UNIVERSITY OF OSLO
Department of Informatics

Investigating the distribution of functionality for building a video server hypercube with IXP2400 cards

Master thesis

Andreas Petlund

11th November 2005

Acknowledgements

I want to thank my advisors, Pål Halvorsen and Carsten Griwodz for excellent guidance and unlimited patience. I also want to thank my fellow students Tom Anders Dalseng and Øystein Yri Sunde for inspiration, discussions and help. Finally, I want to thank my girlfriend Anne for support and structured help, and my family for never giving up.

Abstract

Streamed multimedia is becoming common on the Internet as the bandwidths increase for end-users. To stream data with high bitrates to many concurrent users, servers are needed that can handle these extreme loads. Implementing servers in clusters to be able to meet demands has proved to be a good strategy, providing scalability and performance. One commercial actor that has done this successfully is nCube. Their n4x solution is a server cluster based on a hypercube interconnection topology, and their reported server performance is promising. However, the use of special hardware for offloading routing increases the cost for deploying this system. Network processing units have many similar properties to the special hardware used by nCube. They are fully programmable and optimized for networking tasks. In this thesis, we start by charting the functionality and capabilities of the IXP2400 network processor by implementing a series of test applications. Using knowledge gained from this exploration, the design and implementation of a video server hypercube prototype is done. We present an evaluation of the IXP2400 hardware platform based on the test applications. Our video server cube prototype is also presented and evaluated.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Streaming data with high bitrates to a potentially high number of concurrent users presents us with great challenges regarding scalability and throughput. The network bandwidth rates needed for multimedia streaming systems today strain the processing power of server solutions, stealing resources that could be applied to other important tasks. This thesis will investigate the possibility of using network processing units (NPUs) to offload networking tasks and, in particular, to implement routing in a distributed server cluster topology.

1.1 Background and motivation

The Internet, since its beginning as an experimental network in 1968, has experienced an astounding development. Especially in the last ten years, it has become common property, and a variety of services are now available to the public. During the last years, the bandwidth available to end users has been steadily increasing in most developed countries. With the high bitrates becoming available, the transmission of audio and video has become the new killer application driving the development of networks and infrastructure.

The most common way to access digital multimedia content is still to download a file and play it locally. The alternative is to transmit the data as the content is consumed. This is known as data streaming. Streamed services are becoming more common every day due to the increased capacity available for end users. For instance, most national radio-stations in Norway are being streamed from the servers of the National Broadcasting Company (NRK) [44]. They also offer the possibility to watch recently transmitted TV programs as a streamed service. The fact that the data is streamed implies that the receiver will require a certain quality to be satisfied. If the stream halts for too long, the presentation of the video will stop, and the customer will probably lose interest. Other multimedia content, like teleconferencing, is even more sensitive to changes in the quality of the data transmitted. If the different contributions to a teleconference are delivered at the wrong time, the service will be pointless.

One area of multimedia streaming that is demanding in terms of data rates is the streaming of high quality video. The servers that have to deliver such streams to many concurrent end users will be strained to meet demand unless some techniques are applied to enhance server performance. A multimedia streaming server will not only have to fetch the data, and prepare it for streaming, it will have to handle the transmission of a huge number of network packets. There are many strategies that can be applied to handle these challenges. One of these is to build an immensely powerful server with several CPU's, a multitude of network cards and an extensive amount of memory and storage capacity. Such a single server would be very expensive, and would soon have to give up if the demand were to increase.

To solve the scalability problem, a common approach is to distribute the load on several interacting computers. Several of these approaches will be discussed in this thesis. One variant of this strategy is to deploy a multimedia server as a distributed system consisting of many interconnected computers cooperating to bring the media data to the users. Such a solution would be scalable by adding more computers to the system, and dividing the load between the nodes. Data would have to be collected from the correct node in the interconnection topology, and routed to the egress. The successful interconnection would, therefore, demand resources in terms of routing and other network processing. With the servers busy pulling data from the disks, and preparing it for streaming, reserving resources to process the routing will be a challenge.

One way to ensure effective network processing inside such a topology would be to offload such tasks to a different functional unit. Some commercial actors have done this by implementing routing functionality on custom hardware. The development of hardware, however is an expensive and time-consuming task. On the other hand, network processors optimized for the efficient handling of packets are available today. Although most commonly used in routers and switches, implementations of network processors that can be used in PCs are available. These platforms are highly programmable and configurable to different networking tasks. By building a distributed topology multimedia server, and offloading internal routing tasks to network processors, it would be possible to achieve high performance in a scalable system without having the disadvantages of having to build custom hardware.

1.2 Thesis domain

One commercially available server that has been successful at implementing a distributed server topology is the nCube [42] n4x solution. This is, however, a system that depends heavily on custom hardware for internal routing, making it expensive. The design of special hardware is a task that is time consuming. This makes the time-to-market for such products longer than for programmable solutions. Finding a way to implement a similar server topology without having to use custom hardware would enable much cheaper server solutions with the kind of scalability and performance that is needed to meet competition.

This thesis aims at answering the question of whether a distributed multimedia server cube, similar to the n4x solution, can be implemented using IXP2400 [32] network processors. In pursuing this goal, different areas of the network processor programmability have been explored to find the most effective ways of implementing various applications. We also seek to explore whether offloading network tasks would enhance the performace of a host machine. An implementation of a multimedia server hypercube was made in order to identify strengths and weaknesses of such solutions, and to demonstrate that an NPU offloading of the routing framework in such a solution would work.

The thesis shows that network processors are more efficient at routing tasks than an application running on a Linux host. This proved also to be true for the host application when implemented in the kernel. Moreover, it shows that an interconnected network topology, a hypercube, using IXP2400 cards for routing was far superiour to a switched network topology for delivering data packets. Tests showed that there is a noticeable performace gain by offloading network tasks from the host to the IXP2400 card. The video server cube (***VS*^{3}**) system was implemented, and proved to work with the proposed routing framework. In conclusion, our prototype shows that the IXP2400 NPU efficiently can offload the host machine and provide a favourable way of implementing closely interconnected servers in a cube topology.

1.3 Document structure

In the following chapters, the requirements for successfully implementing a video server cube using IXP2400 cards will be investigated. This will be done by analyzing hardware and software capabilities on a general scale, and with regard to our implementation goal. The requirements and implementation steps taken are presented, and evaluations of the different results are given.

The focus in chapter 2 and 3 is mainly on general aspects of the network processors and the applications and tests made to gain the necessary understanding for programming the IXP2400 platform. Chapter 4 and 5 focus on the requirements for multimedia applications, and the building of the VS^3 server cube application in particular. An evaluation of the results is given for all stages of implementation.

Chapter 2 describes the concept of network processors with the main focus on the Intel internet exchange architecture (IXA). An introduction is given to the hardware boards (Radisys ENP2611 [13]) used in the thesis, and the network processing units that resides on them. The software tools provided by the manufacturers is described, as well as the general programming techniques and strategies.

Chapter 3 presents the applications developed to explore and evaluate the ENP2611 network processor card based on the IXP2400 chipset. We describe the different software and hardware building blocks, and how they perform given different tasks. The results of tests performed on these applications is also presented, and some general conclusions are drawn on how to use these NPUs in general.

Chapter 4 gives an introduction to multimedia systems. Different multimedia applications are presented. The requirements that have to be met when dealing with multimedia applications is discussed. Different solutions for meeting the requirements is also presented. The nCube n4x solution to these challenges is described, as well as some of the protocols that can be used for multimedia streaming.

Chapter 5 describes all the stages in implementing the VS^3 video server cube solution. Different design approaches are presented, and the routing framework is described in detail. The main bottlenecks of this system are located, and the results of tests performed on the system are shown and discussed.

Chapter 6 concludes this thesis, and outlines further work.

The source code for the VS^3 server cube system is included as an appendix. The appendix is divided into subsections based on whether the code is written for μEngines, XScale or Linux host.

Chapter 2

Internet exchange architecture

Network processing Units (NPUs) are special processor architectures used for demanding networking tasks such as backbone routing and switching. This chapter will give a short introduction to NPUs, and especially the internet exchange architecture (IXA) platforms. We will focus on the two network boards that have been available for testing, and especially the hardware and software for the ENP2611 [13], based on the IXP2400 [32] chipset, upon which this thesis bases its studies. The reason why the ENP2505 [12], based on the IXP1200 [31] chipset, is described as well, is that this was the NPU board we used before receiving the ENP2611. Some of the areas which have been explored have also been tested on this platform. The comparison of performance on different levels of the architectures, and the challenges of implementing on each platform have also been important issues.

2.1 Network Processing Units

After 40 years of Moore's law [18], we still see processing power expand at an exponential rate. The successful use of multithreading in the last generations of processors have made the architectures more efficient. This increases the heat produced by the logic units [41]. This has made the silicon manufacturers think about new ways of continuing the current progress. The picture we have been shown of future architectures outlines multiple core processors with simpler structure, thus using more parallel computations.

In a similar manner to how Moore's law applies to microprocessors, we have had an exponential growth in network bandwidth capacity. Figure 2.1 shows how networking bandwidth has increased compared to processing power over the last years. There is no indication of the development rate slowing down, but it will eventually have to halt due to limitations of the transmission medium (if new technologies are not introduced). As of

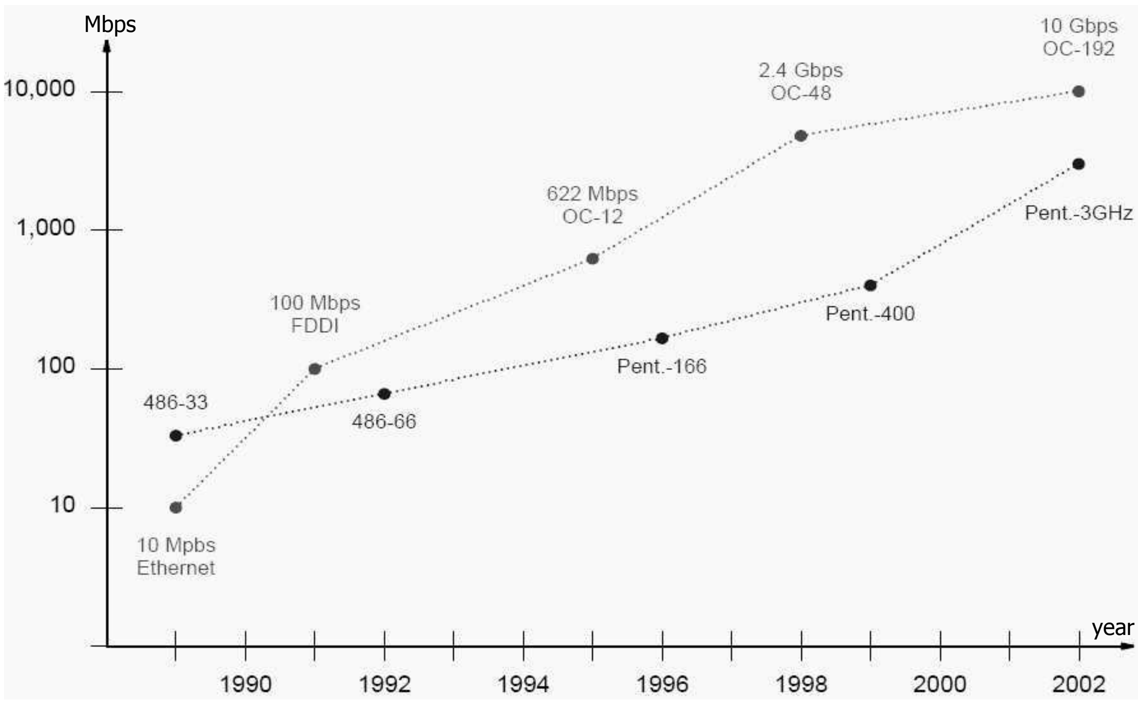


Figure 2.1: Evolution of technologies [22].

yet, the network speed growth is faster than the CPU power growth. Thus today's possible bandwidths sustain an increasing demand for processing power. With the decreasing inter-arrival time of network packets, computers will be strained to keep up with protocol handling, checksumming and other necessary tasks without sacrificing capacity that other processes need.

One solution to this challenge is to offload network tasks to separate functional units. These units will be optimized for efficient packet handling and throughput. Since NPUs are to be optimized for packet handling, the design will differ from traditional computing chipsets. A typical design is a series of several small, symmetric processing units working in parallel. The parallel structure of the packet processing enables the tasks to be performed in a pipeline, with each functional unit performing a special task. This removes the bottleneck of single CPU processing that has to schedule the processing capacity between the different tasks.

A wide range of different companies are manufacturing NPUs for different platforms and purposes, these include Agere [1], AMCC [3], IBM [25], Intel [27], Internet Machines [26], Motorola [38], PMC-Sierra [48], and Vitesse [61]. All these are more or less based on the same offloading ideas, but their implementations vary greatly.

In the past few years, in particular, Intel has focused on network processing, creating a series of network processor platforms called the Internet Exchange Architecture (IXA). This platform group incudes three main series of NPUs, namely the IXP12xx, IXP2xxx and IXP4xx series of network processors [43].

2.2 Intel Internet Exhange Processors

The IXA platform from Intel includes, as of now, three main series: IXP12xx, IXP2xxx and IXP4xx. The IXP12xx series was in common use up to last year, when Intel decommisioned it in favour of the IXP2xxx series. The IXP4xx series is targeted at the home and small business market, while the IXP2xxx series aims at the heavier applications. In this section, the main features of the IXP1200 will be outlined. A more detailed description will be given of the IXP2400 as this is the chipset on which most work has been done in this thesis.

2.2.1 IXP1200

The Intel IXP1200 [31] network processor chipset is designed to meet the wide requirements placed on network equipment in high performance systems and consists of six main functional units as shown in figure 2.2. The core processing unit is a 32 bit StrongARM processor running at 232MHz. There are six special purpose microprocessors called microengines (μEngines) running in parallel, also at 232MHz. Each μEngine can accomodate a maximum of four contexts. The DRAM unit provides acces to memory for storing packet data, and the SRAM unit gives acces to memory for storing shared variables and metadata. The IX bus connects the internal IXP chipset devices, and the PCI bus enables the card to interface with other PCI devices. In addition to these six main components there are 4KB of scratchpad memory used for high-speed communication between functional units. There are also special registers for inter-unit communication.

2.2.2 ENP2505

The Radisys ENP2505 (see figure 2.3) integrates the IXP1200 chipset on a network board. It has four 10/100 Mbit Ethernet ports for communication. For boot code and other stored procedures, there are 8MB of Flash memory. The card has a total of 8MB of SRAM and 256MB of DRAM. In addition there are a serial port for interfacing and debugging, and a PCI connector.

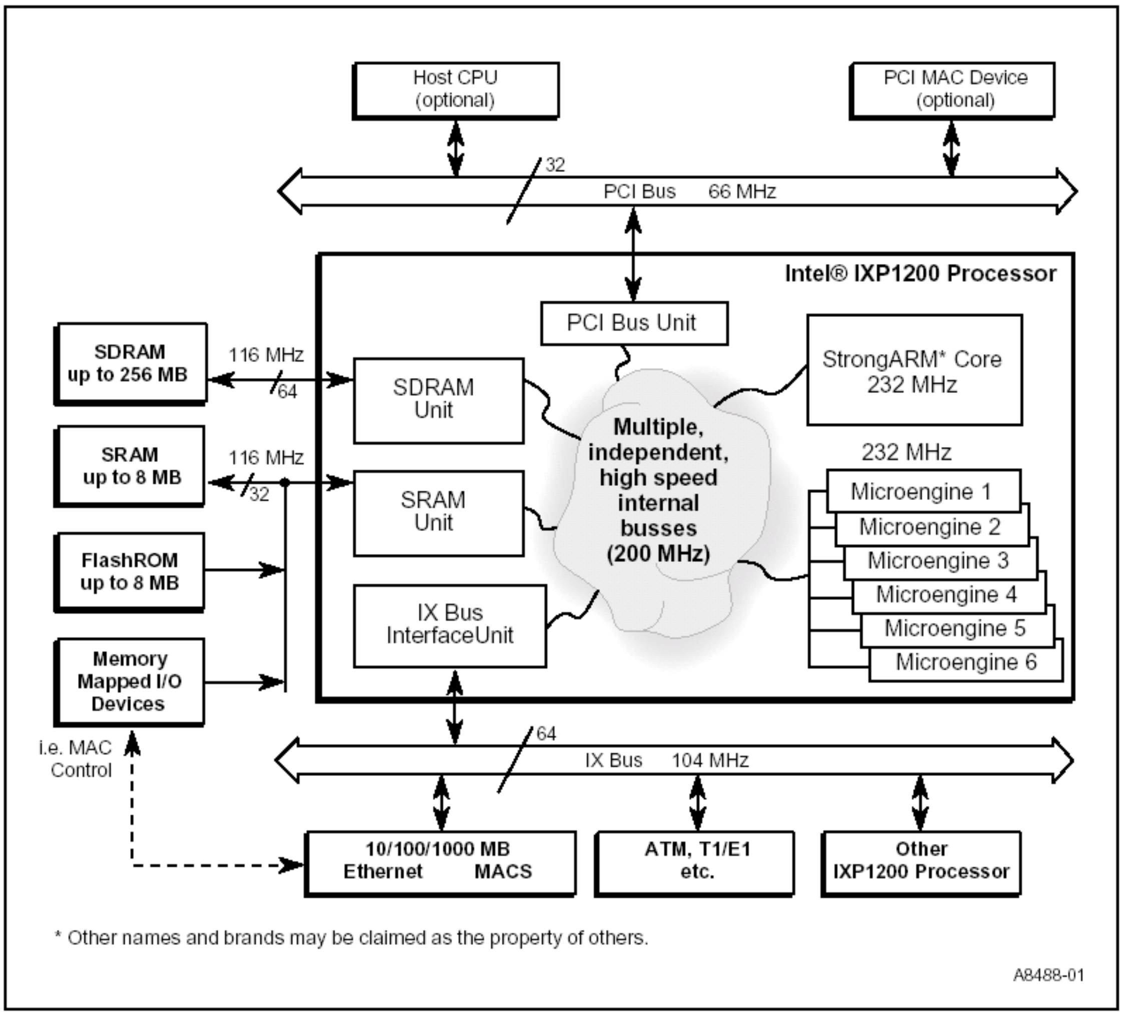


Figure 2.2: IXP1200 block diagram [31].

2.2.3 IXP2400

The IXP2400 chipset [32] is a second generation NPU platform from Intel. It is designed to handle a wide range of access, edge and core applications. It has a more powerful CPU and microengines than the IXP1200 and is better suited for heavy networking tasks. The physical interfaces are customizable and can be chosen by the manufacturer of the device on which the IXP chipset is integrated. The number of network ports and the network port type are also customizable. The major functional blocks of the IXP2400 chipset are shown in figure 2.4:

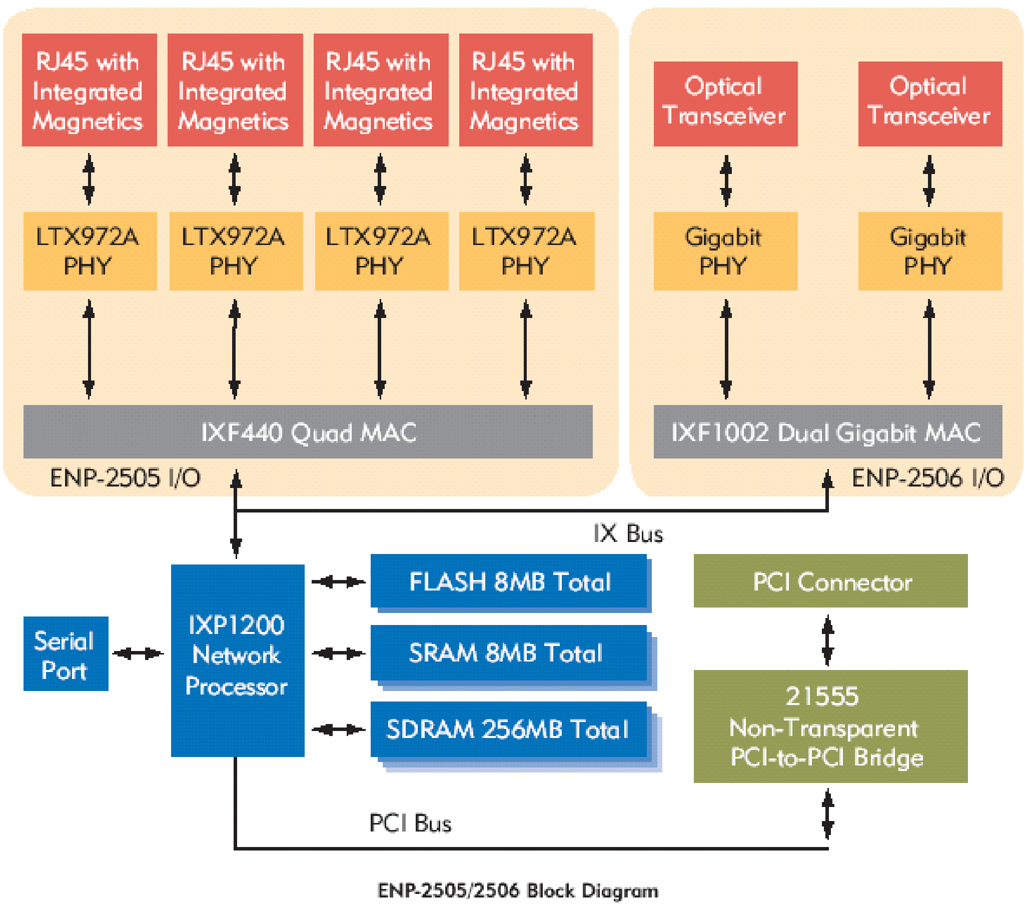


Figure 2.3: ENP2505 block diagram [12].

- **An Intel 600MHz XScale core:** The XScale core is capable of running an independent operating system (in our case, MontaVista Linux for embedded platforms [37]). This unit is used to initialize and manage the chip. In a network application, it typically controls some of the higher layer network processing tasks like updating IP routing tables.

- **8 600MHz μEngines:** For efficient handling of general packet processing, the eight μEngines can be used. These are separate 32 bit programmable units that are specialized for network processing. Each μEngine has a maximum of eight threads (contexts) that can enhance performance further.

- **Two independant SRAM controllers:** The two SRAM controllers can independently access one SRAM channel each. This type of memory is generally used for

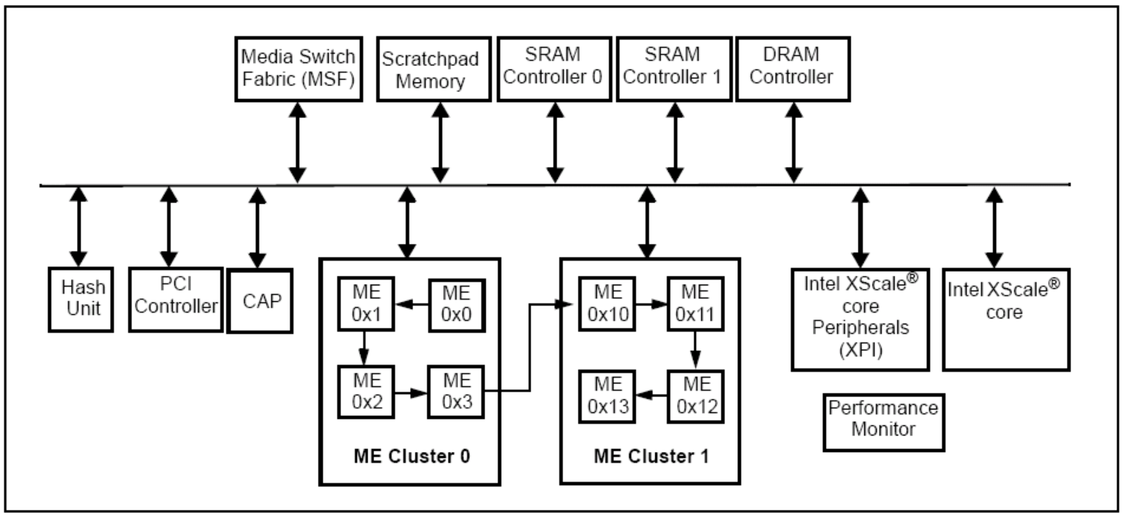


Figure 2.4: IXP2400 block diagram [32].

packet metadata, control data, shared counters and variables. Atomic control operations are supported for synchronization purposes.

- **One DDR DRAM controller:** DRAM memory is generally used to store packet data. When a packet is received, it can be stored here, and all steps of the packet handling pipeline can modify the packet from the same memory area.

- **Media switch fabric (MSF):** The MSF is the IXP interface to the framing hardware. This is where connections with the physical layer is configured. It contains transmit and receive buffers, and packets that are about to be transmitted are divided into smaller MPackets[1] upon transmission to make the IXP chipset compatible with as many different types of media hardware as possible.

- **PCI 2.2 compliant controller:** In order to connect with other components, like a host machine or PCI compilant peripheral devices, the IXP has a PCI controller.

- **Scratchpad, hash and CAP (SHaC) unit:** Three of the most useful functions of the IXP chipset resides in the SHaC unit:

  - **Scratchpad memory - 16KB:** The scratchpad memory is a 16 KB storage for general purpose use with atomic operations and ring support. This is widely used to convey packet buffer handles between different processing units.

[1]To make the IXP system compatible with different physical interface standards, packets are divided into a basic unit called mpackets before they are sent to the interface controllers via the MSF. On the IXP2400 mpackets can be 64, 128 or 256 bytes, but once configured, the mpacket size must remain constant [6].

  - **Hash unit:** The hash unit can be used to achieve hardware supported hash calculations, thus giving increased performance when making lookups in tables etc.
  - **Chip-wide control and status registers (CAP):** The CAP unit is used to handle inter-processor communications.

- **XScale Core peripherals:** These include an interrupt controller, four timers, one serial UART port, 8 general purpose input/output circuits (GPIOs), and an interface for low-speed off-chip peripherals. In addition, the IXP chipset has a performance monitor with registers for analyzing and tuning performance.

The sum of these components is the IXP2400 chipset. In the hardware board that is available to us, this chipset resides on a network card made by Radisys [51].

2.2.4 Radisys ENP2611

The Radisys ENP2611 [13] is a network board that integrates the IXP2400 chipset, peripherals, memory and physical interfaces. Figure 2.5 gives a schematical representation of the hardware. The following main components are present:

- **256MB (DDR) DRAM:** The DRAM is, as mentioned in section 2.2.3, primarily used to store packet data.

- **8MB SRAM:** The SRAM is accessible via two different channels to optimize performance. It is generally used for metadata and shared variables.

- **16MB StrataFlash memory:** As on the ENP2505 this memory keeps the boot code and utilities for the board. The increased size compared to the ENP2505 allows for a Linux kernel to be loaded into the flash memory.

- **Three gigabit ethernet interfaces:** The physical communication with the network is made possible by the optical transceivers that are controlled by the PM3386 (controls two optical interfaces) and PM3387 (controls one optical interface) Gigabit Ethernet controllers. These give the board a total of 3 optical interfaces that can be used freely by the IXP hardware.

- **SCSI parallel interface v3 (SPI-3) bridge FPGA:** The SPI-3 is used for connection between the physical interfaces and the IXP2400 MSF. It is the link between the PM3386 and PM3387 controllers and the IXP2400.

- **Two PCI-PCI bridges:** The first is a non-transparent Intel 21555 PCI-bridge chip which connects the internal 64-bit PCI bus to the backplane host 32 or 64 bit PCI bus. This gives the hardware the possibility to provide inter-processor communication and interrupts. The second is a TI PCI2150 transparent PCI bridge which links the internal 64-bit PCI bus to a downstream 32-bit PCI bus. This is used to connect the debug 10/100 interface to the rest of the chipset.

- **Ethernet controller:** An Intel 82559 PCI Ethernet controller is used to control the 10/100 ethernet debug interface.

- **SPI-3 Option Board connector:** This can be used to connect the chipset to a service specific NPU co-processor that can be used to provide further hardware support for specific tasks.

- **10/100 Ethernet interface:** This debug port can not be used as a part of the IXP topology, but is rather used as a tool for loading images into flash ROM and mounting NFS filesystems by the Linux kernel running on the XScale core.

- **Clock generation circuitry:** The clock generation circuitry includes the general IXP2400 system clock, and the interface clocks for the IXP2400 MSF/FPGA and the FPGA/PM338x interfaces.

- **Reset and initialization circuitry:** This circuitry connects to a switch located on the board to enable a manual reset. Software resets can be initialized both form the board and via the PCI bridge.

- **Power:** Power supply circuitry is needed for the different logical parts of the chipset.

These components are the external hardware needed to complete a functional IXP chipset environment. The hardware manufacturers also provide some software tools to make developing applications for the platform easier.

