DEVELOPMENT AND STANDARDIZATION OF AN EMBEDDED LINUX BASED TRIPLE-PLAY IP SETTOP BOX

A Thesis Submitted to
the Department of Electrical and Electronics Engineering of
İzmir Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in Electrical and Electronics Engineering

by
Haluk ATLI

July 2007
İZMİR

We approve the thesis of **Haluk ATLI**

Date of Signature

……..
Asst. Prof. Dr. Orhan COŞKUN **9 July 2007**
Supervisor
Department of Electrical and Electronics Engineering
İzmir Institute of Technology

……..
Asst. Prof. Dr. Şevket GÜMÜŞTEKİN **9 July 2007**
Department of Electrical and Electronics Engineering
İzmir Institute of Technology

……..
Asst. Prof. Dr. Aylin KANTARCI **9 July 2007**
Department of Computer Engineering
Ege University

……..
Prof. Dr. F. Acar SAVACI **9 July 2007**
Head of Department
İzmir Institute of Technology

……..
Prof. Dr. M. Barış ÖZERDEM
Head of the Graduate School

ACKNOWLEDGMENT

First of all, I would like to express my gratitude to Dr. Orhan Coşkun for his valuable suggestions, support, guidance and assistance in preparation of this thesis.

Besides, I would like to state my appreciation to all my colleagues in Vestel Research and Development for teaching me how to develop innovative products, from the very bottom to top.

I would like to dedicate this thesis as I would dedicate any accomplishment, achievement or success in my life, to my family.

ABSTRACT

DEVELOPMENT AND STANDARDIZATION OF AN EMBEDDED LINUX BASED TRIPLE-PLAY IP SETTOP BOX

With the recent enhancements to the delivery of IP services and of the video codecs such as h.264, transmission of television through IP-based communication systems has been a viable option. An IP settop box (IPSTB) constructs a bridge between a television set and a broadband IP network such as DSL, cable modem, powerline or wireless broadband. IPSTB brings new challenges for the system designers, especially in the areas of inherently organized home networking systems, protocols, and architectures. Future IPSTB products are candidate to converge the information and entertainment technologies. This thesis suggests newly developed device and service discovery methods for the design of an IPSTB software structure that is compatible with the Universal Plug and Play (UPnP) audio video (AV) device descriptions. At the design process, it suggests optimized communication schemes between the servers that are in the control of service providers, and the consumer IPSTBs.

As a newly developed technology, since there is not any standardization for most parts of the overall IPTV system, this thesis takes proven mechanisms as basis and adapts them to the overall design that consists of the hardware drivers, middleware, and the additional programs which helps the middleware to handle the external components of the system connected via USB or serial interfaces. Being an innovative idea, we have used a control system called Virtual Bus Manager so as to communicate between the aforementioned system components. Some system components such as web browser is based on the X Windows architecture, so cross-compiling the X system for the embedded platform has also been a challenge for the feasibility of the final design. Being the second part of the Triple-Play system, Voice over IP application has also been included and based on the compilation of open source software for the corresponding embedded system. Finally, the web browser itself has been based on the popular Gecko web-core that is derived from Firefox.

ÖZET

GÖMÜLÜ LINUX İŞLETİM SİSTEMİ ÜZERİNDE ÜÇ SERVİS DESTEKLİ IP TELEVİZYON KUTUSU GELİŞTİRMESİ VE STANDARTLAŞTIRMASI

Son yıllarda Internet erişim hızlarında görülen artış ve ağ tabanlı hizmetlerin ev kullanıcılarına uygun fiyatlar ile sunulması sayesinde Internet'in kullanımı oldukça yaygın durumdadır. Bu hızlı değişime video sıkıştırma ve gönderme algoritmalarında görülen gelişmeleri de eklersek, ilerleyen dönemlerde ev kullanıcıların canlı yayınları Internet üzerinden izlemesini sağlayan IP televizyon kutularının (IPSTB) kullanımının gözle görülür şekilde artacağı aşikardır. Bu amaç ile kullanılan bir IPSTB, televizyon ile Internet arasında bir köprü kurarak, ağ üzerinden ulaşılabilen hizmetleri bilgisayar ihtiyacı olmadan ev kullanıcılarına sunabilmektedir. Belirtilen cihazın tasarımı, ağ yapısı ve protokollerinin belirlenmesi ile ev ağlarına IPSTB sisteminin eklenmesi konusu birtakım zorluklar barındırmaktadır. Bu tez, bahsi geçen cihazların ev ağlarında standart olarak tercih edilen Universal Plug and Play (UPnP) servis tanımlamalarına uygun yöntemler kullanılarak standatlaştırılması konusundaki çalışmaların bir bütünüdür.

Tez genelinde sistem tasarımı aşamasında servis sağlayıcılar tarafından yönetilen video sunucuları ve ev kullanıcılarının hizmetine sunulan IPSTB'ler arasında verimli bir iletişim yöntemi kullanılması savunulmuş ve verimi arttırmak için kullanılabilecek yöntemler üzerinde durulmuştur. Tezi hazırlarken tasarımın merkezinde bulunan ve cihaz üzerine eklenebilecek tüm özelliklerin sorunsuz bir şekilde çalışmasını sağlayan Uygulama Yönetici (VBM) program geliştirilmiştir. Bu program, yazılım ve donanım birimleri arasında kullanılmıştır. Tez çalışması sırasında cihaz üzerinde Linux tabanlı sistemlerin genelinde bulunan X pencerelemesi kullanılmıştır. Cihaz üzerinde ayrıca Web tarayıcı uygulama olarak Firefox'un da temelini oluşturan Gecko kullanılmıştır.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Internet Protocol Television (IPTV) provides a new and full-of-potential system in which digital services are delivered using the widely accepted IP over a specific and closed network structure via broadband connection (Alfonsi 2005, WEB_1 1981). The close network structure is provided by broadband operators, and although media content is delivered over Internet, it is not free or publicly available to all Internet users. The technology is based on multicasting live channels through Internet, and supplemented with additional protocols such as Session Announcement Protocol (SAP) (WEB_2 2000) which is needed in order to inform the clients about the multicast addresses. Being an alternative for Digital Video Broadcasting (DVB) (ETSI 2003), IPTV is often supplied together with Video on Demand (VOD) and Internet services such as web access and Voice over IP (VoIP) (WEB_3 1999). These services together are called as Triple Play.

IPTV uses standard protocols for media streaming and control. Media servers process live channel content into multicast MPEG/AVC encoded streams which has been sent to the IPSTB with User Datagram Protocol (UDP) (WEB_20 1980). Only the IPSTBs that are joined to the provider-specific multicast group by using Internet Group Management Protocol (IGMP) are served (WEB_10 1989, WEB_11 1997, WEB_12 2002). For the control of on-demand video content, Real-time Streaming Protocol (RTSP) is the choice which effectively utilizes Real time Transporting Protocol (RTP) (WEB_13 1996) at the transport layer (WEB_4 1998). It must be noted that although RTSP over RTP is considered to be the preferred choice for real-time video streaming, none of these protocols are bounded to each other. Although transport protocol choices are common in all IPTV systems, at the phase of device description, discovery, and authentication, operators use different standards that are developed by special third party software companies. For instance, PortalTV™ solution by Kasenna (WEB_5 2006) and iTVManager™ by Minerva Networks are among such middlewares that are proprietary to corresponding companies. The compatibility and standardization of those phases will not only help the expansion of IPTV globally, but also solve many conflicts

between different types of devices and services. This thesis suggests using UPnP standards as reference for the standardization of IPTV devices since UPnP has proved itself in home networks as the top choice in terms of ease of use, compatibility and interoperability between products from different manufacturers. Popular home networks architectures such as Intel Viiv™ utilizes UPnP on their devices, and also organizations like Digital Living Network Alliance (DLNA) (WEB_6 2003) take UPnP as the basis for the home networking guidelines.

For home networks, UPnP offers simple configuration with invisible networking (WEB_7 2000). It provides automatic discovery for numerous device categories with different designs. With UPnP, any device is able to dynamically connect to a network, get an IP address, announce its name, convey its capabilities upon request, and learn about the presence and capabilities of other devices (WEB_8 2002, WEB_9 2002). These steps can be adapted for IPTV devices such as IPSTB and media servers. Besides, UPnP can be broadened to support devices like VOD media servers that are not placed at home, rather on a distant place in service provider's close network structure.

In Chapter 2, we give some insight about the concept of multicasting, especially IGMP that is used in IPTV services (WEB_10 1989, WEB_11 1997, WEB_12 2002). Multicasting is based on forwarding data packets to the right client through the right path. In order to do so, routers need to be compatible with the standard multicasting routing and forwarding protocols.

Chapter 2 also describes the IPTV network architecture. As it can be seen in Figure 1, an IPTV network can be examined in four parts; Content Provider Level is the part where the media is stored, Backbone Network organizes the additional IPTV services that are provided to the end user, Content Distribution Level connects the end user with the backbone network and keeps track of the user subscriptions, and being the main concern of this thesis, the Content Consumption Level is the home network where the end user receives media and services.

Starting Chapter 3, we mention about the problems in the IPTV network structure. Delay and jitter problems, which are the high concern of such a real time media system, have been addressed and aimed to be solved with the common buffer-based designs. Packet re-ordering is another issue that can be solved with transport protocols such as RTP that are specifically developed for transferring real-time media (WEB_13 1996). Beside these, there is another issue called the parsing burden on the application level. Comparing to personal computers, embedded system processors provides lower

processing power so that they spend quite amount of the available resources in order to parse the network packets, distinguish between them, and get the necessary data out of it. Though the parse algorithms claim to be effectively optimized, the parsing itself is a heavy process for an embedded system. In order to solve this difficulty, an alternative packet handling method is suggested which takes the packet before the application and self-parses with an optimizer at the lower level.

UPnP based standardization for the service discovery and description in the IPTV network is also explained in Chapter 3. As it is mentioned before, it is widely known that IPTV standards lack deeply at the device and service description level. UPnP standards has been developed by the UPnP Forum and proved itself in terms of ease-of-use and effectiveness in home networks. UPnP uses Simple Service Discovery Protocol (SSDP) (WEB_14 1999) and standard-based XML descriptions at the device and service discovery level. We propose that UPnP standard can well be adapted for IPTV devices and services.

Besides, Chapter 3 explains the cross-compiling process of the X Windows System which is the required display mechanism for the web-browser. It starts with the brief introduction to the X Windows Systems components and the display mechanism. With display, we mean the input interaction devices such as the mouse and the keyboard, and the screen which consists of so called application windows. During the cross-compilation process, each adjustment has been for the sake of embedded system requirements such as much low processing power compared to the personal computers as well as the limited memory usage.

The thesis continues with Voice over IP. The media format and transfer protocol choices have been given with basic reasons. In fact, for the Voice over IP part of the design, we have gone with open source software called Linphone, and adapted it to our system.

The following parts of Chapter 3 cover our work on the middleware of the system. By middleware, we mean the software with which the users interacts, and which controls and uses the system components such as audio and video decoder. There is an abstraction layer between the system specific low level software and the middleware, which makes the middleware portable for many different embedded systems. We have taken Vestel's well-proven DVB STB software as a basis, and added IPTV functions to this middleware. With IP function, we mean channel streaming over IP and on demand media streaming over IP. We have also worked on the database integration of DVB and

IP channels.

For the control and interaction of different system components, we have developed a Virtual Bus Manager API that is described in detail in the final section of Chapter 3. The embedded platform consists of many peripherals such as USB and serial, so many external components can be integrated to the system. At some points, these components require to interact with the middleware. For the easy and clear handling of such hardware and software parts, the Virtual Bus Manager registers them to its database and all the communication between the components goes though it.

