323 specification. The H.450 protocol was implemented in version 2 to add services, and now (at version 4) it provides a lot more than the 3 services it did in version 2. Non H.450 based services are also a possibility. They are implemented in the GK, which makes the development of proprietary services a possibility. Most services require interactions between several of the protocols. For example, call forward requires components of H.450, H.225 and H.245 to be implemented [12]. We can use a gateway to provide the connection path between the packet switched network (Internet telephony) and the switched circuit network (the PSTN) [6]. All these services mean that H.323 is a viable option for setting up sessions for audio and video telephony.

It is important to understand what the functions of each of the protocols under the H.323 umbrella are and how they fit together to facilitate communication. It is also important to get an idea of what services are already available under the standard so they can be put to use. In our

analysis of the H.323 channels in Asterisk we need to have an idea of the standard so that we can take a critical look at what is being provided by each channel and determine which is the best channel and where it needs to be extended.

2.3 Media Protocols

This section describes the RTP protocol which is often used for the transport of real time multimedia in both the SIP and H.323 standards.

2.3.1 RTP

RTP is a protocol designed for the transport of Real Time Multimedia when there are tight constraints on the QoS [14]. This makes it ideal for the transport of video and voice over the internet, where QoS is often quite poor. Its default behaviour is to operate on port 5004, but it may use a port which has been registered for the particular application that is making use of the protocol. In the remainder of this section we will take a look at 3 aspects of RTP: the protocols components, multicast distribution and the applicable concepts to this project.

The RTP protocol essentially consists of two parts, RTP itself, which is a real-time end to end protocol, and RTCP which is a protocol used to monitor the QoS and convey information about the participants in an ongoing session (loose session control [28]). RTCP facilitates modifications which can be made according to the feedback provided, thus improving total performance [14]. This is particularly useful in the conference environment.

The services offered by RTP include payload type identification, sequence numbering, timestamping and delivery monitoring, which means it is useful for providing transport of data with an inherent notion of time. It will typically run on top of UDP. RTP does not grantee QoS, only improves the possibility of QoS. Reservation of resources is necessary for guaranteed QoS. QoS may be established through the use of another third party protocol such as RSVP [14] for reservation in tandem with RTP for transport and RTCP for monitoring.

RTP also supports data transfer to multiple destinations using multicast distribution, which also makes it a good choice for telephony, as conference facilities are undoubtedly an important service. RTP has been developed with flexibility and scalability in mind [14]. It facilitates QoS

by using timestamping. Packets received after the time required can thus be identified an discarded. The use of sequence numbers allow receiver to reconstruct the packet in the right order. RTP may also provide encryption, but keys need to be exchanged using some other protocol such as SDP. Other functionality include mixers (take media from several users and combine it into one media stream and send that out) and translators (take a single stream and send it out in a different format). [28].

When a host wishes to send a media packet, it takes the media, formats it for packetisation, adds any media specific headers, then prepends the RTP header and places it in a lower layer, such as UDP to be sent [28]. Below in Figure 2.8, is the format of the RTP packet. Table 2.2 provides a textual description of the fields represented in Figure 2.8.

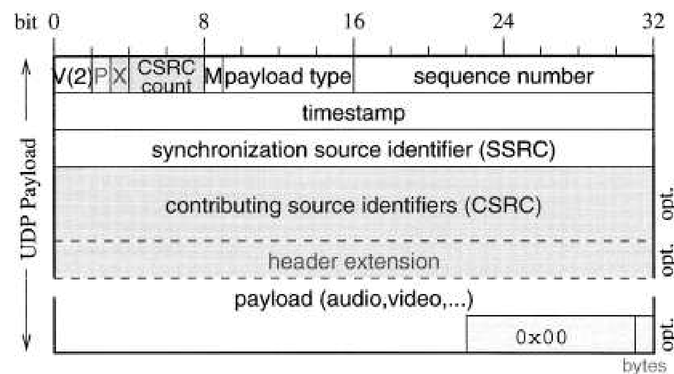


Figure 2.8: RTP packet [12]

Field	Description
V	Protocol Version
X	signals presence of a header extension
P	if set the payload is padded to ensure proper alignment for encryption
M	marker (specific to application typically set to denote boundaries in the data stream [3])

Table 2.2: Descriptions of fields in Figure 2.8

The SSRC identifier is randomly generated and uniquely identifies the source (hence the name synchronisation source identifier) within a multicast group. In the case that two have chosen the same SSRC, they both choose a new one. The CSRC lists all the contributing sources'

SSRCs, the number is indicated by the CSRC count in the header, so for an audio conference it will list all the speakers [28]. The payload type identifies the media encoding using an identifier from IANA [10]. The sequence numbers increment. For more information see RFC1889 [26] and RFC1890 [27].

Each RTCP packet contains a number of elements usually a sender report (SR), which describes the data sent so far, as well as correlating RTP timestamp, a receiver report (IR), which has one block per source, describing loss and jitter, and source destination (SDES), which provides a simple form of session control and can contain contact information, and allow other forms of communication[28].

The requirements for a protocol which transmits in real time (particularly one which is going to be used for internet telephony) are: sequencing, intra-media synchronisation, inter-media synchronisation, payload identification, and frame indication [28]. We can see that this is thoroughly satisfied by RTP. It must be emphasised that though it was primarily designed to satisfy the needs of multi participant multimedia conferences, it is not limited to this application. The storage of continuous data, interactive distributed simulation, and control and measurement applications may find RTP applicable. RTP works really well, and with premium service, it provides almost no delay and jitter for its packets, which makes it ideal for real-time voice and video [28]. Both SIP and H.323 make use of the RTP to exchange data [30].

RTP is used for media transmission for both the H.323 and SIP channels within iLanga. We need to know how the media is being packaged and transported in order to investigate how the channels are handling the media which is being passed through them.

2.4 Summary

This chapter has introduced session based and media based protocols used in real time multimedia. In particular it has described SIP and H.323, two session based protocols, and RTP, a media based protocol. This chapter has described each protocol, and shown their applicability to this project.

Chapter 3

Asterisk and iLanga

Asterisk and iLanga are the main topics of discussion within this project. This chapter expands on both of these frameworks giving a brief discussion on what they are and what they do, detailing their architectures and zooming in on specifics relevant to this project. We also explain how Asterisk fits into iLanga, and why channels are an important focus point of this research.

3.1 Asterisk

Asterisk is a powerful and adaptable suite of integrated telecommunications software which may be implemented to suit a variety of needs. The name comes from the Asterisk symbol *, which in many prominent operating systems, such as unix, dos, windows and linux, represents a wildcard. Wildcards are used to match any filename. This is a powerful image analogous to the design philosophy of Asterisk, “Asterisk is designed to interface any piece of telephony hardware or software with any telephony application seamlessly and consistently” [33], and its goal is in fact to support every possible telephony technology [20].

Asterisk is Open Source, which means that developers can adapt it to suit particular applications and add modules based on their needs. Current implementations run under the Linux operating system, however various windows versions are available. AsteriskWin32 is an example. It is the linux version compiled for windows using cygwin, with a graphical interface.

The remainder of this section details the architecture of Asterisk, explains its configuration, expands on the facilities provided and details a few ways it can be extended.

3.1.1 Architecture of Asterisk

Architecturally, Asterisk is fundamentally simple, but rather different from most telephony products [20]. It essentially acts as a middleware, connecting heterogeneous telephony technologies. It is designed in such a manner so that it is transparent; two phones using completely different voice codecs, and completely different protocols on different platforms speaking to each other seamlessly and accurately as if they were identical. Figure 3.1 illustrates the architecture of Asterisk.

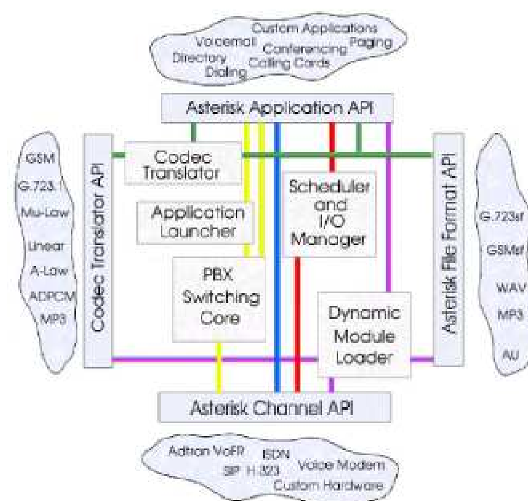


Figure 3.1: Asterisk Architecture [33]

When Asterisk is loaded, the Dynamic Module Loader loads each of the drivers which provide channel drivers file formats, codecs, applications, etc. and links them with the appropriate APIs. The Switching Core then accepts calls from the interfaces and handles them according to the dial plan which is located in a configuration file. The Application Launcher then provides options such as voicemail, dialing outbound trunks, etc. There is also a Scheduler and I/O Manager that may be used by the drivers and applications. The addition of codec translators means that channels with different codecs can speak to each other seamlessly [33]. We can see the interaction of these various different components provides lots of flexibility and makes it possible to implement any particular type of telephony, which is the aim of Asterisk.

Seamless connection is made possible by using four different APIs in which modules are created. They are:

1. Channel API for channels such as SIP and H.323 channels
2. Application API for applications such as the Meetme application (for conferencing), and the Dial application (for dialing other end points)
3. Asterisk File Format API for supporting different file formats such as MP3 and GSM for playback and recording.
4. Asterisk Codec Translator API for providing seamless connection between different codecs

This project is focused on the Channel API (which is expanded in Section 4), however the others are mentioned above for completeness and concept.

3.1.2 Configuring Asterisk

Configuration File	Role
sip.conf	SIP channel and end point configurations
extensions.conf	Configure the dial plan
iax.conf	IAX channel and end point configurations
h323.conf	H323 channel and end point configurations
oh323.conf	Open H323 channel and endpoint configurations
modules.conf	Asterisk modules configuration
manager.conf	Manager configuration
features.conf	Set up call parking
meetme.conf	Set up conference rooms for the MeetMe application

Table 3.1: Configuration files in Asterisk

Asterisk is rather easy to customise using the various configuration files located in `/etc/asterisk`. Table 3.1 shows a summary of some of the configuration files which have been relevant to this project. More specific details on configuring Asterisk may be found in [33, 19].

3.1.3 Facilities Provided

Asterisk has a great wealth of applications in industry as well as the home. Some of its uses include: VoIP gateway, PBX, custom interactive voice response (IVR) server, softswitch, conferencing server, number translator, calling card application, predictive dialer, call queuing system

with remote agents, and remote offices for an existing PBX [33]. It also supports features such as: voicemail, call forwarding, conferencing, call parking and provides a call detail records (CDR) database using MySQL as an add-on [20, 21, 33]. Asterisk includes H.323 gateway functionality (needs to be compiled in addition to other third party modules however) and may operate as a SIP proxy [33]. These facilities make it a multi-faceted PBX suited for small businesses and home users. The only problems Asterisk faces are scalability issues, the lack of an H.323 gatekeeper and the lack of IPv6 support. Because of the nature of translation and the size of the Asterisk channel structure, Asterisk will battle to handle more than around 250 concurrent calls, this raises scalability issues for larger businesses. This can be solved by using multiple servers and creating tunnels between them using the Inter-Asterisk eXchange protocol (IAX), and carefully configuring the dialplan. An H.323 gatekeeper can be provided by using another application, such as Open H.323 gatekeeper, in tandem with Asterisk. By adding other components, such as SIP Express Router (SER) and Open H.323 gatekeeper, in tandem with Asterisk we can also provide extra features, such as call forking, and provide further scalability. This is part of the motivation for the iLanga system's expansion of Asterisk. The lack of IPv6 support is still a problem, however IPv6 is not mainstream yet, and this problem should be rectified in future versions of Asterisk [34].

3.1.4 Extending Asterisk

Developers may extend Asterisk by adding channels, codecs, file formats and applications. This is done by working with the C API. AGI scripts, which are analogous to CGI scripts [31], may also be created to provide further facilities and services such as a weather reader or a cricket score reader. These AGI scripts are called from the dialplan.

The dial plan also provides many facilities for harnessing the power provided by Asterisk. It provides pattern matching algorithms for setting up the dialplan, as well as the use of variables, logical operations and arithmetic operations. It uses the concept of an extension, the number you dial in Asterisk, and it routes the channel that is initiated through a sequence of commands specified for that extension such as dialing another channel or playing back a file.

Asterisk uses a channel based infrastructure. A channel is basically a unit which brings in a call to the Asterisk PBX. Every call is placed or received on a distinct channel. Asterisk uses channel drivers to support each type of hardware [35]. H.323, SIP, IAX, MGCP and ISDN are all currently supported in Asterisk. In chapters 4 and 5, we will expand on channels, and illustrate

how we can create a channel using the channel API. We will also investigate their current support for video, and showing how it is achieved and the pitfalls for channels in which video is not supported.

3.2 iLanga

iLanga is a complete, cost-effective, computer based PBX that has been built at Rhodes University. It is based on three open source components: Asterisk, OpenH323 Gatekeeper (OpenGK) and SIP Express Router (SER). Asterisk as a stand alone component provides limited support for large scale VoIP networks [21]. SER and OpenGK (also known as GNUGK) have thus been added to complement Asterisk by providing further functionality essential for a high quality voice PBX system, extending the VoIP support provided. iLanga provides further abstraction to the concepts of a user and an end point. Different devices using separate channels for the same individual may be grouped under a single user account, which is used when dialing that individual. In this manner, the user will simply dial another individual and need not be exposed to the non-intuitive idea of a specific device as an end point. This may be taken further by providing priority to certain devices and calling different devices at different times of day.

In the remainder of this section we will take a look at the architecture of iLanga, expand on the user interface and conclude with linking Asterisk, iLanga and video expanding on their relationship and where the channel structure fits into the picture.

3.2.1 Architecture

Figure 3.2 illustrates how the components have been integrated to form the iLanga system.

Asterisk [2] is the core switching-software and handles transactions at a call signalling level. It provides support for the integration of multiple protocols by acting as a gateway for cross-communication between different end points. It also deals with the media layer, providing codec translation. This means that Asterisk is able to provide seamless communication between different endpoints. More information about the role of Asterisk in iLanga may be found in [21].

SER [11] is an open, high performance SIP proxy, location and redirect server. Asterisk has a built-in SIP proxy, however it is limited in functionality and only acts as a minimal SIP

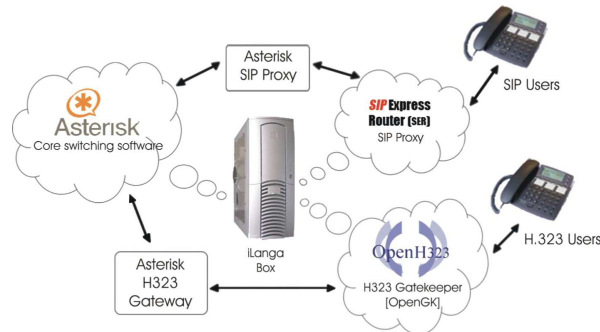


Figure 3.2: iLanga system architecture

location server. It was thus decided to incorporate a more advanced SIP proxy into the iLanga PBX. SER offers more secure registration than Asterisk. It provides secure digest authentication as opposed to the plain text authentication used in Asterisk. In iLanga SIP users register and authenticate with SER before being able to communicate with Asterisk via the Asterisk SIP proxy. Peer-to-peer communication between Asterisk and SER is ensured by configuring the Asterisk dial plan to forward all SIP requests to SER. This architecture means that SIP users can enjoy all the functional benefits of SER such as instant messaging, presence and forking (ringing on multiple active clients all using the same address) and still have access to the functionality and services provided by Asterisk such as conferencing, call forwarding and call parking. By using Asterisk as a gateway, seamless and transparent cross-protocol communication becomes viable. To the SIP user, iLanga appears as a single conglomerate. More information about SER and its integration into iLanga may be found in [21].