2.2.5 New hardware

After some time working on the ENP2505 [12], it became clear that Intel intended to discontinue support of the IXP1200 [31] chipset, and also the IXA SDK 2.01 [29], which was the latest SDK version available for the IXP1200 chipset. At about the same time, we received the ENP2611 [13] cards. The decision was then made to switch to the ENP2611, in the hope that the new Intel SDK 3.51 [30] would be a more stable and thorough release than the 2.01 which had proved to be especially challenging to make work smoothly. This,

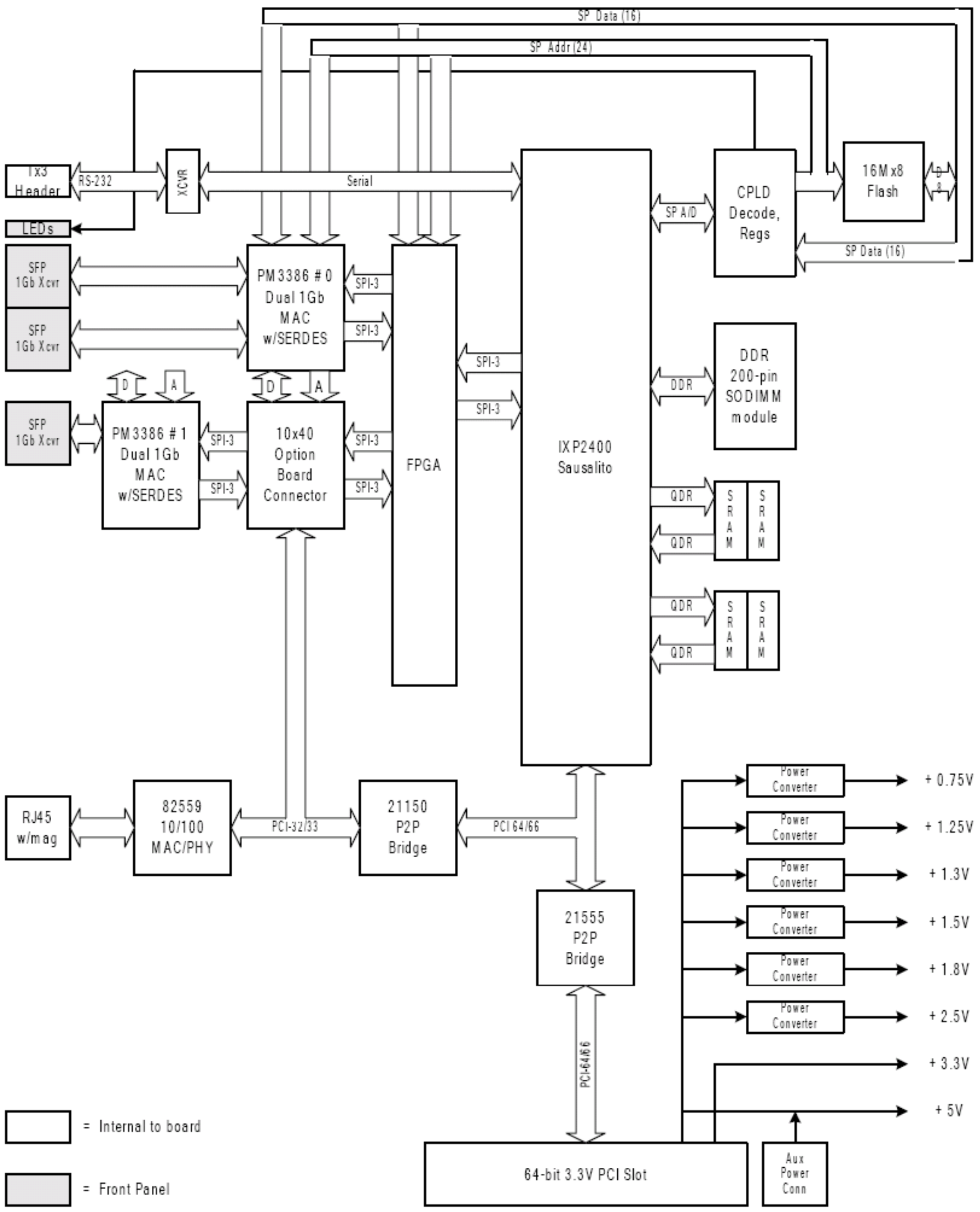


Figure 2.5: ENP2611 Block diagram [13]

combined with the fact that developing applications for a platform that is no longer in use would be of little future value, made it feasible to commence with the hardware change[2].

Component	**ENP2505**	**ENP2611**
Core	StrongARM	XScale
Core clock frequency	232MHz	600MHz
Number of μEngines	6	8
μEngine clock frequency	232MHz	600MHz
Threads per μEngine	4	8
Scratch memory	4KB	16KB
Number of network interfaces	4	3
Type of network interfaces	10/100 Base-T	Gigabit Optical

Table 2.1: Main differences between ENP2505 and ENP2611

The most significant changes in hardware between the two platforms are shown in table 2.1. The structural changes of the Intel IXA SDK versions 2.01 and 3.51 are so significant that porting IXP1200 applications to IXP2400 would have taken excessive work. This task would be further complicated by the fact that the different hardware peripherals would demand different drivers and configuration. The consequence of this was that the applications had to be implemented from scratch on the new platform to make it conform to the new SDK and hardware.

2.3 IXA software libraries and tools

The Intel IXA platform comes with software and libraries to support the programming task. The main bundle is the Intel IXA software development kit (IXA SDK) that provides debug tools, compilers and support libraries. There is also an SDK for the board implementation that contains sample code and board specific drivers. This section will give an introduction to these libraries.

2.3.1 Intel IXA SDK 3.51

The Intel IXA SDK [30] is a set of libraries and reference designs provided by Intel to support programming the IXP chipsets. It also includes some debug tools that interface

[2]This was considered to be profitable even though it ment starting almost from scratch with new hardware and a new microengine programming language (micro-C). The setup and configuration process would also have to be figured out.

with the card, and ease debugging of μEngine applications[3]. The main components of the SDK are as follows:

- **Development Workbench:** The development workbench is a Windows application that can be used to simulate the running of μEngine programs. It can also be used to debug the program directly on the hardware by means of the 10/100 network debug port. Since μEngines cannot directly provide output to screen, this is a tool that is invaluable when it comes to verifying that your program performs as it should. It also has the possibility of simulating a packet stream, so as to see how the application responds to different input. It allows for analysis of the amount of cycles different operations consumes, and to use this to eliminate performance bottlenecks.

- **Software libraries for μEngines and XScale:** Intel's software libraries, both for μEngines and XScale, provides means of executing operations and intrinsics, saving implementation time on several tasks by reducing the necessary number of code lines. There are also libraries for debugging via the XScale, and hardware abstraction libraries for accessing μEngine functionality from the XScale.

- **Compilers, assemblers and linkers for μEngines:** In the IXA SDK 3.5, Intel has added Linux support for the μEngine C language, and provided enhanced compilers and assemblers. This makes development on an all-Linux environment easier than in was using prevous editions like the IXP1200 platform [31][4].

- **Software framework and sample applications:** The software framework is a collection of libraries and sample applications that gives an insight into how different networking functionality can be implemented using standard Intel tools.

There are several layers of abstraction provided with the SDK. On the bottom level microcode instructions can be used to manually set all needed registers. Common instruction combinations are gathered in the intrinsics library for the μEngines. The micro-C language provide an even greater level of abstraction by eliminating the need for assembly-style programming.

When programming for both XScale and μEngines, the hardware abstraction layer (HAL) and operating system services layer (OSSL) libraries provide access to microengine functionality from the XScale. For larger scale programming there are a software framework that manages resources and enables structures and methods for acheiving common networking tasks.

[3]Debug printouts and similar techniques are difficult due to the fact that the μEngines do not allow any screen output. All debugging of this kind will have to go via the XScale.

[4]This was previously only supported for Windows.

We have experienced that the abstractions provided by the software framework often are small, and therefore it may, for some tasks, be better to use the hardware abstraction libraries upon which the framework is built. This is particularly relevant for custom tasks that does not conform with standard network application procedures.

The datatypes used when programming the platform are common for both μEngines and XScale. The byte order of the datatypes is big-endian. The most common datatypes are shown in table 2.2.

Term	**Words**	**Bytes**	**Bits**
Byte	$\frac{1}{2}$	1	8
Word	1	2	16
Longword	2	4	32
Quadword	4	8	64

Table 2.2: IXP2400 data terminology [32].

The IXA SDK is the main tool of development when writing software applications for the IXP2400 platform. Though the software framework is superfluous in many cases, the basic libraries are very useful for development on all layers of the IXP2400 hardware.

2.3.2 Radisys ENP SDK 3.5

Radisys supplies an SDK [33] which provides a range of services that are specific to the ENP2611 board. These services include:

- Drivers for the PM3386 and PM3387 optical interface controllers and for the SPI-3 bridge.
- Sample application code with microblocks for transmitting and receiving packets, that are adapted to the current physical interfaces.
- Additional drivers that provide connection to the host over PCI.
- Kernel image of the Monta Vista Linux [37] operating system that can be used on the XScale.

The drivers for the PM3386m, PM3387 and SPI-3 give us the opportunity of tuning parameters on the hardware devices (like turning on and off hardware ethernet checksum calculation). Prior to running an IXP application, these drivers have to be initialized.

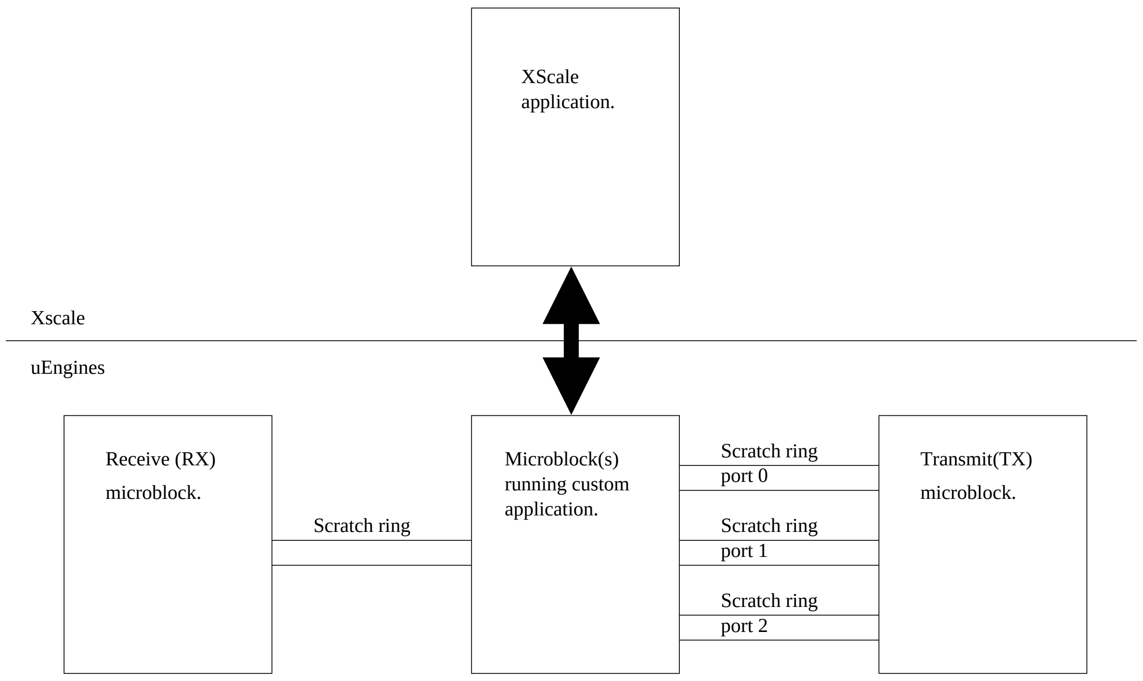


Figure 2.6: RX and TX block functionality and interface.

The microblocks provided have been an important tool. They are in principle the same as the Intel IXA SDK transmit and receive microblock reference designs, but the parameters have been adapted to suit the ENP2611 interfaces. Thus, we can integrate parts of this code to provide us with suitable transmit and receive blocks for our purposes. Figure 2.6 shows a typical application setup. The Receive (RX) microblock receives data from all ports, and passes a reference to the data on one scratch ring (explained later in section 2.3.3). An application on μEngines or XScale can then process the data. To send data, a reference has to be put on a scratch ring. The Transmit (TX) block is designed with three input scratch rings. Which port the packet is to be transmitted on is decided by which scrath ring the reference is passed on. The drivers and board-specific SDK provided by Radisys is an extension of the more extensive SDK provided by Intel to support programming the IXP chipset itself.

2.3.3 IXP programming paradigms

To handle the large throughput that is expected from a networking application, a method that has been proven effective is to give the application a hierarchical structure [6]. Figure 2.7 shows a diagram of the levels of structure. In practice, this means that most of the packets arriving have to be handled by the lower processing levels (i.e., the μEngines). Some packets can be sent to higher levels (XScale) for further processing, and a few packets can be forwarded further (for instance to the host machine). The reason for this is that the performance of the lowest level is highly optimized for effective forwarding. As you look at higher levels, you will find more general purpose processing units that can handle more diverse tasks at a penalty of throughput.

The operational layers are commonly divided in two. The "data plane" handles the high speed processing and forwarding of the majority of network packets. The key elements are real time forwarding and efficiency [41]. The "control plane" involves non-wire-speed general purpose processing that can include table creation and updating, data plane exceptions or computationally intensive tasks [41]. In an IXP2400 setting the data plane typically resides on the μEngines, and the control plane is represented by the XScale.

Both hardware and software design reflects this design philosophy, and as a result, trying to move large datarates between XScale and μEngines will yield poor results[5]. To ensure efficient packet handling, steps must be taken upon implementing applications, so that the majority of tasks are performed on the data-plane (if the tasks are not so processor-intensive that they hamper the data pipeline throughput).

Packet handling As the goal of the lower layers of the network processor is to handle the packets as swiftly and efficiently as possible, it is essential to avoid unneccesary copy operations. Once the packet has been written to DRAM, it resides in the same memory buffer until packet processing is finished, and the packet is transmitted or discarded. What passes between the different processing units is a longword with data (buffer handle) from which the packet location and metadata can be derived.

In the reference design from Intel, the RX and TX microblocks use buffer handles as the means to convey the reference to a packet between processing units. From each buffer handle, the location of two related buffers can be extracted:

- **Packet metadata:** The metadata contains information on the packet size, the port that received the packet, the port that is to transmit the packet, the linking of buffers into larger packets etc (see figure 2.8). This data resides in SRAM. (More details can be found in the IXA framework reference manual [28].)
- **Packet data:** Located in DRAM, this is the buffer where the packet data is actually stored. The RX block reserves some space in front of the packet and after the packet ends in case additional headers have to be appended or prepended.

[5]We have achieved datarates of up to 100 Mbps between XScale and μEngines in experiments using batch queuing of packets on a scratch ring.

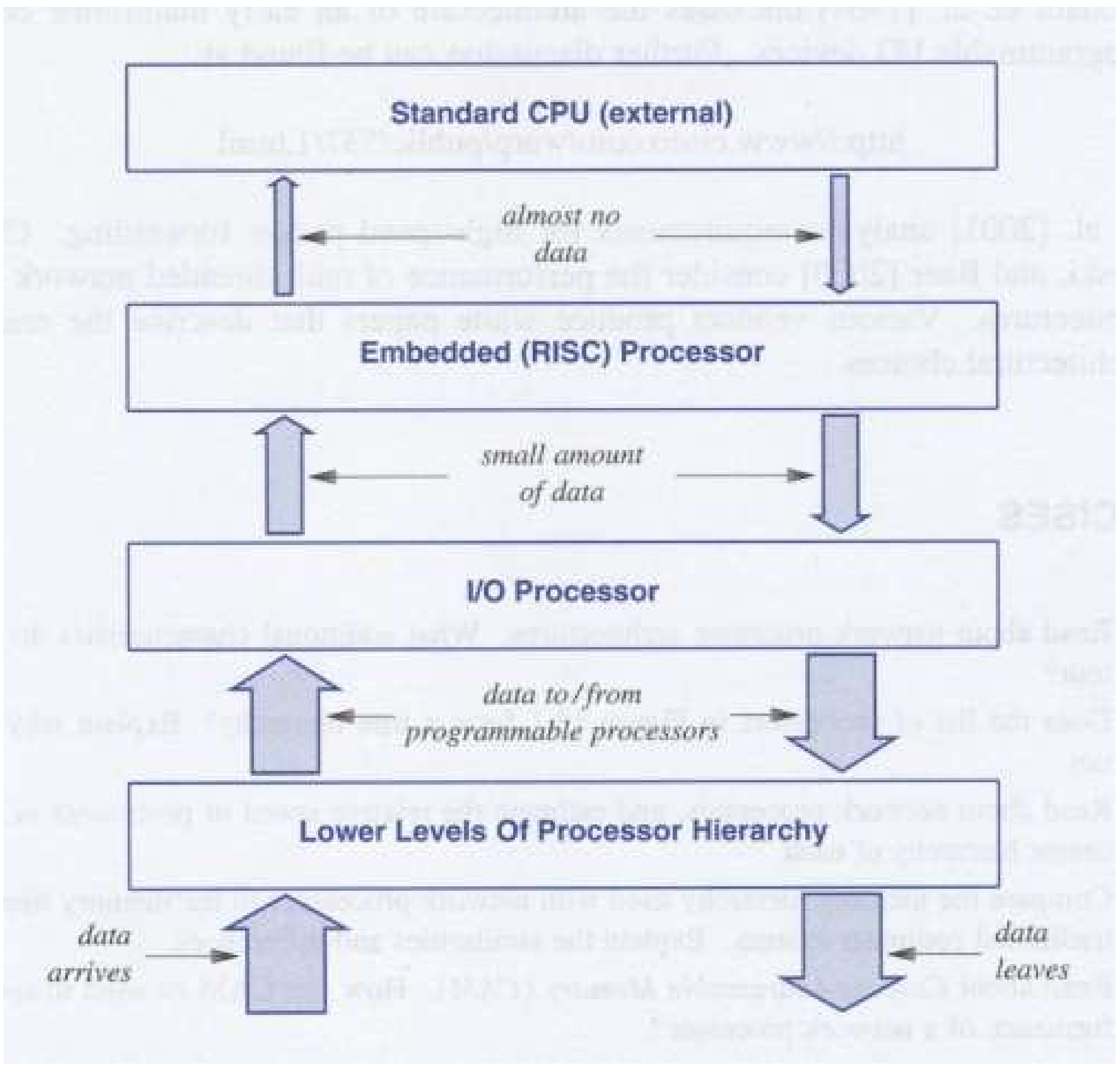

Standard CPU (external)
almost no data
Embedded (RISC) Processor
small amount of data
I/O Processor
data to/from programmable processors
Lower Levels Of Processor Hierarchy
data arrives
data leaves

Figure 2.7: IXA Hierachy model [6]

The metadata buffers, although customizable in size, are typically 32 bytes long. The data buffers are defined by the need for space (2048 bytes would be an appropriate size if you want to store ethernet packets with an maximum transmission unit (MTU) of 1500 bytes). To be able to use the current buffer handle structure, the size of both the metadata buffer and the data buffer have to be power of two. Figure 2.9 give an example of how metadata and data offsets are extracted from the buffer handle in a typical data structure. The SRAM and DRAM offsets calculated from the buffer handle give the starting position of the respective buffer. The packet data in DRAM, however, starts a number of bytes into

```
typedef __declspec(packed) union {
  struct {
    dl_buf_handle_t buffer_next;

    uint16_t buffer_size;
    uint16_t offset;

    uint32_t packet_size :  16;
    uint32_t free_list_id : 4;
    uint32_t rx_stat :      4;
    uint32_t header_type :  8;

    uint16_t input_port;
    uint16_t output_port;

    uint32_t next_hop_id :  16;
    uint32_t fabric_port :  8;
    uint32_t reserved :     4;
    uint32_t nhid_type :    4;

    uint32_t color_id :     4;
    uint32_t reserved_1 :   4;
    uint32_t flow_id :      24;

    uint16_t class_id;
    uint16_t reserved_2;

    uint32_t packet_next;
  };
  uint32_t value[8];/* aggregate for the above fields */
} dl_meta_t;
```

Figure 2.8: Struct describing packet metadata.

the buffer. This is to leave room in front of the packet in case we need to prepend the packet data with a new header. The offset where the packet data begins can be found in the metadata (see the “offset” field in figure 2.8). The 24 least significant bits provides us with the offset of metadata in SRAM and to the data in DRAM. The two most significant bits indicate if the buffer is start of packet (SOP), end of packet (EOP) or both. This is to make sure that packet sizes larger than the internal buffer size can be accomodated. Bits

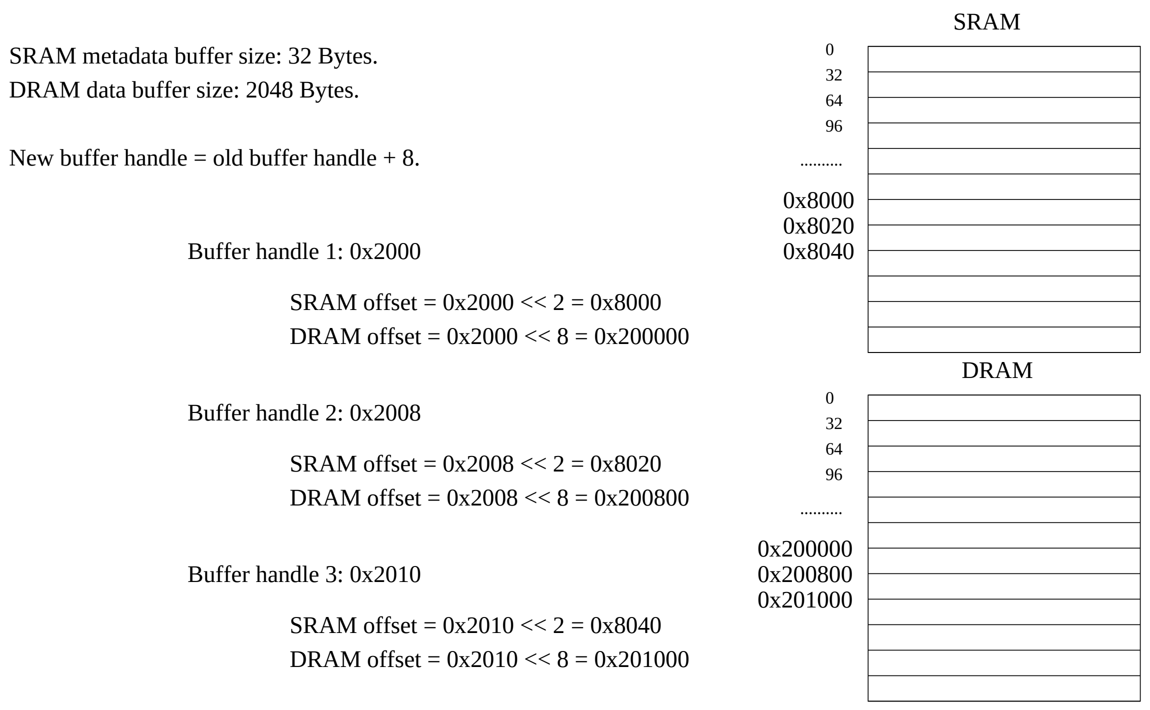

SRAM metadata buffer size: 32 Bytes.
DRAM data buffer size: 2048 Bytes.
New buffer handle = old buffer handle + 8.
Buffer handle 1: 0x2000
SRAM offset = 0x2000 << 2 = 0x8000
DRAM offset = 0x2000 << 8 = 0x200000
Buffer handle 2: 0x2008
SRAM offset = 0x2008 << 2 = 0x8020
DRAM offset = 0x2008 << 8 = 0x200800
Buffer handle 3: 0x2010
SRAM offset = 0x2010 << 2 = 0x8040
DRAM offset = 0x2010 << 8 = 0x201000
SRAM
0
32
64
96
....
0x8000
0x8020
0x8040
DRAM
0
32
64
96
....
0x200000
0x200800
0x201000

Figure 2.9: Calculation of metadata and data offsets from buffer handle values.

24 through 30 state the cell count, indicating how many cells the buffer contains[6].

Scrath ring mechanics The SHaC unit of the IXP chipset contains 16KB scratchpad memory that is accessible to the XScale and all μEngines. Listed below are its most important properties:

- Normal read and write: The memory is accessible on 32 bit boundaries. That implies that you cannot read or write less than 32 bits at a time. You can read or write up to 16 longwords with a single command.
- Atomic read-modify-write operations: You can set or clear bits, increment or decrement, add or subtract in an atomic operation. These operations can also return the pre-modified value of the written data.
- Sixteen hardware assisted rings for interprocess communication: These rings are implemented as FIFO-queues with a head and tail pointer.

[6]Cells are used for Asynchronous Transfer Mode (ATM) processing. When not in ATM mode, bits 24 through 30 can be used for other purposes.

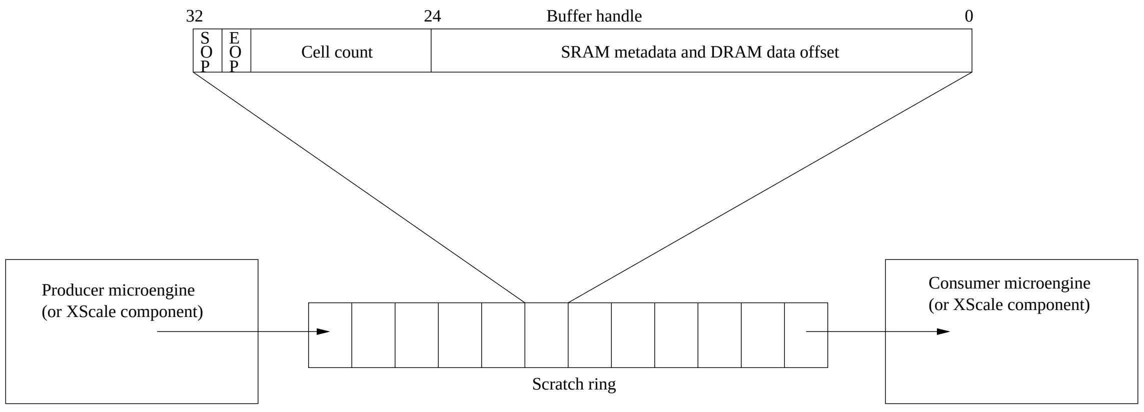

32
24
Buffer handle
0
SOP
EOP
Cell count
SRAM metadata and DRAM data offset
Producer microengine (or XScale component)
Consumer microengine (or XScale component)
Scratch ring

Figure 2.10: How buffer handles are passed on scratch rings between processing components.

The most important generic use of scratch memory is as scratch rings. Given the buffer handle structure described above, you can assign a scratch ring as "egress" for one μEngine and as "ingress" for another one (or the XScale core). It is, in other words, used to facilitate message passing between different structural blocks. Since the location of the packet metadata can be derived from the buffer handle, all necessary information can be supplied by passing this one buffer handle on a scratch ring. Figure 2.10 shows the use of a scratch ring to move buffer handles from a producer component to a consumer component. For optimization purposes, however, it is possible to pass more information about a packet or buffer on the ring. This can save SRAM accesses in cases where every cycle is valuable.

2.3.4 MontaVista Linux

The final piece of software needed to use the ENP2611 as an operational network platform is an operating system running on the XScale core. For the standard ENP SDK, this OS is a small Linux distribution optimized for embedded platforms. The kernel image is developed by MontaVista [37], and provided with the Radisys SDK. The MontaVista preview kit for ENP2611 is, as the name suggests, a limited edition for evaluation purposes. It is based on the 2.4.14 Linux kernel. The hardware drivers provided by Radisys and Intel extend the distribution functionality, and give the programmer full control of the IXP hardware. There is, however, an initiative in the open source community to provide up to date Linux kernels for this platform [34].

2.4 Summary

There is a wide range of possible uses for NPUs. Though their use, up to now, mostly has been in free-standing embedded devices like routers and managed switches, network boards are being developed that can add NPUs to conventional machines. In environments that have need for heavy network processing, these can provide valuable offloading functionality.

The parallel processing done by the IXP chipsets provides effective packet processing pipelines. More complicated tasks, like managing data structures and handling exeptions, can be forwarded to the core CPU, though most packets should be handled on the data plane. The extensive programmability of the IXP platform give us the opportunity of building custom functionality into μEngines and core CPU and adapting the NPU functionality to our needs.

There are many aspects of NPU functionality that have to be explored in order to implement an application which uses the host, the NPU core and NPU data plane. In the next chapter, we will explore some of these key features on the ENP2611 to try to evaluate the performance of the hardware, and how such an application best can be implemented.

Chapter 3

IXP2400 Evaluation

We have, for some time now, been implementing applications on the ENP2505. The ENP2611 was radically different, both in hardware functionality and in SDK structure and use. To be able to efficiently program the IXP2400 hardware platform, and learn the basics of configuring and using the hardware, it was necessary to start with small applications and expand the use as the needed knowledge was aquired. With a hardware platform so different from the programming environments that is common knowledge, aquiring the information needed and successfully applying it was a great challenge. This chapter describes the most important applications developed in this process, and some measurements that have been valuable, both for designing the video cube solution, and for planning IXP2400 application development in general.

3.1 Exploring the IXP2400 hardware

This section describes the applications developed to explore the IXP2400 platform. The first challenge was to be able to successfully transmit and receive packets. The next goal was to modify packet data, and, finally, to do some packet processing on the XScale. The ability to move data efficiently between the host and the IXP was also an area in question (this is discussed in more detail in section 5.4). One of the main challenges was to find methods of implementation that are as simple as possible, but still effective enough to solve the task at hand. This was emphasized by the multitude of different approaches that, judging by the sample code applications provided in the IXA SDK, all would achieve the same goal.

After reviewing the domain of the server cube application, it became clear that the implementation would need hardware resources on three different levels (see section 2.3.3).

The media stream was to be delivered from a server application running on the host machine. Some of the setup and routing structure maintenance had to be done on the XScale, but most of the large-scale packet processing had to be done on μEngines. To ensure that an expedient implementation could be made on the ENP2611 with IXP2400 chipset, the workings of this hardware would have to be explored and some central questions would have to be answered:

- How can operations be implemented efficiently on the μEngines?
- Should we use microcode or micro-C on μEngines?
- How can data be transported to and from the XScale level?
- How large bitrates can be moved to the XScale?
- How can control-plane program functionality be implemented on the XScale?
- How can data be transported to and from the host machine?