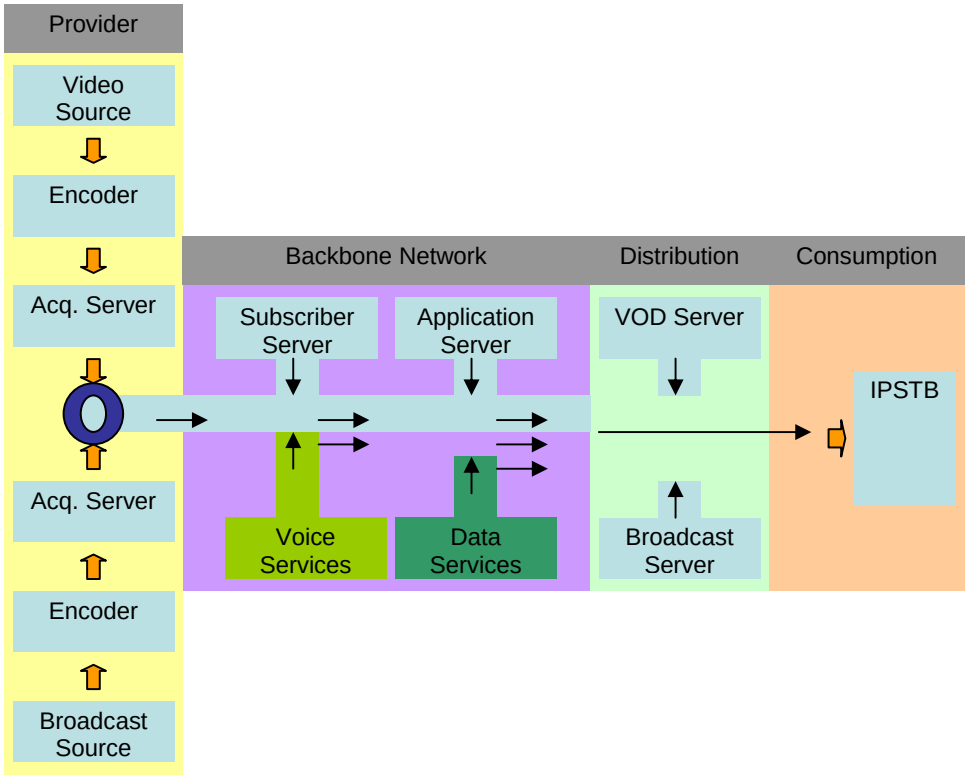

Figure 1. 4-level IPTV Network Architecture

CHAPTER 2

IPTV PROTOCOLS AND NETWORK ARCHITECTURE

2.1. Multicast Network Outline

IPTV based on the model where the media servers that are located at the Headend of the network stores live video content, which is conveyed to any number of subscribed clients. A concurrent transmission of the content from source to destination without any replication at the source-end conserves bandwidth significantly. Multicasting sends the source information to several receivers without adding any extra load on the source or the receivers while using the least required bandwidth (WEB_10 1989). In other words, the bandwidth used to transfer a live video content by the server does not change with the increase in the number of receivers. Multicast packets are replicated in the network by the routers that are capable of multicast features. Multicast-capable routers forward multicasting packets based on the forwarding information built by Routing Protocols, like Multicast OSPF (MOSPF) (WEB_15 1994), Protocol Independent Multicast (PIM) (WEB_16 1998), and Distance Vector Multicast Routing Protocol (DVMRP) (WEB_17 1988).

Other than multicasting, two additional packet transmission methods are widely used in IP networks; broadcasting and unicasting. VOD services use unicasting whereas broadcasting does not find any suitable application in IPTV applications. The reason for this is although broadcasting is an efficient way to transmit one packet to many clients; it is not possible to restrict which clients get these packets. Like multicasting, broadcasted packets are replicated by routers and forwarded to all IP hosts in the local network. Multicasting enables the service providers to define a group of receivers which is a more suitable method considering the fact that IPTV is a subscriber-based system. Besides, broadcasting restricted to use “FF:FF:FF:FF.FF.FF” layer-2 MAC address, and “255.255.255.255” layer-3 IP address. IPTV is a multi-channel system that includes 50-100 live channel content in general. Due to this, restricting the system with only single layer-2 and 3 addresses does not make sense. Another problem with broadcast messages is that their time-to-live (TTL) value is 1, meaning that they cannot pass a router and

only able to be delivered within the Local Area Network (LAN). IPTV services are provided over Internet and require higher TTL values. As a result, broadcasting is an efficient way to transfer packet in the LAN and heavily used by protocols like DHCP and ARP, however, it cannot be used during transmission of live media content.

Unicasting, on the other hand, can be used for on-demand applications but cannot be afforded for delivering live content to all subscribers. For an average IPTV system with 50.000 subscribers, a 1 Mbps standard definition live channel will consume 50 Gigabit-bandwidth if we use unicasting as the delivery method whereas with multicasting, it only consumes 1 Megabit of the available bandwidth. Unlike live channels, on-demand video is not delivered all the time. As the name suggests, it is only delivered on demand, so a separate and single connection between the server and the client makes sense for on-demand applications.

The following sections further explain why multicasting is the appropriate choice for delivering the live content to many clients.

2.1.1. Multicasting Group Theory

A collection of receivers involved in receiving a particular data stream define a multicasting group. The group members neither have any organized network topology nor any environmental boundaries. In other words, each receiver can be located anywhere on the network provided that they are connected via Multicast-capable routers. In order to join a particular multicasting group, IP hosts use Internet Group Management Protocol (IGMP). An IP host should be a member of a particular IP group to receive the data streams.

2.1.2. Forwarding Multicast Packets

In unicast routing, packets are forwarded according to the IP address of the destination. As a result, under secure network conditions, an IP unicast packet is sent through the network along a particular path from the source to the destination. However, in multicast routing, the source sends the content to an arbitrary group of receivers that are represented by a multicast group address. The multicast router should determine which direction is upstream, which is toward the source, and which direction is

downstream, which is away from the source. For the case of multiple downstream paths, the router must replicate the packet and forward the content down to the proper downstream paths. The routers determine the upstream and downstream directions by investigating the forwarding information, which is a part of a multicast group distribution tree. Multicast-capable routers use multicast routing protocols to create multicast distribution trees which control the path that multicasting traffic takes through the network. How precisely a "source to group" multicast distribution tree is build "end to end" is not in the scope of this thesis and is not required to understand the communication between the router and final receiver. Routers use IGMP to periodically question the LAN to find out if known group members are still active. If there are two or more multicasting routers available on the LAN, one of the routers is chosen to take the responsibility of questioning the LAN for group members.

Based on the information that is learned from the IGMP, a router can decide which multicast traffic needs to be forwarded to each of its IP subnets. In other words, IGMP is used to complete building each multicast group distribution tree branches. Only one router which is just located near a LAN that has an IP host joined to the multicast group is responsible to forward multicasting group traffic to the specific LAN.

2.1.3. Multicasting Group Addresses

A unique multicasting destination address range is used to classify an arbitrary listener group of IP hosts. The multicasting address range is only for the destination address of multicasting traffic. Source address for multicast packets is the unicast source address of the stream. The Internet Assigned Numbers Authority (IANA) controls the assignment of multicasting addresses. Class D address space has been assigned to be used for multicasting. Multicasting group addresses are in the range of 224.0.0.0 to 239.255.255.255.

2.2. IGMP Protocol

2.2.1. IGMP Version 1

- Query

Query message used by the local router that support multicasting. The aim is to be informed about which multicast group has still members so that the multicast router continues to send packets. If there is not any response to three successive query messages, the router stops sending data to this specific group.

- Report

IGMP report message is used by the clients in order to join a specific multicast group.

2.2.2. IGMP Version 2

IGMP Version 2 is basically same as Version one with additional messages for the sake of decrease latencies which are causes by delayed Query messages.

- Leave Group

Being a new message, Leave Group is sent by the host clients to the specific multicast group, meaning that the host client will no longer be active in the group and receive data.

- Membership Query

After a leave group message, multicasting router asks to the other members of the group whether they want to continue receiving multicast packets. Some routers use regular Query messages instead of the Membership Query.

2.3. IPTV Network Architecture

The IPTV network can be considered as a four-level hierarchy; Content Provider Level, Service Management & Operations Level, Content Distribution Level, and Content Consumption Level which is the Home Network.

2.3.1. Content Provider Level

The top level of the IPTV network architecture is the video content provider network. It is the source of the digital video content, known also as the IPTV Headend. Live digital video content arrives at the Headend by satellite receivers or via other dedicated data networks and is being processed into multicast MPEG/AVC encoded

streams. The main Headend, which is also called the Super Headend, most of the times stores national channels and also recorded popular on-demand digital videos. For local channels, Regional Headend sites are used in support of redundancy, scalability and performance reasons.

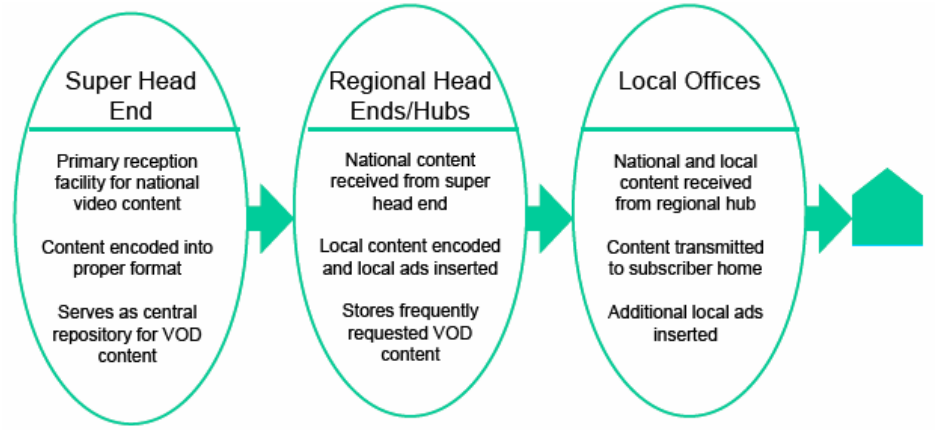

Figure 2. Headend Structure

In Figure 2, we represented this multi-headed structure of Content Provider Level. Besides digital video sources the Headend also include several IPTV and VoD services supporting components:

- A DHCP Server; the DHCP Server is aimed at providing the end client STB with IP addresses based on authorization information found in the DHCP queries and based on service access control authentication and authorization policies.
- A Multicast Router; the Multicast Router functions as an IGMP Queerer.
- IPTV and VoD services management tools; IPTV and VoD services management tools such as subscriber billing and information collections are running on a dedicated host or group of hosts.

2.3.2. Service Management & Operations Level

From the Headend, a group of encoded video streams is transported over the service provider's IP network, which is also known as The Backbone Network, or Service Management & Operations Level. This level interconnects Application and Content Service Providers (ASPs) (e.g. IPTV Service Providers) as well as Network Service Providers (NSP) (e.g. ISPs) on one side to the Content Distribution Level, to

which the subscribers are connected. Each of these parts is unique to the service provider and usually includes equipment from multiple vendors. These networks can be a mix of well-engineered existing IP networks and purpose-built IP networks for video transport. Thus, the IPTV and VoD content provider must provide such dedicated IPTV and VoD VLAN with the exact downstream bandwidth resources needed for the worst case bandwidth consumption case. It must be noted that over subscription is unacceptable for video streaming applications. Additionally, digital streams QoS in terms of constant and minimal delay must be ensured.

2.3.3. Content Distribution Level

Content Distribution Level connects the end-users with the backbone network, which in turn is connected to the service providers. Being at the edge of the provider network, it is perfectly positioned to enforce per-subscriber service level agreement. The connection between the service provider and the subscriber can be accomplished using a variety of technologies. Service providers are commonly using Digital Subscriber Line (DSL) technology to serve individual households and to provide the required bandwidth to run the IPTV services. They also are beginning to use fiber technology like Passive Optical Networking (PON) to reach homes.

In IPTV and VoD service agreements, it is enforced that the client runs the exact number of STB he is paying for and listens only to those channel packages he subscribed.

2.3.4. Content Consumption Level: Home Network

IPTV and VoD services are assumed to be part of a triple play aiming where a single Ethernet port is assumed to provide the Voice (VoIP), IPTV broadcast and VoD services. Common residential networks usually include an unmanaged 100Mbps Ethernet switch. IPSTB is connected to that switch with a home LAN gateway for streaming IPTV and VOIP services. It must be noted that the home network is not required to include QoS enforcement measures or control multicast traffic distribution.

CHAPTER 3

IPTV PROJECT

3.1. IP Network Delay and Jitter

The end-to-end IP network delay varies over time, that is, the amount of time it takes for a single packet to go over the network can be different for any given packet, and this phenomenon is known as network jitter. The network delay is composed of two components; propagation delay and queuing delay. The propagation delay depends on the path the data travels through network capacity and on the size of the data being transmitted. The queuing delay is due to the limitation of bandwidth resources in a given path. As the availability of the bandwidth changes, the data must be delayed until the resource becomes available. The delay mechanism is implemented at gateways and routers.