OpenGK [17] is an open source, full featured H.323 gatekeeper. Asterisk does not provide H.323 management or alias address functionality (these are functionalities provided by a gatekeeper), it just behaves as an H.323 terminal or gateway [33]. A gatekeeper is an important component of an H.323 network [12]. Some of its responsibilities include: management, authentication and alias address management. These are essential for H.323 users, and therefore OpenGK was added to iLanga. In iLanga, H.323 users are managed by OpenGK and communicate with Asterisk via the Asterisk H.323 gateway. Asterisk is configured to act as an H.323 gateway to achieve peer-to-peer communication between OpenGK and Asterisk. This architecture means that H.323 users can enjoy all the benefits of having a gatekeeper and still have access to the functionality and services provided by Asterisk. These include conferencing, call forwarding and call parking. Seamless commu-

nications with other protocols is also provided through the H.323 gateway. More information about OpenGK and its integration into iLanga may be found in [21].

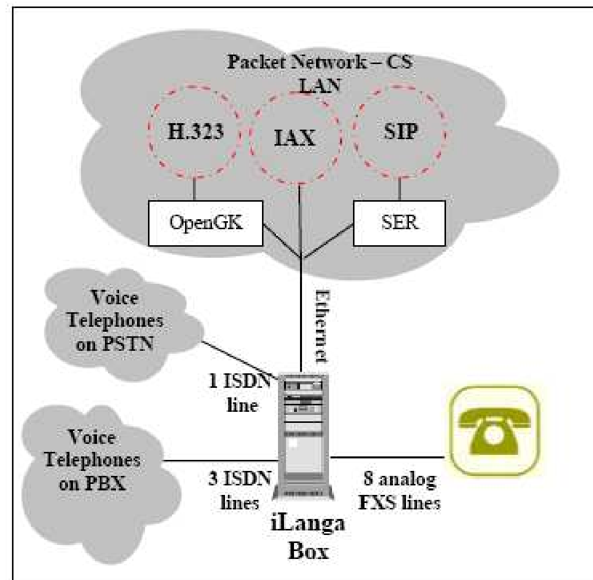


Figure 3.3: iLanga implementation [21]

The final component required by a complete voice PBX is a breakout into the PSTN to provide the ability to call outside of the network. This is provided by installing an Integrated services digital network (ISDN) interface using a Zaptel card. The final implementation of iLanga is illustrated in Figure 3.3. The combination of these three components, the ISDN interface and the features provided by Asterisk result in a complete voice telecommunications package, iLanga. iLanga provides call forking (from SER), voice mail, call forwarding, call transfer, call parking and an IVR system (for example a menu system: “Press 1 for...”). It is also not difficult to design custom services such as a weather reader or cricket score reader.

3.2.2 User interface

A Flash based, user-friendly interface has been developed for iLanga, as detailed in [8]. A snapshot of the user interface may be seen below in Figure 3.4.

Figure 3.4: iLanga graphical frontend [8]

Further extensions to this interface have been identified as being necessary. These are highlighted and their implementations explored in chapter 6.

3.2.3 iLanga, Asterisk and Video

iLanga is a powerful, cost effective voice PBX, however the current lack of support for video is a problem. The channel based architecture of Asterisk is extensible and provides support for video. This means that it is possible to bring video into iLanga using the channel structure of Asterisk. It is for this reason that the Asterisk channel structure is the particular focus of this investigation. The implementations and issues of channels and video are explored in the next chapter.

3.3 Summary

This chapter has described Asterisk and iLanga, which provide the framework that this project is built on. It has highlighted the architectures and facilities provided by each of these telephony products and demonstrated the extensibility inherent in their architectures. It has also established the relationship between Asterisk and iLanga, and explained why channels in Asterisk are the focus for video in iLanga. From this point onwards, this project will be focusing on Asterisk and iLanga. Firstly, in chapters 4 and 5, we will be discussing channels in Asterisk, their implemen-

tations and video support within these channels. We will then, in chapter 6, take a look at some extensions we have made in the iLanga user interface.

Chapter 4

Channels in Asterisk and their Implementation

This chapter delves into the heart of Asterisk, exposing channels in Asterisk, explaining how to implement them, and exploring the availability of video within the SIP channel and the lack of video within H.323 channels. It begins by introducing the channel concept, and then explores calls in Asterisk, and takes a look at the dial application. It then moves on, elaborating on the physical channel structures within the source code, and detailing the functions available in the channel API. It then takes a look at how frames operate, and concludes by discussing how to design a channel module, looking at the example channel we have created. This chapter reveals the documentation created by the author from investigations into the source code and conceptual experiments performed during this project. It reveals the author's understanding of channels gained from the source code.

4.1 Channel Concept

The channel based architecture is largely where the power of Asterisk lies. Each end of the call is abstracted into a channel and they are bridged by Asterisk to provide communication between them. For each of the channels, the relevant channel drivers handle the appropriate signalling, and pass codec translations to the Asterisk Core before transferring the media between the endpoints. This results in seamless conversation between the two ends. This means that the handling of the media largely depends on the implementation of the codecs within the Asterisk core.

Figure 4.1: Use of channels between disparate protocols

Figure 4.1 illustrates the channel concept in action. In this example we have two endpoints using different voice codecs and different signalling protocols. Person A is utilising a SIP phone with the audio encoded using the GSM codec, while person B is using an H.323 phone and the G.711u codec for audio encoding. The channel based architecture of Asterisk creates the possibility of seamless communication between these endpoints. Person A will be using the SIP channel driver, while person B will be using the H.323 channel driver. These channel drivers are created as modules and compiled as system object files. The channel drivers are registered when Asterisk starts up, and used by Asterisk in the call.

4.2 A call in Asterisk

When the call is initialised, Asterisk creates two channel structures using the appropriate channel drivers. It also determines which codecs are going to be used and finds the least common denominator between the two, using a translator function in Asterisk. Both of these channels are created using an instance of a universal structure called `ast_channel`. These structures are bridged for signalling and media using a function `ast_bridge_call`. This allows communication to then take place between these two endpoints. A call is initialised using the dial application.

4.3 The dial application

This section takes a look at the Dial application in Asterisk, highlighting its usage for initiating calls, and explaining how it operates in view of the importance of understanding how the channels are utilised and used by this application.

4.3.1 What is the dial application

The code for the dial application is found in `app_dial.c`. This code is compiled as a module, and registers an application called Dial in its `load_module` function. The dial application is called

<table>
<tr><th>Argument</th><th>Description</th></tr>
<tr><td>SIP/2000</td><td>Dial a single SIP client registered as 2000</td></tr>
<tr><td>SIP/2000&IAX2/3000</td><td>Dial two clients, one registered as 2000 with the SIP channel and the other registered as 3000 with the IAX2 channel</td></tr>
<tr><td>... | 6000</td><td>Specify a timeout for the dialing of a call</td></tr>
<tr><td>... | ... | tT</td><td>Set options to allow caller and callee to transfer the current call</td></tr>
</table>

Table 4.1: Example arguments for the dial application

by Asterisk to make a call with another channel, as mentioned earlier.

It requests one or more channels and places specified outgoing calls on them. As soon as a channel answers, the Dial application will answer the originating channel (if it needs to be answered) and will bridge a call with the channel which is answered first. All other calls placed by the Dial application will be hung up. If a timeout is not specified, the Dial application will wait indefinitely until either one of the called channels answers, the user hangs up, all channels return busy or an error occurs.

4.3.2 Calling the dial application

The dial application is called by executing the application Dial within Asterisk. The dial application requires an argument to be passed to it with information about the channels that are going to be dialed, how long the time out should be (if there is one), and other options such as enabling call transfer during a call. The general format of the argument that may be passed is:

```
technology/number & technology2/number2 ... | timeout | options | URL
```

Table 4.1 shows some example arguments which may be passed to the dial application. Within the table, the ellipsis means that zero or more characters may be filled into that field. The only compulsory section of the dial argument is the first `technology/number` field shown in the first row of Table 4.1. There is no limit to the number of `technology/number` fields specified in the argument. All of these will be dialed and the bridge will be established with the first channel to answer. More information about the options and arguments that may be passed to the dial application may be found by running `show application dial` in the Asterisk CLI.

4.3.3 Extensions and the dial application

Extensions are one of the core concepts in Asterisk. The extensions, defined in `extensions.conf`, are the numbers that Asterisk recognises as valid. When an extension is dialed, the extension in the context which the user has been placed during their registration is executed in order of priority. The following is a possible extract from `extensions.conf`

```
[default]
exten => 4000,1,Answer
exten => 4000,2,Playback(file)
exten => 4000,3,Hangup
```

This extract is a subset of the default context. If we are a user placed in the context default, and dial 4000, the extract above is parsed in order of priority, so the call is answered (1), a file is played back to the user (2), and then the call is hungup (3). Playback is an application registered with Asterisk in the same manner as the Dial application. Below is another possible extract from `extensions.conf`.

```
exten => _XXXX,1,Dial(SIP/${EXTEN})
```

This will execute the application Dial which is registered by the module `app_dial.so` which is a compiled version of `app_dial.c`. The dial application sets up a call using arguments defined earlier. This example uses the pattern matching facilities available in Asterisk to match any four digit number. The Dial application is called, if any four digit number is dialed, and the argument SIP/{four digit number} is passed to it.

4.3.4 How the dial application operates

Figure 4.2 and 4.3 are flow charts illustrating the operation of the dial application. The numbers within the circles may be used to track the operation of the parts of the dial application, illustrated in Figure 4.2, and used to simplify the reading of the flowchart. The basic process consists of four steps.

1. Parsing the dial arguments, setting the options and creating a list of possible channels
2. Waiting for an answer from one of the channels
3. Bridging the channels
4. Closing off the application, updating variables and killing channels

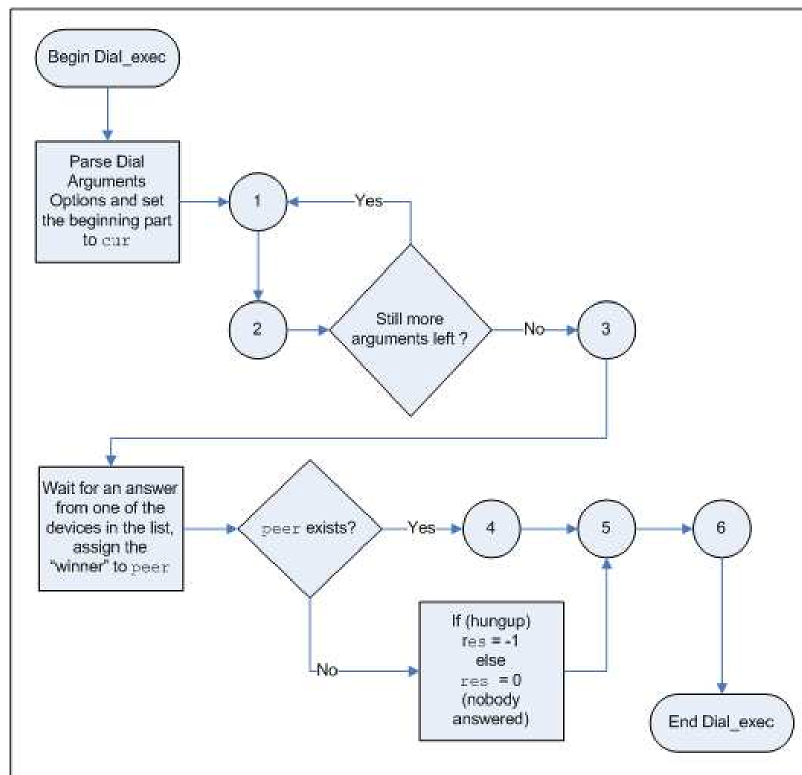


Figure 4.2: Dial Application Flow Chart

Figure 4.3 (a) shows the setting up of the channel list. We have previously discussed the arguments that may be passed to the dial application, this process takes all the devices listed in the dial argument, and puts them in the list. An interesting method is used to check whether it is available. For each device, the device gets called to check whether everything is alright (ie. it is available), and only if this is the case then it is added to the list. Once the list is established, a function, `wait_for_answer` is executed. It uses the channel API function `ast_waitfor_n`, and returns the channel which answers the call, so that it may be bridged with the channel making the call. Figure 4.3 (b) illustrates the bridging of the calls. `ast_bridge_call` calls `ast_channel_bridge` which bridges signalling and media between the channels. This is detailed in section 4.5, which deals with the channel API. Figure 4.3 (c) illustrates the ending of the dial application.

4.3.5 Importance of understanding the dial application

It is important to understand how channels are managed in Asterisk so that when we create or alter a channel it may still be used by Asterisk as intended. The dial application is important as

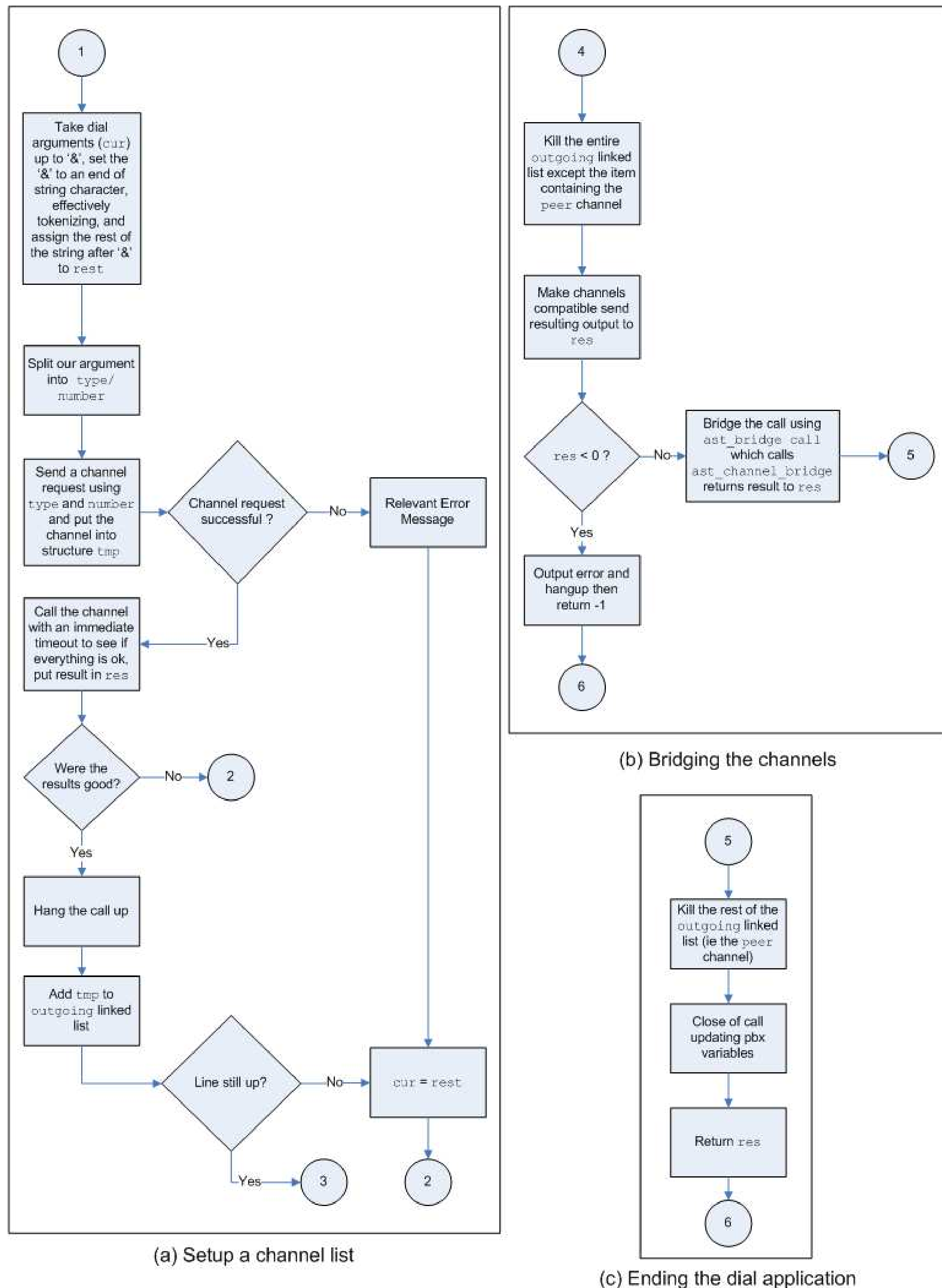


Figure 4.3: Parts of the dial application

it is the usual method of establishing a call between two endpoints in Asterisk.