The questions regarding the IXP platform, when not exhanging data with the host, is answered in this section, the questions that imply communication with the host machine will be further discussed in section 5.

3.2 Static forward

The key task when designing a network application is the ability to receive and transmit packets efficiently. With the SDK supplied by Radisys, a small application was included that had microblocks designed for these tasks and configured to work with the ENP2611 hardware. This is an application with a three microblock structure as shown in figure 3.1. The microblocks perform the following tasks:

- Microblock 1 - RX microblock: This microblock receives data from the medium. It then reassembles the MPackets (see section 2.2.3) into the original ethernet packets. The packet is written to DRAM, metadata is created in SRAM and the buffer handle (see section 2.3.3) is enqueued on a scratch ring.
- Microblock 2 - Packet echo: In this microblock, 5 longwords, including buffer handle and input port, are read from the RX scratch ring. The input port number is checked and the buffer handle is enqueued for sending on an outbound scratch ring (TX) based on which input port it arrived on: $0 \rightarrow 1$, $1 \rightarrow 2$, $2 \rightarrow 0$.

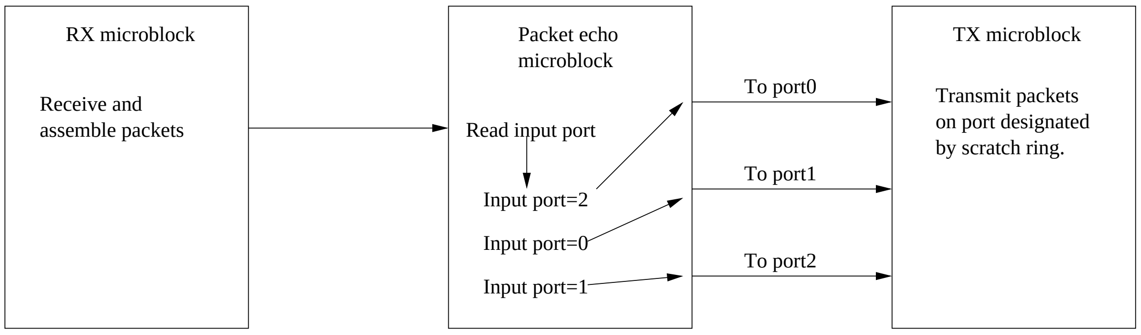


Figure 3.1: Radisys static forward application block diagram.

- Microblock 3 - TX microblock: The buffer handles are dequeued from three different scratch rings by this microblock. An output port is then selected based on which scratch ring the handle arrived on. The packet is split into MPackets, and transmitted on the chosen port.

The RX and TX microblocks maintain statistics on the number of received and transmitted bytes and packets on each port. These variables are shared, and can be read by the XScale application. The XScale application is also responsible for initializing the spi3br and pm3386 (see section 2.3.2) drivers, load μEngine object file images into the μEngines and start the μEngines.

It is worth noting that the RX microblock puts not one, but 5 longwords on the scratch ring bound for the packet echo microblock. The first longword is the buffer handle, the four next are relevant metadata. The reason for this is probably that the receiving microblock will save at least one SRAM access by having received the packet offset on the scratch ring.

After some attempts of trying to design RX and TX code from scratch, it was decided to try to integrate the microblocks from the static forward application into a custom application by replacing the “packet echo” microblock. This proved to be a successful strategy, and enabled the building of the next application.

3.3 IP header switch application

In order to find out more about the performance of μEngines and XScale, a simple application that had to modify packet data was implemented. The first implementation performs all task on μEngines, the second one forwards all packets to the XScale for processing. The designs are described in the following sections.

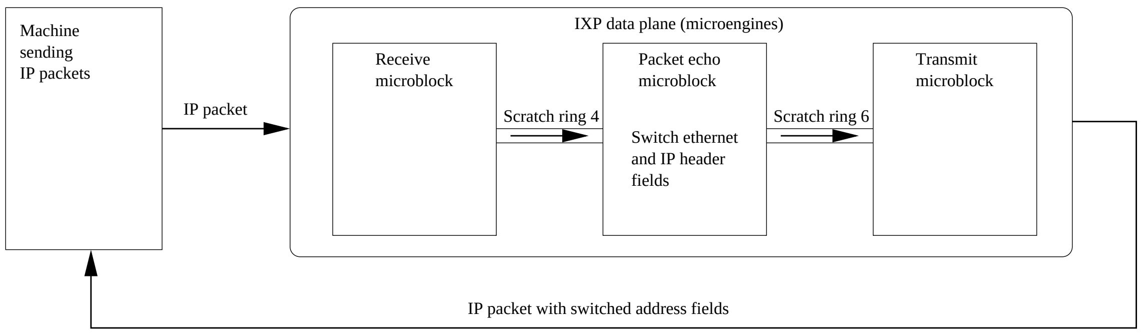

Machine sending IP packets
IP packet
IXP data plane (microengines)
Receive microblock
Scratch ring 4
Packet echo microblock
Switch ethernet and IP header fields
Scratch ring 6
Transmit microblock
IP packet with switched address fields

Figure 3.2: IP header switch application on μEngines.

3.3.1 μEngines implementation

To try to measure the time consumed by a simple application, and to answer the question of micro-C or microcode, it was decided to make a simple modification of the static forwarding application. This application also enabled us to test the mechanisms needed to change the packet data in DRAM from the μEngine level. The following changes were implemented:

- The packet echo microblock (originally written in microcode) was replaced with a version written in micro-C.
- The packet was to be transmitted on the same port it arrived.
- The source and destination ethernet addresses were switched.
- The source and destination IP addresses were switched.

The use of micro-C proved to make the programming of μEngines easier. The code was easier to read and survey, and a breakdown of the microcode produced by the compiler showed that the amount of cycles used was not far from the microcode version.

The greatest change from the static forward application was that this application would actually modify the incoming packets before transmitting them. The RX and TX microblocks from the static forward applications were reused. To avoid having to calculate ethernet checksums in software, ethernet checksumming in hardware was enabled for the interfaces. This was done in the driver initialization of the XScale code. The packet echo block was given one context on one μEngine, running in a loop receiving packets, switching headers, and transmitting packets. Figure 3.2 shows a diagram of the microblock usage and packet flow.

The next thing that had to be explored was the passing of buffer handles to the XScale, and how packet processing could be performed there. From the example applications in the IXA SDK, there seemed to be several feasible approaches.

3.3.2 XScale implementation

To test the communication capabilities between μEngines and the XScale, it was decided to implement the IP address switch functionality on the XScale. In achieving this, a choice had to be made between using the kernel-mode IXA Software framework structure, or to manage with the hardware abstraction layer (HAL) libraries.

The software framework was first tried, in the hope that it would provide greater flexibility and a larger degree of abstraction. The experience from this experiment was that most of the methods introduced a greater aspect of complexity without enhancing functionality, at least not for such simple tasks. One more complicating factor was that the software framework required the application to be implemented as a kernel module, thus excluding the possibility of making use of general user-level libraries in the development. The main argument for running the XScale part of a network application in kernel mode, was to avoid extra context switches (making it more efficient). The nature of IXP hardware (see section 2.3.3), however, restricts the data flow between μEngines and XScale. This means that the bottleneck will not be XScale processing, but the moving of data between XScale and μEngines.

Another consideration that had to be made was that, in the server cube application, data had to be moved between the host and the XScale. One way of implementing this, that looked promising was to use a socket on the ENP2611 debug port (see section 5.4). Implementing in kernel mode would make programming on a socket interface much more difficult. The sum of these arguments led to a decision to redesign the application using the HAL and operating system services layer (OSSL) libraries in user mode. Figure 3.3 shows the basic structure of this application.

The mechanism that seems to be the best for getting the buffer handles to the XScale is to spawn a thread that is dedicated to listening for an interrupt triggered by the μEngine code. There is functionality for this in the HAL. When the interrupt is generated, a given procedure is called. This procedure processes the packet, and if a packet has to be sent to another stage in the processing pipeline, it can be enqueued on the appropriate scratch ring. The μEngine that wants to send a packet to the XScale simply inserts the buffer handle on the scratch ring designated for communicating with the XScale and generates the appropriate interrupt.

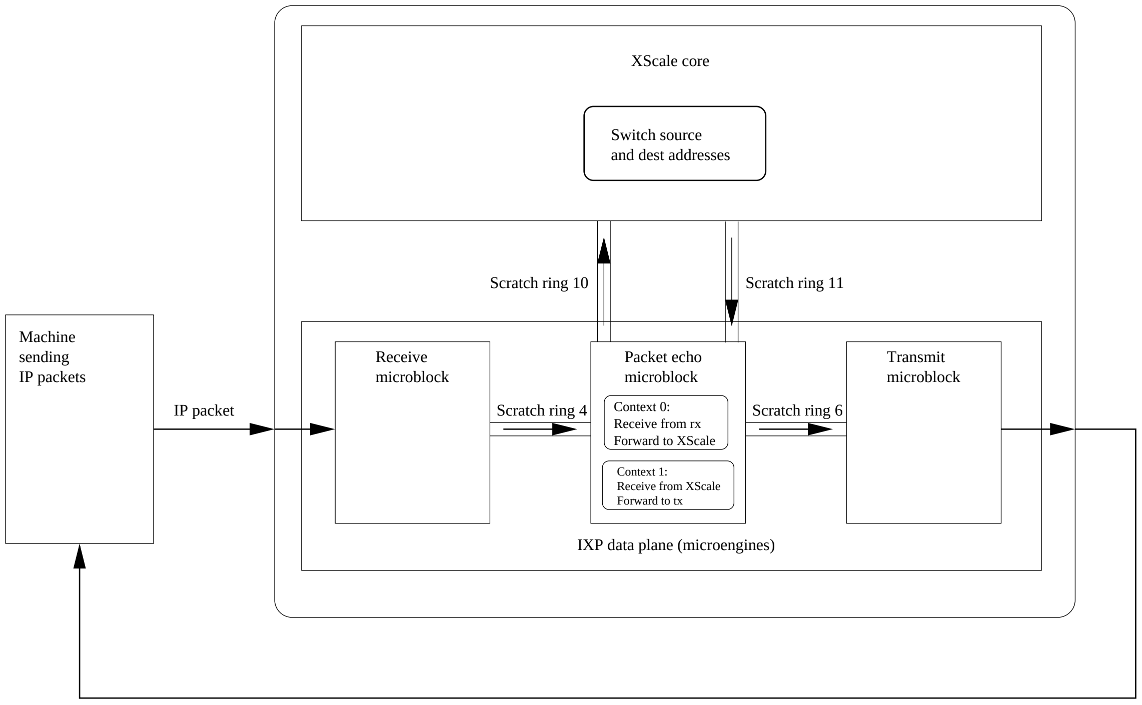

XScale core
Switch source and dest addresses
Scratch ring 10
Scratch ring 11
Machine sending IP packets
IP packet
Receive microblock
Scratch ring 4
Packet echo microblock
Context 0: Receive from rx Forward to XScale
Context 1: Receive from XScale Forward to tx
Scratch ring 6
Transmit microblock
IXP data plane (microengines)

Figure 3.3: IP header switch application using μEngines and XScale.

3.3.3 IP header switch application evaluation

In order to determine the general abilities of the different networking components of the IXP2400, the IP header switch application was created. This application receives an IP packet, switches the IP source and destination addresses, switches the source and destination ethernet addresses, and transmits the packet on the same interface that it arrived on. The nature of IP checksumming (a ones compliment sum), makes it unnecessary to recalculate the IP checksum.

In the following sections, the result of tests performed on this functionality implemented on different areas of the IXP platform and on a Linux host machine is presented. The measurements are done by timing a packet on the way out from one machine. The packet is processed on another machine, and returned. The difference between the send time and the receive time is then taken. The reason why the timing is done on the sending machine, not the processing machine, is to ensure that the measurements will be comparable. The difference in platforms and implementations would lead to results that could not be compared if measured on the processing machine. When looking at the results (in μs) we have to take into account that the times are not only processing time for the machine performing the IP header field switch, but the processing time of the sender upon send and receive

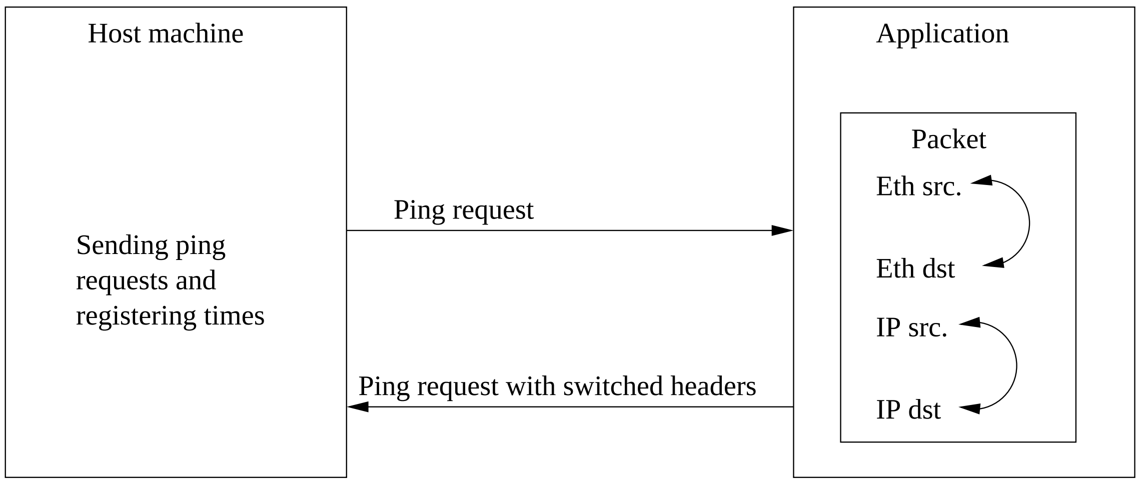

Host machine
Sending ping requests and registering times
Ping request
Ping request with switched headers
Application
Packet
Eth src.
Eth dst
IP src.
IP dst

Figure 3.4: Generic test setup.

as well as the transmission times over network link and busses.

Test setup The purpose of the test is to measure the time used by an application to switch the source and destination IP header fields and the source and destination ethernet header fields. The application that performs the task is implemented in four different ways: On a Linux host in user space, on a Linux host in kernel space, on an IXP card on XScale and on an IXP card on μEngines.

The test itself is done by pinging the interface that the IP header switch application is assigned to. The ping packet will be echoed back to the sender, which will generate a reply. The reply is also echoed to the sender. The measurement of elapsed time is done by having tcpdump listen to the interface on the sending host. Each ping request (or reply) has a sequence number. By calculating the time between ICMP packets with matching sequence numbers, the processing time for the packet pair can be acquired. In this test set, two packet sizes was used: 98 and 1497 bytes packets. This can help us to determine how big part of the whole operation copy operations related to the larger packet size represents. Figure 3.4 shows the generic test setup for the four tests.

The *first* implementation was done on a standard (SUSE 9.3) Linux host with a user space application. To be able to manipulate the IP header freely, a packet socket was created. This socket makes a copy of received packets that complies with the packet socket configuration[1]. The packet is then copied to a local application buffer, and the source and destination IP addresses are switched. The packet is, finally, returned to the

[1]The implemented packet socket was configured to receive all IP packets on eth0 (gigabit interface).

sender via the packet socket. The kernel ethernet layer takes care of prepending the packet with the correct ethernet header, and the packet is then transmitted on the interface. In order of preventing the kernel from generating replies to the received packets (the user space application should have total control), IP packets on the given interface is firewalled. Figure 3.5 shows a diagram of the test setup for this implementation.

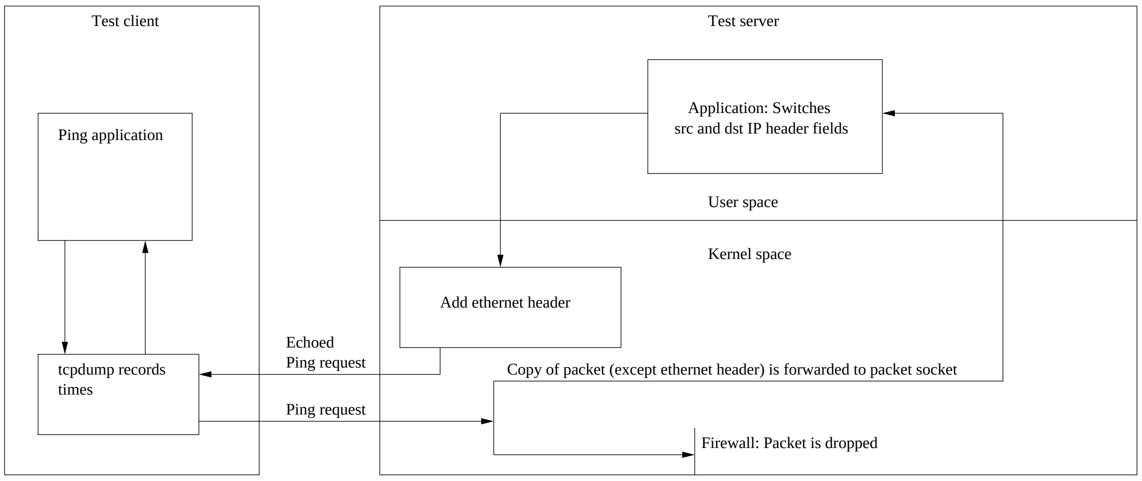

Test client
Ping application
tcpdump records times
Test server
Application: Switches src and dst IP header fields
User space
Kernel space
Add ethernet header
Echoed Ping request
Copy of packet (except ethernet header) is forwarded to packet socket
Ping request
Firewall: Packet is dropped

Figure 3.5: User space packet echo application test setup.

The *second* implementation uses Linux iptables to perform the header switch in kernel space. This eliminates the need of context switches and copy operations from kernel to user space. There is, however, a possibility that the complexity of iptables state matching and routing operations can slow the process down a little compared to what an application dedicated only to doing this one task can perform. The test was done on the same host as the user space implementaton test. In figure 3.6, the test setup for this implementation is shown. The iptables NAT PREROUTING table rule rewrites the IP header destination field to the sending machine's address. The packet is then sent to routing. The Linux routing table directs the packet back to the interface on which it was received (based on the rewritten IP header dst field). Finally, the iptables NAT POSTROUTING table rule rewrites the source IP header field with the IP address of the local machine, and the packet is transmitted.

The *third* test implementation uses the IXP card, but performs the header switching task on the XScale core. The task is completely independent of the Linux Host, and all steps are performed on the IXP platform. Figure 3.7 describes the test setup for this implementation. The application receives the packets in the RX block and forwards them to the packet_echo block. They are then sent to the XScale core by calling the "A" interrupt. The XScale core takes care of switching source and destination headers, and sends the

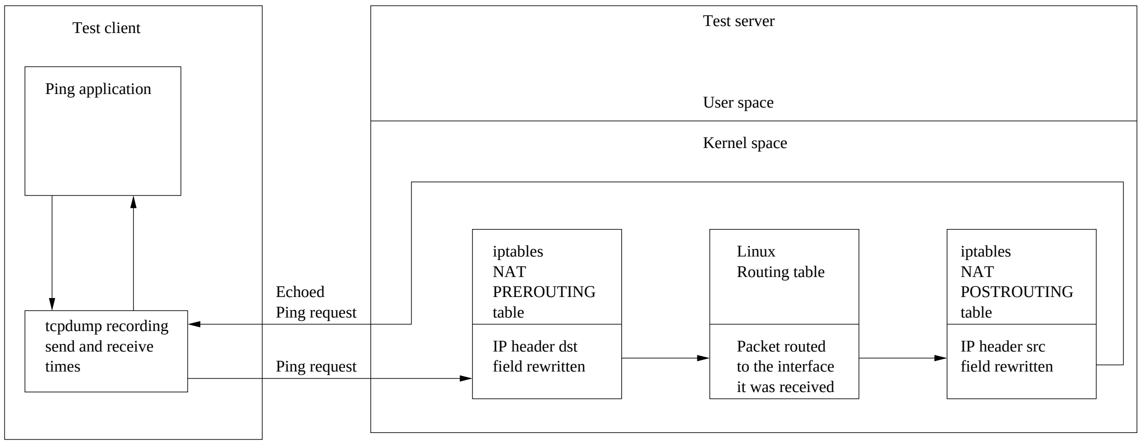

Test client
Ping application
tcpdump recording send and receive times
Echoed Ping request
Ping request
Test server
User space
Kernel space
iptables NAT PREROUTING table
IP header dst field rewritten
Linux Routing table
Packet routed to the interface it was received
iptables NAT POSTROUTING table
IP header src field rewritten

Figure 3.6: Kernel space packet echo using iptables.

packet back to the packet_echo block. The packet is finally passed to the TX block and transmitted on the same port it was received.

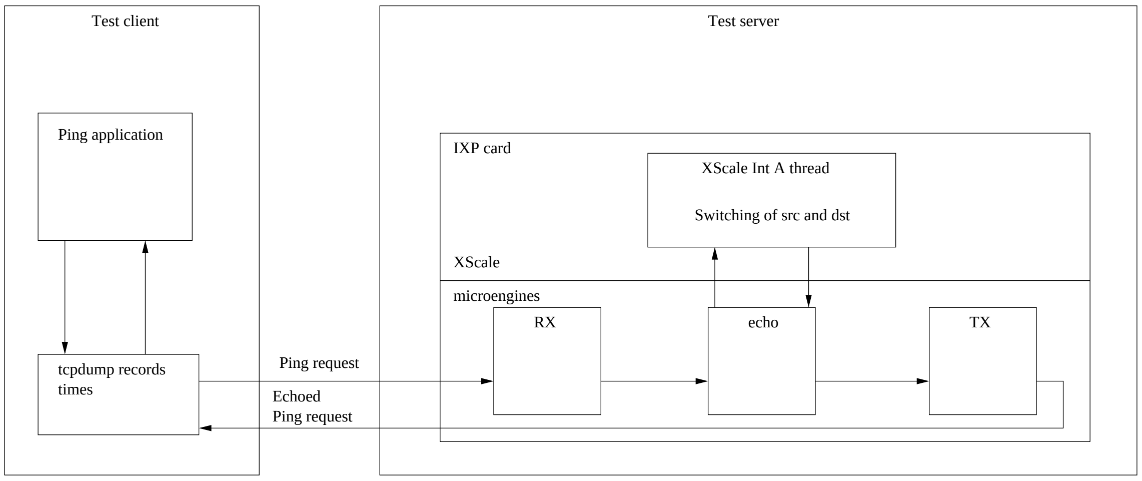


Figure 3.7: IXP XScale ICMP echo test setup.

In the *fourth* IP header switch implementation to be tested, all packet processing is done on the IXP μEngines. In an application such as this, with limited processing power needs, we can expect a fairly efficient processing due to the parallel processing capabilities on the μEngines. If the need for processing power should increase, it is possible to distribute processing further. In this implementation, however, the task of switching IP source and destination addresses (and mac adresses) is done on a single microengine. A diagram of the test setup is shown in figure 3.8. The ICMP packets are received by the TX block, and forwarded to the packet_echo microblock. The ethernet source and destination fields are switched, then the IP source and destination fields are switched. The packet is then sent to the TX block to be transmitted back to the host who sent it.

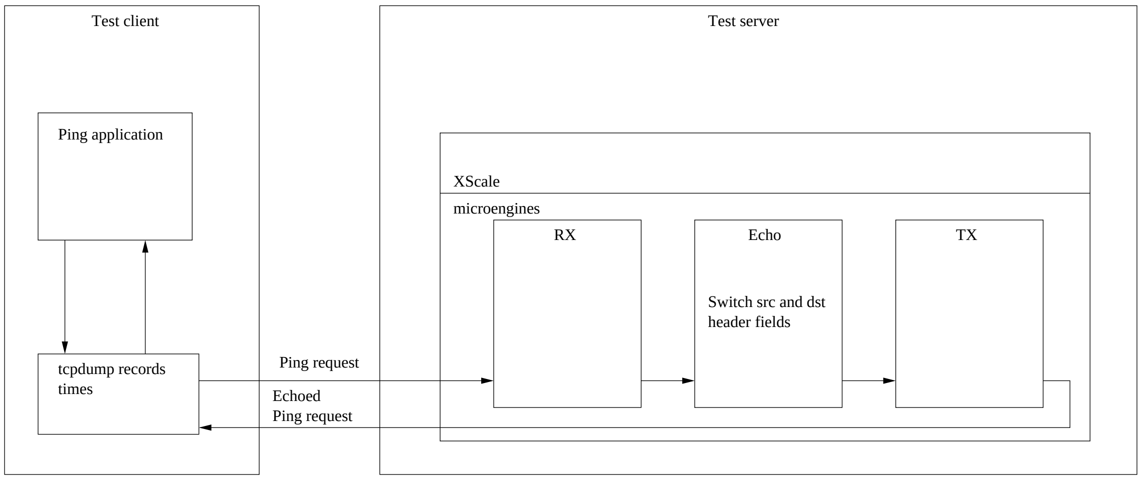


Figure 3.8: IXP μEngine ICMP echo test setup.

Packet size	Max time	Min time	Avg time	Median	Std. dev.	Count
Linux host user space implementation						
98 Bytes	19331 μs	11 μs	111 μs	105 μs	310 μs	99380
1497 Bytes	16216 μs	16 μs	172 μs	174 μs	102 μs	99960
Linux host kernel space implementation						
98 Bytes	19723 μs	11 μs	111 μs	101 μs	378 μs	98902
1497 Bytes	19092 μs	16 μs	168 μs	178 μs	182 μs	99806
IXP XScale implementation						
98 Bytes	1145 μs	54 μs	123 μs	109 μs	44 μs	100000
1497 Bytes	1146 μs	110 μs	171 μs	173 μs	45 μs	100000
IXP μEngines implementation						
98 Bytes	1169 μs	28 μs	98 μs	99 μs	38 μs	100000
1497 Bytes	1150 μs	84 μs	151 μs	162 μs	50 μs	100000

Table 3.1: IP header switch application times.

Evaluation The purpose of the above tests was to compare the processing time of the same functionality implemented on the IXP platform and on a host machine. Table 3.1

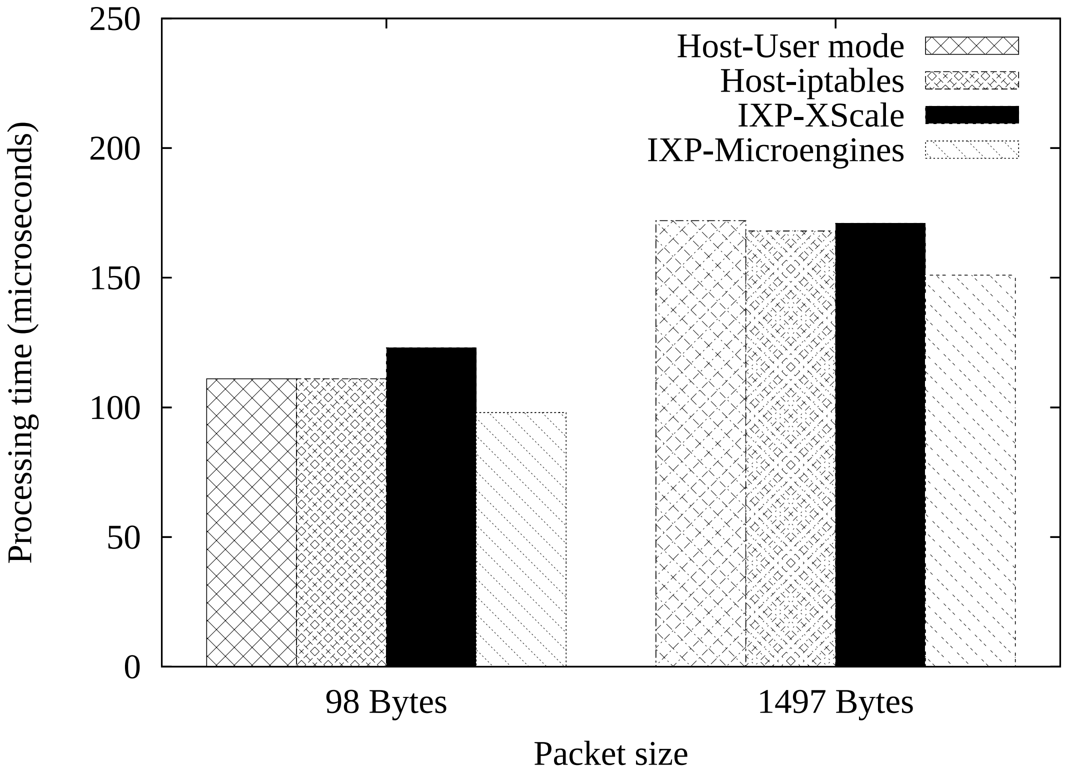


Figure 3.9: Comparison of average times for the IP header switch application.

shows the times and statistical data from the four IP header switch tests. A comparison of the average times from the tests can be found in figure 3.9.

The fact that there are similar results on the iptables implementation and user space implementation for the 98B/packet tests on the host can possibly be ascribed to the string matching support of iptables that makes it necessary to do the packet inspection on the application layer [5]. The routing process will also consume some cycles. In the user mode application, no other logic than switching the headers is applied. The packet is copied directly from the network layer, and immediately returned, thus bypassing several costly operations.

The method that produced the highest processing time for the 98B/packet tests, was the XScale implementation. This is not unexpected as the architecture is not supposed to do processing of high-performance tasks on this level, but rather handling of exceptions and data structure updating (see section 2.3.3).

It is reasonable to assume that the applications will have to spend more time on copy operations for the larger packet size, and that this is what evens the numbers somewhat.