The IPSTB can use a smoothing buffer to accommodate network jitter. Although buffering is very effective in fixing this problem, it also introduces a start up delay. There is a tradeoff between the maximum amount of jitter that a STB can tolerate, and the startup or channel change time. Over-buffering finally causes high channel change times whereas under-buffering makes the system vulnerable to jitters. By improving the network jitter characteristics, service providers can improve the STB channel change time. Thus, jitter tolerance should be configurable to allow tuning of this parameter to match a given network. Also, if the jitter is very large, more advanced techniques may need to be employed to reduce the amount of buffering and so reduce the impact of jitter compensation (Pourmohammadi-Fallah et al. 2005, Rosado-Sosa et al. 1998).

3.2. Packet Loss/ Reordering

Due to the “best effort” character of UDP/IP, some packets may be lost. It should also be noted that sometimes the order in which they are received by the IPSTB might be changed. Loss and reordering of video packets can produce video artifacts such as complete loss of the corresponding frame or macro blocking. On top of that, audio can

exhibit dropouts, clicks, and pops. The severity of these degradations on video for example, depends on whether the I, B or P frame packets have been impacted. In the worst case, loss of a single packet may affect the entire frame, and in the best case only part of it will be disturbed. As compression rates increase and the use of Advanced Video Coding (AVC) becomes more widespread, loss of data becomes even more critical. Therefore, the system must at all cost prevent data losses.

The IPSTB should be designed to recover from audio or video packet loss with minimal artifacts. In doing so, lip-sync must be maintained at all times. Recovery from stream errors is even more important for audio than video. This is because audio degradations are perceptually more irritating than video degradations. The audio decoder should be able to repeat/insert one or more frames to replace lost audio packets.

Since an IP packet can reach its destination out of order, the IPSTB should be able to perform packet re-ordering. Note that this problem is potentially more severe for UDP, due to the absence of a sequence number in the UDP header. However, the Real-time Transport Protocol (RTP), which sits on top of UDP takes care of this, and can also reduce packet loss by the use of Forward Error Correction (FEC).

3.3. Broadband Access and Home Networking

The IPTV network in the home can be divided into two segments. These segments can be defined as “The Broadband Access Network” and “The Home Network”. These two distinct regions show different requirements with respect to the underlying system. The Access Network contains a broadband connection, such as a DSL line or cable modem, and this provides connectivity to the service provider supplied IPTV services, as well as the wider Internet; whereas the Home Network comprises the consumer equipment forming an IP Local Area Network (LAN) within the home. This includes equipment such as routers, switches, wireless LAN, home plug systems, etc…

An IPSTB may be required to be connected to both the Broadband Access Network and the Home Network. Furthermore, an advanced IPSTB may also be required to act as a source of IP streams and so deliver content to another STB. The range of issues encountered in these distinct networks and usage scenarios, including the differences in the underlying physical network, leads to variety of solutions for the delivery of IPTV services. Certain protocols used in one scenario, may not map well

into another scenario.

3.4. The Transporting Protocol Choice

Many of today's IPTV deployments utilize UDP as the delivery mechanism due to its simplicity and easy implementation. One of the primary reasons for this is because of its inherent support for multicasting. In addition, UDP is a simple mechanism that does not require significant overhead on either the STB client, or the servers. UDP is "connectionless", making suitable for broadcasting. Being an alternative to UDP, RTP is also by some IPTV systems for multicasting live video content. Together with RTSP, RTP is also used for on-demand content transmission.

At the other hand, TCP (WEB_18 1981) is also a somewhat potential alternative to UDP, though it is important to note that TCP cannot easily be used to send/receive multicast streams due to its retransmit functionality. The TCP acknowledgement (ACK) feedback to the transmitter does not fit well and so is quite unsuitable in single-to-many applications. Also, TCP can cause limitless delays to build up in the system due to retransmissions, and this is inappropriate for delivery of real-time services.

TCP/IP is quite often used for streaming content in the Home Network via standards such as UPnP and standard based organizations such as DLNA (using HTTP for example) (WEB_19 1996). Traffic of this type requires the packets to go over all the way up and down the IP stack that is implemented on the Operating System (OS). This requires a redundant and huge amount of CPU processing. This is not a problem for low bit rate content (music and pictures for example), but if you want to stream multiple HD content around the home you are going to be in a world of hurt using HTTP. The buffering and start up delays typical of HTTP streaming media players are another aspect that will not sit well in the living room, where users expect faster response to channel change commands. So, even as HTTP/TCP are not suitable for streaming live content, or in single-to-many streaming scenarios, these protocols work fine for streaming pre-recorded data, and are typically used for content transfer within the digital home.

3.5. Application Level Parsing Burden

Figure 3 describes the data path for a conventional network application. The Ethernet driver is a software module which is responsible to communicate with the Ethernet controller. Once the Ethernet driver sees a new packet is received by the controller, it links that packet into the OS specific data structures and passes it on to the TCP/IP stack for further processing. The packets typically arrive into a 'socket', and in the figure below there are two sockets opened by the application, one for audio/video payload and a second for some other (perhaps unrelated) data. For audio/video payloads, the application does minimal processing and then routes the data to the audio/video/playback driver for decoding and displaying on a TV set.

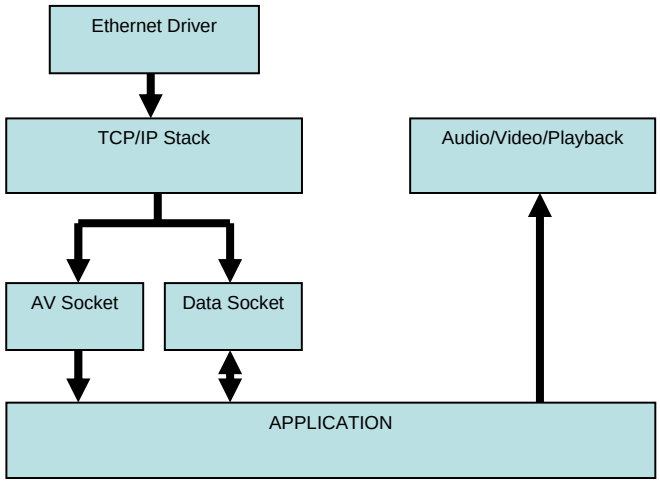

Figure. 3. Conventional Network Data Path

The TCP/IP stack is OS specific and it handles packets received from all network devices. The stack implements basic TCP/IP protocols, such as ARP, TCP, UDP and ICMP as described by the internet standards. Once an IP packet enters the TCP/IP stack it passes through many stages of protocol handling, those stages include:

- checksum of the IP header
- rule based header processing
- protocol specific checksum (UDP or TCP)
- defragmentation
- data accumulation in the socket buffer

At the end, if the IP datagram arrives at the socket buffer, it is copied into the user supplied buffer. After the video payload has arrived into the application, the application

needs to communicate with a driver and send data to the driver for further processing by the audio/video hardware. With that said, it should become clear that it is challenging to complete all these operations in the limited amount of time, which puts a real burden on the host CPU. In order to achieve optimal performance, it's important to reduce the CPU overhead per packet.

One can change the header structures for the sake of decreasing the header portion for each packet; however this also leads to an out-of-standard packet header arrangement. Some unnecessary elements in the header such as double checksum and parts that are allocated for future usage can be eliminated. However, for an embedded Linux system, such a non-standard packet would mean nothing for the transfer protocol (TCP) that is provided with the standard kernel. This solution will require a specific application level receive socket API that is able to understand this specific header structure.

Another solution is to parse the data before it is started to be parsed by the TCP API. This solution is explained at the following part.

3.6. Path Optimization

With the assumption that all audio and video traffic meets certain criteria, such as:

- UDP (or RTP/UDP) protocol would be used to send audio and video payload
- UDP packets occupies single IP datagram (no fragmentation)

And given that such assumptions hold true, which is the case for a standard IPTV communication, the audio and video payload could easily be isolated and sent via a shorter path as follows:

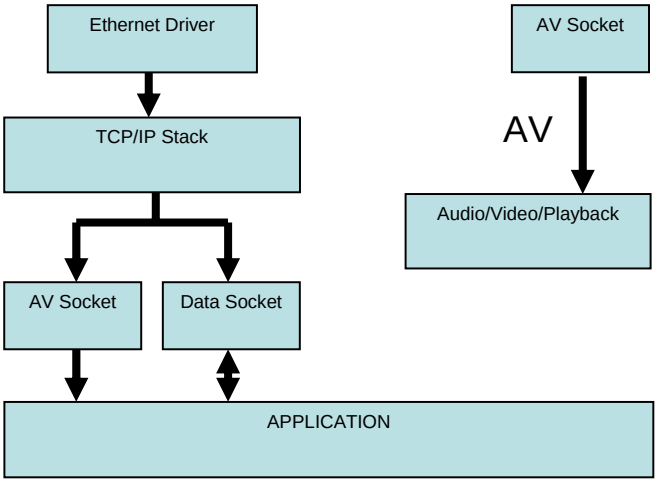

Figure 4. Optimized Network Data Path

Figure 4 shows the optimized network data path. In this optimized data flow scheme, Ethernet packets are captured from the output of the Ethernet driver before they reach the TCP/IP stack. Packet match filters that are installed by application to detect audio/video packets process each received packet. If the packet doesn't match, the filter returns the packet back to the TCP/IP stack for normal processing. If the packet does match, it is sent directly to the audio/video hardware for further decoding.

Ideally, at first, a network packet reaches the Optimizer block. If the packet matches any installed 'rule', it's identified according to the rule and placed into a 'received' queue. If no match is found, then the packet is forwarded to the TCP/IP stack. Playback application works together with the optimizer. It pulls packets from the received queue, and once the packet data is successfully delivered to the playback hardware, then it is moved into the 'pending' queue. When the playback hardware finishes processing the network packet, it then releases the descriptor and associated memory back to the OS buffer pool for following memory allocation steps.

This system is suggested for the sake of acceleration in terms of packet processing. Besides, since there is no dedicated playback buffer and the playback engine takes data directly from the network buffer, this system theoretically improves system performance. At the other hand, there are some issues that need to be taken into account:

- Most of the systems uses a playback buffer in order to properly compensate for the network jitter, thus removing this opportunity might affect the whole design.
- As an alternative, the network buffer should be increased in order to

compensate for network jitter.

3.7. UPnP Based Standardization

One of the main problems with the current IPTV narrowcast and broadcast service is that it lacks standard definitions on top of the streaming protocols. For IPTV services, the system designers are unable to follow a standard file/channel structure in terms of discovery and description. Likewise, enhanced VOD features such as multiple trick modes during on demand services change for different service providers. Due to this, providers come up with different middlewares that are capable of different services but not compatible among each other. While standard based transfer protocols such as UDP and RTP have been widely used by the IPTV solution providers, none of the solutions are fully interoperable and compatible in the best sense. Table 1 below shows the standard and non-standard parts for IPTV services.