4.4 The channel structure

The channel structure of Asterisk is the structure in which the mechanisms for a particular channel type are established. The channel structures are setup by an `ast_request` which calls the function specified as the request function during the registration of the channel. The channel structure is essentially composed of two components, the `ast_channel` struct and the `ast_channel_pvt` struct.

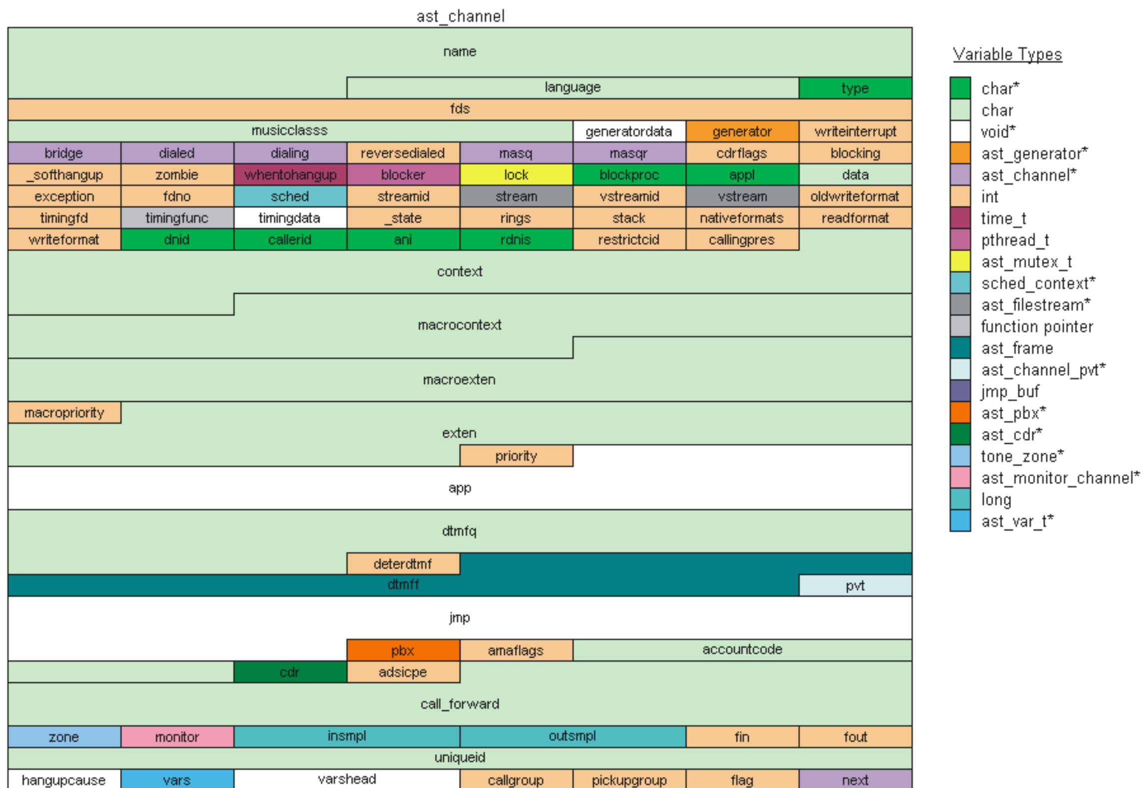


Figure 4.4: ast_channel structure

The `ast_channel` struct contains many variables, locks and descriptors used for monitoring the channel's operation and storing information about the channel. It is found in the file `channel.h`. Figure 4.4 shows the `ast_channel` structure. Within this structure, there is a variable `pvt`, which is a pointer to a variable of type `ast_channel_pvt`.

ast_channel_pvt

void* pvt	ast_frame* readq	int [2] alertpipe		ast_trans_pvt* writetrans
ast_trans_pvt* readtrans	int rawwriteformat	int rawreadformat	function pointer send_digit	function pointer call
function pointer hangup	function pointer answer	function pointer read	function pointer write	function pointer send_text
function pointer send_image	function pointer send_html	function pointer exception	function pointer bridge	function pointer indicate
function pointer fixup	function pointer setoption	function pointer queryoption	function pointer transfer	function pointer write_video

Figure 4.5: ast_channel_pvt structure

Figure 4.5 illustrates the private structure of the channel. It contains information about the media and translators being used and the functions which are called by the channel API for signalling, and the reading and writing of media frames and dial tones. These functions include the functions for calling, answering and hanging up a call made using the channel. `ast_channel_pvt` also contains a pointer to a structure called `pvt`. This is the specific channel's private structure, i.e in the case of the SIP channel, the SIP private structure. This structure usually contains locks, sockets and media information particular to an instance of a particular channel type. These structures are called and altered by the channel API using functions specified in `ast_channel_pvt`.

4.5 The channel API

The channel API consists of a number of functions designed to facilitate communication between the channel drivers and the Asterisk core in aid of seamless connection between protocols, discussed previously. The channel API consists essentially of a number of functions, and two particular structures. These structures are `ast_channel` and `ast_channel_pvt`. Files of interest include `channel.c`, `channel.h`, `channel_pvt.h`, and `res_features.c`. The table that follows contains a few of the functions from the channel API which are used when dealing with channels and discussed in this text. Appendix A contains more functions from the channel API which are useful. These tables have been produced during the investigation, and expand on the function's purpose, how they are used within Asterisk and what they return. Functions such as `ast_call`, `ast_read`, `ast_write` use functions which are defined within the relevant channel driver as they are specific to their channel type. `ast_channel_register` and `ast_channel_unregister` are used in the respective functions `load_module` and `unload_module` contained in the channel modules. They are used to register a channel driver module (also referred to as a channel driver and a channel module) to support a protocol of type of hardware.

`ast_call_bridge` is called to bridge a call. It operates as an "infinite loop", only breaking out of the loop once the bridge is broken or the channels are hangup. Within this "infinite loop", there are calls to `ast_channel bridge` which performs the transfer of media between the channels using `ast_read` and `ast_write` statements which operate according to the settings made for media compatibility.

Media compatibility is established using the `ast_channel_make_compatible` function. This function makes calls to `ast_set_read_format` and `ast_set_write_format` to configure the channels with appropriate formats and set variables in the channel structure and the private channel structure.

Function	**Description**	
ast_request	Description:	Requests a channel of a given type, running the code from the channel driver of that type
	Example call:	ast_request(type, format, number) where type is a character array containing the type of channel (eg. SIP), format is an integer referring to the format of the data and number is of any type and contains the number being called.
	Returns:	ast_channel* (Pointer to an asterisk channel), null if unsuccessful
ast_channel_register_ex	Description:	Registers a particular channel with a function to poll devicestate
	Example call:	ast_channel_register_ex(type, description, capabilities, requester, devicestate) where type a character array containing the type of channel, analogous with type specified in other functions, description is a character array containing the description of channel being registered, capabilities in an integer representing the capabilities of the channel, requester is a pointer to a function returning an ast_channel struct in the channel driver which is run when an ast_request is passed and devicestate which is a pointer to a function returning an integer which returns the device state. This function has a parameter data for the number of the device whose state the driver is interested. This is optional for a channel driver.
	Returns:	int, -1 if there is an error, 0 if successful

Features	Description	
ast_channel_register	Description:	Registers a particular channel, used in the load module to register channel with Asterisk, uses ast_chan_register_ex. but with devicestate function set to null
	Example call:	ast_channel_register(type, description, capabilities, requester) where type a character array containing the type of channel, analogous with type specified in other functions, description is a character array containing the description of channel being registered, capabilities in an integer representing the capabilities of the channel, and requester is a pointer to a function returning an ast_channel struct in the channel driver which is run when an ast_request is passed.
	Returns:	int, -1 if there is an error, 0 if successful
ast_channel_unregister	Description:	Unregisters a channel with the Asterisk system
	Example call:	ast_channel_unregister(type) where type is a character array containing the type of channel to be unregistered, eg. SIP
	Returns:	void
ast_hangup	Description:	Calls a hard hangup of the channel, stopping streams, and destroying the channel, uses the hangup function in the channel driver
	Example call:	ast_hangup(chan), where chan is of type pointer to ast_channel struct and is the structure of the channel we wish to hangup
	Returns:	int, 0 if successful otherwise the value returned by the hangup function in the channel driver
ast_answer	Description:	Answers the call, calls a function in the channel driver
	Example call:	ast_answer(chan) where chan is the pointer to the ast_channel struct of the channel of interest
	Returns:	int, -1 if it is being hung up, the result returned from answer in the channel driver if the state of the channel is ringing, otherwise 0

Features	Description
ast_call	Description: Makes a call to a channel, using functions in the channel driver Example call: ast_call(chan, addr, timeout) where chan is a pointer to the ast_channel struct of the channel we wish to call, addr is a character array for the destination of the call and timeout is an integer for the time waited for a connection Returns: int, -1 if a call function doesn't exist or a hangup is scheduled for the function, otherwise the results returned by the function in the channel driver
ast_indicate	Description: Indicates a condition on a channel such as busy, ringing or congestion Example call: ast_indicate(chan, condition) where chan is a pointer to the ast_channel struct of the channel we wish to send an indication to, and condition is an integer representing the condition which we wish to indicate Returns: int, -1 if hangup is scheduled of invalid condition, 0 if channel doesn't support it, or the result returned from the channel driver
ast_waitfor	Description: Wait for input on a channel Example call: ast_waitfor(chan, ms) where chan is a pointer to the ast_channel struct of the channel we are waiting for and ms an integer for the length of time we can wait for Returns: int, -1 if there is an error, 0 if nothing ever arrived, the number of milliseconds remaining otherwise
ast_waitfor_n	Description: Waits for input from a group of channels Example call: ast_waitfor_n(chanlist, number, ms) where chanlist is an array of pointers to the ast_channel struct of the channel we are waiting for, number is an integer containing the number of channels in the list and ms an integer for the length of time we can wait for Returns: ast_channel*, the channel with activity, otherwise null
ast_read	Description: reads a frame from a channel using a function in the channel driver Example call: ast_read(chan) where chan is a pointer to the ast_channel struct of the channel we are reading from Returns: ast_frame*, a frame, null on error

Features	Description	
ast_write	Description:	write a frame to a channel using function in the channel driver
	Example call:	ast_write(chan, frame) where chan is a pointer to the ast_channel struct of the channel we are writing to and frame is a pointer to ast_frame struct which is the frame we are writing to the channel
	Returns:	int, -1 on error, 0 if the functions don't exist, and the result from the function in the channel driver otherwise
ast_write_video	Description:	write a video frame to a channel using a function in the channel driver
	Example call:	ast_write_video(chan, frame) where chan is a pointer to the ast_channel struct of the channel we are writing to and frame is a pointer to ast_frame struct which is the frame we are writing to the channel
	Returns:	int, -1 on error, 0 if the functions don't exist, and the result from the function in the channel driver otherwise
ast_set_read_format	Description:	sets the format to be read by a channel
	Example call:	ast_set_read_format(chan, format) where chan is a pointer to the ast_channel struct of the channel for which we are setting the format, and format is an integer representing the format being read by the channel
	Returns:	int, -1 on error, 0 otherwise
ast_set_write_format	Description:	sets the format to be written by a channel
	Example call:	ast_set_write_format(chan, format) where chan is a pointer to the ast_channel struct of the channel for which we are setting the format, and format is an integer representing the format being written by the channel
	Returns:	int, -1 on error, 0 otherwise
ast_channel_make_compatible	Description:	Attempt to make two channels compatible with each other
	Example call:	ast_channel_make_compatible(chan, peer) where chan and peer are pointers to the ast_channel structs of the channels we are making compatible in terms of codecs
	Returns:	int, -1 if there is no path to translate, or an error in setting the 2 channel's read or write formats, 0 on success

Features	**Description**	
ast_channel_bridge	Description:	Create a bridge between two channels in terms of media
	Example call:	ast_channel_bridge(chan1, chan2, config, destframe, destchan) where chan1 and chan2 are pointers to the ast_channel structs which we are bridging, config is a pointer to the ast_bridge_config struct containing the configuration of the bridge, destframe and destchan are double pointers to ast_frame and ast_channel structs respectively. They are used to pass the destination channel and frames so that functions such as ast_call_bridge may use this data.
	Returns:	int, -1 on an error to do with channels waiting to hang up or existing bridges, 0 on success
ast_transfer	Description:	transfer a channel if it is a supported function within that channel driver
	Example call:	ast_transfer(chan, dest) where chan is a pointer to the ast_channel struct of the channel we are transferring and dest is a character array representing the destination of the transfer
	Returns:	int, -1 if channel is being hung up or a transfer function doesn't exist otherwise return the result from the function in the channel driver
ast_channel_alloc	Description:	Create a channel structure
	Example call:	ast_channel_alloc(needalertpipe) where needalertpipe is an integer representing a boolean value determining whether we set up this part of the structure in out allocation
	Returns:	ast_channel*, null if shutting down or error otherwise return a new allocated channel
ast_channel_free	Description:	Destroy a channel structure
	Example call:	ast_channel_free(chan) where chan is a pointer to the ast_channel struct of the channel we are destroying
	Returns:	void

Features	Description	
ast_bridge_call	Description:	Bridge a call allowing for parking and transfer using asterisk core. This function calls ast_channel_bridge
	Example call:	ast_bridge_call(chan, peer, config) where chan and peer are pointers to the ast_channel structs which we are bridging and config is a pointer to the ast_bridge_config struct containing the configuration of the bridge
	Returns:	int, -1 on error, 0 on success.

It is necessary for each channel driver to establish their own private structure. The usage of the channel API will be expanded in the sections on the example channel and SIP channel. We will look at this after a brief look at how frames work within Asterisk between channel drivers.