We can see that the overall times have increased for all implememtations when the packet size is 1497B. The average processing time is very similar for the user space implementation, the iptables implementation and the XScale implementation when the packet size is 1497B. The reason why the XScale results have improved in comparison to the other two may be attributed to the fact that no copying of the packet occurs on the IXP platform after it is received. Thus, more resources should be saved when the packetsize is larger. The margin between the μEngines implementation and the other three has also increased slightly. The difference between the XScale and μEngines results for 1497B is, however, very close to the difference between the same implementations with 98B.This makes sense when considering that the cost of copy operations is the main thing separating the two experiments, and the fact that no excessive copying takes place on either of the IXP implementations.

The IXP μEngine implementation has the shortest average time of the four by a reasonable margin for both packet sizes. This is probably due to the fact that none of the components in the processing pipeline has to wait for any system resources other than the read and write operations. The system is entirely and solely dedicated to the one task.

Although the processing speed of each functional unit of the IXP is slower than on the generic computer used in the test, it seems that the fact that the platform is dedicated to this task gives an advantage in comparison to the host implementation. For the implementation that had to pass all the packets to the XScale, however, the results were poorer. This matches the expectations outlined in section 2.3.3 about IXP programming paradigms. For the implementation of other applications, this implies that packet altering microblock applications does not stand back in performace to similar implementations on a generic host, but that XScale processing should be used for exception packets. The XScale was, however, not so slow as to exclude using it to handle situations that could arise quite frequently.

Offloading network functionality to μEngines will, in other words, not only free host resources, but also improve on the processing speed as compared to the what the host would be able to handle.

3.4 Performance gain by offloading

In this section, the possible system performance gain by offloading network services will be tested and discussed. As presented in section 2.1, the offloading of network tasks is a sensible step to take in order to free more system resources on the host.

As the host is given extra network load, an increase in the elapsed time spent on other processes is expected due to the fact that both the kernel and, in most cases, user space applications will have to process network data. When offloading the network task to the IXP, there should be no noticable increase in the processing time.

The following set of tests will try to indicate what the expected gain from offloading a "low-cost" task to a network processor can be. "Low-cost" in this context is that the process of switching IP source and destination header fields of ICMP packets requires little computational power. The fact that transport layer protocols are not involved is also a simplifying factor.

3.4.1 Test

The way the tests were conducted was to measure the time spent in the process of compressing a set of folders containing about 870MB of data with the "tar" command. While this is done, the system is subject to flood pinging with packets of 1497 bytes. The ping packets are processed with the IP header switch applications described in section 3.3.3. The time spent was measured by running the "tar" command through the "time" command. This gave three results for each experiment:

- Real time: The total elapsed time for the command to run. This includes user space time, kernel space time and the time the process is swapped away, and not running.
- User time: The active time the process has used in user space.
- Sys time: The active time the process has spent in kernel space.

A total of four tests was done, all with different types of IP header switch applications running on the tested host. Table 3.2 shows the data for the performed tests. The statistical data for the first test shows how much time was spent on the process with no extraordinary network load. This should be used as a reference. In the second test, a user space application is used to switch IP headers on the received network packets while the tar process is running. The third test's results display the times for the tar process while the IP header source and destination field switch was handled by iptables in the kernel. In the fourth test the switching is offloaded to the IXP card, and performed on the μEngines. Any interaction with the host system that may occur is in order to read from the NFS file system.

3.4.2 Discussion

Figure 3.10 shows the average load times for this test divided in real time, user time and sys time. Figures 3.11, 3.12 and 3.13 show a zoom in of the top of each group from figure 3.10.

Time class	Max time	Min time	Avg time	Median	Std. dev.	Count
Reference time - No extraordinary load.						
Real time	153.07s	132.57s	142.42s	142.35s	4.0946s	201
User time	74.5s	73.16s	73.41s	73.39s	0.1367s	201
System time	6.94s	6.32s	6.6105s	6.61s	0.1152s	201
User space IP header switch.						
Real time	165.53s	142.12s	153.43s	153.61s	4.7797s	201
User time	80.17s	77.18s	77.62s	77.57s	0.3219s	201
System time	10.23s	7.57s	9.3971s	9.44s	0.4537s	201
Iptables IP header switch						
Real time	162.21s	136.06s	146.56s	146.46s	4.6399s	201
User time	76.89s	75.85s	76.2s	76.18s	0.1639s	201
System time	8.71s	7.36s	7.8907s	7.88s	0.2067s	201
IXP μEngines IP header switch						
Real time	156.13s	130.68s	141.12s	141.13s	4.2010s	201
User time	73.88s	73.05s	73.3687s	73.36s	0.134s	201
System time	6.83s	6.14s	6.4602s	6.46s	0.127s	201

Table 3.2: Time used for tar process with different IP header switch implementations.

Because offloading the IP header switch application to the IXP card should leave the host undisturbed to attend to other tasks, the “no-load” tests should yield the same results as the IXP-implementation. There is, however, a slight difference in the numbers. It seems that the times for the IXP implementation tests are slightly lower than for the “no-load” tests. This is probably due to the difference in network setup that is necessary to perform the tests. This can lead to system services slightly changing behaviour, and the results may therefore be somewhat affected.

The trend for the user space IP header switch implementation test is as expected. The numbers tells us that when subjected to this network load, the time used by the tar process increases by 8,73 percent compared to the IXP implementation test. The user space load times and kernel space load times confirm this tendency.

The iptables implementation load test times show a significant improvement over the user space implementation tests. It is still somewhat higher than the IXP implementation test times. The elimination of costly context switches, and also copy operations can probably explain most of the improvement. The elapsed time for this test is still 3,86 percent slower than for the IXP implementation test.

The IP header switch application applied to ICMP packets is a very simple task. For more complex protocol handling tasks, the difference in times would be bigger since the kernel tasks would have priority over the tar process. The potential of the IXP card to handle the

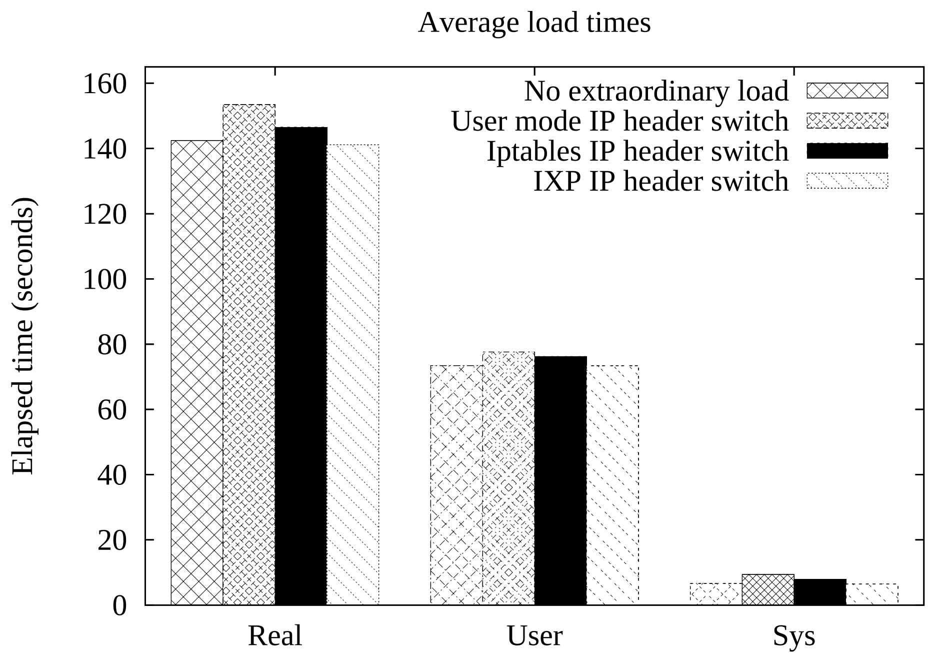


Figure 3.10: Times spent on tar process with different implementations of IP header switch application.

more complex tasks with equal efficiency (due to pipelining) increases the probability of even higher benefits from offloading tasks.

3.5 Summary

In this chapter we have evaluated the ENP2611 NPU board. The evaluation have consisted of tests related to packet handling on μEngines and on theXScale. We have also examined the possible performance gain by offloading simple network functionality.

The IP header switch tests confirm the control plane / data plane implementation model, but also indicates that reasonable processing speed can be achieved on the XScale. This has to be taken into consideration when partitioning tasks between IXP system layers. The load tests shows that offloading network tasks to a NPU will free considerable resources. When the bandwidth increases, this gain will probably be crucial to high-throughput applications like a multimedia server.

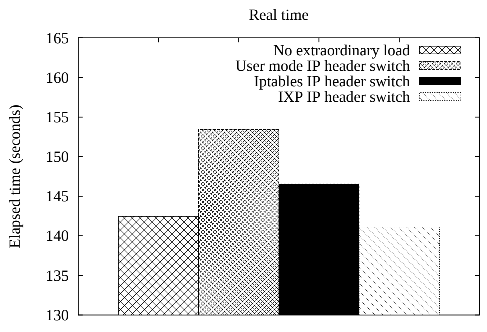


Figure 3.11: Average total time spent on tar process.

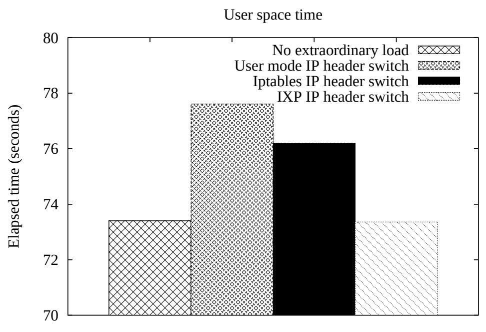


Figure 3.12: Average time spent in user space for tar process.

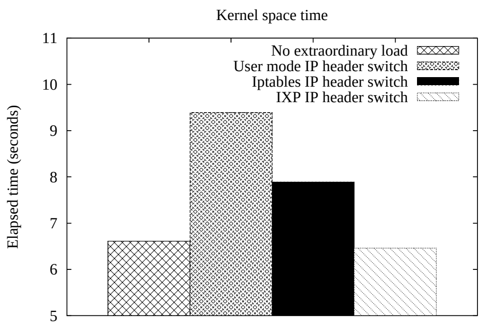


Figure 3.13: Average time spent in kernel space for tar process.

Chapter 4

Multimedia systems

In order to understand the mechanisms needed to implement a solution in the area of multimedia systems, we have to take a look at the properties of multimedia systems and the challenges they present. This chapter will present some multimedia applications and the requirements needed by such systems. An overview of some existing solutions and related protocols will also be presented.

4.1 Multimedia applications

One attempt to define the term multimedia tells us that it is more than one concurrent presentation medium [40]. In this thesis, we will refer to multimedia as one or more presentation media delivered from one computer to another over a network connection. Note that this does not exclude the cases where only one medium (i.e., only video) is delivered. The main point is that the systems that is described has the functionality required to deliver more than one type of media.

There exists many different multimedia applications today, and new, inventive uses are steadily being developed. Some common areas of use today can be:

- **Teleconferencing:** Giving people located physically far apart the possibility of speaking to each other and see the other participants of the meeting. Previously used with only sound over regular phone lines, teleconferencing over data links opens up new possibilities like sharing presentations, documents or mark up a common whiteboard.

- **Video on demand (VoD):** Video (and audio) delivered to a client upon request. This gives the customer the possibility of requesting a video whenever he/she wants.

The greatest challenge in this area has been the large datarates involved. However, the improvement of the network infrastructure makes these solutions more common. An example of an operative VoD site is SF-anytime [58].

- **Audio on demand:** Due to effective compression techniques like RealAudio [53], mp3 [39] and Ogg Vorbis [45], the streaming of audio has been a familiarity on the internet for several years. Common streamed audio services can be radio channels broadcasted over Internet, or sound used to add effect to a website. The data stream can also provide metadata for the audio transmitted (like telling you which radio program you are listening to).

- **Gaming applications:** The number of people using online games and virtual communities have exploded the last years. As an example, there are more than 2 million users of the role playing game “World of Warcraft” [62]. The data that has to be transmitted is mainly information about player actions and environment, although pictures, sound and executable code are also common to exhange. In this field there are a lot of challenges related to keeping the game world consistent to all players, and to minimize the data that has to be passed to each player.

- **Education:** There are many ways to use multimedia services for educational purposes. Students can be shown presentations, ask questions (written or by audio transfer), solve graphical tasks or participate in discussions. Lectures can be transmitted, both live and on demand, allowing students in remote locations the same possibilities as those able to travel to the lecture location.

These are only some of the myriad of multimedia applications that are in use today. With the ascent of common broadband networking, different multimedia content delivered via network to end users are increasingly common. This presents us with some challenges, given the fact that multimedia usually have some properties that are different from the kind of data that has been the most common to access over the Internet until recently.

4.2 Multimedia requirements

When a server has the task of delivering multimedia content, there are some differences in requirements compared to most other networking applications. The most important differences are:

- High data rates: Each second of a movie combining audio and video will require a relatively large amount of data to be transferred per time unit over the network in order to yield satisfying results.

- Large amount of data: The data that have to be delivered occupy much storage space, and the internal server operations needed to handle the data will consequently be costly (due to copy operations, buss transfers etc.).

- Many concurrent streams: If several users request content at the same time, the server will have to provide output of very high bit rates to many streams.

- Time-sensitivity: The frames delivered will have to reach the client within certain time-limits to ensure that the presentation will be satisfactory. In applications with several interacting participants, like teleconferencing or online games, the data has to reach all participants in due time in order to make the application work.

In addition, there are challenges related to the combination of different types of media into streams. As an example, there can be a video conference that includes a presentation, video and audio. All of these components have to be delivered, in time, to all of the different participants.

Two very common media types are audio and video. An example of audio data is a 16 bit, 44KHz compact disk (CD), with a play time of 77 minutes and 20 seconds. The data on this CD will occupy 780.56MB [14]. If we were to stream this data uncompressed, the bitrate would be 1.3Mbps. This, however, is vastly inefficient. To solve this problem it is usual to compress the data before transmitting. Audio data like this can be compressed without losing any of the original sound data. Using free lossless audio compression (FLAC) [14], this audio file can be reduced to the size of 413.46MB, giving us a bitrate of 713Kbps. Applying a lossy compression we can further reduce the needed bitrate. With Moving Pictures Expert Group (MPEG) 1/2 Layer 3 (MP3) compression, the bitrate can be reduced to 128Kbps without noticeable loss of audio quality (as subjectively estimated by the listener) [16][1].

Video streams will require even higher bitrates than audio. A phase-alternating line (PAL) DVD standard can have a resolution of 720x576 pixels, and a framerate of 25 frames per second. This will give an uncompressed bitrate of about 249 Mbps (given a color depth of 24bit/pixel). This would be almost impossible both to store and send efficiently with todays technology. On regular DVD records, however, the data is compressed with MPEG2. This gives a maximum bitrate of about 9 Mbps for the video stream [11]. With the help of MPEG 4 encoding, the bitrate can be brought down to 1-2 Mbps without apparent loss in visual quality [9].

Even with the help of advanced compression techniques, we can see of the above examples that the datarates involved are still formidable. This spurs us to find ways of implementing server solutions that are able to meet the demands that multimedia applications presents.

[1]When lossy compression is applied, it is impossible to restore the data to its original form.

4.3 Improving server performance

There are several ways to improve operating systems and traditional server models to enhance performance when applied to multimedia content. Some of the approaches are:

- Enhance server internals and software to allow greater efficiency (on one server). Such enhancements can be to optimize disk performance by placing data on the disk in ways that is ideal for the type of media [24], or to eliminate copy operations when moving data from disk to transmission medium [21]. Enhancements can also be made to how different protocols are processed, saving precious cycles.
- Build server clusters that behave like one server as seen from the outside. These can be directly interconnected, or connected by a switched network topology. Examples of existing solutions are nCube n4x [42], IBM VideoCharger [25] and Oracle interMedia [46].
- Distribute delivery through proxies. Examples of systems with support for this are Apple Darwin streaming server [8] and Komssys [36]. This kind of approach can be further enhanced by applying multicast techniques like gleaning [23] and patching [20].

In addition, there are different ways to combine one or more of the above components to accommodate the needs of the system that is to be designed.

Next, we will present some server implementation strategies, and show how these deals with the challenges that comes from multimedia requirements..

4.4 Multimedia server implementations

There are may strategies that are applicable when it comes to implementing adapted multimedia servers. In this section, three implementation strategies will be described, namely single server, server clusters and proxies. It has to be said that these strategies are not mutually exclusive. A good single server implementation can be multiplied to build a server cluster structure, which again can be enhanced using a proxy strategy.

4.4.1 Single server implementation

There are many multimedia single-server implementations on the market today, although most of them can be combined to form different types of clusters. Server implementations that are in production today are amongst others: “Quicktime” [50], “Real Helix Server” [54], “Alex Arachnid” [2], “Apple Darwin” [8] and “IBM VideoCharger” [25].

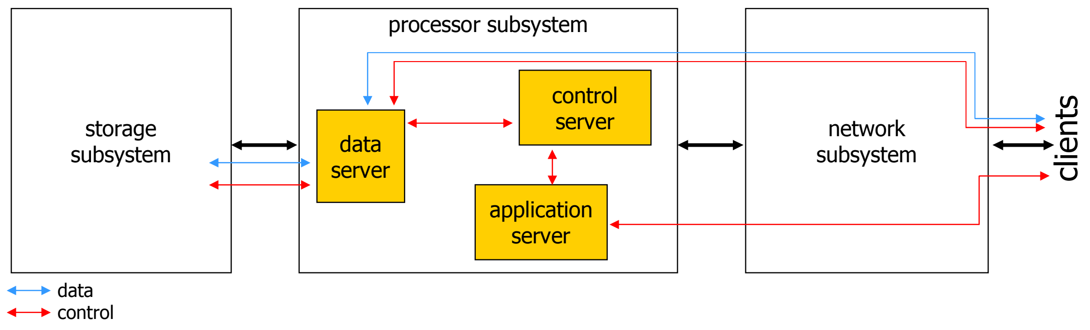


Figure 4.1: Simple general server architecture [23]

When deploying a single server to serve a (potentially) large amount of users, it is common to spend large amounts on hardware (e.g., disks, memory and CPU power) to allow the server to handle periods of great traffic. Most of these resources will then be idle most of the time. Another aspect of the single server solution is that it scales poorly. That implies that hardware upgrades, or even buying a new machine, will be required to meet demand.

Figure 4.1 depicts a common layout for a multimedia server. The diagram divides the server components in three main parts, namely storage, processor and network subsystems. The storage subsystem keeps the data that is to be transmitted, and the networking subsystem handles the actual transfer of data to the client. The processor subsystem is responsible for all the tasks required to control and synchronize the operations. The diagram also divides the typical tasks of the processor subsystem in three; Data server, application server, and control server. The application server functionality is typically the interaction with the user. This part can give a list of available material, organize billing and keep track of users. The data stream itself is delivered by the data server. The control server can be used to guide the whole process, synchronize the operations, and make sure only valid, authorized transactions are committed.

In order to make the servers better adapted to multimedia transactions, there are many alterations that can be made to architecture and operating systems. File systems that specialize on continuous data and large files can be implemented. Examples of specialized file systems are Minorca, Fellini and Presto [23]. We can use multi-processor (and multi-core) architectures to make data processing more effective. Operating systems can be modified to make the data-path more efficient (i.e., DROPS [10] and INSTANCE [47]). These kind of alterations, especially the ones that imply custom built hardware or over-provisioning, are expensive. In order to make cost-efficient server solutions, it is common to use other strategies to meet the scaling obstacles.

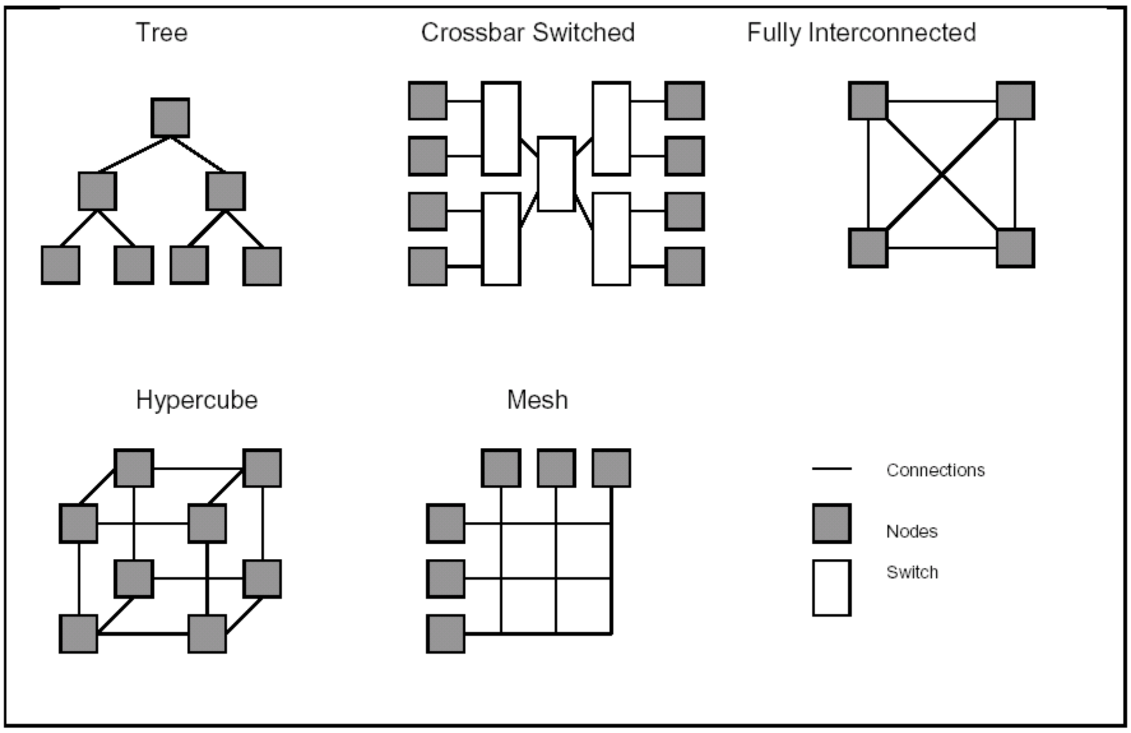


Figure 4.2: Interconnection topologies

4.4.2 Server cluster implementation

To meet the demand for scalability, it is possible to distribute the multimedia server over several interconnected computers. These systems can behave as one server as seen from the outside, but will internally distribute the tasks between several machines. It is also possible to distribute the content between several physically separated servers.

Different topologies offer different benefits to the cluster. This, and the cost of implementing the system (number of connections per node, cost of switches, etc.), must be taken into consideration when deciding upon a topology. Figure 4.2 shows some popular interconnection topologies [19]. The fully interconnected solution, where every node has a direct connection to every other node, is the one with the greatest capabilities in terms of communication, but it is also the most expensive to implement in most cases [19]. From this extreme there are interconnect topology variants which have to pass some data through other nodes to reach its target. For the tree topology, the root is the access point of all data, and therefore a possible bottleneck. Mesh or hypercube topologies will have more evenly distributed data, but will require more complex search and routing algorithms. Switched networking topologies are well-known, and scales well. The switching mechanisms, however, increases the inter-node latency.

In order to be scalable, most video server solutions have implemented support for clustering in one way or another. This kind of support can range from simple load-balancing algorithms which divide the load between different free-standing servers to more complex interconnection topologies and strategies. Examples of server solutions with support for clustering is “Real Helix” [54] and “IBM VideoCharger” [25]. There are also server solutions that are not based on switched network clustering, but on direct interconnection topologies. An example of this is the “nCube n4x” [42] server solution [4]. This cluster solution is based on a hypercube topology, with directly interconnected nodes.

4.4.3 Proxies

The main idea of proxies is to move the most popular content closer to the end users, and in this way reduce traffic on the main content server(s) and backbone network. A statistical rule of thumb is that a great majority of the requested content is represented by just a few of the most popular titles [63]. This empirical observation makes it possible to design caching strategies that make proxies keep the most wanted media, and forward requests for less popular media upward in the media server hierarchy. Another effect of this strategy is that the proxy servers can actually be moved closer to the end user physically, thus reducing network traffic.

Figure 4.3 shows a possible proxy server layout. In this diagram, the master servers have available all the offered content. When requests are made from the end users, the requests are sent to the master servers which provide the content. The in-between proxy servers can then begin to cache the most requested titles of their region. Multimedia server solutions that support proxying include “Real Helix” [53] and “Komssys” [36],

4.5 Hypercube/n4x server solution

In the myriad of different architectures and server topologies, one system that possesses many of the wanted characteristics is the c-cor n4x [4] (previously nCube n4x [42]). The solution is, however, based on custom hardware, and is quite expensive to deploy. This section will try to describe some of the key aspects of the n4x solution.

4.5.1 Hypercube multicomputer structure

This subsection will describe why a hypercube is a good base for an interconnection topology. To show this, we first have to explain the geometrical properties of a hypercube

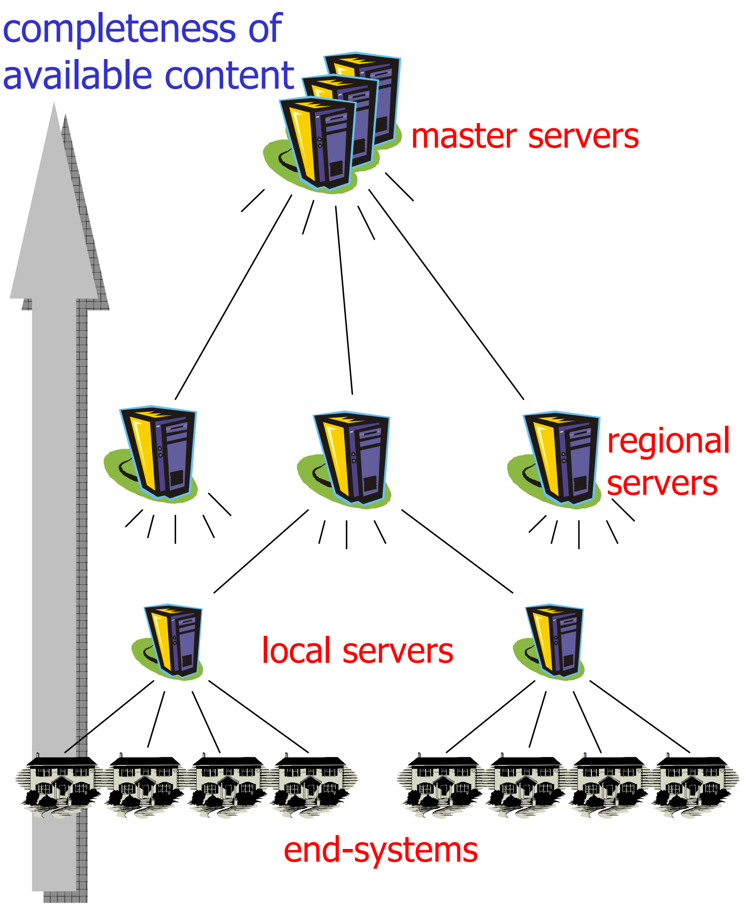


Figure 4.3: Diagram of a possible proxy layout [23].

structure. This is the “Free On-line Dictionary of Computing” (foldoc) [15] definition of a hypercube:

> A cube of more than three dimensions. A single ($2^0 = 1$) point (or "node") can be considered as a zero dimensional cube, two (2^1) nodes joined by a line (or "edge") are a one dimensional cube, four (2^2) nodes arranged in a square are a two dimensional cube and eight (2^3) nodes are an ordinary three dimensional cube. Continuing this geometric progression, the first hypercube has $2^4 = 16$ nodes and is a four dimensional shape (a "four-cube") and an N dimensional cube has 2^N nodes (an "N-cube"). To make an N+1 dimensional cube, take two N dimensional cubes and join each node on one cube to the corresponding node on the other. A four-cube can be visualised as a three-cube with a smaller three-cube centred inside it with edges radiating diagonally out (in the fourth dimension) from each node on the inner cube to

> the corresponding node on the outer cube.
>
> Each node in an N dimensional cube is directly connected to N other nodes. We can identify each node by a set of N Cartesian coordinates where each coordinate is either zero or one. Two node will be directly connected if they differ in only one coordinate.
>
> The simple, regular geometrical structure and the close relationship between the coordinate system and binary numbers make the hypercube an appropriate topology for a parallel computer interconnection network. The fact that the number of directly connected, "nearest neighbour", nodes increases with the total size of the network is also highly desirable for a parallel computer.

There are several advantages to an interconnected topology in comparison with a switched network topology. The direct connections reduces latency, which switched connections would introduce. Powerful switches are expensive, so a switched network doesn't scale well in terms of cost [19]. It must also be considered that when implementing an application for an interconnected network, yau are free to make optimized protocols that can enhance performance for your task.

In principle, the hypercube topology can be used to implement different sorts of parallel multicomputers. The topology has a balance between the fully interconnected network, where a connection point to every other node must be supplied, which would be expensive, and topologies where the data have to travel a long distance from one node to another. A multicomputer is a cluster of interconnected cooperating computer nodes, behaving as one computer. It distributes tasks and information by message-passing (as opposed to a *multiprocessor* , where several processors share a common memory area) [52]. Seen from the outside, it will behave as one server instance.

The qualities of hypercube topology ensure that when the system size increases, no new bottlenecks are introduced. Figure 4.4 shows how nodes are added in the cube. The amount of possible routes for information to take in the cube makes an even distribution of message passing possible.