Table 1. IPTV standard and non-standard parts

Service	**Service Discovery**	**Service Description**	**Service Handling**	**Media Transfer**
Video on Demand (VOD)	---	---	Real time Streaming Protocol	Real time Streaming Protocol (Unicast)
Channel Broadcast over Internet	---	---	Internet Group Management Protocol	User Datagram Protocol (Multicast)
Voice over IP	---	---	Session Initiation Protocol	Real time Streaming Protocol

Same standardization problems had been existed for home networks before technology-leading companies created a joined organization called Digital Living Network Alliance (DLNA) (DLNA 2006) and came up with design guidelines based on the UPnP AV device descriptions and AV media file types. The UPnP AV architecture

classifies each home networking device with an associated XML description file in which device specific services are explained. Each device provides its XML description from a specific URL, and sends the URL address to other UPnP-enabled home network devices during the device discovery process automatically. UPnP provides that the device discovery and description requests/replies are done automatically between themselves. Figure 5 shows a basic instance of standard UPnP AV 1.0 “GetMediaInfo” service description. In this service definition, the only input variable is InstanceID which is a unique number for each file on the server. Rather than the direct URL, UPnP uses this unique number to obtain the information about the corresponding file.

```
<?xml version="1.0"?>
<scpd xmlns="urn:schemas-upnp-org:service-1-0">
.
.
<action>
<name>GetMediaInfo</name>
<argumentList> <argument> <name>InstanceID</name>
<direction>in</direction>
<relatedStateVariable>A_ARG_TYPE_InstanceID</relatedStateVariable>
</argument> <argument> <name>MediaDuration</name>
<direction>out</direction> <relatedStateVariable>CurrentMediaDuration</relatedStateVariable>
</argument> <argument> <name>CurrentURI</name>
<direction>out</direction> <relatedStateVariable>AVTransportURI</relatedStateVariable>
</argument> <argument> <name>CurrentURIMetaData</name>
<direction>out</direction>
<relatedStateVariable>AVTransportURIMetaData</relatedStateVariable>
</argument> <argument> </argumentList>
</action>
```

Figure 5. UPnP AV 1.0 “GetMediaInfo” service description

Table 2. Home Network Solutions to IPTV non-standard parts

Service	Service Discovery	Service Description	Service Handling	Media Transfer
Video on Demand (VOD)	Simple Service Discovery Protocol	XML based standard descriptions via HTTP	Real time Streaming Protocol	Real time Streaming Protocol (Unicast)
Channel Broadcast over Internet	Simple Service Discovery Protocol	XML based standard descriptions via HTTP	Internet Group Management Protocol	User Datagram Protocol (Multicast)
Voice over IP	Simple Service Discovery Protocol	XML based standard descriptions via HTTP	Session Initiation Protocol	Real time Streaming Protocol

For this basic XML, the server output is the URI of the specific file, MetaData, and media duration information. The structure of the XML file is quite flexible and allows additional arguments. Table 2 shows our suggested protocols that are used in UPnP and can easily be adapted for non-standardized IPTV services. With such standardization, devices from different manufactures and services from different providers become compatible and interoperable. We present three examples which demonstrate standardization of IPTV specific services while using UPnP architecture as template. This first example explains how we can retrieve information about on-demand video content that comes from the service provider. The second example give details about constructing the live channel list on IPSTB with simple UPnP steps comparing to time-consuming process that is required for DVB. Our final example describes how to transfer Electronic Programming Guide (EPG) data between the service provider and IPSTB by using UPnP.

```
<?xml version="1.0"?>
<scpd xmlns="urn:schemas-upnp-org:service-1-0">
<action> <name>GetVideoInfo</name>
<argumentList> <argument> <name>InstanceID</name>
<direction>in</direction> <relatedStateVariable>CurrentInstanceID</relatedStateVariable>
</argument> <argument> <name>MediaDuration</name>
<direction>out</direction> <relatedStateVariable>CurrentMediaDuration</relatedStateVariable>
</argument> <argument> <name>MediaType</name>
<direction>out</direction> <relatedStateVariable>CurrentMediaType</relatedStateVariable>
</argument> <argument> <name>MediaLength</name>
<direction>out</direction> <relatedStateVariable>CurrentMediaLength</relatedStateVariable>
</argument> <argument> <name>MediaGenre</name>
<direction>out</direction> <relatedStateVariable>CurrentMediaGenre</relatedStateVariable>
</argument> <argument> <name>MediaProvider</name>
<direction>out</direction> <relatedStateVariable>CurrentMediaProvider</relatedStateVariable>
</argument> <argument> <name>MediaTrailerURL</name>
<direction>out</direction> <relatedStateVariable>CurrentMediaTrailerURI</relatedStateVariable>
</argument> <argument> <name>MediaPhotoURL</name>
<direction>out</direction> <relatedStateVariable>CurrentMediaPhotoURI</relatedStateVariable>
</argument><argument><name>MediaPrice</name> <direction>out</direction>
<relatedStateVariable>CurrentMediaPrice</relatedStateVariable>
</argument> <argument> </argumentList> </action>
```

Figure 6. Video on Demand services description in XML format

The XML structure in Figure 6 shows an example service specifically constructed for Video on Demand application. The service provides on-demand media content information between the IPSTB and service provider. This service yet can easily be enhanced for transmission of live channel information that is used in IPTV applications. In UPnP AV structure, services definitions are described as actions. Each service starts with <action> syntax and ends with </action>. Service name is defined as "GetVideoInfo" in Figure 6. Service parameters are labeled in an argument list that starts with <argumentList> syntax. Services would possibly have more than one parameter with specific names and use. The direction value of the service parameter shows whether it is an input or an output value for the specific service. The usage of GetVideoInfo service is as follows; IPSTB asks for information about a specific on-demand video file. Service provider responses with the URL address that contains the XML file with the requested information for the video content. IPSTB then gets the information from the provided URL. This is depicted in Figure 7.

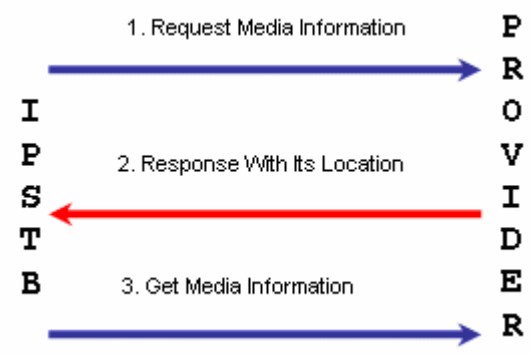

Figure 7. UPnP way of getting media information

Our second example shows how to construct the channel list on IPSTB simply by using standard UPnP steps. Similar to the channel scan operation on standard DVB based STBs, an IPSTB needs channel list information so as to display the content to the user. DVB based STB searches through specific frequencies that correspond to channels' transponder information. The search process depends on the satellite, and generally takes 10-15 minutes of time. Channel transponder values differ for each satellite and change frequently. Hence, the users need to make channel scan operation regularly if they want to watch the newly added channels. Users also need to do channel scan to update the channels that changed their frequency and symbol rate. Users need to add new transponder values to the transponder list manually. For an average user, this process is not only time consuming but also hard to complete so that a professional assistance is definitely needed. On the other hand, an IPSTB can efficiently construct the channel list with a simple step by step cooperation with the service provider. Similar to the example GetVideoInfo service depicted in Figure 6, a well-defined XML file is enough to store the up-to-date channel list on a specific URL so that the STB checks the list on each start-up automatically, and adds the new channels to the list if they are available. The URL for the up-to-date channel list can be either statically present on IPSTB or retrieved by requesting from the service provider. The steps for retrieving the XML file will be same with the steps for GetVideoInfo service.

The Electronic Program Guide (EPG) has received more interest from the end user recently. In simple term, EPG is the list of daily - or weekly - programs that is broadcasted via DVB by the channel providers. Not all channels has adapted their system for EPG, however it is becoming more common day by day. EPG has a specific program id (PID), and broadcasted with specific time intervals such as 30 seconds. The

reason for that is the lack of two-way communication of DVB. For a DVB system, STB is always in idle receive state and unable to communicate with the channel provider. Likewise, the provider is unable to distinguish between the recipients since it uses open area broadcasting. Due to the lack of communication, channel provider needs to broadcast EPG data periodically whether there is a demand from the STB or not. On the other side, during EPG receiving period, the receiver needs to run a separate thread which checks incoming data whether it is EPG data or not. This system is certainly not efficient and cannot be afforded on IPTV communication due to the bandwidth and system requirements.

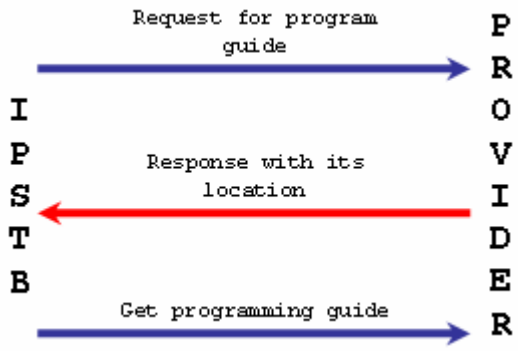

Figure 8. UPnP way of EPG transfer

IPTV is not bounded with communication restrictions that are seen in DVB; each IPSTB can communicate with the service provider and get the program guide upon a request. An efficient and easy process is the use of XML based standard description files which are located on specific destinations. An UPnP-way of getting the program guide is depicted in Figure 8. Similar to the GetVideoInfo service defined previously, a three-step data retrieval process is sufficient for EPG information. IPSTB starts the process by requesting the EPG data for specific channel. Service provider responses with the URL that the EPG data for requested channel is stored. IPSTB then simply retrieves the data from that location.

3.8. Cross Compiling X Window System for IPSTB Platform

3.8.1. A Brief History of X Window System

X Window System (also abbreviated as X or X-Windows) was developed to provide a distributed mechanism to run applications (referred to as clients) across a network on a variety of different hardware platforms. Providing highly portable code is the focus, so that the system could be implemented on standard PCs, other stand alone embedded devices with their own OS, and workstations as well. The system can run under several operating environments and provide high performance. Besides, in keeping with open systems concepts, X is free from user interface policy. The ability of X to run on and across different platforms makes it perhaps one of the most innovative technologies to be developed in computing in recent years. And for this reason, major vendors (IBM, AT&T, DEC, HP, and Sun) have embraced X as their display standard for advanced workstations.

3.8.2. The X Architecture

3.8.2.1. Client and Server Structure

The X architecture is based on a client-server relationship. The server (called the X server) is responsible for the display. A display is considered an X station (a computer that runs the X server) with one or more monitor, keyboard, and mouse. A single instance of an X server is called a display; therefore, a display device may have multiple displays by running multiple X servers.

X is a network windowing system that allows a client to run on either the server host or any other host on the network. The server sends and receives display information (i.e., screen-writing commands, keyboard hits, or mouse movements). This capability implies that a single display station could have numerous clients from all over the network sending messages to it.

3.8.2.2. X Server

The server is responsible for the below tasks:

- Providing multi-client access.
- Receiving and understanding client messages.
- Sending user responses in the form of messages to the clients.
- Performing all drawing.
- Storing and maintaining data structures for limited resources (color maps, cursors, fonts, and graphic information known as “graphics context”).

As it is mentioned before, the X server can handle client requests over a network. This provides for the distributed nature of network computing. It is feasible to connect several processor classes in the network to handle specific applications. What this really means is that additional computing power can be added incrementally, allowing for a more controlled increase in computing power. When clients make requests to the server, the server must be able to decipher what is being requested.

One of the primary tasks that the X server provides is the control of input and output functions. Whenever a key is pressed, the server must determine which client should be informed of it, if any. Similarly, when the mouse is moved the server must inform one or more clients.

As a way of improving performance, the X server has the responsibility of performing all drawing that the client’s request. So, when a client intends to draw a line or a character of text, it makes a request to the server, and the server performs all the necessary steps for generating the graphics. This greatly improves performance since the client need not be concerned about the “how” of drawing. Additionally, the X server maintains the resources required for generating the graphics. Fonts, colors, window coordinates, and other graphical pieces of information are the kinds of resources stored by the server. These data items are given identification codes by the server so that clients may refer to them. This makes more sense than sending an entire graphics structure with large amounts of information over a network. By doing it this way, X reduces the network traffic and leaves bandwidth for other messages.

3.8.2.3. X's Window System

X is built using a hierarchical window system as a window tree. There will always be one root window. The root window always covers the entire display area, and all other windows are its descendants. Additionally, windows on the same level in the tree are called siblings while the parent of a child is called an ancestor to the child.

An X window can be viewed as a "round-tangle" - usually, but not quite always, a rectangle. The upper-left corner of an X window is considered to be its origin. The x-axis increases as you go left while the y-axis increases as you go down. All coordinates in X are relative, that is, from the window's point of view all things start at 0,0. If a window is created and is offset from the root window at 100,100 the root window's perspective of the child's origin is 100,100 while the child views it as 0,0.

The X server is where all windows live. Whenever a window is created, the application must ask the X server to do it. The server adds the newly created window to the window tree when the request is made. Once added, the client may request to be mapped and hence viewable by the user.

If the ancestor isn't there, the child may be mapped, yet won't be visible. Only when all of its ancestors are mapped will the child have the chance to be visible. When a larger window overshadows another, it makes sense that the user won't see the smaller one. This can be corrected by placing the smaller one higher in the stacking order. The stacking order is simply the way X views its windows. In this way, the little one gets recognized as being on top of the larger one, therefore obscuring a part of the larger. It can be viewed as a deck of cards. If one wants the card on the bottom of the deck, he/she would take it and move it to the top. The card that was on the top is now obscured (probably) by the one just placed on top. If a child lies outside of its parents' boundaries, it will not be seen. As soon as the child comes back to the parents' boundaries, it will be viewable provided neither of the other two situations listed occurs.