4.6 Frames in Asterisk

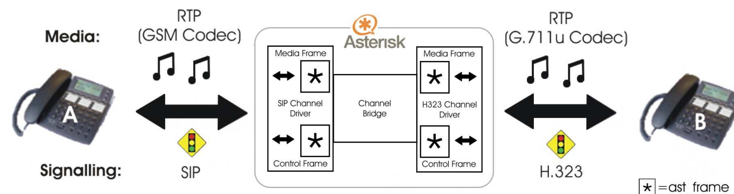


Figure 4.6: Asterisk frames

In Asterisk everything is packaged as frames. These frames are passed between the channels via the Asterisk core. Figure 4.6 illustrates how the media and signalling are processed using frames by the channel drivers. `ast_read`, `ast_write`, `ast_write_video`, `ast_senddigit` and `ast_indicate` all send or receive frames within the Asterisk channel driver. These are defined in the private structure of the channel driver.

ast_frame

int frametype	int subclass	int datalen	int samples
int malloccd	int offset	char* src	void* data
timeval delivery	ast_frame* prev	ast_frame* next	

Figure 4.7: ast_frame structure

These frames are packaged in a structure `ast_frame`. This is illustrated in Figure 4.7. It has a double linked list based structure with pointers to the next frame and the previous frame. `mallocd` illustrates whether the data has been allocated using malloc so that it can be freed when the frame is destroyed. `src` is a pointer to a character array, containing a description of the source, and is included in the structure for debugging purposes. The data length is contained in `datalen`, and the data is stored in the location pointed to by the `data` pointer.

Frame Type	Subclass	Content Type	Integer Value
AST_FRAME_DTMF	A digit	DTMF Digits	1
AST_FRAME_VOICE	AST_FORMAT	Voice data	2
AST_FRAME_VIDEO	AST_FORMAT	Video data	3
AST_FRAME_CONTROL	AST_CONTROL	Control frame	4
AST_FRAME_NULL	-	Empty frame	5
AST_FRAME_IAX	-	IAX Private frame	6
AST_FRAME_TEXT	-	Text messages	7
AST_FRAME_IMAGE	-	Images	8
AST_FRAME_HTML	AST_HTML	HTML	9
AST_FRAME_CNG	level of CNG in -dBov	Comfort noise	10

Table 4.3: Frame Types in Asterisk

Table 4.3 contains a list of possible values for the `frametype` and the possible subclasses. Video frames may have H.261 or H.263 as subclasses, audio frames may have GSM and G.711u as subclasses, and control frames may have congestion, ringing or busy as possible subclasses for the `frametype`. An expanded list of possible subclasses and the values for the constants may be found in the source file `frame.h`. We will now take a look at how to design a channel in Asterisk.

4.7 How to design a channel in Asterisk

In order to design a channel in Asterisk, we need to use the channel API and create a module which registers the channel and provides the functions necessary for signalling and media transfer. The main functions of these modules are to interpret various signalling, passing the appropriate calls to the Asterisk core as control messages such as busy, ringing, hangup, and

congestion, and passing the media as frames to Asterisk by creating read and write functions.

The remainder of this section describes three aspects of channel creation: modules, commands in the command line interface (CLI) and outputting to logs, and concludes with a look at the example channel which we have created for proof-of-concept.

4.7.1 Modules in Asterisk

A module in Asterisk consists of a few essential components: the `load_module` function, the `unload_module` function, the `usecount` function, the `key` function and the `description` function. The `load_module` and `unload_module` functions are executed when Asterisk starts and loads modules, and when Asterisk shutsdown and kills the modules respectively, and are also used for registering the module with Asterisk and setting up CLI commands. The `description` function returns a character array containing the description of the channel, in this case "Example Channel (EG)". The `key` function returns the ASTERISK_GPL_KEY character array, while the `usecount` function returns the amount of times the module has been used. This function uses a lock. In a channel driver, we also need to be able to request information about channel. In Asterisk we may run commands in the CLI for this purpose.

4.7.2 Adding commands to the CLI

The Asterisk CLI is a powerful management console provided with Asterisk. It is used to get information about the status of the PBX and its modules. It may also be used to find out information about the modules registered and get help on how to use them. In order to add commands to the CLI, we need to run the function `ast_cli_register`. This is done in the following manner:

```
ast_cli_register(&cli_eg_info);
```

This is done in the `load_module` function. We perform an `ast_cli_unregister` in the exact same manner in the `unload_module` to unregister the CLI command. `cli_eg_info` is defined in the following manner:

```
static struct ast_cli_entry cli_eg_info =
        { { "eg", "info", NULL }, eg_info, "Example Channel Information",
          eg_info_usage };
```

When we run the command `eg info` in the CLI, the function `eg_info` is executed. If we wanted to create a command such as `this is a test` then we would define the first part of the struct

above as `{ "this", "is", "a", "test", NULL }`. If we seek help by running the help command the string "Example Channel Information" is displayed next to "eg info". If we run `help eg info`, the character array, `eg_info_usage` gets displayed. It is defined as follows:

```
static char eg_info_usage[] =
"Usage: eg info\n"
"       Displays Information about Example Channel\n";
```

Functions such as `eg_info` which we have linked to the command `eg info` need to be able to output information to the CLI.

4.7.3 Outputting to the logs

The ability to output notices and other information to the CLI is also important in a channel driver. This information can be both informative and used for debugging when problems arise. Thus it is crucial for both the developers and the users of the PBX that notices and other messages are included. In order to have this functionality we need to include the header file `logger.h`. Below are two examples of sending messages to the CLI.

```
ast_log(LOG_NOTICE, "This is the example channel in Asterisk\n");
ast_verbose("== Initiating a new call to the example channel ==\n");
```

The first displays a notice in the CLI. This notice includes date, time, source file and line number in the source. This is useful for debugging, however LOG_NOTICE may be replaced by LOG_DEBUG, which only display when in debugging mode. Asterisk can be run in verbose mode. This may be set by running Asterisk with a -vvv option on startup (for a verbosity of 3) or by running `set verbose 3` command in the CLI. Messages such as the second example using `ast_verbose` will only be displayed when the verbosity is greater than 0.

4.7.4 The example channel

In order to verify an understanding of the essentials for a channel, we decided to create an example channel which just outputs to the CLI when it is requested and called. No media facilities were implemented in this channel, however, the evidence below shows the example channel to be operational.

```
Asterisk 1.0.9, Copyright (C) 1999-2004 Digium.
Written by Mark Spencer <markster@digium.com>
=========================================================================
Connected to Asterisk 1.0.9 currently running on g02z0525-1 (pid = 30823)
== Creating a new Example Channel :) ==
== Initiating a new call to the example channel ==

...
g02z0525-1*CLI> show channels
        Channel  (Context    Extension    Pri )   State Appl.         Data
   EG/5555-ec4c  (default    s            1   )      Up AppDial       (Outgoing Line)
   SIP/600-ca00  (local-from-sip 5555         1   )    Ring Dial          EG/5555
2 active channel(s)
```

This channel was built after evaluation of the SIP channel and the channel API. Within the `load_module` function, we register the channel using the type "EG" and the description "Example Channel (EG)" which is put into a character array `tdesc`. We set the request function to `eg_request`. This function calls another function `eg_new` which returns the channel type. It begins by using `ast_channel_alloc` to allocate a channel structure. In this example channel, we just define functions for call and hangup in the channel private structure. Since it is necessary to have a private structure for the instance of the channel (ie. an eg private structure), we take a character array containing the text "pvt" and set it to the structure so that channel will be operational. When the function `eg_request` is called, it outputs `"== Creating a new Example Channel :) =="` to the CLI if Asterisk is running with a verbosity greater than zero. The call and hangup function pointers are linked to `eg_call` and `eg_hangup` respectively. This is a rather basic channel. It is necessary to take a look at a fully functioned, operational channel within Asterisk in order to grasp the deeper concept of the channel API. We have chosen the SIP channel as our test case.

4.8 Summary

This chapter has described channels, and the channel API. It has shown how channels are initialised, how calls are made, and how the modules are structured. It has also demonstrated an example channel created for proof-of-concept.

Chapter 5

The SIP and H.323 Channels

In the previous chapter, we have exposed the channel-structure of Asterisk. This chapter moves on from these foundations and explores the SIP and H.323 channel implementations, discussing their availability of video. It concludes with a discussion on the inheritance of features within Asterisk. This chapter presents the authors work, investigations and experiments with the SIP and H.323 channels.

5.1 The SIP Channel

Asterisk provides a SIP channel driver to support SIP end points. This channel includes a SIP registrar proxy, however lacks the ability of a forking proxy. Endpoints may be registered with Asterisk if they have a user account set up in the configuration file `sip.conf`. More details about this configuration may be found in [33].

The remainder of this section will look at 6 aspects of the SIP channel: The channel driver, channel registration, the `devicestate` function, the private structure (`sip_pvt`), media handling and signalling.

5.1.1 The SIP channel driver

The SIP channel driver is written as a module for Asterisk in C, and may be found as the file `chan_sip.c`. This is compiled as a system object, `chan_sip.so`, and loaded by Asterisk on startup to provide support for SIP end points. Previously we have taken a look at the channel structure, the channel API, and an example channel. This reveals a lot about how channels

are constructed and how they operate within the Asterisk core, however analysing an existing channel driver is necessary to make the concepts more clear.

5.1.2 Channel registration

When the module is loaded, the `load_module` function is called. This function initiates a call to register the channel with Asterisk. As we have seen in the channel API, there are two functions which may be used to register a channel with Asterisk, `ast_register_channel` and `ast_register_channel_ex`. The SIP channel calls `ast_register_channel_ex`, setting the request function to `sip_request` and the `devicestate` function to `sip_device_state`. Within the `load_module` function there are also a number of calls to a function `ast_cli_register`. This function registers commands with the CLI so that information about the channel's status may be obtained when we are running the CLI. This makes commands such as `sip show users` a possibility. We also notice the use of `ast_mutex_lock` and `ast_mutex_unlock`. This is used so that threads may use data concurrently avoiding reader-writer problems.

5.1.3 Devicestate

`sip_devicestate` is the function passed as the function to determine the device state during the registration of the channel. It returns an integer indicating the state of the device registered with the channel driver. The specific device is indicated in the parameter data. Values returned are either AST_DEVICE_INVALID (4), AST_DEVICE_UNAVAILABLE (5), or AST_DEVICE_UNKNOWN (0). These constants may be found in `channel.h`.

5.1.4 The SIP Private Structure

Within the private channel structure, there is a pointer to another private structure for the individual channel. This is an important part of a channel structure. If this is not declared, calls will not be possible using the channel module that has been created. Many functions within the channel API check to see whether this structure exists. Within the SIP channel there is such a structure, `sip_pvt`. This structure contains variables for sockets, configuration options, locks and media data. Two important variables are `rtp` and `vrtp`. These are pointers to an `ast_rtp` type which is used for media transmission by this channel.

When `ast_request` is called in the channel API, the function `sip_request` is called. This returns an `ast_channel` type. It is within this function that we setup the channel for the call.

`sip_alloc` is called to setup the private structure and `sip_new` is called to set up the channel that is returned. The function `sip_new` gets passed parameters for the private structure, the state which indicates the device state and title which indicates which registered device we are dealing with. `sip_new` creates and returns a channel structure. It calls `ast_channel_alloc` to create the channel structure, and then assigns the values for the channel's private structure `pvt`. These include setting up the functions that get called for reading and writing media, as well as ones for initiating a call.

```
tmp->pvt->pvt = i;
tmp->pvt->send_text = sip_sendtext;
tmp->pvt->call = sip_call;
tmp->pvt->hangup = sip_hangup;
tmp->pvt->answer = sip_answer;
tmp->pvt->read = sip_read;
tmp->pvt->write = sip_write;
tmp->pvt->write_video = sip_write;
tmp->pvt->indicate = sip_indicate;
tmp->pvt->transfer = sip_transfer;
tmp->pvt->fixup = sip_fixup;
tmp->pvt->send_digit = sip_senddigit;
```

The above extract of code is a rather important part of setting up the channel. The first line is where the SIP private structure gets assigned and the rest is where the function pointers get assigned which are use by methods such as `ast_call`, `ast_read`, and `ast_write` for their purposes within the channel. `ast_read` and `ast_write` are used for media transfer, while `ast_call` is used to initiate a call with the channel.

5.1.5 Media Handling

```
tmp->fds[0] = ast_rtp_fd(i->rtp);
tmp->fds[1] = ast_rtcp_fd(i->rtp);
if (i->vrtp) {
        tmp->fds[2] = ast_rtp_fd(i->vrtp);
        tmp->fds[3] = ast_rtcp_fd(i->vrtp);
```

The above segment of code is used to set up the media frame descriptors within the channel structure. The SIP channel needs to use RTP for media transfer. Asterisk provides a module for RTP. It is contained within `rtp.h` and `rtp.c`. The functions `ast_rtp_fd` and `ast_rtcp_fd` are contained within the RTP module. The transmission and reception of media, as mentioned in

the Section 4.5, is done using ast_read and ast_write. Within this channel, sip_rtp_read is called by sip_read which is the function assigned for reading, as has been seen previously. sip_rtp_read receives two parameters. The first is for the channel we are interested and the second is for the SIP private structure associated with that channel. sip_rtp_read checks the frame type using the frame descriptor number (fdno) and returns the frame which is read either from the rtp or vrtp streams using ast_read_rtcp for the rtcp frames and ast_read_rtp for the rtp frames. sip_write accepts two parameters, one for the relevant channel and the other for the frame we are writing. The frame type is assessed using the frametype variable in the frame structure, and if it is video then it is written to the vrtp stream, and if it is voice then it is written to the rtp stream. This writing is done using the ast_rtp_write function passing the ast_rtp stream and the ast_frame which we are writing.

Within load_module, ast_rtp_proto_register is used to register RTP for the SIP channel. The address of sip_rtp (&sip_rtp) is passed as a parameter, and the definition is shown below. This is necessary for the usage of RTP.

```
static struct ast_rtp_protocol sip_rtp = {
        get_rtp_info: sip_get_rtp_peer,
        get_vrtp_info: sip_get_vrtp_peer,
        set_rtp_peer: sip_set_rtp_peer,
        get_codec: sip_get_codec,
};
```

sip_get_rtp_peer returns the rtp stream (of type ast_rtp) and sip_get_vrtp_peer returns the vrtp stream. sip_set_rtp_peer utilises ast_rtp_get_peer and the sockets redirip and vredirip for the rtp and vrtp streams respectively. These sockets are located in the SIP private channel structure. Further discussion of the RTP functions mentioned is beyond the scope of this project.

5.1.6 Signalling

In terms of the signalling, the channel driver contains a number of functions for transmitting the packets discussed in Section 2.2.1. These are all prefixed with transmit such as transmit_response, transmit_invite, and transmit_notify. A number of sockets are established in the SIP private structure, and within the code for transmission. They are used throughout the code to send signalling data. SDP information from the SIP packets is used to interpret media types as video or audio and set up the various translations required for using the RTP functions discussed in the

previous section. These translations are performed by the translators in the Asterisk core according to translation paths determined usually by running `ast_make_compatible`, and stored in the channel structure.

5.2 SIP Video

We have found that the SIP driver provided with Asterisk does provide support for video if configured correctly. The SIP private structure, `sip_pvt`, contains two pointers, `rtp` and `vrtp`, for media streams. The channel driver uses the SDP information from the SIP packet to determine whether the media is video or audio as discussed earlier. Based on this information, it then utilises the `vrtp` stream for video based media or the `rtp` stream for voice based media. In this section we will outline how to configure the SIP channel to be capable of video, and demonstrate the results.