4.5.2 The c-cor n4x multimedia server architecture

The n4x solution [4] makes use of the Hypercube topology to implement a multimedia server cluster that allows for a great bandwidth potential. This is made efficient by high-speed message routing hardware [4]. The solution is scalable because you can expand it exponentially (see section 4.5.1) There is no single point of failure; routing is dynamically processed. The routing is acieved through hardware especially built for this task. This also helps performance when data can be routed by the least trafficked route, or avoid broken routes. Since there are only one copy per content-item, it maximizes storage space. The

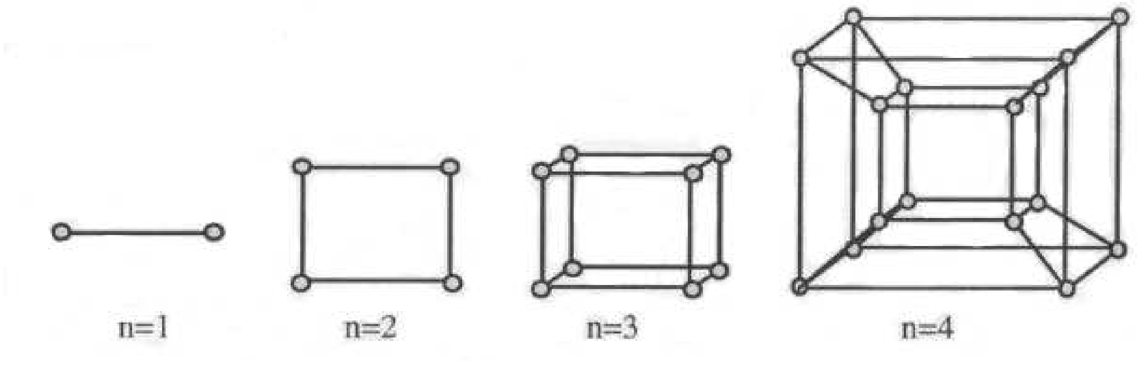


Figure 4.4: Hypercube scaling

system boasts ability to serve over 60000 concurrent streams with rates of 16Gbps per node and over 2 Tbps systemwide [42]. This is achieved by using the topology to evenly distribute data load internally in the cube. Each node has 8 full-duplex, high-speed ports for intra-cube communication. Internal load-balancing is achieved by striping[2] the data over all disks in the cube. The redundant array of inexpensive disks (RAID) striping also ensures that if a whole node (and all its disks) fails, the system can continue operation uninterrupted.

The n4x MediaHUBs are where the data is stored. These nodes are interconnected with eight hypecube connectors. This can accomodate a hypercube with $n = 8$ or a total of 256 nodes. These require no common memory access, they use message-passing to move data. The messages are controlled by an adaptive routing system. This system will find the most efficient route from point to point, and will avoid heavily trafficked routes or broken links.

The interconnection is supported by special hypercube connector hardware. This hardware supports routing logic by using a built-in vector processor unit [42]. This ensures that sufficient resources are available for video data retrieval and streaming on the server.

To transmit the data delivered by the cube, n4x uses hardware implemented interface modules customized for the needed transmission type. Examples of such modules are: QAM Cable, ATM, DVB-ASI, Ethernet and Gigabit Ethernet. These interface modules have hardware support for i.e. multiplexing, encoding or forward error correction (FEC) operations needed to support its interface type/model. This enables high output datarates from each hypercube node (MediaHUB).

[2]The data for each media item is evenly distributed among all nodes, requiring a smaller amount of data to be fetched from each node, thus reducing the possibility of bottlenecks.

The custom built components are assembled on a motherboard with Intel [27] 860 chipset and an Intel Xeon processor. The disk array are controlled by Qlogic 12160 [49] SCSI controllers. Put together, these components form a customizable, scalable server solution.

The next section will present some protocols that are instrumental in achieving better multimedia streaming performance. The protocols presented are used for setting up and performing streaming of real-time data.

4.6 Protocols

To set up a multimedia stream, we have to make use of some tools to control the stream (start, stop, pause the stream etc.), to convey information about how the stream progresses, and to be able to ship the data with the necessary information. This can for example be accomplished with the following protocols:

- Real-Time Streaming Protocol (RTSP) [56]: Used to issue control commands to a multimedia server. Enables the client to for instance play, stop or pause a media stream. Also used to setup the stream, and convey necessary information about port numbers for server and client.
- Real-Time Transport Protocol (RTP) [55]: Used for sending media. Provides services like payload type identification, timestamps, sequence numbering and delivery monitoring.

RTP Control Protocol (RTCP) is used in conjunction with RTP to allow the service to optimize streaming performance based on feedback from the client. This protocol is used to send information between the client(s) and server(s) to keep track of the progress and quality of the stream(s). The server(s) can then make adjustments to compensate for any events that may occur during transmission.

4.6.1 RTSP

The RTSP protocol [56] is an application-layer protocol for controlling media streams. An RTSP session is identified by its ID on the server and is completely independent of an eventual TCP connection. Consequently, the client can open and close several TCP connections, and still control the same RTSP session. RTSP can also be conveyed by user datagram protocol (UDP). In most aspects, RTSP is very similar to HTTP/1.1 [17]. The protocol supports the following operations:

- Retrieval of media from media server.

- Invitation of a media server to a conference.
- Addition of media to an existing presentation.

To be able to send a request for media, the client has to have some info about it. This can be placed in a description of the media, and can be made available in different ways, like publishing it on a web server, or making it available through the media server itself. The description will also give information about the transport methods the server is capable of. The alternatives are:

- Unicast with the client determining the port.
- Multicast with the server determining the address and port.
- Multicast with the client determining the address and port.

The stream controlled by RTSP may be sent via a separate protocol, independent of the control channel. A common protocol combination to use is RTP/UDP. The stream is totally independent of RTSP, and once started, it will continue until the media is exhausted if no more commands are received. The RTSP session on the other hand will have to maintain a state depending on which commands it has received. The specification states that the RTSP session identifier has to be randomly generated, and has to be at least eight octets long to make guessing the ID more difficult.

The following RTSP methods will affect the sessions state:

- SETUP: Initiates an RTSP session and allocates resources for a given stream.
- PLAY and RECORD: Start transmission of an already configured stream.
- PAUSE: Halts a stream. Does not free server resources. The stream can be restarted if wanted.
- TEARDOWN: Releases the RTSP session, and frees the resources allocated for the stream.

The location of the media controlled by RTSP is defined by the RTSP uniform resource locator (URL). It is equivalent to a HTTP URL, and describes the server address, eventual port number and the absolute path of the media on the server [56]:

```
rtsp_URL  =   ( "rtsp:" | "rtspu:" )
              "//" host [ ":" port ] [ abs_path ]
host      =   <A legal Internet host domain name of
               IP address (in dotted decimal form)>
```

The RTSP message is text-based, and each line is terminated with CRLF. The character set used is ISO 10646.

When a request is received and interpreted a response will be sent. The response can have the following format: [56]

```
Response     =     Status-Line
             *(    general-header
             |     response-header
             |     entity-header )
             CRLF
             [ message-body ]
```

The status line consists of the RTSP version type, a status code and a reason-phrase. The reason-phrase is intended to give a short textual explanation of the status code. The status codes can be classified as follows:

- 1xx: Informational - Request received, continuing process
- 2xx: Success - The action was successfully received, understood, and accepted
- 3xx: Redirection - Further action must be taken in order to complete the request
- 4xx: Client Error - The request contains bad syntax or cannot be fulfilled
- 5xx: Server Error - The server failed to fulfill an apparently valid request

The response header gives additional information that could not be included in the status field. This includes additional information about the resource requested like for instance server and client port numbers for the media stream.

Each RTSP request carries a sequence number labeled "CSeq". The sequence number is incremented by one for each new request. If a given request is not acknowledged, the eventual retransmission of the request have to carry the same sequence number as the original request.

Figure 4.5 shows a typical RTSP session. The first step for the client is to retrieve information about the stream. In this example, it sends a regular HTTP request to a web server, and gets the description file in response. This file could have been made available from the media server too, accessed with an RTSP DESCRIBE request. The client now sends a SETUP request, asking the server to reserve resources for the new stream. If all is well, the server responds with an OK message containing the information needed by the client to receive the stream. When the client is ready, a PLAY request is sent to the server. This tells the server to commence streaming the data. The server sends an RTSP

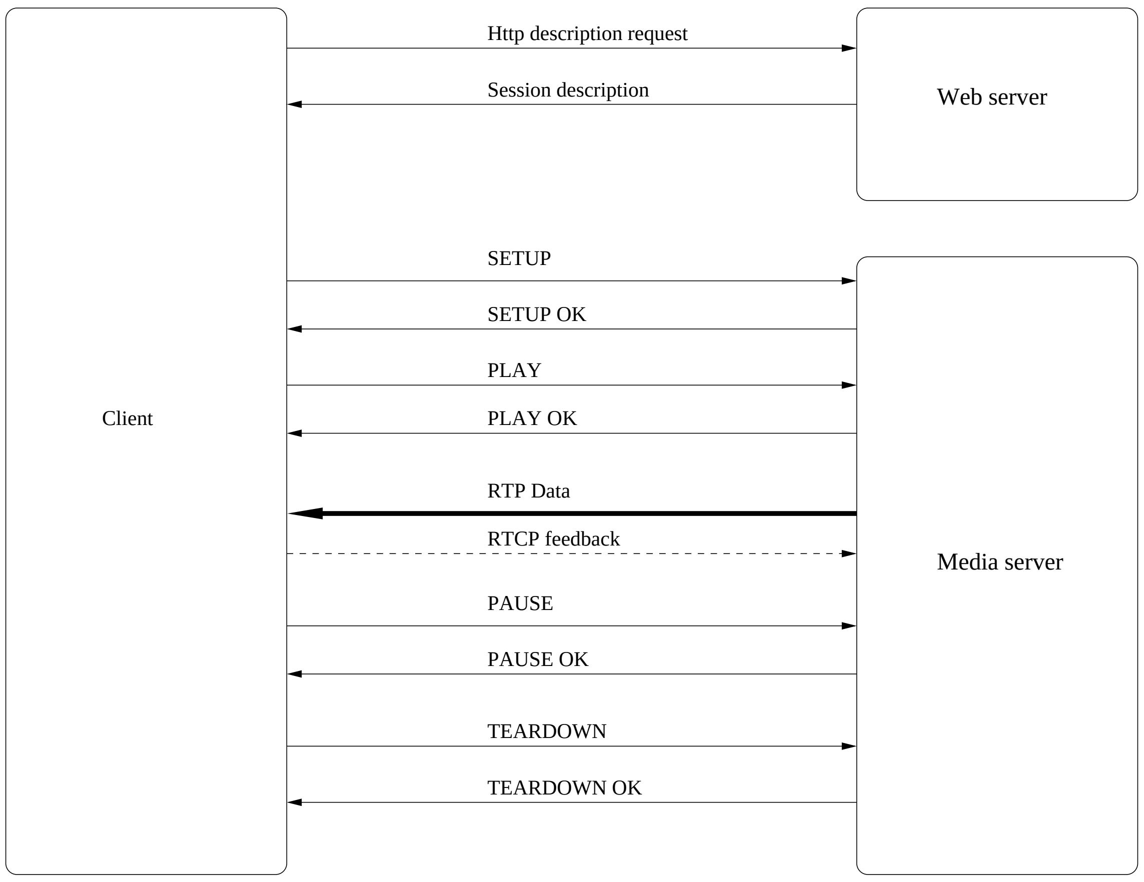


Figure 4.5: A typical RTSP session [7]

OK to the client, and starts to send RTP packets with the requested data. If supported, the client occasionally sends RTCP feedback messages to the server to report on the state of the stream. If a PAUSE or STOP message is received, the server will have to take the appropriate action. To end the RTSP session, and free all related resources, a TEARDOWN message is sent from the client.

4.6.2 RTP

RTP [55] is a protocol which provides a set of services for real-time media streaming purposes. These services include payload type identification, sequence numbering, timestamping and delivery monitoring. RTP is usually being used on top of UDP, but it is also possible to use it with other suitable underlying network protocols. The protocol was primarily designed to facilitate multimedia conferences, but is in common use as a means

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|V=2|P|X|  CC   |M|     PT      |       sequence number         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           timestamp                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           synchronization source (SSRC) identifier            |
+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
|            contributing source (CSRC) identifiers             |
|                             ....                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4.6: RTP header format.

to transport any kind of real-time media stream.

The RTP header is shown in figure 4.6. The version (V) field of the RTP header states which version is used in the implementation. The version designated by RFC3550 is 2.

The padding bit (P) states that at least one octet at the end of the payload is padding. How many octets are padding can be found by reading the last octet. This padding is necessary for some encryption algorithms.

The extension bit (X) states that exactly one header extension follows the RTP header. The potential header extension has a field that states its own length. It is recommended to try to manage challenges using conventional means, and try to avoid using the header extension.

The source of an RTP session is identified by a 32 bit synchronization source (SSRC) identifier (figure 4.6.2). This makes the source independent of network addresses for identification. Packets from a given synchronization source share the same timer and sequence number space. In that way the client can group the received packets by SSRC for playback (for instance where several different RTP sessions are received on the same port). The SSRC is to be chosen randomly, and is meant to be globally unique inside a specific RTP session.

If there are contributors to an RTP session (for instance if several media streams are mixed into one), the RTP header is appended with the SSRC's for the contributing streams. This could for instance happen in an audio conference, where the mixer could identify the speakers contributing to the audio by appending their contributing sources CSRCs to the RTP header.

The CSRC count (CC) field contains a number indicating how many CSRC's are appended to the header.

The usage of the marker bit (M) is defined by a profile. It can be used to indicate significant points in a stream, like frame-boundaries.

The payload type field (PT) contains a code identifying the format of the payload media. Some standard types are defined in RFC3551. If a receiver does not recognize the payload type, the received packets will be discarded.

The sequence number is incremented by one for each packet sent. The initial value is to be chosen randomly to make attacks on encryption more difficult.

The timestamp field represents the sampling time for the payload data. It is derived from a time reading, or from the nominal sample instance. The progression rate of the timestamp can therefore be different depending on the sample rate of the payload media and format. The initial value should be chosen randomly. If, for instance, several packets belong to the same video frame, they should have the same timestamp. They would, however, have consecutive sequence numbers.

An RTP session can be monitored and enhanced using RTCP in tandem with the session. This is done by sending RTCP sender and receiver reports that indicate how all of the parts involved in an RTP session are doing.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      defined by profile       |           length              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        header extension                       |
|                             ....                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4.7: RTP header extension format.

RTP header extension (figure 4.7) is a mechanism that allows for custom extensions to be added to the RTP header. It is possible to implement servers and clients that support these extensions, but systems that do not support the custom add-on must be able to ignore the extension without losing any of the original RTP functionality. It is recommended in the specification that other methods should be used to overcome any challenges that an extension would help solving.

4.7 Summary

What is wanted from a multimedia server solution is high performance to serve the clients during eventual peaks, scalability to meet an increasing demand, and reliability to make the users trust the supplier [59]. In this chapter, we have seen that multimedia applications have a range of stringent requirements that must be met. The methods for achieving this are to build optimized servers, to group the servers in special topologies and to distribute the content to proxies, depending on demand. To support such services, enhance transmission and optimize streaming, we can make use of special protocols like RTSP, RTP/RTCP that improve the streaming performance and adaptability.

The c-cor n4x addresses many of the multimedia challenges mentioned. The n4x approach achieves better multimedia streaming performance and is a flexible, freely expandable topology, but uses special hardware that offloads the main processing platform. There are custom hardware both for routing inside the server topology and for multeplexing, encoding and transmitting data out of the cube with different network standards. Although effective, this kind of custom hardware is expensive to develop.

Today, more and more parts of standard computer components are given their own processing power in order to offload the central systems. In the wake of this development, there have surfaced new platforms of programmble hardware with special properties and uses. One common example of this is the graphics processing unit (GPU) commonly used to offload demanding graphical calculations. Another platform that is maturing is the Network Processing Unit (NPU). These are programmable units optimized for network functionality. The NPU's are mass-produced, and relatively cheap. The possibility then arises for programming these units to perform tasks that previously would have to be hardware-implemented, like hypercube routing mechanisms in a server solution.

In the next chapter, the implementation of the basic functionality of a video server cube (VS^3) will be described. The server cube will have a hypercube design topology, and IXP2400 NPUs will be used to handle the routing process.

Chapter 5

The VS^3 Video Server Cube

This section describes the steps taken to implement a limited working multimedia server cluster based on a hypercube topology. Small parts of a similar system are first implemented on IXP1200[31] boards as an assignment in the inf5070[23] course, and this system is also described. From there, each step in testing the hardware, and deciding on implementation strategies up to the final implementation is discussed. A major part of this work has been to get knowledge of the IXP2400[32] hardware, the Intel IXA SDK 3.51[30] and the possibilities and disadvantages of different approaches. This has led to several separate experiments to measure how different implementations and components perform given specific tasks. The results of these, and the increasing understanding of the hardware and programming platform have resulted in several incremental design steps that has culminated in the current server solution. The implementation process resulted in several intermediate designs that were rejected. The purpose of these designs, and the reason why they finally were discarded will be discussed. Finally, approaches on how to further improve the system, and remove the current bottlenecks will be presented.

5.1 Hypercube server general design

The system we want to make is an implementation of a multimedia server cluster with hypercube topology similar to the nCube n4x system. The system should use message-passing to communicate internally, and have no shared memory areas between nodes. To make routing efficient, and to offload the host, we want to use IXP2400 network processor cards to handle the routing operations. In figure 5.1, the basic design is outlined. We want to use the Linux hosts as data and streaming servers, and the IXP cards to take care of operations related to locating the media and routing packets to the correct egress node. When a data packet is to be sent to an outside client, the streaming application has to

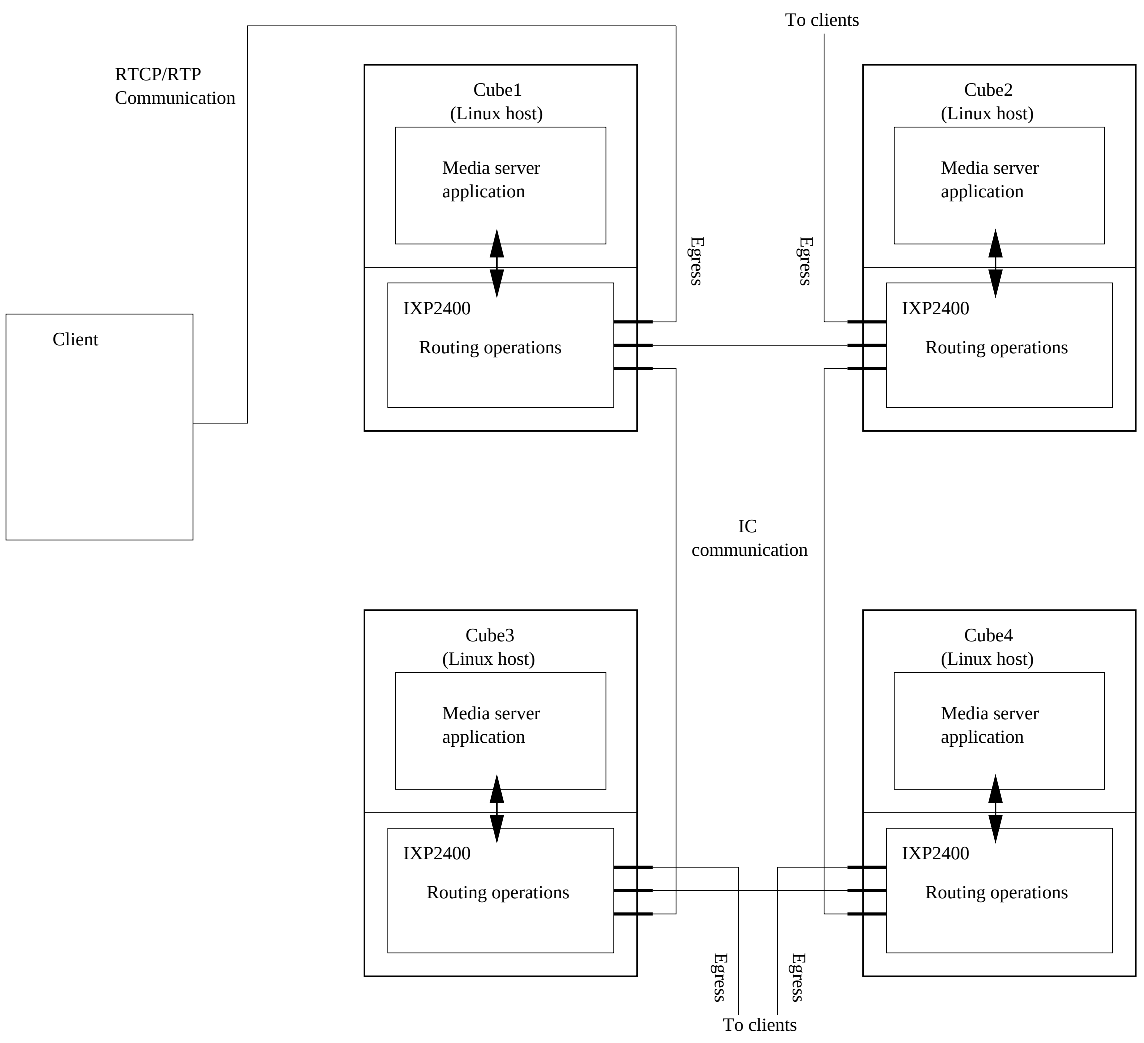


Figure 5.1: Basic design of multimedia server cube with IXP cards.

send it to the IXP card, it will then be routed to the egress node and sent to the client. An adaptive Domain Name Service (DNS)[1] will make sure that requests are distributed evenly between the egress nodes. The cluster will look like one server as seen from the client.

RTSP is the protocol we want to use for setting up and controlling the media stream. The streamed data itself shall be transported using RTP. For intra-cube messages and routing, a new Intra-Cube (IC) protocol is designed.

[1]This service could either communicate with the server cluster to find the egress node with the most free resources, or could statically select a new node each time, in a round robin fashion.

As discussed in chapter 4, there are many optimizations that can be applied to a multimedia server cluster. In this assignment, we have implemented the basic functionality needed by a server cluster capable of delivering a video stream. Optimizations like striping the data over all nodes, and dynamically adaptive routing will have to be built into the system at a later stage due to the limited timespan of this master thesis. The *n*-value of the hypercube topology is limited to 2.[2] The reason for this is that the ENP2611 network cards have 3 optical interfaces, and one of these interfaces is used for the egress connection. There is the possibility of inserting more than one card into each host to increase the possible *n*-value of the topology, but this option has not been thoroughly explored due to the increase in complexity this would lead to, and the number of available cards.

5.2 The legacy design

The design of the server system is based on the nCube n4x system. nCube claims that their solution is able to serve 2000 concurrent users with a bandwidth of3,75Mbps per user with a rack of 8 MediaHUBs (cube with $n = 3$) [42]. Their solution is, however, heavily based on custom hardware, and as of such expensive to develop and produce. The IXP cards are optimized for packet handling and have several network interfaces, in a manner similar to the custom hardware of the n4x system. This makes it possible to envision a similar design using programmable IXP cards for cube interconnection.

The work began with some basic attempts to implement parts of a routing functionality on the IXP1200 card as an assignment for the course INF5070 [23]. We describe the main features of this design in the following section.

5.2.1 The INF5070 implementation

The assignment for INF5070 [57] was to implement an n4x-like solution with RTP based transport and RTSP based control utilizing IXP1200 [31] cards and a given programming framework based on the Intel IXA SDK 2.01 [29]. We chose to implement the routing part, and postpone the RTP/RTSP part. The software framework that was provided had a structure that delivered all packets to the StrongARM core (Figure 5.2). The choice was made to use a source routing algorithm with information conveyed through an intra-cube (IC) protocol. The purpose of the IC-protocol was to handle:

- Routing between cluster nodes.
- Transportation of video data to the egress-node.

[2]A hypercube topology with $n = 2$ consists of a total of $2^2 = 4$ nodes.

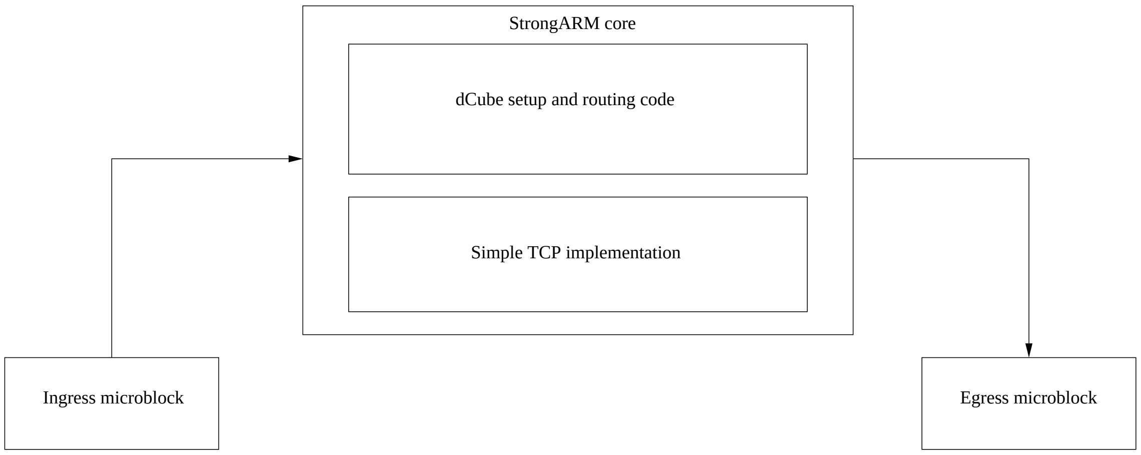


Figure 5.2: Dataflow inf5070 implementation

- Forwarding of video-playback-control to the node serving the stream.
- Handling requests for specific video files, locating the file and setting up the stream.

The final result of the project was an application that managed to stream the requested data from one node to another using intra-cube routing. The bitrates was, however, very limited because of the fact that all traffic, routing, setup information and data packets, had to pass through the StrongARM core on each and every node it passed through. The performance was also reduced by the fact that the data itself was read and streamed by an application running on the StrongARM. This demanded too many resources to achieve high performance.

5.3 SDK code base

The SDKs from Radisys and Intel provide an extensive code base from which applications can be adapted. The majority of these examples, however, focuses on standard networking tasks, and could not easily be used as a basis for the VS^3 application. In the Radisys SDK static forward application, there were components that could be used successfully. To build any network application, functionality has to exist for receiving and transmitting data. This functionality was provided by the RX and TX microblocks from the static forward application. With an asynchronous interface of passing buffer handles on scratch rings, new components could be fit seamlessly in between of these microblocks.

With RX and TX functionality being the only "recyclable" component, the following stages had to be built from the bottom: Microblocks for data plane processing, XScale components for control plane processing and a streaming application located on the host machine. There were made designs that aimed to be able to use existing host server applications, thus eliminating the need to implement this. The next sections will discuss different designs for implementing the VS^3 system.

5.4 XScale-Host communication

In order to pull data from the host to XScale and μEngines, the ideal solution would be one that could accommodate very high data rates. Due to the large scope of the assignment, the task of implementing this efficiently was temporarily postponed. A strategy for achieving this can be found in the further work section (see section 6.3). The method that was ultimately chosen was to transmit the data over the 10/100 debug Ethernet port. Based on this decision, three different design choices were outlined and tested:

- **Use network file system (NFS) to read the file directly from the XScale application:** Using NFS would provide an abstraction of all operations that are needed to get the file from the host to the XScale. The performance of the transfer itself, however, would be reduced due to the nature of NFS services. NFS, based on remote procedure calls (RPC) will, in addition to moving the file data, also use much resources on consistency checking and synchronization. The NFS approach would also imply that all tasks related to reading data, generating packets and sending these packets have to be handled by the XScale core. For a scenario with many streams, the load would probably be too big for the XScale core, and lead to performance reduction.

- **Create a raw socket and forward packets directly to the host:** Another approach was to create a raw socket on the XScale. This would give the opportunity of creating tailor-made packets and transmitting them to the host without creating a two-way socket connection. Any available multimedia streaming server software could then be used on the host machine, but some manipulation of packets would be needed to make the server software work as wanted within the cube structure . This concept will be described in more detail in section 5.5.

- **Create a regular TCP socket to communicate with an application running on the host:** The regular socket alternative would demand a server application on the host that could receive a connection from the XScale. The performance of the data transfer would be limited by the TCP/IP stack processing both on XScale and on the host machine. It would, however, provide the service of a byte stream between the applications. The most attractive element of this solution is that it would simplify some aspects of implementing the streaming (host) application. More information about this design approach can be found in section 5.6.

The implementation work began using the raw socket strategy. This allowed the use of a server system that already existed (i.e. komssys [36]) on top of the routing framework. The next section providess a description of the implementation strategy, and explains why it eventually was discarded in favour of the regular TCP socket design.

5.5 The raw socket design approach

After exploring the hardware possibilities in section 3.1, the first structural design for the whole server cube application was made. If stream setup was to be done by RTSP, a TCP connection would have to be set up. A solution where the Intra-Cube (IC) routing system would forward all TCP packets to the host machine containing the requested media was outlined. The task of delivering packets to the correct egress node would rest on the routing layer (IXP cards), and each host machine would behave like a freestanding multimedia server, oblivious of the routing layer below.

5.5.1 TCP Handling

For this approach to work, the TCP connection from the client would have to address the machine that hosts the wanted media, not the egress machine. In order to do this, the egress node (the node receiving the initial TCP SYN request) would have to do a three-way handshake, then receive the RTSP SETUP message. This is because we don't know where the media file is located upon receiving the initial request, and the client expects a complete TCP connection before sending the RTSP SETUP request. The machine that hosts the file will have to be found (the mechanisms for this are presented in section 5.6). When the route between the egress node and the machine delivering the media has been found, the XScale application on the machine will have to perform a three-way handshake to connect to the application on the host. It can then forward the RTSP SETUP packet. In this way, the host application will never know that the TCP connection has been routed through several other machines, and will believe that it is communicating directly with the client. There are, however, some more complications to doing this. The sequence (seq) and acknowledgment (ack) numbers of the different TCP connections will be different, and will have to be simultaneously translated to match. The content of the RTSP packets will have to be modified to make the server stream to the XScale (which will take care of routing and forwarding). All these extra steps will have to be done on the IXP platform as a part of the routing framework.