3.8.2.4. X Has Its Own Protocol

X has its own protocol which is nothing more than an agreed upon form of communication. In the X world, this is accomplished via network packets (chunks of information). There are four kinds of packets that exist in X: events, replies, requests,

and errors. Events are things like a keyboard press, a mouse movement, or a screen update. Just like in the real world, events can happen at any time. Clients must be ready and able to cope with this. The client will never be able to know for sure if it has to respond to a keystroke or to repaint itself. In order to accommodate this, the client must inform the server of the events it cares about, and then provide processing for when the event is reported.

Requests are generated by a library function (Xlib) and sent to the server. A request packet might contain drawing information, or perhaps a question about a specific window. When the request is for locally stored information, it quite obviously does not have to go through the server. Otherwise, the request is passed to the server where it is acted on and a reply is sent. Replies come from the server to Xlib only when a previous Xlib request was for server based information. This situation is considered a round-trip, and as might be inferred, is quite costly. After all, the client pays the price for the request and must also pay for the reply.

Since X is network based, and it is more efficient to send chunks of data across a network, the designers chose to have the server queue up events and errors while the client queues up requests. The server gives the server queue to the client when the client initiates the transaction. At that time the client gives the server its queue.

3.8.3 Why is Cross-Compiling X Needed?

Simply, X is the general dependant requirement for most of the popular programs that are able to run on a host Linux PC. If one need to take the full advantage of running a Linux operating system on an embedded device, the first and the foremost job is for sure to cross-compile X for that corresponding platform. Unfortunately, although the open source suppliers and developers of X precisely states that it should be able to run on variety of systems including popular ARM and MIPS based platforms, its binary and rpm releases are only for i386 based PCs, meaning that it would not run on any MIPS board, or on any other alternative solutions.

3.8.4 How to Cross-Compile X?

The term “Cross-Compile” basically means to compile a program with its dependencies (objects, other supporting binaries, etc…) so that it will not run on the system that the compilation process takes places, but rather on a different system with different characteristics (processor, OS, etc…). Specifically, our aim is to cross-compile the famous X Windows System so that it will run as a daemon process on a third party embedded system with a MIPS processor.

We have been supplied with the cross-compile tool chain (mipsel-linux-gnu tool chain) by the platform providers. The latest version of X provided by the XFree86 organization is 4.6.0 which can be download as 7 separate .tar.gz balls from its web site. Normally, during the compilation process of any Linux based program, the first step is to create the Makefile by using the configuration file and setting the design parameters according to your design goals. However, for XFree distribution of X source code, the mechanism differs from the standard. For the sake of configuring the compilation parameters, some additional definition files have been provided. Those are;

- xf86site.def, which covers all possible configuration parameters as C comments.
- host.def, that is empty at first, but created by selecting the appropriate definitions from xf86site.def and defining either YES or NO.
- cross.def, which is not required unless the process is aimed to be crossed. Cross.def needs to be included in host.def, and defines cross-compile specific parameters.

After carefully examining all parts of xf86site.def, we have generated the host definitions accordingly (the comments are used so as to give some insight about each corresponding definition).

Host.def:

/

It is much preferred if the libraries are generated as shared objects rather than the static .a files. The aim is to copy the shared object files under the common /usr/lib once they are created and use from that location. DefiningSharedLibX11 as NO makes the libraries static.

**/*

#define SharedLibX11 YES

/

Since the embedded system has rather strict memory requirements, we have no space and no need for documentation.
**/*
#define BuildDocs NO

*/**
For our final compilation, we am planning to define BuildServersOnly as YES due to again memory requirements. However, during development process, we need to use some example programs that are automatically compiled with X, and uses X.
**/*
#define BuildServersOnly NO

*/**
The embedded system we am working on has no additional video card available, which is most of the time the case for that kind of system. Therefore, we need to use the frame buffer as the display device. The below four definitions are required for using X with frame buffer (resultantly creates Xfbdev binary).
**/*
#define KDriveXServer YES
#define KdriveServerExtraDefines -DMAXSCREENS=3
#define TinyXServer YES
#define XfbdevServer YES

#define ProjectRoot /usr/X11R6
#define UseSeparateConfDir NO
#include <cross.def>

As it can be seen at the very last line of host definitions, being a cross-compile process, we have also used additional cross.def file in which target specific compilation definition are included:

Cross.def:
*/**
This file contains redefinitions of some symbols to enable cross compilation: e.g. paths for include files and paths to compiler images. It will have to be edited to reflect these given one's local configuration.
**/*
*/**
Although it is not the default platform choice, X supports MIPS based architectures, it is defined below.
**/*
#undef i386Architecture
#define Mips32Architecture
#undef OptimizedCDebugFlags
#define OptimizedCDebugFlags -O2
#define ServerCDebugFlags -O2
*/**

The standard definition needs to be MIPS compatible

**/*

#undef StandardDefines

*#define StandardDefines -Dlinux -D__mips__ -D_POSIX_SOURCE *

-D_BSD_SOURCE -D_GNU_SOURCE -DX_LOCALE

*/**

The third party MIPS cross compiler binaries, libraries and headers are under the /opt/nxlinux/gcc/gcc-3.3.4-glibc-2.3.2/mipsel-linux-gnu/ directory

**/*

#define StdIncDir /opt/nxlinux/gcc/gcc-3.3.4-glibc-2.3.2/mipsel-linux-gnu/include

#define PreIncDir

#undef PostIncDir

#define PostIncDir /opt/nxlinux/gcc/gcc-3.3.4-glibc-2.3.2/mipsel-linux-gnu/include

*/**

C Compiler needs to be defined mipsel-linux-gnu-gcc

**/*

#undef CcCmd

#define CcCmd mipsel-linux-gnu-gcc -mhard-float

*/**

C++ Compiler needs to be defined mipsel-linux-gnu-g++

**/*

#undef CplusplusCmd

#define HasCplusplus YES

#define CplusplusCmd mipsel-linux-gnu-g++

#define DoRanlibCmd YES

#define RanlibCmd mipsel-linux-gnu-ranlib

#undef ExtraLoadFlags

#define ExtraLoadFlags

#define FbNoPixelAddrCode

#undef TermcapLibrary

#define TermcapLibrary -ltermcap

#undef LdPostLib

#define LdPostLib -L/opt/nxlinux/gcc/gcc-3.3.4-glibc-2.3.2/mipsel-linux-gnu/lib

#undef ExtensionOSDefines

#define ExtensionOSDefines

*#define ServerXdmcpDefines /**/*

#define HostCcCmd cc

#include <cross.rules>

3.9. Voice over Internet Protocol (VOIP)

3.9.1. H.323

ITU-T specifies H.323 as to transmit audio, video, and data through the widely used IP network. The specification also includes Internet itself. It contains not only the choice of protocols, but also specific components:

- Transport Protocol: RTP/RTCP
- Call Signaling and Media Control (System Control): H.225, H.245
- Data: T.120
- Audio: G.711, G.722, G.723, G.728, G.729
- Video: H.261, H.263

The H.323 protocol suite is based on several protocols. The protocol family supports call admissions, setup, status, teardown, media streams, and messages in H.323 systems. These protocols are supported by both reliable and unreliable packet delivery mechanisms over data networks. The multipoint controller (MC) supports conferences between three or more endpoints in a multipoint conference. MCs transmit the capability set to each endpoint in the multipoint conference and can revise capabilities during the conference. The MC function can be resident in a terminal, gateway or gatekeeper.

In addition, there is a H.323 proxy server which is a proxy specifically designed for the H.323 protocol. The proxy operates at the application layer and can examine packets between two communicating applications. Proxies can determine the destination of a call and perform the connection if desired. Although most H.323 implementations today utilize TCP as the transport mechanism for signaling, H.323 version 2 does enable basic UDP transport. Also, other standards bodies are investigating the use of other reliable UDP mechanisms to create more scalable signaling methods.

3.9.1.1. RAS

RAS signaling provides pre-call control in H.323 networks where gatekeepers and a zone exist. The RAS channel is established between endpoints and gatekeepers across

an IP network. The RAS channel is opened before any other channels are established and is independent of the call control signaling and media transport channels. This unreliable UDP connection carries the RAS messages that perform registration, admissions, bandwidth changes, status, and disengage procedures.

3.9.1.2. Gatekeeper

Auto discovery enables an endpoint, which might not know its gatekeeper, to discover its gatekeeper through a multicast message. Because endpoints do not have to be statically configured or reconfigured for gatekeepers, this method has less administrative overhead. The gatekeeper discovery multicast address is 224.0.1.41, the gatekeeper UDP discovery port is 1718, and the gatekeeper UDP registration and status port is 1719. The following three RAS messages are used for H.323 gatekeeper auto discovery:

- Gatekeeper Request (GRQ): A multicast message sent by an endpoint looking for the gatekeeper.
- Gatekeeper Confirm (GCF): The reply to an endpoint GRQ indicating the transport address of the gatekeeper's RAS channel.
- Gatekeeper Reject (GRJ): This advises the endpoint that the gatekeeper does not want to accept its registration. This is usually due to a configuration on the gateway or gatekeeper.

3.9.1.3. Media Control/ Transport (H.245/RTP/RTCP)

H.245 handles end-to-end control messages between H.323 entities. H.245 procedures establish logical channels for transmission of audio, video, data, and control channel information. An endpoint establishes one H.245 channel for each call with the participating endpoint. The reliable control channel is created over IP using the dynamically assigned TCP port in the final call signaling message. The exchange of capabilities, the opening and closing of logical channels, preference modes, and message control take place over this control channel. H.245 control also enables separate transmit and receive capability exchange as well as function negotiation, such as determining which codec to use.

The following procedures and messages can be used to enable H.245 control operation:

- Capability Exchange: Consists of messages that securely exchange the capabilities between two endpoints, also referred to as terminals. These messages indicate the terminal's transmit and receive capabilities for audio, video, and data to the participating terminal. For audio, capability exchange includes speech transcoding codecs such as G-series G.729 at 8 kbps, G.728 at 16 kbps, G.711 at 64 kbps, G.723 at 5.3 or 6.3 kbps, or G.722 at 48, 56, and 64 kbps. It also includes International Organization for Standardization (ISO) series IS.11172-3 with 32-, 44.1-, and 48 kHz sampling rates, and IS.13818-3 with 16-, 22.05-, 24-, 32-, 44.1-, and 48 kHz sampling rates; and GSM full-rate, half-rate, and enhanced full-rate speech audio codecs.
- Master-Slave Termination - Procedures used to determine which endpoint is master and which endpoint is slave for a particular call. The relationship is maintained for the duration of the call and is used to resolve conflicts between endpoints. Master-slave rules are used when both endpoints request similar actions at the same time.
- Round-Trip Delay - Procedures used to determine delay between the originating and terminating endpoints.
- Logical Channel Signaling - Opens and closes the logical channel that carries audio, video, and data information. The channel is set up before the actual transmission to ensure that the terminals are ready and capable of receiving and decoding information. The same signaling messages establish both uni-directional and bidirectional channels. After logical channel signaling is successfully established, the UDP port for the RTP media channel is passed from the terminating to the originating endpoint. Also, when using the Gatekeeper Call Routed model, this is the point at which the gatekeeper can divert the RTP streams by providing the actual UDP/IP address of the terminating endpoint.

3.9.1.4. Termination of a Call

Either endpoint participating in a call can initiate call termination procedures. First, the endpoint must cease media transmissions (such as audio, video, or data) and close all logical channels. Next, it must end the H.245 session and send a release complete message on the call signaling channel, if it's still open or active. At this point, if no gatekeeper is present, the call is terminated. When a gatekeeper is present, the following messages are used on the RAS channel to complete call termination:

- Disengage Request (DRQ): Sent by an endpoint or gatekeeper to terminate a call
- Disengage Confirm (DCF): Sent by an endpoint or gatekeeper confirming disconnection of the call
- Disengage Reject (DRJ): Sent by the endpoint or gatekeeper rejecting call disconnection

3.9.1.5. Media Transport (RTP/RTCP)

RTP provides media transport in H.323. More specifically, RTP enables real-time, end-to-end delivery of interactive audio, video, and data over unicast or multicast networks. Packetization and transmission services include payload identification, sequencing, timestamping, and monitoring. RTP relies on other mechanisms and lower layers to ensure on-time delivery, resource reservation, reliability, and QoS. RTCP monitors data delivery as well as controls and identifies services. The media channel is created using UDP, where RTP streams operate on an even port number and the corresponding RTCP stream operates on the next-higher (odd) port number.