5.2.1 Testing Environment

Figure 5.1: Test setup for SIP video

Figure 5.1 shows the setup of the environment used for testing SIP video. We are using two Logitech webcams, each connected to a Pentium 4 3.0 GHz machine running Windows XP. For a video client, we have chosen to use Windows Messenger 5.0. While there are many SIP softphones available, there are very few soft-videophones available. Messenger seems to be the best client available, and is available freely from the web. Newer versions of Messenger have dropped the support for setting up SIP directly, so thus Messenger 5.0 is used. We setup Messenger to use the Asterisk box as the SIP server. This machine is also a Pentium 4 3.0 GHz machine running the current stable version of Asterisk, Asterisk 1.0.9, on Gentoo Linux.

5.2.2 Configuration

To make video possible in Asterisk we need to edit the main SIP configuration file of Asterisk (`sip.conf`). Below is an extract of what the configuration file should contain to enable video in the SIP channels.

```
[general]
...
videosupport=yes
allow=h261
allow=h263
```

This configuration allows the usage of the codecs H.261 and H.263, and enables video support. H.261 and H.263 are the prominent video codecs used for video telephony in commercial video phones and most video conferencing systems. Asterisk does support these codecs, so it is important that our channels are setup with the ability to make use of these codecs.

The next step is to add a registration account for the endpoint to use when signing in. This is also done in the sip.conf file. Below is another extract from this file to set up an account for registration with the Asterisk SIP proxy.

```
[4000]
type=friend
context=default
username=4000
secret=1234
host=dynamic
callerid="SIP Video Phone"
```

This is identical to the configuration used for setting up a SIP voice client in Asterisk. In this example, we have created an account 4000 on our SIP proxy with a password of 1234 and a caller id of "SIP Video Phone". These settings just need to be inserted into the video phone along with the IP address of our Asterisk box. Video calls will then be possible through Asterisk. More information about setting up sip.conf may be found in [19].

In order for us to be able to dial the video phone from another video phone, we need to insert an extension into the Asterisk dialplan. This is located in another configuration file called extensions.conf. Below is an extract from this configuration file which makes dialing our video phone from another phone a possibility.

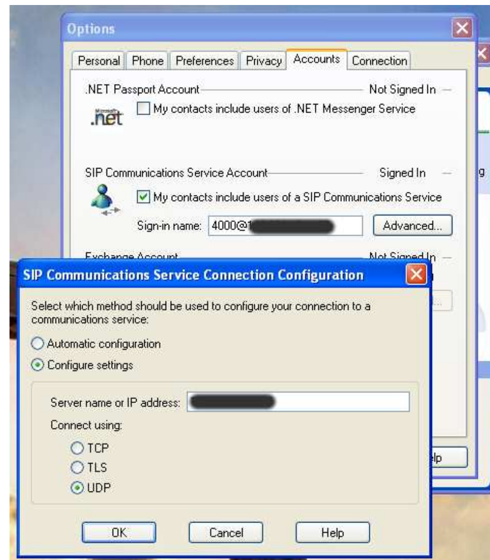


Figure 5.2: Setup of windows messenger 5.0

```
[default]
exten => 4000,1,Dial(SIP/4000)
```

We have now registered the video phone as a number 4000. By dialing this number from another phone we will be able to communicate with our video phone. To read more about the dialplan see [19].

Figure 5.2 illustrates the setup of Windows Messenger as a video phone. First you open the option dialog box, and navigate to the Account tab. Next, you tick the box for a SIP Communications Service, and enter the SIP URL as your Sign-in name. The SIP URL is of the form username-in-asterisk@ip-address-of-server. Finally, you click on the Advanced button and select Configure settings. You then input the server IP address and choose UDP as the protocol for connection. This configuration results in a video phone using Windows Messenger.

5.2.3 Results

The results of a call between two Windows Messenger endpoints may be seen in Figure 5.3. As, this figure shows, the video call was completed successfully.

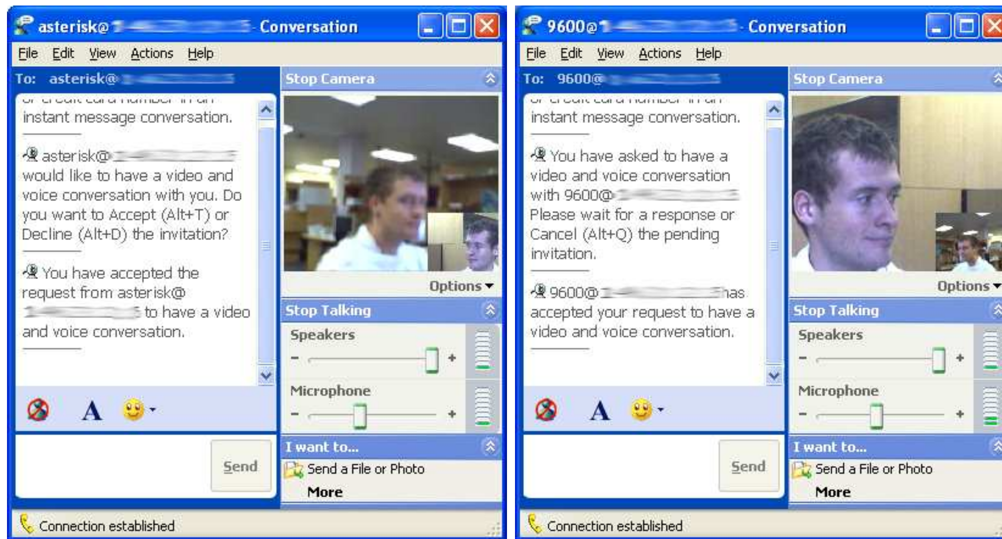


Figure 5.3: SIP video in operation

5.3 H323 Channel Implementations

A default installation of Asterisk does not include a channel driver compiled for H.323 support in Asterisk. We have sourced four different channel drivers for H.323. For each of these drivers, one has to collect the correct versions of their H.323 libraries, and then compile the channels with the correct configuration. In this section we will take a look at the channels, their basic architectures, advantages and disadvantages.

5.3.1 The OH323 channel

The OpenH323 (OH323) channel driver [16] was the first H.323 channel available for Asterisk. It utilises the OpenH323 stack, which is a complete open source H.323 implementation used by products such as gnomemeeting. Since its initial releases, it has undergone a lot of improvements and developed into a really robust H.323 channel driver. It is now capable of a much higher load and is still an active and developing project. OH323 is not included with Asterisk because of licencing issues. We have installed versions 0.5.9, 0.5.10, 0.6.0, 0.6.6, and 0.7.2 of this channel during this project, using the versions of OpenH323 recommended in the installation instructions of each version. We have used all these drivers for audio based calls, speaking with other channels such as the SIP channel. iLanga currently uses version 0.5.10 of the OpenH323. Since this channel driver has proven reliable, and is still developing, it is the choice channel implementation

for this project.

5.3.2 The H323 channel

The H323 channel driver was written by Jeremy McNamara to provide better usage of the RTP stack than the older versions of OH323. It claims to support a higher load than the older OH323 channel drivers [19]. This H.323 channel is included with Asterisk. It has based its signalling on the older versions of the OH323 channel driver, but has been altered to use the Asterisk RTP media stack for translation rather than requesting raw audio for translation purposes. There is a marked similarity to the SIP channel code in terms of RTP, however it lacks the implementation of a `vrtp` stream and does not use the `ast_rtp_bridge` for bridging between two H323 channels but rather a H.323 channel bridge similar to the one in older versions of the OH323 channel. The signalling uses the OpenH323 stack, however the back end used to interface with OpenH323 limits the version that may be used with OpenH323 1.12.2 and Pwlib 1.5.2 which are rather outdated. This means that improvements in the OpenH323 stack in terms of speed an functionality may not be taken advantage of with this channel.

5.3.3 The OOH323 channel

Objective Systems have developed an H.323 channel driver, Objective Open H323 (OOH323) [18], in C++. It is based on their own H.323 stack, which is limited in comparison to the mature OpenH323 stack. It is rather new and unexplored, but looks like in time it will be very promising. Currently the stack provides support for most of the common voice and video codecs, but the channel driver only provides support for a limited amount. The channel driver currently only supports ulaw, gsm, g729a, g723.1 and rfc2833. There are development plans to include support for more audio codecs as well as video. It is currently included in the asterisk-addons package for the development version (unstable) of Asterisk 1.2.0, which has just recently been released as beta. The OOH323 channel is not considered suitable for iLanga, as it is currently only available in Asterisk 1.2.0 which hasn't been released as a stable version. We have installed and tested this channel in Asterisk 1.2.0 beta using both version 0.7.2 and 0.7.3 of the OOH323 stack.

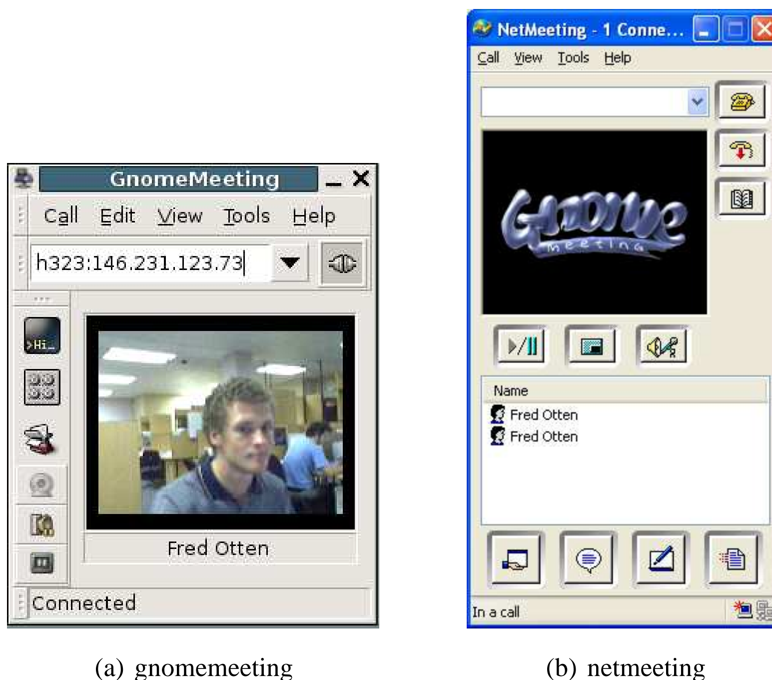
5.3.4 Channel Woomera

Channel Woomera [13] is a channel for the Woomera framework. Woomera is a basic text protocol designed by Craig Southern . Currently woomera only supports H.323 using the OpenH323

stack but will soon support the OPAL VoIP abstraction layer which will allow it to speak to many other protocols. It is not an ideal companion to Asterisk, however since it translates the signalling to a common protocol which is part of the advantage of Asterisk. This could also lead to a problem of generality, where new features are not supported by Woomera and hence are not available to the end points. We can accomplish the same results, if not better, using generic channels for the various protocols because we don't have a go-between. The advantage however is that as Woomera develops, new protocols will be supported by it, and this will be inherited into Asterisk through Woomera. We have installed and compiled the channel on Asterisk 1.2.0 beta.

5.4 No H323 Video

Netmeeting, which is provided in the default installation on Windows XP, is used as a client for the H.323 endpoints. We can show that that the OpenH323 and Pwlib installed on the server supports video. Gnomemeeting is an application, similar to netmeeting, available in Linux for H.323 video. By initiating a call between netmeeting and gnomemeeting, we can show that the OpenH323 and Pwlib installed on that machine are configured correctly for video.



(a) gnomemeeting (b) netmeeting

Figure 5.4: gnomemeeting speaking with netmeeting

Figure 5.4 illustrates a call between netmeeting and gnomemeeting running on the server. We can see that video is operational. Because the sever does not have a webcam installed, it just sends a gnomemeeting logo which moves around. We can see the gnomemeeting logo coming though to netmeeting and the video coming through to gnomemeeting. This proves that the current installation of OpenH323 and Pwlib supports video.

None of the H.323 channels we have discussed support video. We have attempted to initialise video calls with all four of the channels mentioned in the previous section, but it did not work with any of them. A closer look at the source code reveals that the interface between OpenH323 and the channel drivers for the OH323 channels and H323 channels does not contain routines to video in terms of media and signalling.

```
static struct ast_rtp *oh323_get_vrtp_peer(struct ast_channel *chan)
{
        return NULL;
}
```

This extract is taken from the H323 channel's code, which uses the Asterisk RTP stack. This clearly does not send any video. Attempts to model the code on the SIP channel driver failed, as the socket information could not be extracted using the interface to OpenH323. Adding vrtp pointers thus proved futile, as the signalling could not be handled with the current implementation. Thus providing video would incorporate a thorough investigation into the OpenH323 libraries and creating an interface for the channel driver to use for signalling and media handling, which is beyond the scope of this project, but is suggested as a future extension.

5.5 Inheritance of features

There are many facilities available to the channels in Asterisk. These include services such as conferencing, call transfer, call parking, music on hold, and a large range of applications, written using Asterisk Gateway Interface (AGI) Scripts and the C API. This provides a lot of flexibility, and allows easy service creation within Asterisk. We have found that the call parking, call transfer and music on hold to all be operational with the SIP video channels. Call parking freezes the last video frame received, and plays music on hold as it does on the voice end points. Call transfer works really well. Conferencing, however does not work. The MeetMe application

which is used for setting up conferences claims to support video, however in reality that support is not yet available, because of the lack video mixers in the MeetMe application.

5.6 Summary

This chapter has described the channel implementations available for SIP and H.323 within Asterisk. It began with a detailed look at the SIP channel driver, and then elaborated on the configuration of SIP video. It also took a look at the various H.323 channels we have installed, and highlighted the lack of H.323 video support, and where the problem lies with evidence of the H.323 stack supporting video. It concluded with a summary of the inheritance of features in which we saw that call parking and call transfer are successfully carried over to video.

Chapter 6

iLanga User Interface and Extensions

The iLanga user interface provides a web based system for the PBX. It contains a directory listing, facilities for listening to voicemail and a database of calls made with their costs. It is designed to be extensible. In this chapter we briefly discuss the architecture of the web interface, and show and explain the extensions that we have made for providing call parking and call transfer facilities within the interface.

6.1 Architecture

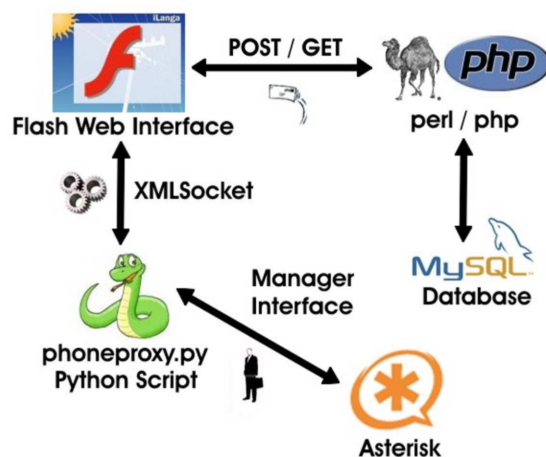


Figure 6.1: Architecture of the web interface

The basic architecture of the iLanga User interface is shown in Figure 6.1. The interface is a Flash project designed using Macromedia Flash MX 2004 Professional. This interface uses

PHP and Perl scripts to get data from a MySQL database which contains information about the users, their devices and the calls that have been made. This information is then displayed using a variety of movie clips (basic graphical components in Flash) and text in Flash. The Flash interface communicates with a Python script using XML Sockets, a feature readily available in Flash. This Python script then communicates with the Asterisk Manager Interface, passing only necessary responses back to the User interface using XML.