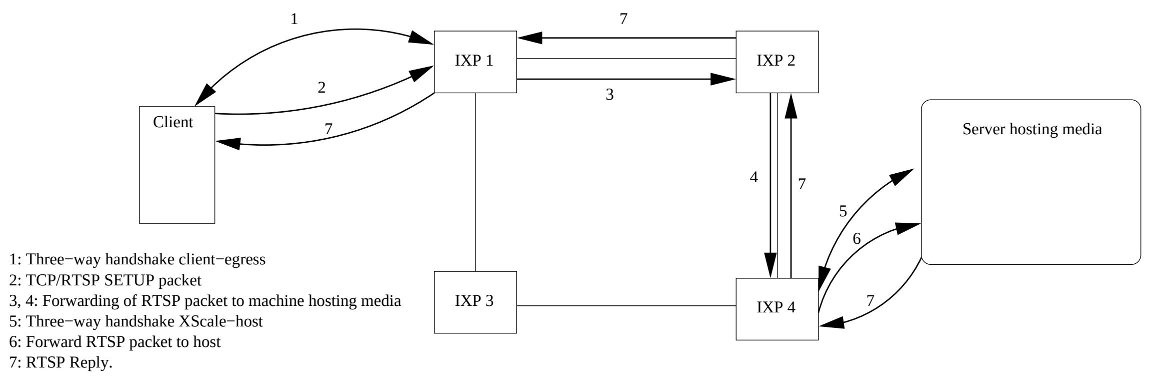

1
7
IXP 1
IXP 2
2
3
Client
7
Server hosting media
4
7
5
6
1: Three−way handshake client−egress
2: TCP/RTSP SETUP packet
3, 4: Forwarding of RTSP packet to machine hosting media
5: Three−way handshake XScale−host
6: Forward RTSP packet to host
7: RTSP Reply.
IXP 3
IXP 4
7

Figure 5.3: Design with forwarding of TCP to machine hosting the file.

In figure 5.3[3], the set up of a stream with RTSP is shown. At the egress (IXP 1) the TCP three-way handshake is done (1). The RTSP SETUP packet is then received (2). The packet has to be inspected, and the file must be found in the cube. Info about the TCP session and the IC-session has to be kept both at the egress and at the machine that has the file. This is to ensure that packets from the outside can be routed to the correct host (egress IC information) and that correct headers and routing information can be supplied to build the packets that are outbound (media server IC information). After receiving the RTSP SETUP message, it is forwarded to the correct machine (3, 4). A three-way handshake is then done between the XScale and the host machine (5). The RTSP/TCP packet can be modified to conform with this TCP session, and forwarded to the host machine(6). Finally, RTSP replies can be processed the same way, and the server can start streaming RTP packets to the client (7).

When using raw sockets, received packets are forwarded both to the kernel and to your socket. This means that kernel-level filtering has to be introduced in order to stop all packets bound for your application from being answered by the kernel processes. The Monta Vista preview kit Linux kernel did not have support for iptables in the distributed version, so in order to make this work, the kernel had to be reconfigured and recompiled. Since no extra software was included with this distribution, a cross-compile of iptables had to be made for the XScale. These steps made the necessary filtering possible.

The advantages of this design approach would be that the media streaming server software would not have to be aware of the routing framework below, thus making it easy to replace the streaming server software. It became evident, however, that the process of maintaining several parallel TCP sessions for every TCP stream, together with the process of translating IP addresses and sequence numbers for the streamed packets would introduce a degree of complexity that would make the system error-prone and difficult to

[3]The process of locating the node that has the media file is not represented in the figure

implement. The conclusion was that the disadvantages of complexity and structure were greater than the advantages gained by this approach. The work with this design-option was discontinued before the routing framework was completed. This was due to the estimation that redesigning the software structure would result in an implementation that was simpler and more likely to be completed inside the time-limits of the thesis.

It was obvious that another approach had to be taken to make a design that was expandable, yet simple enough to implement given the time limit. The new design structure would have to give the egress node responsibility for handling TCP sessions. The next section presents the final design.

5.6 Egress TCP design

The complexity problems introduced by the TCP-translation solutions made it necessary to investigate alternative approaches. An obvious solution was to let the egress node handle the TCP connection, and let the rest of the control mechanism be implemented by the cube application. A consequence of this is that the media streaming server application on the host machine would have to be aware of the underlying mechanisms.

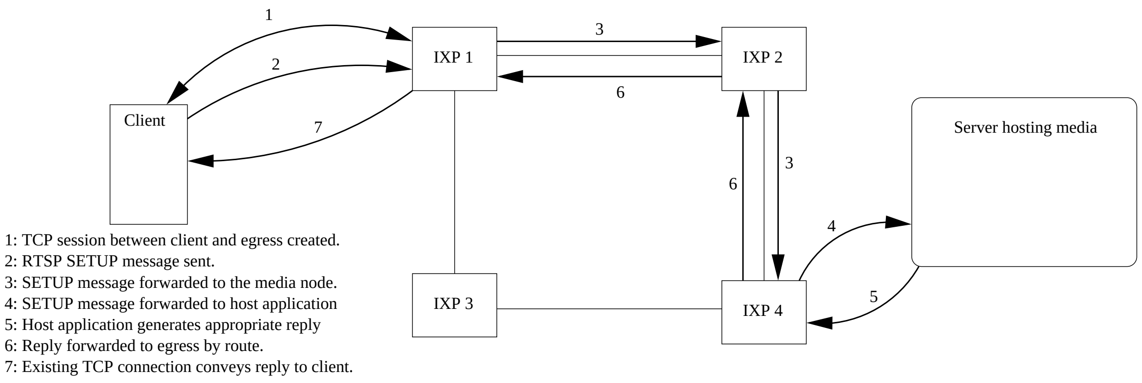

1
2
7
3
6
3
6
4
5
Client
IXP 1
IXP 2
IXP 3
IXP 4
Server hosting media
1: TCP session between client and egress created.
2: RTSP SETUP message sent.
3: SETUP message forwarded to the media node.
4: SETUP message forwarded to host application
5: Host application generates appropriate reply
6: Reply forwarded to egress by route.
7: Existing TCP connection conveys reply to client.

Figure 5.4: Design with egress handling TCP connections.

Figure 5.4 shows the setup process for a stream using the egress TCP design. The main differences from the raw socket design is that the node receiving the request handles all TCP operations. The first step, when a request is made, is to setup a TCP session, and give the correct replies to the client (1). When the client sends the RTSP SETUP request, the receiving node has to acknowledge the packet, and forward the request to the node that has the media that was requested[4]. (2, 3 and 4). The appropriate reply is generated

[4]The process of locating the media is not represented in the figure, but explained in more detail in section 5.7.4.

by the host application (5 and 6) and the egress node wraps the reply in the appropriate TCP header, and transmits it to the client (7).

Not only would this structure make several aspects of routing and TCP handling easier to implement, it would also make a better base for expanding the application when the primary bottleneck, namely host-μEngine dataflow, could be overcome. Using direct memory access (DMA) to move data between the host and the μEngines would be a natural way to improve on this bottleneck. This option will be further discussed in section 6.3. In order to achieve this, the host application (or the operating system) would have to be aware of the underlying structure. Another consideration is that the RTSP standard states that it should be possible to create several RTSP connections on one single TCP connection. This would be difficult to implement using the raw socket design.

One of the moments that were crucial in deciding on implementation strategies was how the routing itself would be handled in the cube. The next section will describe how this was designed and implemented.

5.7 Cube routing

Several advantages are obtained by implementing a solution in a distributed hypercube. In a cube with n > 1, there is more than one possible route between any two nodes. This gives us the opportunity to avoid congested routes, or to bypass possible broken links.

There are various routing strategies that can be applied to a network topology like the hypercube. One concern in a setting like this must be that the strategy will allow for a certain degree of adaptability. This is to be able to cope with eventual broken links and avoid heavily trafficked routes.

The choice fell on source routing as the routing strategy to implement. Using this strategy, the sender determines the route that has to be taken. The routing information is included in the packet header when the packet is transmitted. This eliminates the need for routing tables at every node. There was two main reasons for this choice. The solution would be fairly simple to implement, and the processing speed for each packet would be high.

There is also possibilities for extending the routing strategy with features that will add adaptivity. There is, amongst aothers, an internet-draft proposing a dynamic source routing strategy for ad-hoc networks [35]. Some strategies on how to improve adaptability for our source routing strategy will be found in the section 6.3.

Below is a description of how the source routing strategy is implemented in the current video cube routing framework.

- When a new setup request is received, the message is broadcasted through the cube.

- For each node the request visits, the port where it arrived is pushed on the routing field in the IC header.
- When the request arrives at the node that has the media file, it saves the route carried in the packet. It then sends an IC packet via the route to the egress node, instructing it to setup an IC session to be able to route new control packets directly to the node that has the media file.
- The stream then appends all RTP packets with an IC header that routes the packet to the egress node.
- The port number popped from the route field of the IC header can have one of the following values:
  - 0x0: Transmit the packet on port 0.
  - 0x1: Transmit the packet on port 1.
  - 0x2: Transmit the packet on port 2.
  - 0xff: Forward the packet to the XScale and pass it to `tcp_send`.

There are several advantages to this routing strategy. It is fairly simple to implement. It requires no routing information on intermediate nodes, only on the egress and the node that has the media file. It is very efficient to execute at each intermediate node, the microblock code only have to pop the port number and transmit the packet.

The disadvantages, however, touch some of the central concepts of implementing the system as a Hypercube. The current implementation does not take into consideration that the route have to be redirected if traffic patterns changes and transmission slows down. The option of redirecting the route if one or more links are broken also has to be added. It would also be profitable to be able to reserve resources along the route to guarantee that the stream will perform as wanted.

These disadvantages are subjects that should be explored further. Our source routing algorithm could probably be kept without losing the possibility of rerouting in the case of a broken link. A way of doing this could be to keep all routing messages that arrive at the node containing the media[5] in the IC struct. If a broken link is detected, we could generate a message that tells the IC session to switch to the next route that has not been tried. For large cubes (big n-value) it would probably be more efficient to map possible routes on system startup (and maybe with a predefined interval), and store an index to a route in the IC session struct. It is also possible to transmit "ping" packets along different routes, to try to measure any difference in latency. This, however, can not be done too

[5]All possible paths from the egress to the node inside the limits of the defined TTL.

frequently without affecting the streaming performance that should be prioritized. An approach with reserving resources along a route would probably be a much better alternative than switching routes "on the fly", given that this approach easily could lead to streams jumping back and forth between route alternatives. The ability to reserve resources along a route is also possible. Each node could keep a record of how many pass-through streams they support. When the packet locating the file is transmitted through the system, it could sum the pass-through number for each node it visits. The media server could then choose the route with the lowest sum.

5.7.1 Intra-cube header and extension

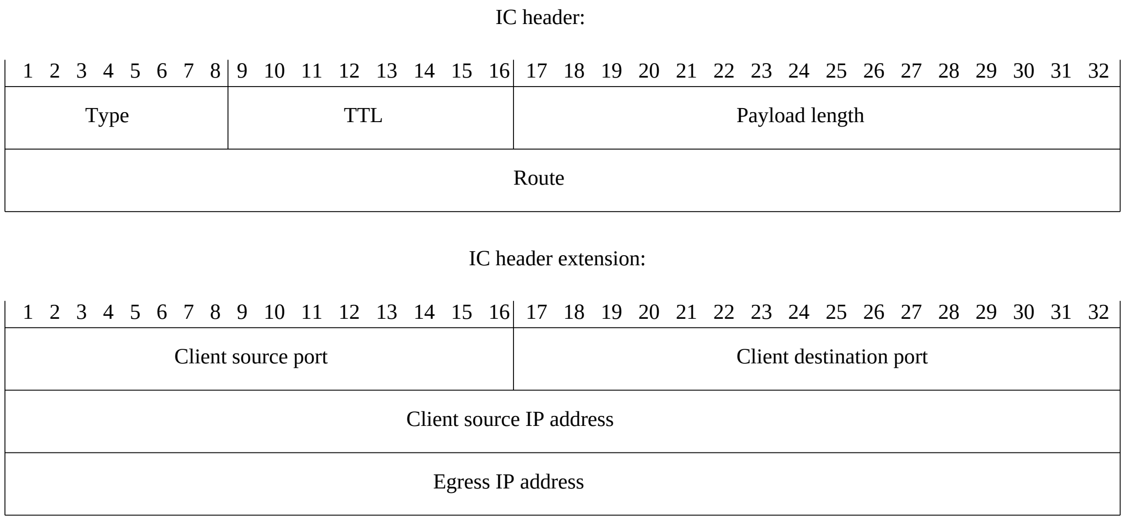


Figure 5.5: IC header and header extension format.

Figure 5.5 shows the format of the IC header. The basic IC header is designed to fulfill the following tasks:

- Separate setup and control packets from data packets.
- Limit the number of jumps a packet can circulate in the cube.
- Give information about payload length (when pushing data through the host-XScale socket.
- Provide routing information for the packet.

The purpose of the IC header extension is to carry information necessary to create IC session structs and to carry information about IC sessions to the egress when packets must be sent through `tcp_send` (port numbers and IP addresses is needed to look up the correct TCP struct).

5.7.2 IC packet types

In the system, as it is implemented in this thesis, there are five different IC packet types: ICH_FIND_FILE, ICH_ROUTE_FB, ICH_CTRL_FB, ICH_CTRL_MSG, and ICH_RTP. The first three of these packet types implicates that a header extension will follow immediately after the basic IC header.

The ICH_FIND_FILE is first created at the egress when a new SETUP request arrives. The IC header is appended with the IC header extension because the machine hosting the wanted media has to set up an IC session to be able to stream the content. The ICH_FIND_FILE packet is broadcasted throughout the cube until the node hosting the wanted media file is located.

When the file has been located, the node hosting the wanted media generates a packet of the type ICH_ROUTE_FB. This packet is transmitted back to the egress through the route found by the ICH_FIND_FILE packet. This packet also carries the header extension. This is because when the packet arrives at the egress, an egress IC session has to be created. This session is used to be able to route following RTSP control packets directly to the correct machine in the cube.

The ICH_CTRL_MSG package is created when the egress receives an RTSP control packet other than SETUP. The IC session that matches the request is found, and the packet is forwarded to the correct host through the cube.

When an RTSP control message is processed by the media streaming server, a reply is generated, and has to be sent to the egress through the cube. These packets are wrapped in IC headers of the ICH_CTRL_FB type. A header extension is used with these packets to look up the correct TCP session at the egress. This could have been avoided by adding a IC session index field to the regular IC header, and use this to look up the egress IC session. There are two reasons why this solution was implemented:

1. The main bulk of data packets does not need the extra information. Transmitting it through the cube with each RTP packet would be a waste of resources.

2. The RTSP traffic load in the cube is negligible compared to the RTP data, so the extra 24 bytes on these packets will not make a noticeable impact on cube performance.

The last IC packet type is the ICH_RTP. This is by far the most common in the cube since it wraps all RTP data packets. The purpose of this header type is to bring the payload to the egress, and subsequently to the client, as fast as possible.

5.7.3 Intra-cube session

The purpose of the IC session is to keep the necessary data to handle the functionality needed by the cube infrastructure. As of now, the following data is kept by the IC session struct as shown in figure 5.6:

- **Status:** Information concerning the state of the session. Used to identify active sessions, sessions about to be set up and closed sessions.
- **Route:** The current route the packets have to travel. For the egress node this is the route to the machine serving the media, for the media server, it is the route to the egress.
- **Client source port and server port:** Used to generate UDP headers.
- **Source and destination IP addresses:** Used to build IP headers.
- **Source and destination MAC addresses:** Used to build ethernet headers.

```
typedef struct{
  uint8_t    status;
  uint32_t   route; /* Route */
  uint16_t   sport; /* Client source port */
  uint16_t   dport; /* Server port */

  uint32_t   saddr; /* Client source IP addr. */
  uint32_t   daddr; /* Egress IP addr */

  char       eth_src[6]; /* Ethernet src address */
  char       eth_dst[6]; /* Ethernet src address */
} IC_session_t;
```

Figure 5.6: IC session struct.

The IC session is first created at the server hosting the media when the requested media file is found. The egress saves the session data when the route is set up, and the host is updated with the session data when the SETUP message is forwarded there. The way the program is implemented now, the media server on the host builds RTP, UDP, IP, ethernet and IC headers for the data packets. This means that the XScale IC session on the media server machine is superfluous. The reason it was implemented in this way was to facilitate the transition from a solution where the server handles header generation, to a future design where this functionality is handled by μEngines (aided by the XScale). The different aspects of this are discussed in more detail in section 6.3.

Regarding the general design structure and the routing framework that had been decided upon, tasks had to be divided between the different hardware layers available (host, XScale or μEngines). The next section will describe where the different functionality was placed, and the advantages and tradeoffs this led to.

5.7.4 Partitioning of tasks

The desicions on where to locate the different application parts was based on the general programming paradigms for networking applications (see sections 2.3.3) and on the results of the preliminary tests done on the IXP2400 hardware (see section 3). The way tasks are divided between the different subcomponents in this design is as follows:

- Host machine: The host machine runs the streaming server application. It receives RTSP requests, generates replies, creates streaming threads and forwards data to the XScale. The media data is stored on the host disk, and the host server is responsible for retrieving it.

- XScale core: The XScale core handles the TCP connections (if egress). It receives data from the host, and forwards it to the μEngines for routing to the egress node. It also receives control packets from μEngines, and forwards them to the host.

- μEngines: The μEngines receives control packets on the egress port. They filter all packets not TCP/IP bound for port 9070[6] and forwards control packets to the XScale for TCP handling. The μEngines also receive IC packets and route them according to the port number given in the route header field.

The IP header switch tests (see section 3.1) showed that the μEngines was very efficient on a simple task that implicated manipulating packet headers. Consequently, it was decided to implement the bulk of the routing functionality on the μEngines. Filtering of unwanted data that could arrive on the egress port was also to be implemented here.

[6]This port is used because Komssys [36] at first was intended to be the streaming server application run on the host. The default RTSP listening port for this system is TCP/9070.

Since TCP related traffic would represent a small part of the total, it was acceptable to place the main functionality on the XScale. This decision was also supported by the fact that maintaining data structures is a less complicated task on the XScale.

The media streaming application, performing a processor-intensive task, was to be implemented on the host. It was decided that the streaming application would also generate all the necessary headers to leave all but the routing to the μEngines. When direct transfer of data from the host to the μEngines is implemented, it will be natural to let the microblocks generate some of the headers. As it was, however, the data was moved to the μEngines via the XScale, consuming a lot of processing power from both μEngines and XScale alike. To counter this effect, as much of the packet generating functionality as possible was placed on the Host.

Next, descriptions of how the aforementioned components were implemented on the different hardware layers follow. The structure of the components and the main challenges regarding each step will also be discussed.

5.7.5 μEngine tasks

The *ICrouter* microblock used in the egress TCP design consisted of two active contexts. This was to better exploit the available resources by switching contexts when memory-accesses or other operations that leave the code waiting for results are performed.

First context The packet handling mechanisms used for the first context of the *ICrouter* microblock are shown in figure 5.7. When a packet arrives at any port, the first thing to be checked is whether the packet is received on the egress port (port 0) or any of the intra-cube ports (port 1 or 2). If the packet is received on the egress port, it is sent through a filter dropping all packets not TCP/IP bound for port 9070, thus eliminating all traffic except the packets bound for this server solution[7]. If the given criteria is met, the packet is forwarded to the XScale for TCP processing. If the packet received arrived at an intra-cube port (port 1 or 2), it is by definition an IC packet with an IC header (see section 5.7.1) and must be routed according to the given routing algorithm. If the packet is of the type ICH_FIND_FILE, it is sent to locate the host containing the wanted media. Such packets are forwarded to the XScale which performs the rest of this processing. All other packets are to be routed to the port given by the route field in the IC-header. The Time To Live (TTL) is decremented. If the TTL is zero, the packet is dropped. If the port number popped from the IC header route-field is 0 (egress port), the IC-header is stripped and the packet is transmitted. Packets bound for port one and two is transmitted on the popped port.

[7]To support RTCP reports, the filtering mechanism has to be able to open up the corresponding port for the connections, depending on the RTP transport method. This is as of yet not implemented.

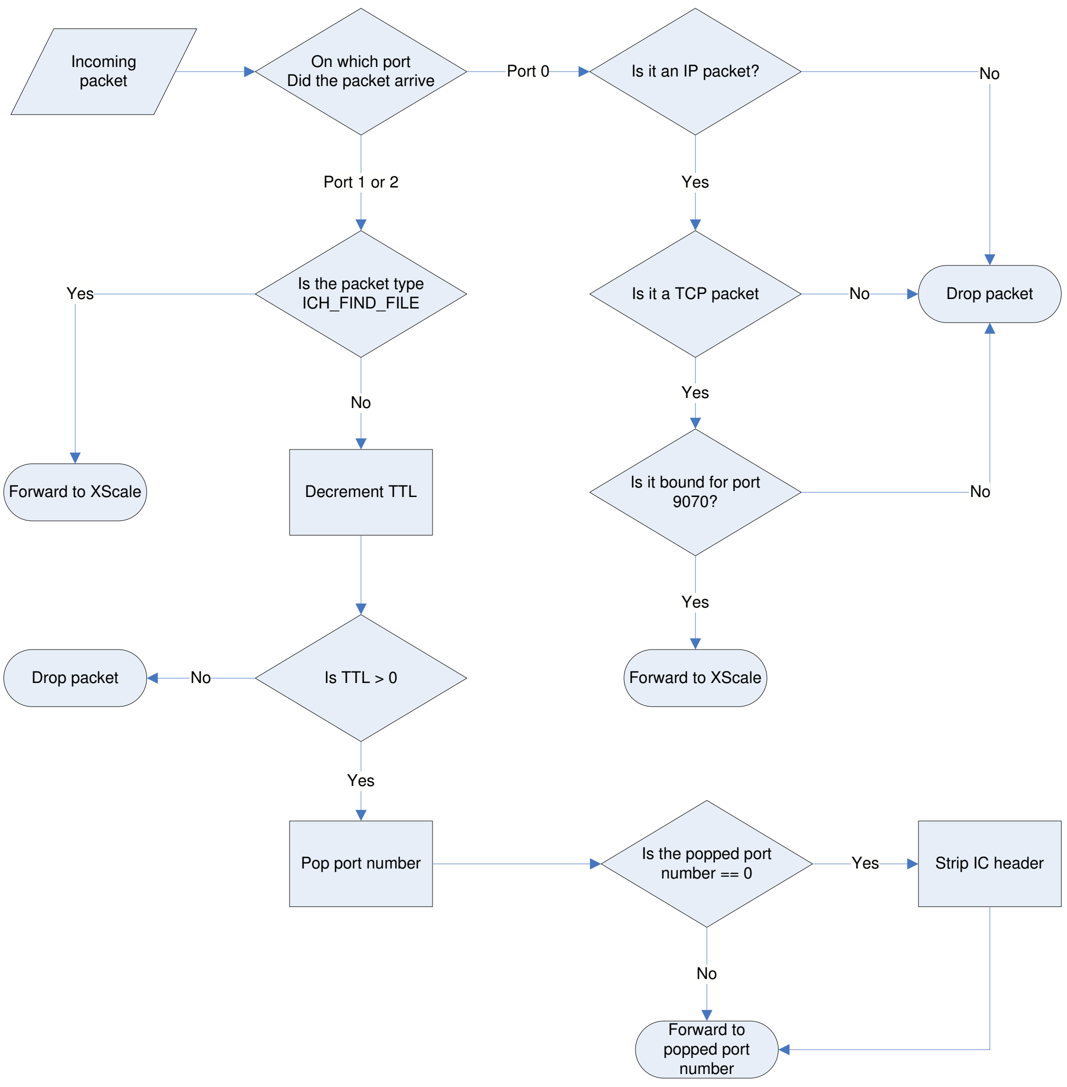


Figure 5.7: Flow chart for *ICrouter* microblock when a packet is received.

Second context The second context of the *ICrouter* μEngine receives buffer handles from the XScale, and transmits the packets referred to on the port specified by the packet metadata. All setup of packet and header is done by the media server on the host, or by the XScale. Functionality for batch processing of packets[8] is ready to use, but not yet supported by the XScale code.

[8]In this case batch processing means caching packets and sending them to the μEngines collectively to achieve higher transfer rates and more efficient resource utilization.

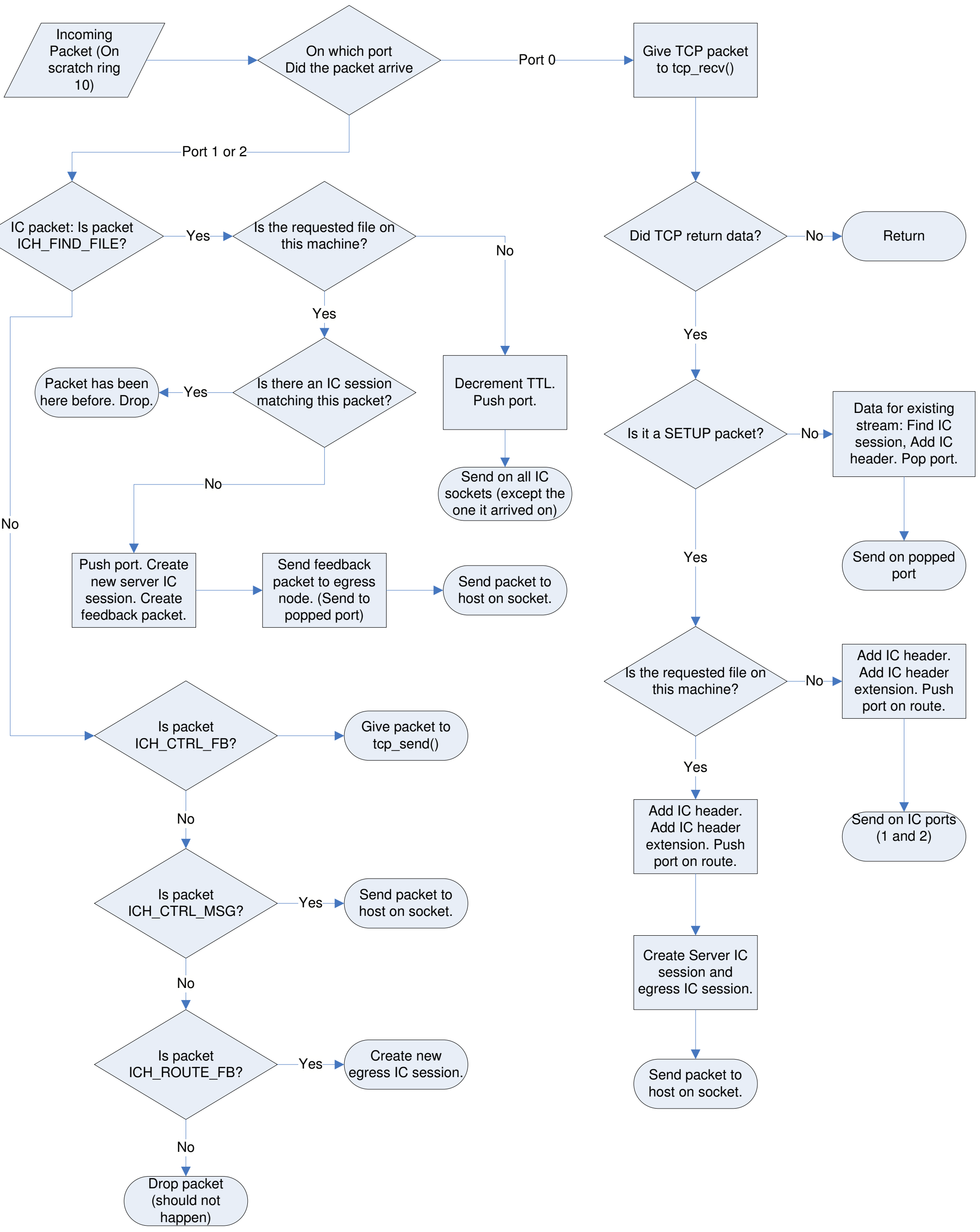
5.7.6 XScale tasks

The IXA SDK has support for spawning specialized threads that will call a specified method upon receiving one of two interrupts branded “INT_A” and “INT_B”. In this implementation, only one such thread for handling data sent by the *ICrouter* microblock to the XScale is used. The procedure for handling packets forwarded from the XScale is described schematically in figure 5.8. The first thing that is done, as in the *ICrouter* microblock, is to check which port the packet was received on.

If a packet is received on the egress port (port 0), we know that it is a TCP packet, and it will be passed to the `tcp_recv` method. This method returns one of three values:

- **B_TCP_NO_DATA:** The received packet is successfully handled, but contained no data. This is for example the case for SYN, ACK or FIN packets.
- **B_TCP_ERROR:** The packet is not successfully processed. This can happen if the TCP session is not found, or an error is detected with respect to sequence numbering.
- **B_TCP_DATA:** There is a data payload in the TCP packet. The method also sets references to the payload and the payload size in order to enable data processing.