3.9.2. Embedding VoIP

An IP phone is a device that is connected to the PC with a common interface (USB interface is dominant) and allocated as system's audio input and output during the case of a call. Additional soft phone software is provided with the phone, and it is used to connect the popular communication programs (such as Skype, MSN Messenger, Yahoo! Messenger, etc...) with the IP phone so that any call would be directed to it. The

target embedded system has two USB 2.0 interface, and an additional USB 1.1 interface, thus it would be a wise decision to use any common IP Phone with USB connector.

In order to make the system understand that an IP Phone is connected to the system, ALSA network drivers need to be implemented during the kernel compilation process. If ALSA is fully compiled, the system will be automatically understand an audio input-output device is connected to itself, and reserve the newly connected device as the system audio device - that is actually what must be the case for a standard plug-and-play device. After the successful compilation of ALSA drivers, target system recognized the IP phone. That has been the phase on which the VoIP hardware had almost been ready.

Standard VoIP software is capable starting/accepting a session with the usage of Session Initiation Protocol (SIP) and sending/receiving voice over IP with Real Time Streaming Protocol (RTSP). With these standards in mind, the highly popular Linphone open-source VoIP software have been chosen as the basis; optimized and added some enhancements onto it for the target system. Linphone gives the basic support for five types of codec choices; among with GSM standard has been preferred.

Thanks to the low dependency demands of it, the cross compilation process of Linphone has been nothing more than choosing the host and target as mipsel base compilers. The resultant binary works with the ALSA drivers, thus the most important requirement for the embedded system is this appropriate sound driver. The system is implemented so as to work with several VoIP proxies and public SIP servers over the Internet such as;

- http://www.antisip.com/
- http://iptel.org/
- http://www.freeworlddialup.com/
- http://www.le5.com/
- http://www.realtunnel.com/

3.10. Working on the Middleware

As its name suggests, middleware is in the middle of the design. It provides interaction between the user and the device via its user interface. Doing so, it uses low

level APIs and drivers by integration them into the abstraction layer, and calling with its own functions. Only with this abstraction, it is possible to uses the same middleware on different platforms and systems as long as the sources are compatible.

The middleware that is dealt on the IPSTB design is stable Satellite Digital Video Broadcast (DVB-S) receiver software with no IP features at all. It consists of general parts such as database, event and listener components, basic graphic components, and DVB-S transport stream parser parts. Without interfering with the stability of it, the middleware is targeted to be optimized and enhanced with IP STB features. These include;

- A setup interface for IP network connection
- Integration of IP multicast display
- Integration of Video on Demand display
- Database integration for IP channels

3.10.1. Setup Interface for IP Network Connection

A standard DVB-S (ETSI 2003) receiver has two ways of finding new channels and adding them to the channel list; the automatic search that goes through a previously recorded transponder values and gradually find the channels, and the manual search that requires the user to enter the transponder information for a specific channel and searches with those values. In IP world, channels can be added by parsing a specific .xml file as it is discussed in UPnP section, or the same manual search methods can be applied as well. Below functions are written with this intention:

```
void AssignIPSearchParameters(*ipstreaminfoptr)
{
        unsigned char ip1,ip2,ip3,ip4;
        char temp[4];
        char *start, *end;
        U32 temp_address = 0;
        U16port=VEL_GetInt(IPPORTEDITBOX);
        char* ipAddr=VEL_GetText(IPADDRESSEDITBOX);
        char* providerName=VEL_GetText(IPPROVIDERNAMEBUTTON);

        start = ipAddr; end = strchr(start, '.');
```

```
        strncpy(temp, start, end-start);
        temp[end-start] = 0;
  ip1=atoi(temp);
        start = end+1; end = strchr(start, '.');
        strncpy(temp, start, end-start);
        temp[end-start] = 0;
  ip2=atoi(temp);
        start = end+1; end = strchr(start, '.');
        strncpy(temp, start, end-start);
        temp[end-start] = 0;
  ip3=atoi(temp);
        start = end+1;
        strcpy(temp, start);
        ip4=atoi(temp);
        temp_address+=ip1;
   temp_address <<=8;
        temp_address+=ip2;
   temp_address <<=8;
        temp_address+=ip3;
   temp_address <<=8;
        temp_address+=ip4;
        ipstreaminfoptr->IP_address = temp_address;
        ipstreaminfoptr->IP_port = port;
}

void StartIPManualSearch(*Comp, Data)
{
        U32 port = VEL_GetInt(IPPORTEDITBOX);
        char* ipAddr = VEL_GetText(IPADDRESSEDITBOX);
        char* providerName = VEL_GetText(IPPROVIDERNAMEBUTTON);
        printf("ipAddr =
%s\npPort =
%d\nproviderName =
%s\n",
ipAddr, port, providerName);
        tuneIPToAddress(ipAddr,port);
        VDB_SetQuery();
        VDB_FindFirst();
        if(addNewChannelsEvent == FALSE)
                addNewChannelsEvent = VEL_CreateEvent();
```

```
        if(stopSearchEvent==FALSE)
                stopSearchEvent = VEL_CreateEvent();
        VEL_CreateTrigger(stopSearchEvent, ExitScanMenu, FALSE);
        searchTypeFlag = IPSEARCHFLAG;
}
```

3.10.2. Integration of IP Multicast Display

The hybrid design is targeted for ease of use so that the user will simply be able to switch between IP channels and DVB-S channels. At the low level, the system uses the same decoding procedure for DVB-S and IP streams. The main difference is the source of the stream. The distinction has been done with a global state variable that distinguishes between IP and tuner sources. All tuner functions have been deactivated during IP streaming and vice versa. This has been proven to be necessary since the system is unable to decode two streams simultaneously; it only supports one.

The distinctions have been done by implementing separate audio and video modes and different functions at the abstraction layer:

```
void managerFunction(vid_req, aud_req)
{
   if((vid_req == NONE) && (aud_req == NONE))
   { /*Nothing to do*/ }
   else if((vid_req == VIDEO_START) && (aud_req == AUDIO_START))
   {  /*Start decode for both*/
      av_decodeStart(); }
   else if((vid_req == VIDEO_STOP) && (aud_req == AUDIO_STOP))
   {  /*Stop decode for both*/
  av_decodeStop(); }
   else if((vid_req == IP_AV_STOP))
   {  /* Stop ipstream for both */
                av_decodeStop_IP(); }
   else if((vid_req == IP_AV_START))
   {  /* Stop ipstream for both */
                av_decodeStart_IP(); }
   else
   { printf("***** NO ACTION IS DEFINED *****\n"); }
}
```

```
void av_decodeStart_IP(void){
        bresult result = berr_external_error;
        bplayback_socket_params socket_params;
        bplayback_file_t socket;
        bplayback_params playback_params;
        bplaypump_open_params playpump_open_params;
        bband_t playback_parser_band;
        SetPlaybackHandle(bplayback_ip_open());
        bplaypump_get_open_params(B_ID(0), &playpump_open_params);
        playpump_open_params.buffer_size *= 2;
        playpump_open_params.num_descriptors *= 2;
        SetPlaypumpHandle(bplaypump_open(B_ID(0), &playpump_open_params));
        bplayback_socket_params_init(&socket_params, GetPlaybackHandle());
        socket = bplayback_socket_open(GetPlaybackHandle(), &socket_params);
        bplayback_params_init(&playback_params, GetPlaybackHandle());
        playback_params.index_format = bindex_format_none;
        stream = bplayback_start(GetPlaybackHandle(),
GetPlaypumpHandle(), &mpeg, socket, &playback_params);
        if(stream){         result
= bdecode_start(GetDecodeHandle(), stream, GetDecodeWindowHandle());
        }
}
```

3.10.3. Database Integration for IP Channels

As it is stated before, the base middleware designed for satellite receivers; this is also valid for the database construction. It was possible to add the IP channels to the already defined database as themselves, but in order not to distract the stability of the code, database entries of the IP streaming has been separated. Excluding the features that might potentially be used both for DVB-S streams and IP streams (PID values, encryption information, AV specific information, etc…), below table shows the database entries that is essential for DVB-S, but is not valid for IP (with instances):

Table 3. Instance DVB-S Transponders

Satellite	Hotbird	Astra
Transponder Frequency	11960	12645
Transponder Symbol Rate	27500	08888
Transponder Polarization	Vertical	Horizontal

Although the entries are different for IP case, a quick intuition will show that a similar structure is used. Instead of getting the stream from the satellite, IP channels are provided by Service Providers. Each Service Provider has a different multicast network in itself to which users log and authenticate in order to get IP streams.

What distinguishes different DVB-S channels that are broadcasted from the same satellite are their transponder values; being frequency and symbol rate. Things are similar for IP world, for this time, we have IP address and IP port. The resultant database entry is constructed as follows:

```
typedef struct IPStream_Info
{
/*database index, if need*/
U32 VDB_ip_index;
/*IP Address of the program*/
U32 IP_address;
/*IP Port of the program*/
U16 IP_port;
/*is it new or old parameter*/
BOOL UseKnownMode;
} VTA_IPStream_Info_t;
```

Table 4. Instance IP Multicast Addresses and Port numbers

Service Provider	Provider X	Provider Y
IP Address	225.0.0.12	231.1.1.54
IP Port	1234	1111

Adding a global entry to the database that distinguishes between IP and DVB-S channels, the remaining database entries such as audio and video PID information are left as they are since IP streams and DVB-S streams are basically uses the same

transport stream structure.

3.11. The Virtual Bus Manager

For the control and interaction of different system components, we have developed a Virtual Bus Manager API. The embedded platform consists of many peripherals such as USB and serial, so many external components can be integrated to the system. At some points, these components require to interact with the middleware. For the easy and clear handling of such hardware and software parts, the Virtual Bus Manager registers them to its database and all the communication between the components goes though it.

3.11.1. Structures

The structures, static variables and used definitions are defined below:

- Maximum Pending

Maximum pending is the maximum number of simultaneous connection requests to the server by the client. For such a system of few clients, this would never expected to be so high, thus it is defined as 5 in the header. Its main usage is to avoid simultaneous client attacks to the servers.

```
#define MAXPENDING 5 /* max. connection requests */
```

- Buffer

Buffer size is used for the buffer in which the received message will be stored. The maximum transmission unit (MTU) for a single packet has been defined as 1500 bytes for Ethernet networks. In other words, a single packet's size cannot be larger than 1500 bytes. The buffer size needs to be smaller that this value, it is chosen as 1024, and can be changed according to ones needs.

```
#define BUFFSIZE 1024 /* message buffer size */
```

- Unit Information

TCP clients, which are connected to the virtual bus manager, have been referred as 'unit'. Each unit has some information that needs to be stored in virtual bus manager's small database. These will later be used once the virtual bus manager gives responses to query requests, as well as broadcasting a message. Each unit requires

completing a registration processed to the virtual bus manager with the following information; name, type, remote procedure call (RPC) port number and IP address. For this version of virtual bus manager, name and type needs to be 8 byte in length, whereas RPC port number and IP address length can be changed up to 8 and 15 bytes respectively.

```
#define NAMELEN 8 /* unit name length */
#define TYPELEN 8 /* unit type length */
#define RPCLEN 8 /* RPC port length */
#define IPLEN 15 /* IP address length */
typedef struct UnitInfo
{
        char type[TYPELEN+1];
        char name[NAMELEN+1];
        char rpcPort[RPCLEN+1];
        char ip[IPLEN+1];
}UnitInfo
```

- Function Callback

In order to handle received broadcast messages, a unit needs to point its handler function to virtual bus manager's callback function. An 'AttachCallback' function has been provided with this purpose, refer to the 'Functions' part for more information about it. Unit's handler function needs to have a special form; it should take two char pointer variables; source unit name and the message. The source unit name will hold the name of the unit from whom the message has been broadcasted, and the message will hold simply the broadcasted message itself.

```
/* callback function pointer */
typedef void (*VBM_MessageCallback)(char* sourceUnit, char *message);
```

3.11.2. Functions

- Fill Unit Info

Like the name indicates, VBM_FillUnitInfo function fills up the UnitInfo structure that is passed to the function as the first parameter. The function takes four more parameters. Unit name is the second parameter. It is recorded in virtual bus manager's small database, and used during unit query. The design requires a type for each unit, this has no usage right now, but will be used for group broadcast messages,

and group queries later. Likely, remote procedure call port number has no usage for now, but will be important once the remote procedure mechanism is added to the repertoire. The IP address is simply the localhost '127.0.0.1' if the virtual bus manager and unit run on the same machine.

```
void VBM_FillUnitInfo(UnitInfo* unit,
char* name,
char* type,
char* rpcPort,
char* ip);
```

- Connect

The VBM_Connect function creates a socket between the client and the bus manager, and the communication between these two is provided with this socket. The bus manager than waits for the client to be registered with its name, type and IP. The first parameter is the IP of the system where bus manager is located. If the bus manager and the client are on the same platform, than the IP is simply the local host, or 127.0.0.1. The port number needs to be decided between the bus manager and the client before the interaction takes place.

```
int VBM_Connect(char* ip, int port);
```

- Disconnect

With VBM_Disconnect function, the socket between the client and the bus manager is closed. The bus manager also automatically clear the information about the disconnected client from its database. These can either be dome with VBM_Unregister, or with other methods.

```
void VBM_Disconnect();
```

- Register

VBM_Register takes the UnitInfo structure that is filled with VBM_FillUnitInfo function, and records it to bus managers database.

```
int VBM_Register(UnitInfo *unit);
```

- Unregister

VBM_Unregister clears the corresponding unit info from bus manager's database.

```
int VBM_Unregister(UnitInfo *unit);
```

- Query Unit

VBM_QueryUnit is used to give search functionality to the bus manager system. It is used by the clients so as to get information about other components that are registered to the bus manager. The function takes the name of the searched unit, and the

bus manager fills the unit structure which is given as the second parameter if it is exists.