The remainder of this section describes the manager interface, the Python script and the extensibility this architecture provides.

6.1.1 Asterisk Manager Interface

The Asterisk Manager Interface (otherwise known as the Manager API) provides a method of performing various actions remotely by sending plain text commands over a TCP connection. This TCP port is opened when the Asterisk PBX starts up if it is enabled in the configuration file `manager.conf`. The manager interface provides external applications with the ability to connect to this port and communicate with Asterisk by writing and reading requests and responses respectively. Various actions such as initialising a call, closing a channel and redirecting a call are possible. The Asterisk Manager Interface also reports the state of devices using a response. Too many connections to the manager interface can cause instability in the system. It is for this reason that we create a middle layer between our Flash interface back end and the Asterisk manager interface using a Python script.

6.1.2 Python script

The Python script is written using the twisted framework. The twisted framework, written in Python, was created for writing networked applications. It includes implementations for many common used network services, and is a perfect choice for managing the TCP connections that are required for interfacing between our user interface and the manager interface. This script may be readily extended to support more features so that more information may be fetched by the user interface from multiple sources using new XML commands in the user interface back end.

6.1.3 Extensibility

The extensibility provided by this architecture is rather useful. It means that we can easily extend the user interface to provide new features for the users. By writing new action scripts, adding new graphical and textual objects to the Flash project and adding to the Python scripts, we can easily add features to the user interface. Features such as call transfer and call parking are not possible on soft phones which do not support the sending of DTMF tones while in a call. We can easily add these features to the user interface. The next section expands on the extensions that we have made to the user interface. Further information on the user interface and the twisted framework may be found in [10].

6.2 Extensions

This section exposes the extensions we have made to the iLanga user interface to add call transfer and call parking. It gives details on the manager interface commands used, and show screen shots of the new interface with details on the operation of these new features.

6.2.1 Call Transfer

Asterisk provides facilities for call transfer. We can make this facility available to the users by adding `tT` to the arguments sent to the dial application when initiating a call. Call transfer is usually done by pressing `#` on the phone and entering the extension to which you wish to transfer the call. Unfortunately, many video phones, especially soft phones such as Window Messenger, cannot send DTMF tones during a call and hence they are not able to use this facility.

The manager interface in Asterisk 1.0.9 has the ability to do call transfer using a `Redirect` action. We thus decided that we will extend the iLanga user interface to provide facilities for call transfer by sending the relevant manager interface commands.

The following command can be used to transfer a call in the manager interface:

```
Action: Redirect
Channel: SIP/200-8f54
Context: local-from-sip
Exten: 9600
Priority: 1
```

The channel field can be obtained using the `Command` action in the manager interface command to run `show channels`. This request returns a response containing the results obtained when running `show channels` in the CLI. The following manager interface command is used:

```
Action: Command
Command: show channels
```

From these results we choose the appropriate channel name, ie. our end of the call. The channel names are quite intuitive, so this process can be easily replicated with a script.

In order to make call transfer a possibility in the user interface we have created a few Flash movies that are used in the interface. We also have written Flash action scripts which send the commands to the manager interface via the Python script. For this, XML Sockets are used in the same manner as before. The next step was to add some code to the Python script so that it can pass back the channel names involved in the call to Flash. Extracts of the source code and explanations are contained in Appendix C.

Figure 6.2: Initiating a call transfer

Figure 6.2 illustrates our updated user interface for call transfer. The status button located at the bottom right corner of the interface is red when we are in a call. If the user takes this button

and drags it to a person in the directory, as indicated in Figure 6.2, they may transfer the call to that person. The user may also click on the person's name in the directory for the same effect. Figure 6.3 then appears requesting confirmation of the call transfer requested. If you click on Yes, then the user interface performs the call transfer using the XML sockets to communicate with the Python script, as described earlier.

An information tab has been made available within the directory which explains how to transfer a call. This is illustrated in Figure 6.4.

6.2.2 Call Parking

Asterisk also provides call parking facilities. We park a call by transferring it to the extension defined as the parking extension. Call parking is configured in the configuration file `features.conf` in Asterisk 1.0.9. In this configuration file we specify the call parking extension and the positions available. For example:

```
[general]
parkext => 400            ; What ext. to dial to park
parkpos => 401-420        ; What extensions to park calls on
context => local-from-sip ; Which context parked calls are in
parkingtime => 600        ; Number of seconds a call can be parked for
```

When a call is parked, the user is read the number of the parking position, and the other end of the call receives music on hold until the parked call is retrieved, the parking time has expired or the parked user hangs up. To park a call using the manager interface, we transfer the call to the parking extension. This is done in the same manner as specified in the previous section on call transfer. We just set `Exten` to 400 (or the relevant parking extension).

Call parking is an important facility in a PBX, so we have decided to add this facility to the user interface. We have done this by creating movie clips for the various graphical element and writing action scripts to send XML to the Python script for the relevant manager interface commands.

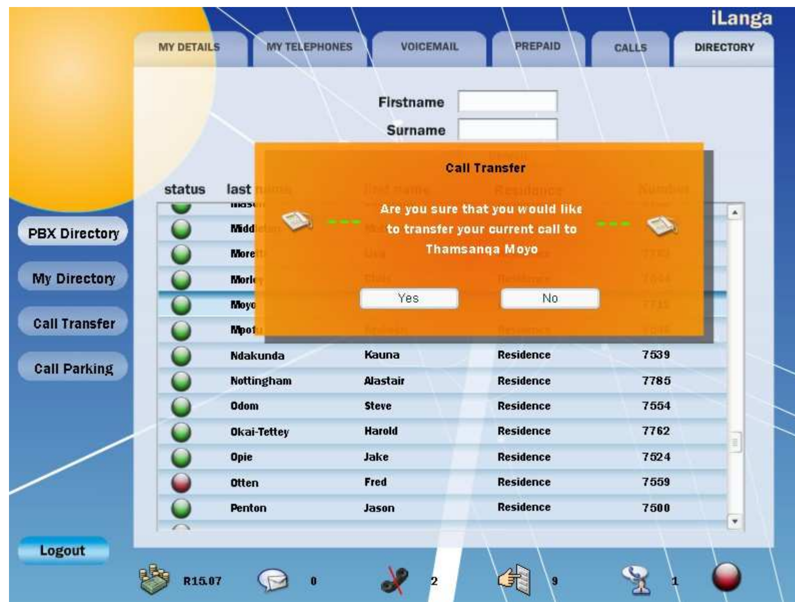


Figure 6.3: Confirmation of call transfer

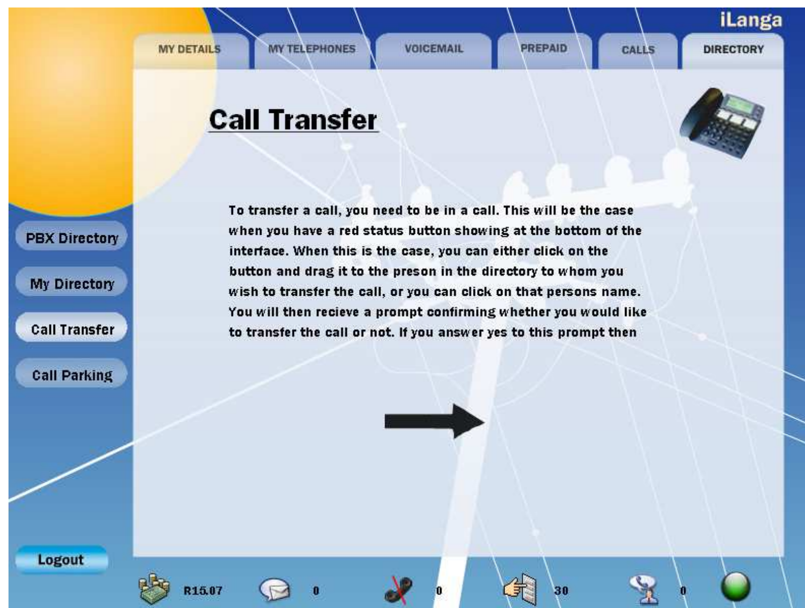


Figure 6.4: The call transfer tab in the updated iLanga interface

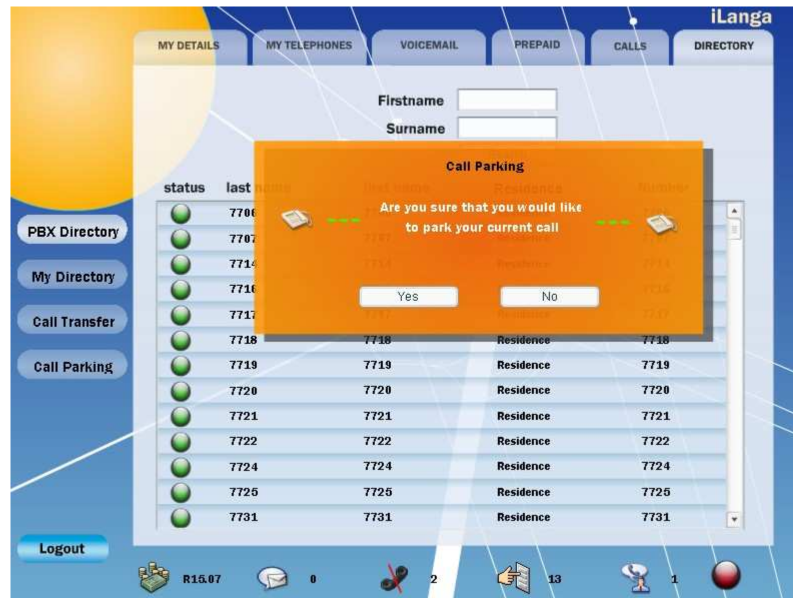


Figure 6.5: Parking a call in the updated iLanga interface

When we are in a call, the colour of the status button in the bottom right corner of the interface is red. We have already noted that dragging it to a person can initiate a call transfer. Clicking on it, however, brings up the option of parking the current call. This is illustrated in Figure 6.5. If the user confirms the request, then the necessary calls are made to the manager interface via the Python script, and the call is parked and the number read back for retrieval. It has been decided not to include the functionality of retrieving a parked call for security reasons. If you know the number then you can retrieve the parked call, as with the usual operation of call parking by dialing the number given when the call was parked. Including the retrieval of parked calls into the interface is a trivial extension. It would just involve sending an `Originate` request to the manager interface with the number that has been read out. The parked calls, and the number required for retrieval, may be retrieved by running the `show parkedcalls` command from the CLI. In the manager interface, we can issue a `Command` action request, or send a `ParkedCalls` action request. This will cause a response containing information about the calls that are currently parked.

Figure 6.6: Indication of having parked a call

When we have parked a call, we change the colour of the status button at the bottom right corner of the interface to orange. This is illustrated in Figure 6.6. The Asterisk Manager interface sends out an Event response `ParkedCall` containing information about a parked call. This information is received and checked whether the user has parked call. Parked calls time out and may be hung up. Therefore we check the parked calls at the beginning of each status check within the interface, and update the status button accordingly.

An information tab has been made available within the directory which explains how to park a call. This is illustrated in Figure 6.7.

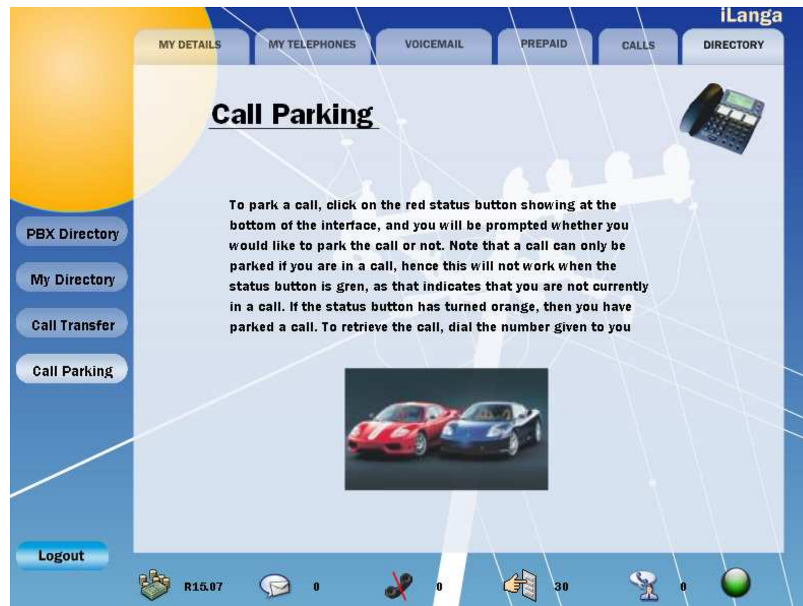


Figure 6.7: The call parking tab in the updated iLanga interface

6.3 Summary

This chapter has demonstrated the extensibility inherent in the architecture of this web interface by adding features for call transfer and call parking. We have briefly described the architecture of this user interface and explained how it co-operates with the Asterisk manager interface via the Python script. We have also introduced the manager interface, and the twisted framework used in the Python script.

Chapter 7

Conclusion

This chapter provides an overview of the document, detailing the achievements and summarising the findings of our investigation into the provision of video capabilities in iLanga. It also details further extensions possible in this field, and concludes the writeup with final words on the subject.

7.1 Document Summary

Chapter 2 introduced the session based and media based protocols used in real time multimedia. In particular it described SIP and H.323, two session based protocols which we deal with in this project. RTP, the media based protocol used by these protocols was also described. Chapter 3 described Asterisk and iLanga, the framework this project is built on. The architectures and facilities provided have been highlighted, and extensibility discussed. Asterisk, the core of iLanga, uses a channel-based architecture. This was explored in depth in chapter 4, which takes a look at channels, the channel API and how they are used. Chapter 4 also demonstrated an example channel created for proof-of-concept. Chapter 5 continues from the previous chapter. It took a look at the SIP channel in detail and elaborated on the configuration of SIP video. It also looked at the various H.323 channels we installed, and highlighted the lack of H.323 video support, and expanded on the problem. It finally concluded with a look at the inheritance of features. This revealed that call parking and call transfer were successfully carried over to video channels. Chapter 6 demonstrated the web interface and expanded on the extensions made.

7.2 Video in iLanga

We have shown that video is available in iLanga through the media capabilities available in Asterisk. It does, however, largely depend on the implementation of the particular channel. Tests have revealed that SIP video is available, while the H.323 channel drivers do not currently support video. This is because of their channel implementations in terms of handling media and interfacing with an external H.323 stack.

7.3 Inheritance of features

Natively Asterisk provides facilities such as call parking and call transfer to the voice channels. By using the manager interface, and the iLanga user interface, we have performed call transfer and call parking on our video endpoints. This establishes the inheritance of these features are inherited to video channels. We have found that the MeetMe application, which provides conferencing facilities to channels, does not support video because of it lack of video mixers.

7.4 Summary of Findings

This project has resulted in the following research findings:

- Channels can be constructed using the channel API.
- Video is possible in iLanga through Asterisk.
- The SIP and IAX2 channels support video.
- The H.323 channel does not support video due to implementations of RTP and the interface with external stack.
- The features, such as call parking and call transfer, available for voice channels are inherited to video channels. When a call is parked on a video channel, music on hold gets played, as with voice channels, while the screen displays the last video frame received.
- The MeetMe application, for conferencing in Asterisk, does not support video as claimed.