Int_A thread processing When a packet is sent to the XScale, the Int_A procedure is invoked. If the input port is 0, we call `tcp_recv`. If `tcp_recv` returns B_TCP_ERROR or B_TCP_NO_DATA, the TCP implementation have processed all required replies, and the call returns without providing any data payload. If data is received, we know that it is an RTSP control packet. The RTSP command will then have to be identified. If the packet is a SETUP request, the program will try to find the file on the local machine. If the file is not found on the machine, an IC header and an IC header extension (containing information that is required to setup a new IC session) are prepended to the RTSP packet (for details, see section 5.7.1). The port number that the packet is received on is pushed on the route field of the IC header, and the packet is sent on both IC ports (port 1 and 2). An egress IC session is not created yet, because the route to be used is determined by the packet that arrives first at the host machine that has the requested file. If we assume that the packet travelling the path with the least load is the first to arrive, this will help the system give the stream an effective route during the current conditions. If the file is found on the machine, however, both a server IC session, and an egress IC session is created (given that this machine is both server and egress). The RTSP packet is prepended with an IC header and header extension, and forwarded to the host on the socket interface.



Incoming Packet (On scratch ring 10)
On which port Did the packet arrive
Port 0
Give TCP packet to tcp_recv()
Port 1 or 2
IC packet: Is packet ICH_FIND_FILE?
Yes
Is the requested file on this machine?
No
Yes
Is there an IC session matching this packet?
Yes
Packet has been here before. Drop.
Decrement TTL. Push port.
Send on all IC sockets (except the one it arrived on)
No
No
Push port. Create new server IC session. Create feedback packet.
Send feedback packet to egress node. (Send to popped port)
Send packet to host on socket.
Is packet ICH_CTRL_FB?
Give packet to tcp_send()
No
Is packet ICH_CTRL_MSG?
Yes
Send packet to host on socket.
No
Is packet ICH_ROUTE_FB?
Yes
Create new egress IC session.
No
Drop packet (should not happen)
Did TCP return data?
No
Return
Yes
Is it a SETUP packet?
No
Data for existing stream: Find IC session, Add IC header. Pop port.
Send on popped port
Yes
Is the requested file on this machine?
No
Add IC header. Add IC header extension. Push port on route.
Send on IC ports (1 and 2)
Yes
Add IC header. Add IC header extension. Push port on route.
Create Server IC session and egress IC session.
Send packet to host on socket.

Figure 5.8: Flow chart for XScale int_A thread.

If a packet is received on one of the IC interfaces (port 1 or 2), we know that the packet is an intra-cube packet and has an IC header. The IC packet received can be one of the five types mentioned in section 5.7.1.

As shown in figure 5.8, the handling of each packet depends upon which IC header type the packet has. If the packet is tagged "ICH_FIND_FILE", the next step is to check if the file resides on this machine. If the file is not found, the TTL is decremented, the port the packet is received on is pushed on the route, and the packet is transmitted on all IC ports except the one it arrives on. If the file is found on this machine, a search is done to try to find an IC session (see section 5.7.3) that matches. If a matching IC session is found, the same setup packet has arrived before through another route, and the packet is dropped. If no matching IC session is found, a new server IC session is created. This session keeps the return route that this stream is to use. To make sure that control messages arrive at the server by the correct route, a feedback IC message is created (" ICH_ROUTE_FB") and sent through the route to the egress. The original SETUP packet is then passed to the host on the XScale-host socket.

If the packet received is of the type "ICH_CTRL_FB", the payload is an RTSP reply message. The payload is passed to `tcp_send` along with IP addresses and TCP ports (to be able to find the correct TCP session), and transmitted to the client.

When a packet of the "ICH_CTRL_MSG" type arrives, the payload is an RTSP request bound for the media server. This packet is directly forwarded to the host on the XScale-host socket.

The last IC packet type handled by the "int A" thread is the "ICH_ROUTE_FB". This packet type is transmitted to the egress machine after a route has been found and the streaming server has created its IC session. An egress IC session used to guide incoming RTSP packets to the correct machine is created, and the packet is discarded.

If a packet arrives that matches none of the above categories, the packet is dropped, but this should not happen if the system works correctly. The reason "ICH_RTP" never reaches this stage of processing is that once sent into the routing framework, these packets never surface to be examined by the XScale, but are forwarded straight to the egress node.

5.7.7 Host-XScale data path

In addition to the "int A" thread, there is a program loop running on the XScale that is polling the host-XScale socket for incoming data. This data is formatted in such a way that an IC header always arrives first. This header contains a byte count for the packet payload, so that the packets can be distinguished from one another. If the packet is of the type ICH_ROUTE_FB or ICH_CTRL_FB, we know that an IC header extension comes after the header (see section 5.7.1). If the destination port (popped from route) is 0xff and the packet is of the type ICH_CTRL_FB, the packet is passed to `tcp_send`. In all other cases, the packet is routed as is on the port popped from the route field of the IC header.

5.7.8 Host tasks

The host application is implemented in userspace with a TCP socket providing the data stream to the XScale. For each RTP stream, a processing thread is generated. It opens the file, prepends each data chunk with the needed headers, and transmits the data on the host-XScale socket.

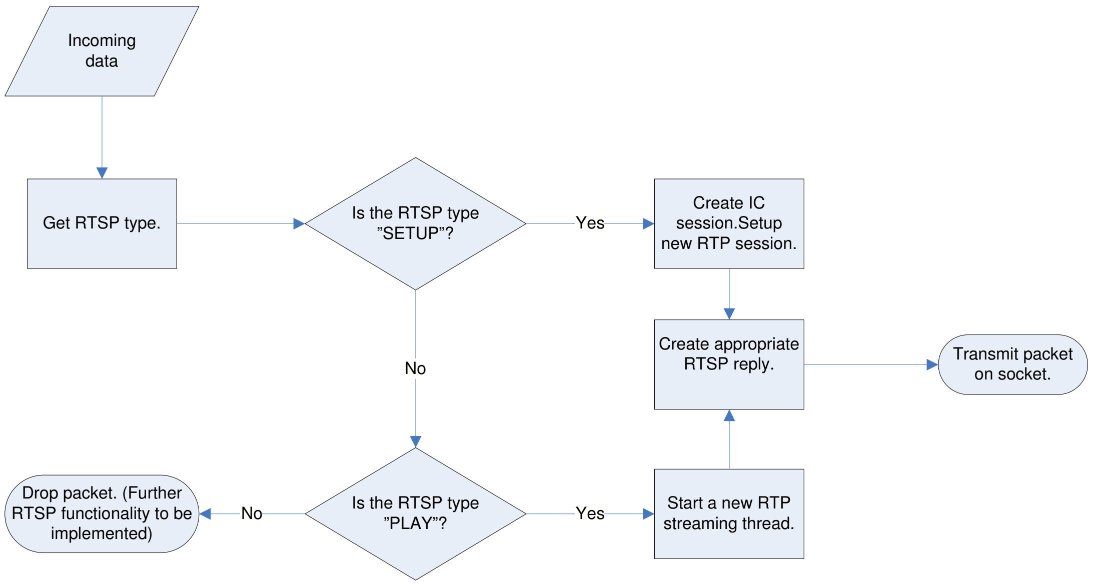

Incoming data
Get RTSP type.
Is the RTSP type "SETUP"?
Yes
Create IC session.Setup new RTP session.
No
Create appropriate RTSP reply.
Transmit packet on socket.
Drop packet. (Further RTSP functionality to be implemented)
No
Is the RTSP type "PLAY"?
Yes
Start a new RTP streaming thread.

Figure 5.9: Flow chart for Host when receiving data from XScale.

Figure 5.9 shows the data flow for the host application when an incoming packet is received. When data is received from the XScale, the first step is to get the payload size and the IC type. Any IC header extension is also read. The RTSP packet type is then read. If it is a SETUP packet, a new IC session is created, and a new RTP session (see section 5.8) is prepared. We already know that the file is located on this machine. This was checked on the XScale. The RTSP reply to the host is then generated stating the server port numbers, the reply is wrapped in an IC header, and the IC packet is sent on the socket. If a PLAY message arrives, an RTP streaming thread is created. The program the generates an RTSP reply message, and transmits the reply on the socket to the XScale.

5.8 RTP-session

Though the current version of the cube does not implement genuine RTP functionality, it keeps some information about each RTP stream. This information is kept in the RTP session struct, which is comprised of the following components:

- **Thread ID:** An identifier for the streaming thread. Used to send signals or forcefully shut down the stream.
- **Client RTP Port:** The port negotiated by RTSP to deliver data to the client.
- **Client RTCP port:** The port negotiated by RTSP to deliver RTCP data. Not in use as of this version.
- **Server RTP port:** The server port delivering the stream. In this implementation, this port is a "phoney", because a regular socket mechanism is not used.
- **Server RTCP port:** The port used by the server to send RTCP packet. Not in use as of this version.
- **Index of IC session:** The index of the IC session corresponding to this RTP session. Used to look up IP and MAC addresses.
- **Filename:** The name of the wanted media file. Used to open the file for streaming at the server.

As the description shows, there are many crucial RTP components that is not yet implemented. These include RTCP feedback and rewinding or forwarding the stream. Eventual RTCP packets will be discarded by the μEngine code (see section 5.7.5). The current implementation supports the streaming of a file from beginning to end, as is, and with a fixed bitrate. The reason for this is that the focus has been on implementing a working routing framework for the cube and exploring the benefits of offloading this kind of routing functionality on the IXP μEngines.

When an RTSP PLAY message is received, the media server on the host creates a new media streaming thread. The first operation that is performed when a new thread is started is to build the RTP, UDP, IP, ethernet and IC headers that will wrap the media data. Since only a few of the header fields are changed from packet to packet, these headers are saved, rather than creating the entire header again for each data packet. This is done by having only one transmission buffer of the current MTU size. The headers are kept statically in the beginning of the buffer, and the data from disk is copied into the payload part. For each of the data chunks read from disk, the length field is updated in the IC, IP and UDP header. The sequence number and timestamp are updated in the RTP header. Then, checksums are calculated for UDP and IP, and the packet is sent to the XScale on the socket. The thread then goes to sleep until it is time for another packet to be sent.

5.9 Microblock designs

How the different tasks should be prioritized between the microblocks was an issue that strongly depended on the choice of routing algorithm. Another deciding factor was the method of moving data from the host to the μEngines. Given the possibility of using 8 μEngines in parallel, there were many tasks that could be distributed and optimized by distributing them on several μEngines. The fact that each μEngine also supports eight program threads, further enhanced this possibility.

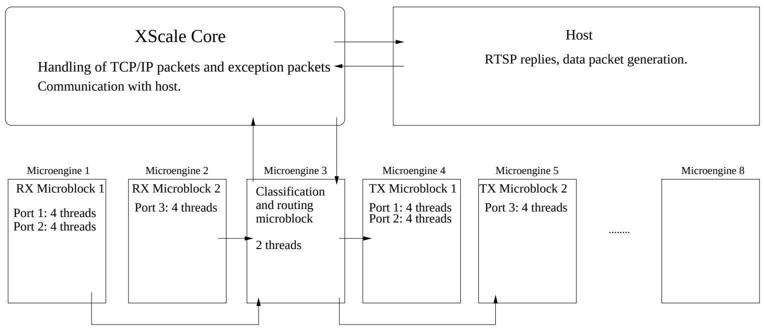
If the network processing application that is to be written expects a very heavy load, steps can be taken to enhance RX and TX performance. The reference designs of RX and TX microblocks provided by Intel give the programmer the possibility of distributing the task (RX or TX) on two microblocks (figure 5.10(a)). Four threads are assigned to handling each of the ports for receive and transmit, giving the performance of the microblocks a potential efficiency increase in comparison to the one-microblock RX and TX designs. The actual effects of this enhancement depends on the transmission medium. Implementing this solution reduces the chance of queues that in the worst case can lead to packets being dropped. This measure would, however, be in vain if the packet processing between RX and TX is slower than the receive and transmit process. In order to meet with this problem, steps should be taken to optimize other packet processing tasks. Figure 5.10(b) shows the same application with the two main tasks split between two μEngines.

As long as each step in the pipeline between RX and TX is kept at a level where fewer cycles is consumed for each packet than would the RX and TX block, the packet flow would be optimal. From this a rule of thumb for microblock design can be derived:

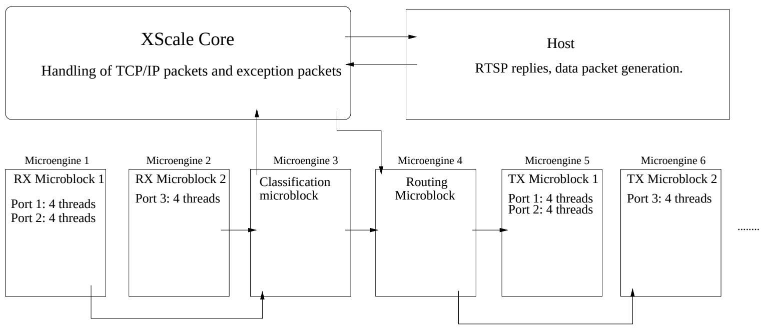
$$C_{PP} \leqslant C_{TX} \leqslant C_{RX} \tag{5.1}$$

where C represents cycles used, PP is each individual microblock used in packet processing, TX is transmit microblock(s) and RX is receive microblock(s). In practice this means that the system should be able to transmit packets at least as quick as they can be received, and every individual stage of the packet processing pipeline in between should spend less time on processing one packet than the transmit block(s).

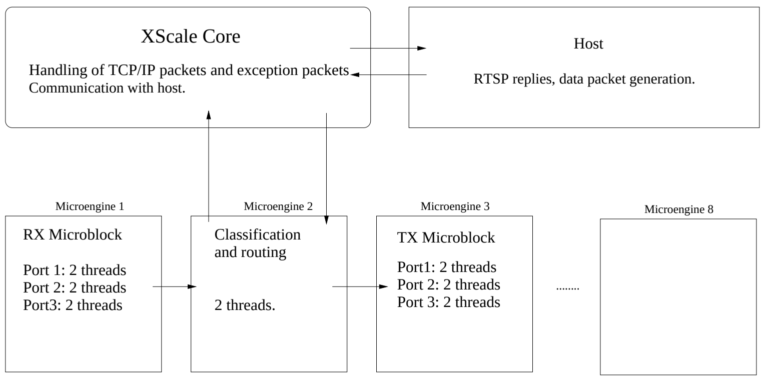
The handling of exception packets and other traffic that has to be forwarded to the XScale will of course impede the packet processing pipeline severely. It is therefore crucial to make sure that the bulk of the packet traffic is handled at the μEngines. In the implemented system, however, the XScale-host transfer bottleneck slows the system down to a point where the μEngines can easily handle the traffic load it is presented with. The packets are, as of now, being generated on the host, then moved by TCP socket to the XScale for then to be sent to the μEngines. Until a more efficient way of moving data from the host to the μEngines can be implemented (see section 6.3), it is sufficient to keep the RX and



(a) RX and TX blocks split on two μEngines.



(b) Two microblocks for RX, TX and packet processing.



(c) The current microblock design.

Figure 5.10: Three microblock structuring alternatives.

TX functionality on one microblock each, and the packet processing on one microblock utilizing two program threads. This design is visualized in figure 5.10(c).

When data can be moved directly from the server to the μEngines, more of the functionality of generating packet headers can be delegated to the μEngines. This will be discussed in more detail in section 6.3.

5.10 Evaluation

This set of tests is done with the purpose of measuring the forwarding time of packets routed in the cube server implementation. Since the performance of data transport from the host down to the μEngines yields bitrates of less than 10Mbps due to the lack of efficient host-μEngine communications, a test of the system as a whole would give little or no information about how packets would move through a heavily loaded system once this bottleneck is removed. The test focus is, therefore, on the underlying routing mechanisms. The results from the μEngine measurements are then compared to packets routed through a conventional switched network to try to determine the possible performance gain by implementing the routing in this network topology using IXP cards.

5.10.1 Packets routed through a switched network

The first test emulates a server cluster interconnected by a dedicated conventional switched network. Figure 5.11 shows the test setup for this configuration. The test is conducted by transmitting ICMP ping requests from the "Cube1" host destined for "Cube2". The forwarding machines ("Cube2", "Cube3" and "Cube4") have iptables DNAT and SNAT rules that forwards the packet to the next host in the chain, thus sending the packets in a complete circle through the hosts and back to the sender ("Cube1"). There is an iptables rule firewalling ICMP requests in order to prevent the kernel from replying to the messages.

The packets that are sent from "Cube1" are timed on the way out by tcpdump [60]. When they arrive again from "Cube4", the arrival time is recorded. The difference between the send and arrival time give us the time used for routing the packets through the four hosts.

5.10.2 IXP hypercube routing tests

To be able to measure the amount of time spent on the routing process in the IXP cube implementation, a special timestamping mechanism had to be developed. Since every host relates its timestams to its own time domain, the packets had to be timestamped on

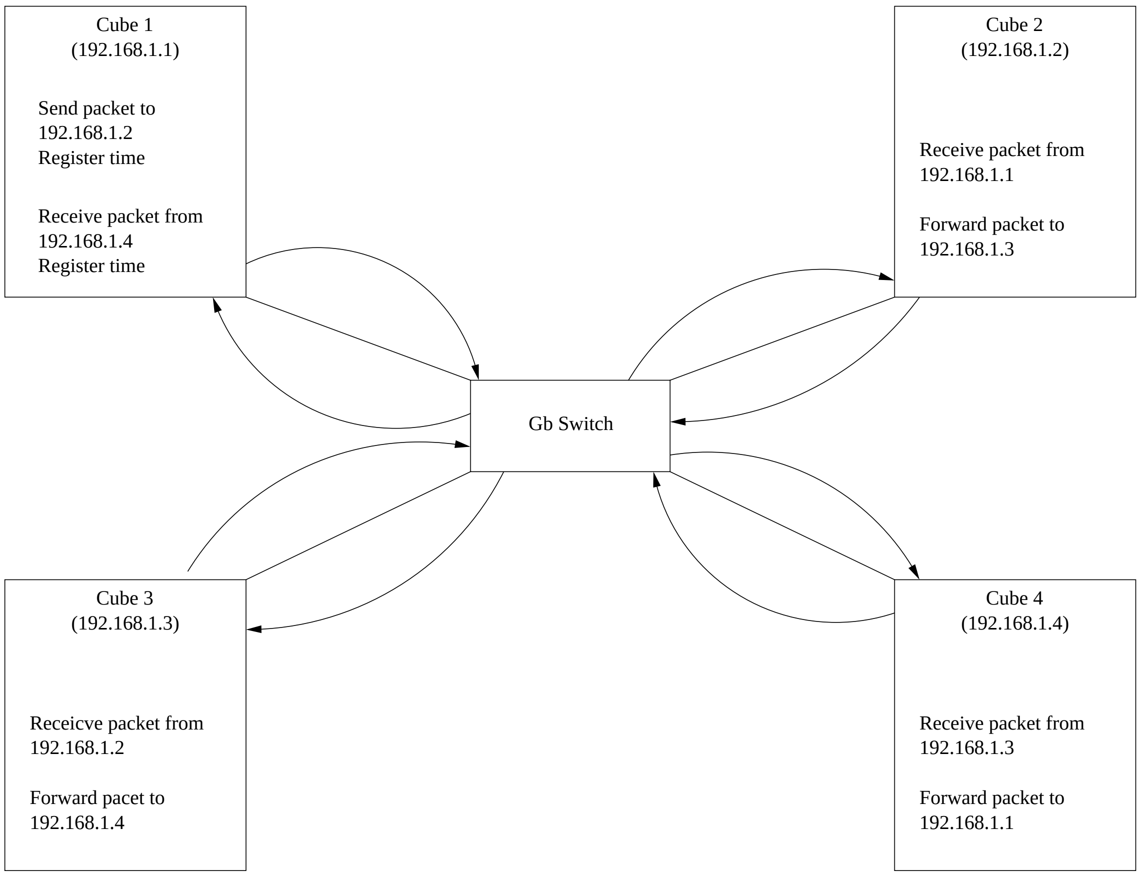


Figure 5.11: Test setup: Routing using switched network.

the same host. To accurately be able to measure the routing mechanisms, the timestamps should be done on μEngines. Tests have shown that even when synchronized at a designated point of code execution, there are still differences in the clocks of each microengine. This implies that a timestamp has to be made by the same IXP μEngine on the same host.

This challenge was solved by creating a new IC header type called "ICH_TIMER". Packets given this designation is to be timestamped, traverse the entire "cube" and return to the sending host. Upon return, the packet is timestamped again on the same μEngine as on the way out. This give us an accurate time for the packet to traverse the entire cube (4 hosts) in a manner similar to the switched network.

The route given to the "ICH_TIMER" packet is processed in the same way as every other IC packet to ensure a realistic estimate of the processing time.

Figure 5.12 shows the test setup for this process. The "ICH_TIMER" packet is created on the host, and passed to the XScale and consequently to the microengines in the same

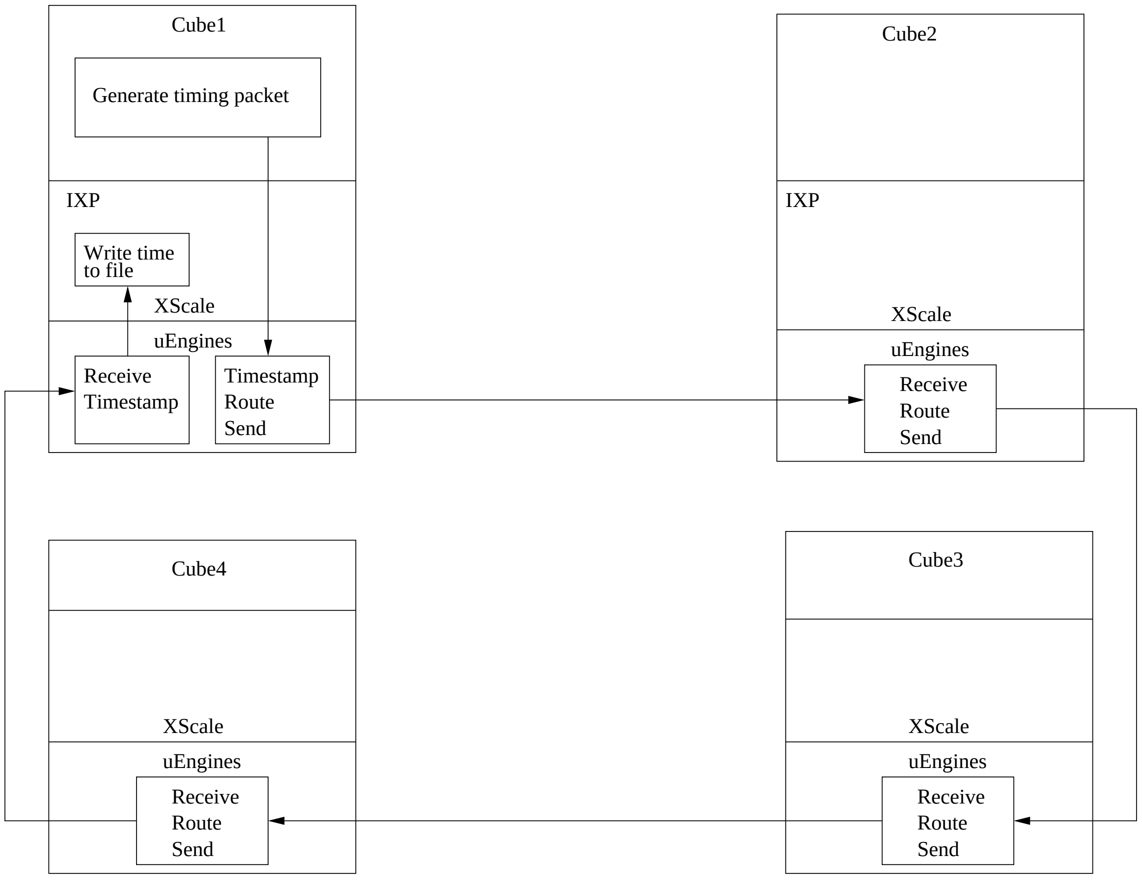


Figure 5.12: Test setup: Timing of routing in the IXP hypercube.

manner as for instance an RTP packet in the ***VS*^{3}** server solution. The packet is then forwarded to the μEngine where it is timestamped. The send port is then popped from the IC header route field, and the packet is sent to the next host.

On the "Cube2", "Cube3" and "Cube4" machines the μEngines receive the packet, checks the packet type to confirm that it is to be routed, routes the packet to the port given by the IC header route field, and transmits the packet.

When the packet arrives back at the "Cube1" machine, the route field of the IC header has no valid port number, telling the system to forward it to the XScale, and the fact that the packet is of the type "ICH_TIMER" tells the μEngine to timestamp the packet before forwarding it to the XScale. The timestamp is made to not overwrite the timestamp made on the way out.

On the XScale the two timestamps are read. The difference is then calculated, and printed to a file on the mounted NFS filesystem used by the IXP card. The packet sizes used for the test was the same as for the switched network test. The results from this test are presented in table 5.1 with the results from the switched network test.

5.10.3 Comparison of routing times

Table 5.1 shows the statistical data of this test and the test discussed in section 5.10.2. Both tests are performed with ICMP packets of 98 bytes and 1500 bytes.

The average times for completing one "circuit" in the machine topology is graphically displayed in figure 5.13. As expected, the total processing time is radically larger for the larger packet size (1500 bytes). We can also see that the benefit of implementing routing in a dedicated interconnected topology (like the hypercube), compared to using a switched network structure, is significant with respect to inter-server communication. This can be attributed not only to the extra transmission times introduced by having the packet move through the switch, but also to the overhead that the layered handling structure of the kernel requires. Since this does not have to be supported by the intra-cube routing mechanisms of the IXP hypercube implementation, the routing costs diminishes. This effect is magnified by the simple nature of the routing mechanism.

Packet size	**Max time**	**Min time**	**Avg time**	**Median**	**Std. dev.**	**Count**
Switched network times						
98 Bytes	9022μs	9μs	135,98μs	129μs	71,481μs	99928
1500 Bytes	2495μs	261μs	265.6μs	265μs	8.2152μs	99991
IXP cube routing times						
98 Bytes	30μs	27μs	28.138μs	28μs	0.3544μs	63965
1500 Bytes	109μs	106μs	107.0034μs	107μs	0.1367μs	63965

Table 5.1: Routing times for IXP cube and switched network routing.

For 98 bytes packets, the switched network implementation is 483% more time consuming than the IXP hypercube solution (a difference of 108μs in average). For 1500 bytes packets, the switched network implementation time is 248% higher than the IXP hypercube time (a difference of 159μs in average). This can be explained by the copy operations occupying a greater share of the total time consumed.

Based on these numbers we can say that a directly interconnected network topology will have a vast advantage in comparison to a switched network solution. An increase in routing algorithm complexity will probably make the difference smaller, but still large enough to make a significant difference.

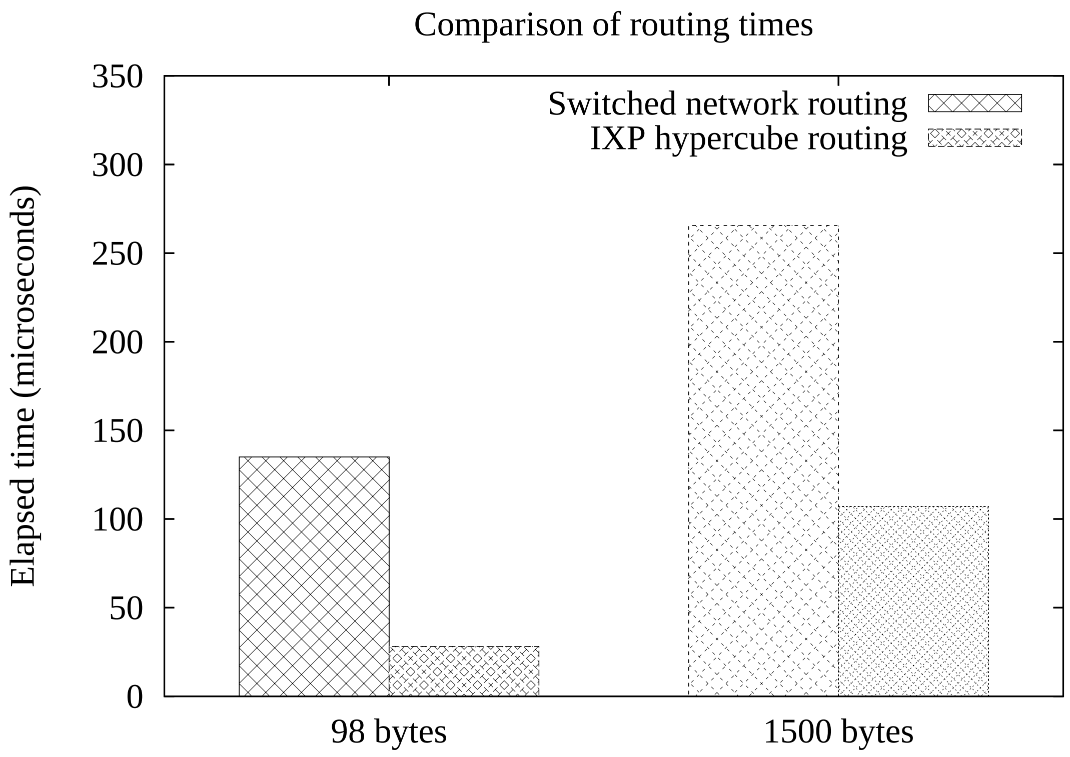


Figure 5.13: Comparison of average routing times.

5.10.4 Processing times given increased load

To try to find out what level of impact an increase in load i.e., will have on the IXP hypercube system, the loop routing test was done again, this time with packets being generated on more than one of the hosts. For each new host generating packets, the load on all hosts increases because all packets have to be forwarded through all hosts. Although the bit rate of each packet stream is not high (about 2.4Mb/s) it will give an indication of how load times will behave.