*int VBM_QueryUnit(char *unitName, UnitInfo *unit);*

- Broadcast Message

VBM_BroadcastMessage simply broadcasts the given parameter to the all units that are registered to the bus manager. If the message means anything to any unit, an action is taken accordingly.

*int VBM_BroadcastMessage(char *message);*

- Attach Callback

VBM_AttachCallback function links the local function that will be used to handle the broadcast messages from the units to the callback that is defined in the Virtual Bus Manager API.

void VBM_AttachCallback(VBM_MessageCallback callback);

3.11.3. Example Usage

```
/*
* vestel – sampleunit.c
* vestel – simple vbm unit example
*/
#include "bus_manager.h"
void printMessage(char* sourceUnit, char *message)
{
fprintf(stdout, "App. message called\n");
fprintf(stdout, "\tSource: %s\n", sourceUnit);
fprintf(stdout, "\tMessage: %s\n", message);
}
int main(int argc, char *argv[])
{
UnitInfo base;
UnitInfo searched;
if(argc != 4)
{
        fprintf(stderr, "USAGE: ./sampleunit <server_ip> <port> <searched unit >
<broadcast message>\n");
        return 1;
        }
/* vestel - fill the unit info */
VBM_FillUnitInfo(&base, "baseunit", "exmodule", "6555", "127.0.0.1");
```

```
if(VBM_Connect(argv[1], atoi(argv[2])))
{
        /* vestel - register */
        VBM_Register(&base);
        sleep(1);
        VBM_AttachCallback(printMessage);
        VBM_QueryUnit(argv[3], &searched);
        fprintf(stdout, "Client:\n\tName; %s\n\tType; %s\n\tRPC port; %s\n\tIP; %s\n", searched.name,
searched.type, searched.rpcPort, searched.ip);
        VBM_BroadcastMessage(argv[4]);
        /* vestel - close socket */
        sleep(1);
        VBM_Disconnect();
        }
return 1;
}
```

CHAPTER 4

CONCLUSION

The newly developed IPTV technology uses media transfer over Internet which brings many challenges with the current network structures and protocols. With the lack of standardization on the description and discovery part of the communication, IPTV network architecture varies for different products. This not only limits the emergence of the IPTV concept as a better alternative to DVB, but also makes designs harder for settop box producers. In order to make products compatible and interoperable with each other, this thesis proposes UPnP standards to be adapted for the IPTV networks. Device and service discovery mechanism with XML based standard descriptions have been projected. Alternative methods such as EPG data transfers have been modified according to UPnP standards. With this standardization, the compatibility of home network products and newly developed IP Settop Boxes aimed to be broaden.

Although personal computers can be used to receive IPTV services, for consumer electronics, embedded devices such as IPSTB are the more convenient way to be used at home. The system requirements such as processing power and memory are stricter for embedded devices; that is why optimization at every stage of communication is necessary. The suggested data parsing at the physical layer brings an efficient alternative for packet handling. This system is suggested in order to accelerate packet processing. Besides, since there is no dedicated playback buffer and the playback engine takes data directly from the network buffer, this system theoretically improves system performance as well.

There is not any standardization for most parts of the overall IPTV system, so the thesis takes proven mechanisms as basis and adapts them to the overall design that consists of the hardware drivers, middleware, and the additional programs which helps the middleware to handle the external components of the system connected via USB or serial interfaces. As an innovative idea, we have used a control system called Virtual Bus Manager so as to communicate between the aforementioned system components. Some system components such as web browser is based on the X Windows architecture, so cross-compiling the X system for the embedded platform has also been a challenge for the feasibility of the final design.

REFERENCES

Alfonsi, B., 2005. “I want my IPTV: Internet Protocol television predicted a winner”, *IEEE Distributed Systems Online*, Vol. 31, Issue 2.

DLNA, 2006. *Digital Living Network Alliance Home Networked Device Interoperability Guidelines Expanded*, DLNA Consortium, Los Angeles.

ETSI, 2003. *Digital Video Broadcasting (DVB); Multimedia Home Platform (MHP) Specification*, ETSI, Paris.

Pourmohammadi-Fallah, Y., Zahir, S., Alnuweiri, H. M., 2005. “A Fast-Start Rate Control Mechanism For Video Streaming Applications”, Consumer Electronics ICCE 2005 Digest of Technical Papers International Conference, pp. 3-4.

Rosado-Sosa, C. and Rubin, I., 1998. “Jitter Compensation Scheduling Schemes for the Support of Real-Time Communications”, Conference Record IEEE International Conference, Vol. 2, pp. 885-890.

WEB_1, 1981. RFC 791 Internet Protocol, 06/05/2007.
http://www.ietf.org/rfc/rfc791.txt

WEB_2, 2000. RFC 2974 SAP: Session Announcement Protocol, 06/05/2007.
http://www.ietf.org/rfc/rfc2974.txt

WEB_3, 1999. RFC 2543 SIP: Session Initiation Protocol, 06/05/2007.
http://www.ietf.org/rfc/rfc2543.txt

WEB_4, 1998. RFC 2326 Real Time Streaming Protocol, 06/05/2007.
http://www.ietf.org/rfc/rfc2326.txt

WEB_5, 2006. Kasenna Portal TV™ Whitepaper, 06/05/2007.
http://www.kasenna.com/downloads/white_papers/PortalTV_paper.pdf

WEB_6, 2003. Digital Home White Paper, 06/05/2007.
http://www.dlna.org/about/DLNA_Overview.pdf

WEB_7, 2000. UPnP Device Architecture Version 1.0, 06/05/2007.
http://www.upnp.org/download/UPnPDA10_20000613.htm

WEB_8, 2002. UPnP AV Transport: 1 Service Template Version 1.01, 06/05/2007.
http://www.upnp.org/standardizeddcps/documents/AVTransport1.0.pdf

WEB_9, 2002. UPnP Content Directory: 1 Service Template Version 1.01, 06/05/2007.
http://www.upnp.org/standardizeddcps/documents/ContentDirectory1.0.pdf

WEB_10, 1989. RFC 1112 IGMP version 1 (IGMPv1) specification, 06/05/2007.
http://www.ietf.org/rfc/rfc1112.txt

WEB_11, 1997. RFC 2236 IGMP version 2 (IGMPv2) specification, 06/05/2007.
http://www.ietf.org/rfc/rfc2236.txt

WEB_12, 2002. RFC 3376 IGMP version 3 (IGMPv3) specification, 06/05/2007.
http://www.ietf.org/rfc/rfc3376.txt

WEB_13, 1996. RFC 1889 RTP: A Transport Protocol for Real-Time Applications, 06/05/2007. http://www.ietf.org/rfc/rfc1889.txt

WEB_14, 1999. Simple Service Discovery Protocol, 06/05/2007.
http://quimby.gnus.org/internet-drafts/draft-cai-ssdp-v1-03.txt

WEB_15, 1994. RFC 1584 Multicast Extensions to OSPF, 06/05/2007.
http://www.ietf.org/rfc/rfc1584.txt

WEB_16, 1998. RFC 2362 Protocol Independent Multicast: Protocol Specification, 06/05/2007. http://www.ietf.org/rfc/rfc2362.txt

WEB_17, 1988. RFC 1075 Distance Vector Multicast Routing Protocol, 06/05/2007. http://www.ietf.org/rfc/rfc1075.txt

WEB_18, 1981. RFC 793 Transmission Control Protocol, 06/05/2007. http://www.ietf.org/rfc/rfc793.txt

WEB_19, 1996. RFC 1945 Hypertext Transfer Protocol - HTTP/1.0, 06/05/2007. http://www.ietf.org/rfc/rfc1945.txt

APPENDIX A

IPSTB APPLICATION PROGRAM INTERFACE

Below list shows the open source software and their dependent licensing information that are used with the target platform:

Description	**Licensing**
uclibc root filesystem binary	GPL
uclibc toolchain binary	GPL
Kernel Source (2.6.x)	**GPL**
- includes UDP multicasting support	GPL
- includes IGMP Management	GPL
- includes SAP client	GPL
IPSTB Related	
- FreeType fonts	Dual GPL or BSD
- libpng (Browser)	General Open Source
- libjpeg (Browser)	BSD
- libexif (Browser)	LGPL
- truetype MSWebFonts	Microsoft EULA
- liveMedia rtsp client	LGPL