7.5 My project achievements

My project achievements include:

- Documenting and creating a flow chart of the operation of the Dial application used for initialising a call in Asterisk.
- Producing a document which explains the channel concept, and expands on the channel API particularly the functions and structures available.
- Implementing an example channel in Asterisk from the knowledge gained. It does not support media transfer, but outputs to the CLI when a call is initialised.
- Finding video to be available in the SIP channel if it is configured correctly.
- Documenting the configuration of both the SIP end points and the SIP channel for video support.
- Documenting the basic operation of the SIP channel module.
- Finding H.323 video to not be available in the four channels we have found, compiled and installed in Asterisk.
- Explaining why this is the case, presenting proof that the OpenH.323 stack provides video.
- Explaining how video is provided in Asterisk.
- Investigating the inheritance of the features available for voice channels to video channels and reporting the results.
- Extending the iLanga user interface to provide support for transferring and parking calls.

7.6 Further Extensions

This research explains channels and explores the availability of video in iLanga. In this section we explain some possible extensions which stem from this research.

7.6.1 Video mail and Video on Hold

The iLanga PBX provides facilities for voice mail and music on hold. The video facilities would be well complemented with video on hold and video mail. The iLanga user interface could also be extended to include playback of the video recorded using video mail.

7.6.2 Legacy video channel

There are many legacy video devices around. From this research on the channel API and the channel structure, a channel may be developed for one of these legacy video devices.

7.6.3 Video MeetMe application

The MeetMe application, as has been mentioned, does not support video. By investigating the MeetMe application and the various MCUs available, a video MeetMe application could be developed for video conferencing.

7.6.4 H323 video within a H323 channel

This research has shown that the H.323 channels available for Asterisk do not support video. We have mentioned that this is because of the handling of media and the interfacing with the external stack. This extension would involve investigating an external stack, and from this information rectifying the lack of support for video. A new H.323 channel could also be built that does not use an external stack.

7.6.5 H264 codec for cell phone technology

The emergence of third generation cell phone technology, and accompanying devices which support video raises a need to provide support for this technology. Most of these devices use the H.264 codec specification for video. This is not available in Asterisk. This extension would involve taking a look at the H.264 specification and the implementation of codecs and translators within the Asterisk core and, from this information, developing codecs and translators using appropriate Asterisk APIs.

7.6.6 Streaming

Facilities such as video on demand may be developed for the iLanga PBX. This would mean that a user may dial a number and watch the associated live television stream associated with this number. This is analogous with the live streaming concept, except using a PBX.

7.7 Final words

iLanga is a full featured PBX developed at Rhodes University which provides support for high quality voice over multiple protocols with accompanying services such as voicemail, call forwarding and call parking. Asterisk, the core of iLanga, utilises a channel-based architecture. We have provided a detailed overview of the channel API, and described how a new channel can be created. We have also presented video using Asterisk, and demonstrated the inheritance of features for a video channel, answering the question posed. The features tested include call transfer, call parking and music on hold.

We have thus investigated the possibility of video in iLanga, and shown that some of the features available for voice are extended to video.

References

[1] J. Arkko, V. Torvinen, G. Camarillo, A. Niemi and T. Haukka. *Security Mechanism Agreement for the Session Initiation Protocol (SIP)*. IETF Request for Comments 3329. January 2003.

[2] Asterisk. *Asterisk Open Source PBX*. website located at: http://www.asterisk.org. 2005.

[3] U. Black. *Voice over IP*. Advanced Communication Series. Prentice Hall. 2000.

[4] G. Camarillo. *SIP: Compressing the Session Initiation Protocol (SIP)*. IETF Request for Comments 3486. February 2003.

[5] Cisco Systems Inc. *Guide to Cisco Systems' VoIP Infrastructure Solution for SIP*. retrieved from: http://www.cisco.com/univercd/cc/td/ doc/product/voice/sipsols/biggulp/bgsip.pdf. 2000.

[6] I. Dalgic and H. Fang. *Comparison of H.323 and SIP for IP Telephony signalling*. Proc. of Photonics East, Boston, Massachusetts. September 1999.

[7] M. Handley, H. Schulzrinne, E. Schooler and J. Rossenberg. *SIP: Session Initiation Protocol*. IETF Request for Comments 2543. March 1999.

[8] J. Hitchcock, J. Penton, A. Terzoli, *The design of a graphical frontend for and Asterisk-based software PBX*, South African Telecommunications Networks and Appliances Conference, September 2004, Spiers.

[9] J. Hitchcock. *Decorating Asterisk: Experiments in Voice over IP Service Creation for a Multi-Protocol Environment*. Rhodes University. May 2005.

[10] IANA. *RTP Parameters*. Internet document located at: http://www.iana.org/assignments/rtp-parameters.

[11] IPTel. *SIP Express Router*. located at: http://www.iptel.org/ser. 2005.

[12] H. Liu and P. Mouchtaris. *Voice over IP signalling: H.323 and beyond*. IEEE Communications Magazine. 2000.

[13] A. Minessale. *Channel Woomera*. located at: http://www.pbxfreeware.org/chan_woomera. 2005.

[14] D. Minoli and E. Minoli. *Delivering voice over IP networks*. Wiley Computing Publishing. 1998.

[15] Nortel Networks. *A Comparison of H.323 v4 and SIP*. 3GPP S2, Tokyo, Japan. Technical Document: S2-000500. January 2000.

[16] OpenH323. *Open H.323 channel driver*. located at: http://www.inaccessnetworks.com/projects/Asterisk-oh323. 2005.

[17] OpenGK. *Open H.323 gatekeeper*. located at: http://www.gnugk.org. 2005.

[18] Objective Systems. *OOH323 channel driver*. located at: http://www.obj-sys.com. 2005.

[19] Various Parties. *VoIP info*. Internet forum located at: http://www.voip-info.org. 2005.

[20] J. Penton and A. Terzoli. *Asterisk: A Converged TDM and Packet-based Communications System*, South African Telecommunications Networks and Appliances Conference (SATNAC), September 2003, Fancourt.

[21] J. Penton, A. Terzoli, iLanga: *A Next Generation VoIP-based, TDM-enabled PBX*, South African Telecommunications Networks and Appliances Conference (SATNAC), September 2004, Spiers.

[22] J. Rossenberg, J. Lennox and H. Schulzrinne. *Programming Internet Telephony Services*. IEEE Internet Computing, Vol. 3, No. 3, pg. 63-72. June 1999.

[23] J. Rossenberg, H.Schulzrinne, G. Camarillo, A. Johnston, J, Peterson, R. Sparks, M.Handley and E. Schooler. *SIP: Session Initiation Protocol*. IETF Request for Comments 3261. June 2002.

[24] J. Rossenberg and H.Schulzrinne. *Session Initiation Protocol (SIP): Locating SIP Servers:*. IETF Request for Comments 3263. June 2002.

[25] H. Schulzrinne and J. Rossenberg. *Internet Telephony: Architecture and Protocols - An IETF perspective*. Computer Networks Vol. 31, No. 3. 1999.

[26] H. Schulzrinne, S. Casner, R. Frederick and V. Jacobson. *RTP: A Transport protocol for real time applications*. IETF Request for Comments 1889. January 1996.

[27] H. Schulzrinne. *RTP Profile for audio and video conferences with minimal control*. IETF Request for Comments 1890. January 1996.

[28] H. Schulzrinne. *SIP - Signalling for Internet Telephony and Conferencing*. Slides - Berkeley Multimedia, Interfaces and Graphics Seminar located at: http://bmrc.berkeley.edu/courseware/ cs298/fall98/w14/slides.pdf. November 1998.

[29] H. Schulzrinne. *The Session Initiation Protocol (SIP)*. Slides: hgs/Tutorial located at: http://www.cs.columbia.ed/~hgs/teaching/ais/slides/sip_long.pdf. University of Columbia. May 2001.

[30] H. Schulzrinne and J. Rossenberg. *Comparison of H.323 and SIP for IP Telephony*. Network and Operating Systems Support for Digital Audio and Video (NOSSDAV), Cambridge, England. 1998.

[31] B. Schwarz. *Asterisk Open-Source PBX*. Linux Journal Vol. 2004, Issue 118, pg. 6. Specialized Systems Consultants, Inc. Seattle, WA, USA February 2004.

[32] Team Solutions. *Video Conferencing Standards and Terminology*. Internet resource located at: http://www.teamsolutions.co.uk/tsstds.html.

[33] M. Spencer, M. Allison, C. Rhodes, et al. *Asterisk Handbook (Version 2)*. Digium. March 2003.

[34] D. Tarrant and T. Hunt. *VoIP - Voice over IP overview*. University of Southhampton. Document located at: http://www.ecs.soton.ac.uk/~dt302/guides/VOIP-Overview.pdf. August 2004.

[35] Unknown Authors. *Implementing a channel*. Asterisk source documentation located in /doc/channel.txt.

Appendix A

More Channel API Functions

Function	Description	
ast_request_and_dial	Description:	Request a channel using the function above, and dials the channel
	Example call:	ast_request_and_dial(type, format, number, timeout, reason, callerid) where type is a character array containing the type of channel (eg. SIP), format is an integer referring to the format of the data, number is of any type and contains the number being called, timeout an integer containing the value for timing out the dial attempt, reason is an integer containing a value representing the reason for failure if it occurs, and callerid is a character array containing the callerid of the channel being setup
	Returns:	ast_channel*, null if unsuccessful
ast_soft_hangup	Description:	Hangsup the channel as above, however does not destroy the channel structure just sets a variable _softhangup to the cause variable (This can be used to safely hangup a call managed by another thread)
	Example call:	ast_softhangup(chan, cause) where chan is a pointer to an ast_channel struct of the channel which is being softly hangup, and cause is an integer used for the value of the variable
	Returns:	int, always 0

Function	Description	
ast_check_hangup	Description:	Determine whether hangup has been requested for a channel
	Example call:	ast_check_hangup(chan) where chan is a pointer to the ast_channel struct for which you are enquiring
	Returns:	int, 1 if hangup requested otherwise 0
ast_channel_setwhentohangup	Description	Place a time limit on when to hangup a channel
	Example call:	ast_channel_setwhentohangup(chan, offset) where chan is a pointer to the ast_channel struct of the channel you wish to place a time limit upon, and offset is a time_t variable which is the time in seconds from the current time that you are requesting a hangup of the channel specified
	Returns:	void
ast_senddigit	Description:	Sends a digit to a channel using a function in the channel driver
	Example call:	ast_senddigit(chan, digit) where chan is a pointer to the ast_channel struct of the channel to which we are sending a digit and digit is a character containing the digit being sent
	Returns:	int, 0 always
ast_channel_masquerade	Description:	Creates a clone of a specified channel, taking the guts of the channel and moving it to another channel, then destroying the old channel structure, leaving the guts in the new channel
	Example call:	ast_channel_masquerade(original, clone) where original and clone are pointers to the ast_channel structs which we are dealing with for the masquerade described above
	Returns:	int, -1 on error, 0 on success

Function	Description	
ast_begin_shutdown	Description:	Initiate a system shutdown, stop channels from being allocated
	Example call:	ast_begin_shutdown(hangup) where hangup is an integer representing a boolean value of whether or not to soft hangup all channels in operation
	Returns:	void
ast_cancel_shutdown	Description:	cancels an existing shutdown, and resumes normal operation
	Example call:	ast_cancel_shutdown()
	Returns:	void
ast_active_channels	Description:	gets the number of active channels
	Example call:	ast_active_channels()
	Returns:	int, the number of active channels
ast_setstate	Description:	change the state of a current channel
	Example call:	ast_setstate(chan, state) where chan is a pointer to the ast_channel struct of the channel we wish to change the state and state is an integer representing the state that we are setting the channel to
	Returns:	int, 0 always
ast_queue_frame	Description:	queue an outgoing frame
	Example call:	ast_queue_frame(chan, frame) where chan is a pointer to the ast_channel struct of the channel and frame is a pointer to an ast_frame which we are going to queue
	Returns:	int, -1 on frame error, 0 otherwise
ast_queue_control	Description:	queue an outgoing control using ast_queue_frame
	Example call:	ast_queue_control(chan, control) where chan is a pointer to the ast_channel struct of the channel and control is an integer representing a control frame
	Returns:	int, -1 on frame error, 0 otherwise

Function	Description	
ast_queue_hangup	Description:	queue an outgoing hangup
	Example call:	ast_queue_hangup(chan) where chan is a pointer to the ast_channel struct of the channel
	Returns:	int, -1 on error, 0 otherwise
ast_change_name	Description:	Change the name of a channel
	Example call:	ast_change_name(chan, newname) where chan is a pointer to the ast_channel struct of the channel we wish to change the name of, and newname is a character array containing the new name
	Returns:	void

Appendix B

Example Channel

B.1 chan_eg.c

```
/*
Fred Otten
Channel Creation Example
Based on Investigations into the SIP, H323 and IAX channel drivers
General Conventions and Basic Requirements

*/
#include <stdio.h>
#include <string.h>
#include <asterisk/lock.h>
#include <asterisk/channel_pvt.h>
#include <asterisk/cli.h>
#include <asterisk/module.h>

#include <asterisk/logger.h>
static char *desc = "Example Channel (EG)";
static char *type = "EG";

static char *tdesc = "Example Channel (EG)";
static char eg_info_usage[] =
"Usage: eg info\n"
"       Displays Information about Example Channel\n";
static int usecnt = 0;

AST_MUTEX_DEFINE_STATIC(usecnt_lock);
static int eg_info(int fd, int argc, char* argv[])
{
        ast_log(LOG_NOTICE, "This is the example channel in Asterisk\n");
        return 0;

}
static struct ast_cli_entry cli_eg_info =

        { { "eg", "info", NULL }, eg_info, "Example Channel Information", eg_info_usage };
static int eg_hangup(struct ast_channel *ast)
{
        return 0;

}
static int eg_call(struct ast_channel *ast, char *dest, int timeout)
{
        ast_verbose("== Initiating a new call to the example channel ==\n");
        return 0;
```

```
}
static struct ast_channel *eg_new(char *title)
{
        ast_verbose("== Creating a new Example Channel :) ==\n");
        struct ast_channel *tmp;
        tmp = ast_channel_alloc(0);
        if (tmp)
        {
                char *p = "pvt";
                snprintf(tmp->name, sizeof(tmp->name), "EG/%s-%04x", title, rand() & 0xffff);
                tmp->type=type;
                tmp->pvt->call=eg_call;
                tmp->pvt->hangup=eg_hangup;
                tmp->pvt->pvt=p;
                ast_setstate(tmp, AST_STATE_UP);
                ast_mutex_lock(&usecnt_lock);
                usecnt++;
                ast_mutex_unlock(&usecnt_lock);
        }
        else
                ast_log(LOG_WARNING, "Unable to create channel");
        return tmp;

}
static struct ast_channel *eg_request(char *type, int format, void *data)
{
        struct ast_channel *tmpc = NULL;
        char *dest = data;
        tmpc = eg_new(dest);
        return tmpc;

}
int load_module()
{
        if (ast_channel_register(type, tdesc, 2, eg_request))
        {
                ast_log(LOG_ERROR, "Unable to register channel class %s\n", type);
                return -1;
        }
        ast_cli_register(&cli_eg_info);
        return 0;

}
int unload_module()
{
        ast_cli_unregister(&cli_eg_info);
        ast_channel_unregister(type);
        return 0;

}
int usecount()
{
        int res;
        ast_mutex_lock(&usecnt_lock);
        res = usecnt;
        ast_mutex_unlock(&usecnt_lock);
        return res;

}
char *key()
{
        return ASTERISK_GPL_KEY;

}
char *description()
{
        return desc;
}
```

Appendix C

iLanga User Interface Extensions

C.1 directory.fla

C.1.1 Action script extracts

Call Parking

```
on(release) {
        var chan = this._parent._parent._parent._parent._parent._parent.mc_statusbar.mc_jason.otherchan;
        if (chan != "")
        {
                myStr = "Action: Redirect, Channel: "+ chan +", Exten: 400,Context: from-manager, Priority: 1;\r\n";
                this._parent._parent._parent._parent._parent._parent.mc_statusbar.mc_jason.myXMLSocket.send(myStr);
        }
        this._parent.removeMovieClip();
}
```

Call Transfer

```
on(release) {
        var chan = this._parent._parent._parent._parent._parent._parent.mc_statusbar.mc_jason.ourchan;
        if (chan != "")
        {

                myStr = "Action: Redirect, Channel: "+ chan +", Exten: " + _parent.callchannel + ",Context: from-manager,
                    Priority: 1;\r\n";
                this._parent._parent._parent._parent._parent._parent.mc_statusbar.mc_jason.myXMLSocket.send(myStr);
        }
        this._parent._parent.userSP.spContentHolder.disableself(false);
        this._parent.removeMovieClip();
}
```

Setting up tabs

```
tabsd=["PBX Directory","My Directory","Call Transfer","Call Parking"]
```

C.1.2 Graphics and movie clips

This section shows the movie clips that have been created in Flash for call transfer and call parking.

Figure C.1: Call Parking dialog box

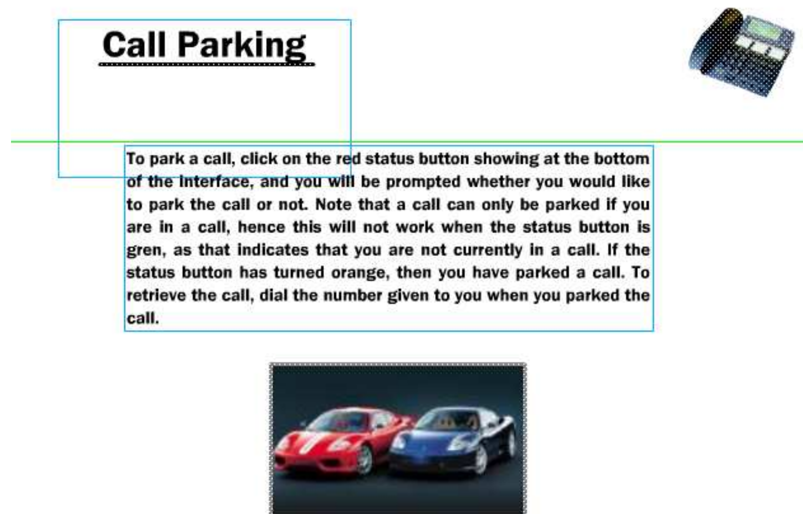


Figure C.2: Call Parking tab in the directory

Figure C.3: Call Transfer dialog box

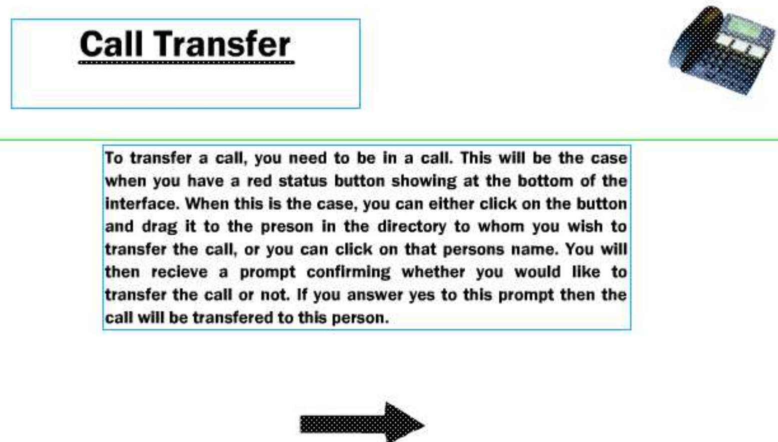


Figure C.4: Call Transfer tab in the directory

C.2 nav.fla

C.2.1 Action script extracts

LED Button

This contains code for the dragging of the LED buttons and the general state.

```
this.buttons = ["red", "green", "Flashing", "orange"];
this.states = ["up", "down", "ringing", "down"];
this.attachMovie("redled","red",10);
this.attachMovie("greenled","green",11);
this.attachMovie("Flashingled","Flashing",12);
```

```
this.attachMovie("orangeled","orange",13);
this.UP = 0;
this.DOWN = 1;
this.RINGING = 2;
this.PARKED = 3;
this.setstate = function(n) {
        this.state = n;
        if(this.state == this.DOWN) {
                this.green._visible = true;
                this.red._visible = false;
                this.Flashing._visible = false;
                this.orange._visible = false;
        }
        else
        if(this.state == this.UP) {
                this.green._visible = false;
                this.red._visible = true;
                this.Flashing._visible = false;
                this.orange._visible = false;
        }
        else
        if(this.state == this.RINGING) {
                this.green._visible = false;
                this.red._visible = false;
                this.Flashing._visible = true;
                this.orange._visible = false;
        }
        else
        if(this.state == this.PARKED) {
                this.green._visible = false;
                this.red._visible = false;
                this.Flashing._visible = false;
                this.orange._visible = true;
        }
}
if (this.state!=3) { this.setstate(this.DOWN); }
this.onPress = function() {
        var m = this._parent._parent._parent;
        m.attachMovie(this.buttons[this.state]+"led","mouseicon",100);
        m.onMouseMove = function() { updateAfterEvent(); }
        var xdiff = this._parent._xmouse-this._x;
        var ydiff = this._parent._ymouse-this._y;
        m.mouseicon.state = this.state;
        m.mouseicon._x = m._xmouse-xdiff;
        m.mouseicon._y = m._ymouse-ydiff;
        m.mouseicon.origx = m.mouseicon._x;
        m.mouseicon.origy = m.mouseicon._y;
        m.mouseicon.orig = this._parent;
        m.mouseicon.onMouseUp = function() {
                stopDrag();
                if ((Math.abs(this._x-700)<5) and (Math.abs(this._y-554)<5)) {
                        if (this.state==3) {
                                // Call currently parked
                        }
                        else {
                                // Park the call
                                this._parent.parkCall(this.state);
                        }
                        m.movemouseicon = true;
                }
                else {
                        m = this._parent;
                        var dt = eval(this._droptarget);
                        if(dt._parent._parent and (dt._parent._parent==m.nav.container.holder.directory.userSP.spContentHolder)) {
                                m.makeCall(dt._parent.number.text,dt._parent.fname.text+" "+dt._parent.lname.text,this.state);
                                this.removeMovieClip();
                        }
                        else {
                                m.movemouseicon = true;
                        }
                }
```

```
         }
         startDrag(m.mouseicon,false);
}
stop();
```

Frame 1

Contains code for XML Sockets and the code called when we click on the directory

```
  myPark = "Action: ParkedCalls\r\n\r\n";
  MyXMLSocket.send(myPark);
  //parkedcall

  if (messageportions[0] eq "Event: Link"){
       ourchan = messageportions[1].substring(10);
       otherchan = messageportions[2].substring(10);

       devData = new LoadVars()
       devData.username = _global.username;
       devData.password = _global.passwd;
       devData.parent = this;

       devData.onLoad = function(success)
       {
             if (success) {
                   var n;
                   var temp = 0;
                   for(n = 0; n < this.num; n++) {
                        if (ourchan.indexOf(this["channel"+n])!=-1)
                        {
                             temp = 1
                        }
                   }
                   if (temp==0)
                   {
                        tmp = ourchan;
                        ourchan = otherchan;
                        otherchan= tmp;
                   }
            }
       }
       devData.sendAndLoad("http://pbx.ict.ru.ac.za/iLanga/userdevices.php",devData,"POST");

}
if (messageportions[0] eq "Event: Unlink"){
       ourchan = "";
       otherchan = "";

}
if (messageportions[0] eq "Event: ParkedCall"){
       leds.setstate(3);

}
function makeCall(dnumber,dname,state) {
         if (state==0){
                 //Call Transfer
                 this.nav.container.holder.directory.attachMovie("calltransfer","calltransfer0",1000);

                 this.nav.container.holder.directory.calltransfer0._x = 180;
                 this.nav.container.holder.directory.calltransfer0._y = 80;

                 if (_global.language == "en") {
                         this.nav.container.holder.directory.calltransfer0.title.text = "Call Transfer";

                         this.nav.container.holder.directory.calltransfer0.qtext.text = "Are you sure that you would like\n
                             to transfer your current call to\n " + dname;
                 }
                 else {
                         this.nav.container.holder.directory.calltransfer0.title.text = "Call Transfer";
```

```
                this.nav.container.holder.directory.calltransfer0.qtext.text = "Are you sure that you would like\n
                    to transfer your current call to\n " + dname;
            }
            this.nav.container.holder.directory.calltransfer0.extension = dnumber;
            this.nav.container.holder.directory.calltransfer0.statusbarroot = statusbarroot;
            }
        if (state==1){
            var statusbarroot = this.nav.mc_statusbar.mc_jason;
            trace(statusbarroot);
            this.nav.container.holder.directory.userSP.spContentHolder.disableself(true);
            this.nav.container.holder.directory.attachMovie("choosedev","choosedev0",1000);

            this.nav.container.holder.directory.choosedev0._x = 180;
            this.nav.container.holder.directory.choosedev0._y = 80;
            trace("testing");
            if (_global.language == "en") {

                this.nav.container.holder.directory.choosedev0.title.text = "Please select the device to use to call "
                    + dname + ")";
                trace("in here");
            }
            else {

                this.nav.container.holder.directory.choosedev0.title.text = "Nceda ukhethe isixhobo osifunayo ("
                    + dname + ")";
            }
            this.nav.container.holder.directory.choosedev0.extension = dnumber;
            this.nav.container.holder.directory.choosedev0.statusbarroot = statusbarroot;
            this.nav.container.holder.directory.choosedev0.phonesSP.contentPath="phoneclips";
        }

    }
    function parkCall(state) {
        if (state==0){
            //Call Transfer
            this.nav.container.holder.directory.attachMovie("callpark","callpark0",1000);

            this.nav.container.holder.directory.callpark0._x = 180;
            this.nav.container.holder.directory.callpark0._y = 80;

            if (_global.language == "en") {
                this.nav.container.holder.directory.callpark0.title.text = "Call Parking";

                this.nav.container.holder.directory.callpark0.qtext.text = "Are you sure that you would like\n to park your
                    current call";
            }
            else {
                this.nav.container.holder.directory.callpark0.title.text = "Call Parking";

                this.nav.container.holder.directory.callpark0.qtext.text = "Are you sure that you would like\n to park your
                    current call";
            }
            }
        if (state==3){
            // Get parked call
        }
    }
```

C.2.2 Graphics and movie clips

This section shows the movie clip created in Flash. This is status button when a call is parked.

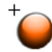


Figure C.5: Status button when a call is parked

C.3 ilangaproxy.py

The `messageRecieved` method processes any message received from the Manager interface. It uses a variable `sendmessage` as a boolean value originally false. This is updated using the methods defined such as `channelFilter`, `mailboxFilter`, `ExtStateFilter`, `isAdminUser`, `parkedCallFilter`, `linkFilter` and `unlinkFilter` which each check the message to see whether it matches their functionality, and if this is the case then they return true, and the `sendmessage` variable then becomes true, which results in the message being sent to the Flash user interface. `parkedCallFilter`, `linkFilter` and `unlinkFilter` are used for call parking and call transfer, and have been created during this project. The manager interface passes a `Link` and `Unlink` event packet when calls are created and destroyed respectively. The contain the channel named necessary for call transfer and call parking so thus they are passed to the Flash interface. `parkedCall` filter manages the `ParkedCall` event packets for updating the status button in the interface.

Listed below are some extracts from the source file:

```
def parkedCallFilter(self,message):
        chan = ""
        parked = 0
        for lin in message:
                k = lin.keys()[0].strip()
                if k == "Event":
                        if lin[k]=="ParkedCall":
                                print "We have a parked call"
                                parked = 1
                if k == "From":      #"Channel":
                        chan = lin[k][0:lin[k].find("-")];
                if k == "Channel":
                        inchan = lin[k]
        if chan == "":
                if parked == 1:
                        fname="/tmp/park%s.dat" % self.username
                        parkfile=open(fname,'r')
                        chan=parkfile.read().strip()
                        parkfile.close()
        if chan != "":
                self.factory.db.query("select * from userdevices where username='%s'" % MySQLdb.escape_string(self.username))
                for result in self.factory.db:
                        if chan.find(result["channel"]) >= 0:
                                self.factory.db.query("select * from userdevices where channel='%s'" % MySQLdb.escape_string(chan))
                                for result in self.factory.db:
                                        print result["username"]
```

```
                                        if result["username"]==self.username:
                                                fname="/tmp/park%s.dat" % self.username
                                                parkfile=open(fname,'w')
                                                parkfile.write(inchan)
                                                parkfile.close()
                                                print "parked %s from %s username %s\n" % (inchan, chan, self.username)
                                return 1

                return 0
        def linkFilter(self,message):
                chan = ""
                temp = 0
                for lin in message:
                        k = lin.keys()[0].strip()
                        if k == "Event":
                                if lin[k]=="Link":
                                        temp = 1
                        if k == "Channel1":
                                chan1=lin[k];
                        if k == "Channel2":
                                chan2=lin[k];
                if temp == 1:
                        self.factory.db.query("select * from userdevices where username='%s'" % MySQLdb.escape_string(self.username))
                        for result in self.factory.db:
                                if chan1.find(result["channel"]) >= 0:
                                        return 1
                                if chan2.find(result["channel"]) >= 0:
                                        return 1

                return 0

        def unlinkFilter(self,message):
                chan = ""
                temp = 0
                for lin in message:
                        k = lin.keys()[0].strip()
                        if k == "Event":
                                 if lin[k]=="Unlink":
                                         temp = 1
                        if k == "Channel1":
                                 chan1=lin[k];
                        if k == "Channel2":
                                 chan2=lin[k];
                if temp == 1:
                        self.factory.db.query("select * from userdevices where username='%s'" % MySQLdb.escape_string(self.username))
                        for result in self.factory.db:
                                 if chan1.find(result["channel"]) >= 0:
                                         return 1
                                 if chan2.find(result["channel"]) >= 0:
                                         return 1

                return 0
        def messageReceived(self,message):
                sendmessage = 0
                sendmessage = sendmessage or self.channelFilter(message) or self.mailboxFilter(message) or self.ExtStateFilter(message) or
                    self.isAdminUser() or self.parkedCallFilter(message) or self.linkFilter(message) or self.unlinkFilter(message)
                if sendmessage:
                        for z in message:
                                k = z.keys()[0].strip()
                                self.sendLine("%s: %s\r\n" % (k, z[k]))
```