APPENDIX B

VIRTUAL BUS MANAGER SOURCE CODE

```
/*
  * vestel - virtual bus manager
  */
#include "bus_manager.h"
#include <semaphore.h>  /* semaphore */
#define CMDLEN 1 /* message type length */
#define KEY 129 /* xor key */
static UnitInfo baseUnit;
static VBM_MessageCallback messageHandler;
static int serverSocket;
static char receivedData[BUFFSIZE];
pthread_mutex_t mutexClient = PTHREAD_MUTEX_INITIALIZER;
/* semaphores are declared global so they can be accessed
* in main routine and in recv thread routine
*/
sem_t vbmSema;
static void* ReceiveThread(void* argList);
void XORScramble(char* message, int messageLength);
int VBM_Connect(char *ip, int port){
        struct sockaddr_in ipServer;
        pthread_t clientTh;
        /* vestel - Create the TCP socket */
        if((serverSocket = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0){
                perror("Failed to create socket"); return -1;
        }
        /* vestel - Construct the server sockaddr_in structure */
        memset(&ipServer, 0, sizeof(ipServer));  /* vestel - clear struct */
        ipServer.sin_family = AF_INET;  /* vestel - Internet/IP */
        ipServer.sin_addr.s_addr = inet_addr(ip);/* vestel - IP address */
        ipServer.sin_port = htons(port);  /* vestel - server port */
        /* vestel - Establish connection */
        if(connect(serverSocket, (struct sockaddr*)&ipServer, sizeof(ipServer)) < 0){
                perror("Failed to connect with server");              return -1;
        }
        sem_init(&vbmSema, 0, 0); /* initialize last param to 0, wait once said */
        /* second param = 0 - semaphore is local */
```

```
/* vestel - start separate recv thread */
        if((pthread_create(&clientTh, NULL, ReceiveThread, NULL))){
                perror("Failed to start a seperate thread");
        }
        return 1;
}
static void* ReceiveThread(void* argList){
        int received = -1;
        char buffer[BUFFSIZE]; /* vestel - to write/read commands */
        char source[NAMELEN];
        char message[BUFFSIZE - NAMELEN - CMDLEN];
        /* vestel - Check the command and check for more incoming data in loop */
        for(;;){
                /* vestel - Receive the packet */
                if((received = recv(serverSocket, buffer, BUFFSIZE, 0)) <= 0){
                        perror("Failed to receive packet from client"); break;
                }
                fprintf(stdout,"Client gets: %s, %d\n", buffer, received);
                /* vestel - check the command */
                switch((int)buffer[0]){
                        case CMD_REGISTER: break;
                        case CMD_QUERY:
                                pthread_mutex_lock(&mutexClient);
                                memcpy(receivedData, buffer, BUFFSIZE);
                                pthread_mutex_unlock(&mutexClient);
                                sem_post(&vbmSema);break;
                        case CMD_BROADCAST:
                                pthread_mutex_lock(&mutexClient);
                                memcpy(receivedData, buffer, BUFFSIZE);
                                /* vestel - call the broadcast message handler */
                                if(messageHandler){
                                        /* +1 so as to skip first byte which is message type */
                                        strcpy(source, receivedData + 1);
                                        /* source[NAMELEN] = '\0'; */
                                        /* an additinal +1 so as to skip the null byte between source and
message */
                                        strcpy(message, receivedData + 1 + strlen(source) + 1);
                                        /* message[strlen(message)] = '\0'; */
                                        messageHandler(source, message);
                                }
                                pthread_mutex_unlock(&mutexClient); break;
                }
                /* clear the buffer */
                memset(buffer, 0, sizeof(buffer));
        }
```

```
}
void VBM_Disconnect(){
        sem_destroy(&vbmSema); /* destroy semaphore */
        VBM_Unregister(&baseUnit); /* unregister the device */
        close(serverSocket);
}
int VBM_Register(UnitInfo *unit){
        fprintf(stdout,"Entered register\n");
        char buffer[BUFFSIZE];
        /* fill the buffer */
        buffer[0] = CMD_REGISTER;
        memcpy(buffer + 1, unit, sizeof(UnitInfo));
        fprintf(stdout,"after memcopy in register\n");
        /* XORScramble(buffer, sizeof(UnitInfo) + 1); */
        if(send(serverSocket, buffer, sizeof(UnitInfo) + 1, 0) != sizeof(UnitInfo) + 1){
                perror("Mismatch in number of sent bytes"); return -1;
        }
        fprintf(stdout,"after send in register\n");
        return 1;
        fprintf(stdout,"Exitting register\n");
}
int VBM_Unregister(UnitInfo *unit){
        char buffer[BUFFSIZE];
        /* fill the buffer */
        buffer[0] = CMD_UNREGISTER;
        memcpy(buffer + 1, unit, sizeof(UnitInfo));
        /* XORScramble(buffer, sizeof(UnitInfo) + 1); */
        if(send(serverSocket, buffer, sizeof(UnitInfo) + 1, 0) != sizeof(UnitInfo) + 1){
                perror("Mismatch in number of sent bytes");
                return -1;
        }
        return 1;
}
int VBM_QueryUnit(char *unitName, UnitInfo *unit){
        char buffer[BUFFSIZE];
        /* fill the buffer */
        memset(buffer, 0, sizeof(buffer));
        buffer[0] = CMD_QUERY;
        strcat(buffer, unitName);
        /* XORScramble(buffer, strlen(buffer)); */
        /* vestel - send the packet to the server*/
        if(send(serverSocket, buffer, strlen(buffer), 0) != strlen(buffer)){
                perror("Mismatch in number of sent bytes");
                return -1;
        }
```

```
        sem_wait(&vbmSema); /* wait till the answer received */
        /* vestel - record result */
        UnitInfo *tempUnitPointer = (UnitInfo*)(receivedData + 1);
        memcpy(unit, tempUnitPointer, sizeof(UnitInfo));
        memset(receivedData, 0, sizeof(receivedData));
        return 1;
}
int VBM_BroadcastMessage(char *message){
        char buffer[BUFFSIZE];
        /* fill the buffer */
        sprintf(buffer, "%c%s%c%s", (char)CMD_BROADCAST, baseUnit.name, (char)0, message);
        /* vestel - send the packet to the server */
        /* XORScramble(buffer, strlen(message) + strlen(baseUnit.name) + 2);          */
        if(send(serverSocket, buffer, strlen(message) + strlen(baseUnit.name) + 2, 0) !=
                strlen(message) + strlen(baseUnit.name) + 2){
                perror("Mismatch in number of sent bytes"); return -1;
        }
        return 1;
}
void VBM_AttachCallback(VBM_MessageCallback callback){
        fprintf(stdout,"Entered attach callback\n");
        messageHandler = callback;
        fprintf(stdout,"Exitting attach callback\n");
}
void VBM_FillUnitInfo(UnitInfo* unit, char *name, char *type, char* rpcPort, char *ip)
{
        int size;   memset(unit, 0, sizeof(UnitInfo));
        size=strlen(name);  strncpy(unit->name, name,size<NAMELEN?size:NAMELEN );
        unit->name[NAMELEN-1] = 0; size=strlen(type);
        strncpy(unit->type, type,size<TYPELEN?size:TYPELEN);
        unit->type[TYPELEN-1] = 0; size=strlen(rpcPort);
        strncpy(unit->rpcPort, rpcPort,size<RPCLEN?size:RPCLEN);
        unit->rpcPort[RPCLEN-1] = 0; size=strlen(ip);
        strncpy(unit->ip, ip, size<IPLEN?size:IPLEN);
        unit->ip[IPLEN-1] = 0; memcpy(baseUnit.name, unit->name, NAMELEN);
        memcpy(baseUnit.type, unit->type, TYPELEN);
        memcpy(baseUnit.rpcPort, unit->rpcPort, RPCLEN);
        memcpy(baseUnit.ip, unit->ip, IPLEN);
}
void XORScramble(char* message, int messageLength)
{
int i = 0;for(i = 0; i < messageLength; i++)
                message[i] = (char)(message[i] ^ KEY);
} /* end of new client functions */
```

```
void VBM_ServerCycle(int serverSock){
	struct sockaddr_in  ipClient;
	int clientIndex, clientSocket;

	#if 0
	/* vestel - first start the GTK on the PC */
	struct sockaddr_in gtkClient;
	int gtkSocket;
	if((gtkSocket = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0){
		perror("Failed to create socket");
	}
	memset(&gtkClient, 0, sizeof(gtkClient));  /* vestel - clear struct */
	gtkClient.sin_family = AF_INET;  /* vestel - Internet/IP */
	gtkClient.sin_addr.s_addr = inet_addr("10.1.10.20");/* vestel - IP address */
	gtkClient.sin_port = htons(6500);  /* vestel - server port */
	/* vestel - Establish connection */
	if(connect(gtkSocket, (struct sockaddr*)&gtkClient, sizeof(gtkClient)) < 0){
		perror("Failed to connect with gtk server");	}
	/* vestel - send the packet to the sgtk erver*/
	if(send(gtkSocket, "1", 2, 0) != 2){
		perror("Mismatch in number of sent bytes"); }
	close(gtkSocket);
	/* vestel - end of first start the GTK on the PC */
	#endif
	/* vestel - Run the server until cancelled */
	for(;;){
		unsigned int clientLen = sizeof(ipClient);
		/* vestel - Wait for client connection */
		if((clientSocket = accept(serverSock, (struct sockaddr*)&ipClient, &clientLen))<0){
			perror("Failed to accept client connection");
		}
		fprintf(stdout,"Client connected: %s\n", inet_ntoa(ipClient.sin_addr));
		for(clientIndex=0; clientIndex<NTHREADS; clientIndex++)
			if(!(tcpClients[clientIndex].unit.name[0]>0))
				break;
		tcpClients[clientIndex].socket = clientSocket;
		/* vestel - start separate thread */
		fprintf(stdout,"Start thread ID: %x\n", clientIndex);
		tcpClients[clientIndex].clientID = clientIndex;
		if(pthread_create(&(tcpClients[clientIndex].clientThread), NULL, VBM_HandleClient,
&tcpClients[clientIndex])){
			perror("Failed to start a seperate thread");
		}
	}
}
```

```
static void* VBM_HandleClient(void* argList){
	/* convert to appropriate type the thread arguments */
	tcpClient* client = (tcpClient*)argList;
	UnitInfo *tempUnitPointer;
	int received = -1;
	char buffer[BUFFSIZE]; /* vestel - to write/read commands */
	char unitName[NAMELEN]; /* vestel - stored for query message */
	int i = 0, j = 0;

	/* vestel - Receive bytes and always check for more incoming data in loop */
	for(;;){
		/* vestel - Receive the packet */
		if((received = recv(client->socket, buffer, BUFFSIZE, 0)) <= 0){
			/* vestel - remove the unite from the client list */
			pthread_mutex_lock(&mutexClient);
			memset(&tcpClients[client->clientID].unit, 0, sizeof(UnitInfo));
			pthread_mutex_unlock(&mutexClient);
			perror("Failed to receive packet from client");
			break;
		}
		/* vestel - xor with KEY, descramble */
/*		for(i = 0; i < received; i++)
			buffer[i] = (char)(buffer[i] ^ KEY);	*/

		fprintf(stdout,"VBM Receives: %s, %d\n", buffer, received);
		/* vestel - check the command */
		switch((int)buffer[0]){
			case CMD_REGISTER:
				/* vestel - add the unite to the client list */
				tempUnitPointer = (UnitInfo*)(buffer + 1);
				pthread_mutex_lock(&mutexClient);
				memcpy(&tcpClients[client->clientID].unit, tempUnitPointer,
sizeof(UnitInfo));
				fprintf(stdout,"Client:\n\tName; %s\n\tID; %x\n\tSocket; %x\n\tType;
%s\n\tRPC port; %s\n\tIP; %s\n",
					tcpClients[client->clientID].unit.name, tcpClients[client-
>clientID].clientID,
					tcpClients[client->clientID].socket, tcpClients[client-
>clientID].unit.type,
					tcpClients[client->clientID].unit.rpcPort, tcpClients[client-
>clientID].unit.ip);
				pthread_mutex_unlock(&mutexClient);
				break;
			case CMD_QUERY:
				/* vestel - get the name */
```

```
                                        strncpy(unitName, buffer + CMDLEN, NAMELEN);
                                        unitName[NAMELEN] = '\0';
                                        /* vestel - smart search */
                                        memset(buffer, 0, sizeof(buffer)); /* clear the buffer */
                                        buffer[0] = CMD_QUERY; /* 1 is query message */
                                        pthread_mutex_lock(&mutexClient);
                                        for(j = 0; j<NTHREADS; j++){
                                                if(tcpClients[j].unit.name[0] > 0){
                                                        if(!(strcmp(tcpClients[j].unit.name, unitName))){
                                                                fprintf(stdout,"Unit found, sending info\n");
                                                                memcpy(buffer + 1, &tcpClients[j].unit,
sizeof(UnitInfo));
                                                                break;
                                                        }
                                                }}
                                        pthread_mutex_unlock(&mutexClient);
                                        /* vestel - send query result */
                                        if((int)send(client->socket, buffer, sizeof(UnitInfo) + 1, 0) !=
sizeof(UnitInfo) + 1){
                                                perror("Failed to send bytes to client");
                                        }
                                        break;
                                case CMD_BROADCAST:
                                        /* vestel - smart broadcast */
                                        pthread_mutex_lock(&mutexClient);
                                        for(j = 0; j<NTHREADS; j++){
                                                if(tcpClients[j].unit.name[0] > 0){
                                                        /* if(strncmp(buffer+CMDLEN, tcpClients[j].unit.name,
NAMELEN)){ */
                                                                if((int)send(tcpClients[j].socket, buffer,
received, 0) != (int)received){
                                                                        perror("Failed to send bytes to
client");
                                                                }
                                                        /* } */
                                                }
                                        }
                                        pthread_mutex_unlock(&mutexClient);
                                        break;
                                case CMD_UNREGISTER:
                                        /* vestel - remove the unite from the client list */
                                        pthread_mutex_lock(&mutexClient);
                                        memset(tcpClients[client->clientID].unit.name, 0, NAMELEN+1);
                                        memset(tcpClients[client->clientID].unit.type, 0, TYPELEN+1);
                                        memset(tcpClients[client->clientID].unit.rpcPort, 0, RPCLEN+1);
```

```
                                        memset(tcpClients[client->clientID].unit.ip, 0, IPLEN+1);
                                        pthread_mutex_unlock(&mutexClient);
                                        break;
                        }
                        /* clear the buffer */
                        memset(buffer, 0, sizeof(buffer));
                }
                fprintf(stdout,"Client exits\n");
                return argList;
}

int main(int argc,char *argv[]){
                int serverSocket;
                if(argc!=2){
                        fprintf(stderr,"USAGE: ./Tcpserver <port> &\n"); exit(1);
                }
                serverSocket = VBM_InitServer(htons(atoi(argv[1])));
                VBM_ServerCycle(serverSocket);
                return 1;
}
```