Table 5.2 shows the statistical data from this test transmitting 1500 bytes packets. To maximize the bitrate given the sending intervals permitted by the host-ixp implementation, a packet size of 1500 bytes was used. The reason why the number of measurements (count) increases by such a degree is that more transmitting nodes give more valid samples. The number of measurements for four nodes should be approximately the same as four times the number of measurements for one node.

Figure 5.14 plots the time spent on traversing the loop given the increased load. The

Transmitting nodes	**Max time**	**Min time**	**Avg time**	**Median**	**Std. dev.**	**Count**
1	109μs	106μs	107.0034μs	107μs	0.1367μs	63965
2	141μs	106μs	107.36μs	107μs	2.6691μs	121395
3	142μs	106μs	108.47μs	108μs	3.532μs	191846
4	202μs	106μs	108.92μs	108μs	4.8546μs	255709

Table 5.2: Routing times on μEngines given increased load.

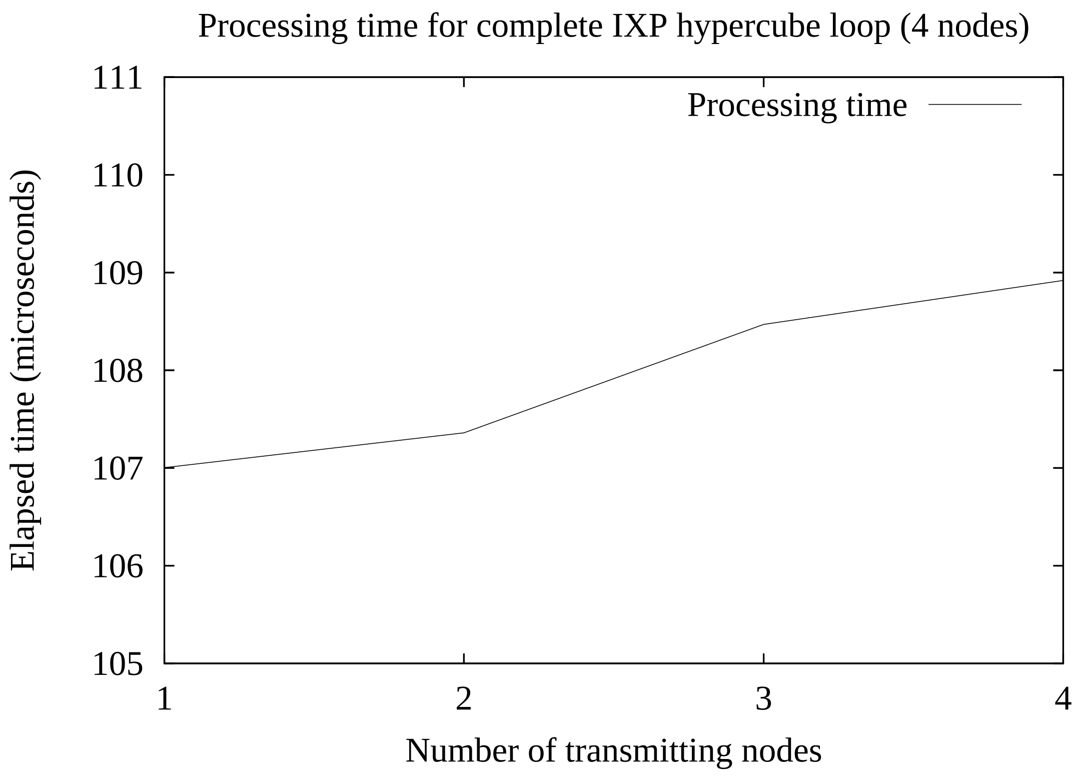


Figure 5.14: Development of processing time with increased load.

graph shows an increase in load time, but not a significant one[9]. This indicates that the system should be able to handle much higher loads, before having to drop packets. The chipset is built to be able to handle 1Gb/s on the three interfaces, so given that an efficient processing pipeline can be implemented, the platform should be able to accomodate high loads.

[9]Note that the y-axis starts on 105, and that the actual difference is below 2 μs

5.11 Summary

This chapter has described the process of implementing our ***VS*^{3}** video server cube with routing functionality based on the capabilities of IXP2400 cards. Several designs were proposed to achieve this. The first design aimed at using an existing server application on the host. This was discarded in our prototype, due to the complexity that this would introduce to packet routing and translation, but should be considered in a later stage. The design that was finally chosen reused the existing RX and TX components from the static forward application. The rest of the components had to be built from scratch. The streaming server application running on the host had to be aware of the underlying routing framework. This was to ensure that a future implementation using DMA to effectively move the data to the μEngines could reuse the code (see section 6.3), and to avoid unneccesary complexity.

A video server hypercube was implemented with routing functionality located on the μEngines, TCP support and uplink to host on the XScale, and a streaming server on the host. The server solution was tested with several streams delivered from the different content servers through the same cube egress. Setup and routing worked as expected, and the video data was delivered as intended. The bitrates, however, were hampered by the host-XScale bottleneck, thus impeding the performance of the server.

Two tests have been performed on the routing framework. The first was a measurement of the time used by a data packet to traverse a loop of four machines in the system. A similar test has been run on a group of Linux hosts linked in a switched network. The second test measured the increased processing time when doubling and quadrupling the load.

These test show, as expected, that the directly connected hypercube topology delivered the routed packets much faster than the switched network could manage. This is a strong argument for implementing data-intensive servers like a multimedia streaming server in such a topology. When increasing the load, there was a slight increase in processing time. To be able to put serious pressure on the routing system, however, packet generators have to be applied, providing heavier loads.

Chapter 6

Conclusion

This chapter concludes the master thesis. We will present a short summary of what has been done, then the most important results of this work, and finally some key issues that should be pursued in the future.

6.1 Summary

In this thesis, we have investigated the behaviour of Intel IXP network processors, with focus on the IXP2400 and the IXA SDK. A series of test implementations have been done to find the strengths and weaknesses of both the hardware and software platform. Tests have been run on the different applications to find out whether offloading network tasks to this platform can be valuable. The knowledge gained from the experiments has been used to construct a multimedia server cube solution, similar to the nCube n4x, using network processors for offloading routing tasks.

6.2 Results

From the work done on the ENP2611 and our $\boldsymbol{VS^3}$ video server cube we have gained valuable knowledge of hardware and software issues related to programming the IXP2400 NPU. We have also run a number of tests that were done to evaluate performance of the implemented systems. The tests performed on the applications showed that offloading network operations on a Linux host frees a noticeable amount of resources, even for simple networking tasks. We could also observe that the time needed to perform a simple task of editing some protocol header fields was performed more efficiently on the NPU than on a Linux host. We conclude that wire-speed processing should, to as large degree as possible, be kept on the μEngines, while processor-intensive tasks and control operations should be put on the XScale. Measurements showed that when using the interrupt "int A" method of passing packet references to the XScale, the processing times were only slightly higher than the results for the Linux host and the μEngines. This shows that the XScale can be used quite efficiently without impeding performance significantly.

From the ***VS*³** implementation we found that the greatest challenge was to move data efficiently from the host to the IXP as there is no support for DMA transfers yet. However, focusing on the internal cube communication, tests on the implemented routing framework showed that this method of intra-cube routing was efficient. Tests showed that, compared to implementing the server cluster in a switched network, the hypercube topology with direct interconnection was far more efficient.

In conclusion, our prototype shows that the IXP2400 NPU can efficiently offload the host machine and provide a favourable way of implementing closely interconnected servers in a cube topology.

6.3 Further work

In this section, we present a short description of what we deem to be the most important issues to address in future work on this subject.

During the course of implementing the ***VS*³**, it soon became clear that the main bottleneck of the system would be the host-IXP data transfer. Hardware documentation and experiences from research projects elsewhere using the same hardware tells us that DMA can be used to move data directly from the host to the μEngines over the PCI bus. This can be done without having to go through the XScale as of the present implementation. This would help the system in two ways:

1. It would dramatically enhance transfer speeds. If a 64 bit bus could be used effectively, it would be able to feed enough data to fully utilize the optical interfaces of the IXP card.

2. It would enable the host to transmit the data only, leaving some or all of the header generation operations to be done on μEngines. Allocating one or more μEngines to the task of packet preparation could be done without stealing resources from neither host nor existing IXP functions.

Work is currently in progress in our research group to find methods of doing this efficiently on the IXP2400.

To be able to reap the full benefits of RTP/RTCP and RTSP on the ***VS*³**, many improvements have to be done. As of this implementation, the system only has support for RTSP

SETUP and PLAY, and basic RTP sending. To enable further control of the streaming, we should complete the RTSP and RTP implementations. To be able to receive RTCP receiver reports, the filter in egress node μEngines must have functionality to dynamically update which ports (and protocols) should be allowed to enter the cube, according to the port numbers conveyed by the RTSP SETUP exchange.

The source routing algorithm should be extended to find a new route if a link is broken during streaming. Another feature that should be implemented is the ability to reserve resources along the route, and to choose the optimal route based on recorded reservations.

Finally, there were some tests that could have been performed to support the data already collected, if time had permitted.

To get an even more accurate comparison of the routing performance compared to Linux hosts, a test could have been run monitoring the routing time for a group of directly connected Linux hosts. By eliminating the time used for switching, we could get more precise data for the time saved by processing packets on the IXP.

It is possible that, even with the kernel having priority, the packet processing time would suffer from a heavily loaded host. We could therefore have measured the time used by a Linux host to process the IP header switching when loaded with heavy work (like a tar process).

To find out which kind of load the μEngines can take before having to drop packets, a packet generator (or several) could be used to strain the IP header switch application.

Bibliography

[1] The Agere website. November 2005, http://www.agere.com.

[2] Alex Arachnid server website. October 2005, http://www.alex.com/pages/anglais/fond_stream.html.

[3] The AMCC website. November 2005, http://www.amcc.com.

[4] The C-COR website. http://www.c-cor.com/.

[5] Anton Chuvakin. Iptables linux firewall with packet string-matching support. Published at SecurityFocus: http://www.securityfocus.com, 2001. October 2005, http://www.securityfocus.com/infocus/1531.

[6] Douglas E. Comer. *Using Network Processors, Intel IXP2xxx version*. Pearson Education, Inc., 2005.

[7] California software laboratories: RTSP techguide. October 2005, http://www.cswl.com/whiteppr/tech/StreamingTechnology.html.

[8] Apple Darwin streaming server. November 2005, http://developer.apple.com/darwin/projects/streaming/.

[9] The DivX website. October 2005, http://www.divx.com.

[10] The Dresden real-time operating system (DROPS) website. November 2005, http://web.inf.tu-dresden.de/SyA/lsbs/project/overview.html.

[11] DVD demystified website. November 2005, http://www.dvddemystified.com/.

[12] RadiSys ENP2505 hardware reference manual. November 2005, http://www.radisys.com/files/support_downloads/007-01266-0002.ENP-2505.pdf.

[13] RadiSys ENP2611 hardware reference manual. November 2005, http://www.radisys.com/files/support_downloads/007-01419-0003.ENP-2611HW.pdf.

[14] Flac - free lossless audio compression website. October 2005 http://flac.sourceforge.net.

[15] The Free On-line Dictionary of Computing. November 2005, http://foldoc.doc.ic.ac.uk/foldoc/index.html.

[16] The Fraunhofer website. October 2005, http://www.iis.fraunhofer.de.

[17] J. Gettys, J Mogul, H. Frystyk, L. Masinter, P Leach, and T. Berners-Lee. RFC3550, 1999. October 2005, http://www.ietf.org/rfc/rfc2616.txt.

[18] Moore Gordon. Cramming more components onto integrated circuits. Electronics Magazine, 1965. October 2005, ftp://download.intel.com/museum/Moores_Law/Articles-Press_Releases/*Gordon_Moore*_1965_*Article.pdf*.

[19] A. Griffiths and G. Metherall. Cluster interconnection networks. Report for CSC433, Monash University, 1999. Available online: http://www.buyya.com/csc433/ClusterNets.pdf.

[20] Carsten Griwodz, Michael Liepert, Michael Zink, and Ralf Steinmetz. Tune to lambda patching. *ACM Performance Evaluation Review*, 27(4):20–26, March 2000.

[21] Pål Halvorsen. Improving I/O performance of multimedia servers. *Thesis for the Dr. Scient. (PhD) degree at University of Oslo, Published by Unipub forlag, ISSN 1501-7710, No. 161, Oslo, Norway*, 2001.

[22] Pål Halvorsen and Carsten Griwodz. INF5060 - multimediakommunikasjon med nettverksprosessorer. IFI, UIO.

[23] Pål Halvorsen and Carsten Griwodz. INF5070 - media storage and distribution systems. IFI, UIO.

[24] Pål Halvorsen, Carsten Griwodz, Vera Goebel, Ketil Lund, Thomas Plagemann, and Jonathan Walpole. Storage system support for multimedia applications, part I and II. *IEEE Distributed Systems Online, Vol 5, No. 1/2, Jan/Feb 2004*, 2003.

[25] IBM VideoCharger. October 2005, http://www-306.ibm.com/software/data/videocharger.

[26] The Internet Machines website. November 2005, http:/www.internetmachines.com.

[27] The Intel website. November 2005, http://www.intel.com.

[28] Intel IXA portability framework reference manual., 2003.

[29] Intel IXA SDK 2.01 programmers reference manual.

[30] Intel IXA SDK 3.51 programmers reference manual. November 2005, http://www.intel.com/design/network/products/npfamily/sdk_download.htm#351.

[31] Intel IXP1200 hardware reference manual.

[32] Intel IXP2400 hardware reference manual.

[33] RadiSys ENP SDK 3.5 programmers guide.

[34] Generic linux kernel for the IXP2xxx hardware open source initiative. November 2005, http://ixp2xxx.sourceforge.net/.

[35] David B Johnson, David A. Maltz, and Yih-Chun Hu. The dynamic source routing protocol for mobile ad hoc networks (DSR), 2004. November 2005, http://www.ietf.org/internet-drafts/draft-ietf-manet-dsr-10.txt.

[36] Komssys pages at sourceforge. October 2005, http://komssys.sourceforge.net.

[37] The Monta Vista website. http://www.mvista.com/.

[38] The Motorola website. November 2005, http:/www.motorola.com.

[39] The fraunhofer MP3 website. November 2005, http://www.iis.fraunhofer.de/amm/techinf/layer3.

[40] Whatis.com - definition of "multimedia". "October 2005, http://searchwebservices.techtarget.com/sDefinition/0,,sid26_gci212612,00.html".

[41] Chuck Narad. Communications and Moore’s Law. Keynote at Intel 10th Academic Forum, 2005. October 2005, http://download.intel.com/corporate/education/EMEA /academicforum/keynotes/Narad/Keynote%20Chuck%20Narad.pdf.

[42] The nCube website. Site decommissioned when the company was sold to c-cor. Old URL: http://www.ncube.com/.

[43] The Intel network processor family website. October 2005, http://www.intel.com/design/network/products/npfamily/index.htm.

[44] The Norwegian broadcasting company (NRK). November 2005, http://www.nrk.no.

[45] The Ogg Vorbis website. November 2005, http://www.vorbis.com.

[46] The Oracle interMedia website. November 2005, http://www.oracle.com/technology/products/intermedia/index.html.

[47] T. Plagemann, V. Goebel, P. Halvorsen, and O. Anshus. Operating system support for multimedia systems, 2000.

[48] The PMC-Sierra website. November 2005, http:/www.pmcsierra.com.

[49] The Qlogic website. October 2005, http:/www.qlogic.com.

[50] Apple Quicktime streaming server. October 2005, http://www.apple.com/quicktime/streamingserver/.

[51] The RadiSys website. October 2005, http://www.radisys.com.

[52] Sanjay Ranka, Youngju Won, and Sartaj Sahni. Programming a hypercube multicomputer. *IEEE Software Magazine*, 1988.

[53] Real Networks website. November 2005, http://www.realnetworks.com.

[54] Real Networks media delivery website. October 2005, http://www.realnetworks.com/products/media_delivery.html.

[55] H Schulzrinne, S Casner, R Frederick, and V Jacobson. RFC3550, 2003.

[56] H Schulzrinne, A Rao, and R. Lanphier. RFC2326, 1998. November 2005, http://www.ietf.org/rfc/rfc2326.txt.

[57] Frank Olaf Sem-Jacobsen, Tom Anders Dalseng, and Andreas Petlund. Achieving intra-cube routing using Intel IXP cards. Assignment in INF5070: Multimedia storage and distribution systems, 2003.

[58] The SF-anytime website. November 2005. http://www.sf-anytime.com/.

[59] Dinkar Sitaram and Asit Dan. *Multimedia Servers - Applications, environments, and design*. Morgan Kaufmann Publishers, 2000.

[60] The tcpdump/libpcap project homepage. November 2005: http://www.tcpdump.org/.

[61] The Vitesse website. November 2005, http://www.vitesse.com.

[62] The World of Warcraft website. November 2005, http://www.worldofwarcraft.com/.

[63] George Kingsley Zipf. Human behaviour and the principle of least-effort. *Addison-Wesley, Cambridge MA*, 1949.

Appendix A

VS^3 video server cube source

A.1 ICrouter microblock source

A.1.1 dl_system.excerpt.h

```
/*************************************************************
 **                                                                      **
 **   This file contains definitions of scratch rings              **
 **   and shared memory areas for the VS3 application.            **
 **   This is only an excerpt.                                        **
 **   The whole file could not be included due to                    **
 **   Intel proprietary regulations.                                  **
 **   To get the whole picture, access the file                     **
 **   dispatch_loop/dl_system.h in the build tree.                **
 **                                                                      **
 *************************************************************/

/*-----------------------------------------------------------------------
 * i) Base address for meta data (buffer descriptors) of packet
 *       buffers - for free list 2 - dcube
 *-----------------------------------------------------------------------*/
#ifndef                 DCUBE_NUM_BUF_HANDLES
#define                 DCUBE_NUM_BUF_HANDLES  1024
#endif

#ifndef                 DCUBE_SRAM_BASE
#define                 DCUBE_SRAM_BASE             0x80000
```

```
#endif

  /* Size of meta-data (buffer descriptor) in bytes for each buffer.
     Should be a power of 2. */
#ifndef         DCUBE_META_DATA_SIZE
#define         DCUBE_META_DATA_SIZE            32
#endif

  /* Total SRAM size allocated in bytes for meta data.
     In simulation we use only 20KB. */
#ifndef         DCUBE_SRAM_SIZE
#define         DCUBE_SRAM_SIZE         (DCUBE_NUM_BUF_HANDLES * DCUBE_META_DATA_SIZE)
#endif

#ifndef         DCUBE_SRAM_MAX
#define         DCUBE_SRAM_MAX          DCUBE_SRAM_BASE + DCUBE_SRAM_SIZE
#endif

  /*------------------------------------------------------------------------
 * i) Base address for packet buffers. This is the actual packet data.
 *      --dcube
 *-----------------------------------------------------------------------*/
#ifndef         DCUBE_SDRAM_BASE
#define         DCUBE_SDRAM_BASE                0x2000000
#endif

/*        Size of one packet buffer in bytes.   */
#ifndef         DCUBE_BUFFER_SIZE
#define         DCUBE_BUFFER_SIZE                       2048
#endif

/* Total DRAM size allocated in bytes for packet buffers. */
#ifndef         DCUBE_SDRAM_SIZE
#define         DCUBE_SDRAM_SIZE          (DCUBE_NUM_BUF_HANDLES * DCUBE_BUFFER_SIZE)
#endif

/*
 * Scratch ring packet_echo --> XScale
 */
#define         PE_XSCALE_COMM_RING     10
#define         PE_XSCALE_COMM_BASE     8192
```

```
#define                 PE_XSCALE_COMM_SIZE           512


/*
 * Scratch ring XScale --> packet_echo
 */
#define                    XSCALE_PE_COMM_RING       11
#define                    XSCALE_PE_COMM_BASE       9216
#define                 XSCALE_PE_COMM_SIZE           512
```

A.1.2 ICrouter.h

```
/*********************************************************
**                                                      **
** Includes and definitions for the ICrouter microblock  **
**                                                      **
*********************************************************/

#ifndef _ICROUTER_H
#define _ICROUTER_H

#include <ixp.h>
#include <dl_system.h>
#include <dl_buf.c>
#include <dl_meta.c>
#include <hardware.h>
#include <sig_functions.h>
#include <ixp_lib.h>
#include <ixp_crc.c>
#include <rtl.c>
#include <ix_cc_microengines_bindings.h>

#define     PKTHDR_CACHE_SIZE           5             /* Number of quadwords to read in isRTSP() */
#define  ETHPROT_IP                    0x0800    /* Ethernet protocol for filter */
#define  IPPROT_TCP                    6             /* IP protocol for filter */
#define  SERVER_PORT                    9070         /* Server port for filter */

extern dl_buf_handle_t dlBufHandle;                      /* The current buffer handle */
extern dl_buf_handle_t dlEopBufHandle;                   /* For large packets, this is the last buffer in the chain. */
extern dl_meta_t dlMeta;                          /* Metadata struct */
```

```
__declspec(gp_reg) int dlNextBlock;                /* Next block that should process the buffer/packet */

/* IC packet types */
enum {
   ICH_FIND_FILE  =  1,
   ICH_ROUTE_FEEDBACK,
   ICH_CONTROL_MSG,
   ICH_RTP
};

#endif /* #ifndef _ICROUTER_H */
```

A.1.3 ICrouter.c

```
/************************************************
**                                              **
**   The ICrouter microblock and support methods   **
**                                              **
************************************************/

#include "ICrouter.h"

/* Read and return the input port from metadata */
int getInputPort(unsigned int bufHandle){
   /* Check if packet came in on port 0 */
   __declspec(sram_read_reg) unsigned int metaData[5];
   __declspec(gp_reg) unsigned int metaOffset;

   SIGNAL sig_sram_rw;

   metaOffset = bufHandle << 2;
   sram_read (metaData, (volatile void __declspec(sram) *)metaOffset, 5, \
                  sig_done, &sig_sram_rw);
   wait_for_all(&sig_sram_rw);
   return (metaData[3] >> 16);
}

/* Read and return the output port from metadata */
int getOutputPort(unsigned int bufHandle){
   __declspec(sram_read_reg) unsigned int metaData[5];
```

```
    __declspec(gp_reg) unsigned int metaOffset;

    SIGNAL sig_sram_rw;

    metaOffset = bufHandle << 2;
    sram_read (metaData, (volatile void __declspec(sram) *)metaOffset, 5, \
                      sig_done, &sig_sram_rw);
    wait_for_all(&sig_sram_rw);
    return (metaData[3] & 0xffff);
}

/* Pop the next port from route field of IC header */
int popPort(){
    uint32_t route;
    char port;
    __declspec(sdram) unsigned char *p_pkt_hdr;

    p_pkt_hdr = (__declspec(sdram) unsigned char *)
        (Dl_BufGetData(dlBufHandle) + dlMeta.offset);

    route = ua_get_u32(p_pkt_hdr, 4);

    port = route & 0xff;
    route >>= 8;
    route |= 0xff000000;
    ua_set_32(p_pkt_hdr, 4, route);

    return port;
}

/* Decrement and return TTL in IC header */
int ttlDecr(){
    __declspec(sdram) unsigned char *p_pkt_hdr;
    int ttl;

    p_pkt_hdr = (__declspec(sdram) unsigned char *)
        (Dl_BufGetData(dlBufHandle) + dlMeta.offset);

    ttl = ua_get_u8(p_pkt_hdr, 1);
    ttl--;
    ua_set_8(p_pkt_hdr, 1, ttl);

    return ttl;
}

/* Get the IC packet type from IC header */
int getType(){
    __declspec(sdram) unsigned char *p_pkt_hdr;

    p_pkt_hdr = (__declspec(sdram) unsigned char *)
        (Dl_BufGetData(dlBufHandle) + dlMeta.offset);

    return ua_get_u8(p_pkt_hdr, 0);
}

/* Strips the IC header (for outbound packets)*/
void stripICHeader(){
    __declspec(sram) dl_meta_t *pMeta;

    pMeta = (__declspec(sram) dl_meta_t *)Dl_BufGetDesc(dlBufHandle);
    pMeta->bufferSize-=8;
    pMeta->offset +=8;
}

/* Remove ethernet checksum automatically generated by hardware */
void removeEthChecksum(){
    __declspec(sram) dl_meta_t *pMeta;

    pMeta = (__declspec(sram) dl_meta_t *)Dl_BufGetDesc(dlBufHandle);
    pMeta->bufferSize-=4;
}

/* Packet filter . Configured to allow TCP packets destined for port 9070 */
int isRTSP(unsigned int bufHandle, unsigned int dlMeta1){
    __declspec(gp_reg) unsigned short ethtype, dstport;
    __declspec(gp_reg) unsigned char ipprot;
    __declspec(sdram) unsigned char *p_pkt_hdr;
    __declspec(dram_read_reg) unsigned int pkthdr_in[10];
    __declspec(local_mem) unsigned int temp[10];
    SIGNAL_PAIR sig_dram_rw;

    /* set the meta data accordingly */
    dlMeta.value[1]                              = dlMeta1;
    dlBufHandle.value                            = bufHandle;

    p_pkt_hdr = (__declspec(sdram) unsigned char *)
        (Dl_BufGetData(dlBufHandle) + dlMeta.offset);

    dram_read(pkthdr_in,(volatile void __declspec(sdram) *)p_pkt_hdr,
                    PKTHDR_CACHE_SIZE, sig_done,&sig_dram_rw);
    wait_for_all(&sig_dram_rw);

    /* Copy the header to local memory */
    temp[0] = pkthdr_in[0];
    temp[1] = pkthdr_in[1];
    temp[2] = pkthdr_in[2];
    temp[3] = pkthdr_in[3];
    temp[4] = pkthdr_in[4];
    temp[5] = pkthdr_in[5];
    temp[6] = pkthdr_in[6];
    temp[7] = pkthdr_in[7];
    temp[8] = pkthdr_in[8];
    temp[9] = pkthdr_in[9];

    /*Read eth header protocol type */
    ethtype = ua_get_u16(temp, 12);
    if(ethtype != ETHPROT_IP)
        return 0;

    /* Read IP protocol field */
    ipprot = ua_get_u8(temp, 23);
    if(ipprot != IPPROT_TCP)
        return 0;

    /* Check dst port */
    dstport = ua_get_u16(temp, 36);
    if(dstport != SERVER_PORT)
        return 0;

    return 1;
}

/* Main procedure containing the processing loop.
```

```
      Upon receiving a new packet on the scratch ring,
      the packet is processed and dropped or forwarded
*/
main() {

    __declspec(sram_write_reg) unsigned int txReq1, txReq2, txReq3;
    __declspec(sram_read_reg) unsigned int rx_msg1[5], rx_msg2, rx_msg3;
    __declspec(gp_reg) unsigned int ipsrc, ipdest, ethsrc1, ethdest1;
    __declspec(gp_reg) unsigned short ethsrc2, ethdest2;

    int sig;
    SIGNAL sig_scr_created;
    SIGNAL sig_new_packet;
    SIGNAL sig_sram_read;
    SIGNAL sig_sram_write;
    SIGNAL sig_scr_get;
    SIGNAL sig_scr_put;
    SIGNAL sig_get_meta;
    SIGNAL sig_flush_meta;
    SIGNAL sig_never;
    SIGNAL sig_buf_alloc;

    __assign_relative_register(&sig_scr_created, 13);
    __assign_relative_register(&sig_new_packet, 14);

    /* Wait for signal from XScale before proceeding
         This is to prevent the system from starting to
         forward packets until the XScale functionality is ready */
    cap_fast_write( 0, csr_thread_interrupt_a);
    wait_for_all(&sig_scr_created);

    /* Context 0: Read from RX(sr4), Filter packets.
         Forward according to Packet type */
    if (ctx() == 0) {
        while(1){

            /* Read from scratch ring */
            scratch_get_ring(rx_msg1,
                                         (void*)(POS_RX_RING_OUT << 2),
                                         5,
                                         sig_done,
                                  &sig_scr_get);
        wait_for_all(&sig_scr_get);

        /* If data is received, commence processing */
        if (rx_msg1[0]){
          /* Set metadata offset and buf handle values
              (used by helper functions */
          dlMeta.value[1]               = rx_msg1[2];
          dlBufHandle.value             = rx_msg1[0];
          txReq1 = rx_msg1[0];

          /* Remove eth checksum (added by hw) by
              decrementing bufferSize in metadata by 4 */
          removeEthChecksum();

          /* Read the input port number */
          if(getInputPort(rx_msg1[0]) == 0){
            /* External packet, filter */
            if(isRTSP(rx_msg1[0], rx_msg1[2])){
              /* Forward to XScale */
              scratch_put_ring(&txReq1,
                                  (void*)(PE_XSCALE_COMM_RING << 2),
                                  1,
                                  sig_done,
                                  &sig_scr_put);
              wait_for_all(&sig_scr_put);
              /* Invoke int_a to wakeup XScale packet processing */
              cap_fast_write( 0, csr_thread_interrupt_a);
            }
          }else{
            /* Internal packet. Route according to IC header */
            U8 port, ttl, type;

            type = getType();
            if(type == ICH_FIND_FILE){
              /* File locating packet – Always forward to XScale */
              scratch_put_ring(&txReq1,
                                  (void*)(PE_XSCALE_COMM_RING << 2),
                                  1,
                                  sig_done,
                                  &sig_scr_put);
              wait_for_all(&sig_scr_put);
              cap_fast_write( 0, csr_thread_interrupt_a);
            }else{
              /* Other packet. Route according to route field of IC header */
              /* Decrement ttl */
              ttl = ttlDecr();
              if(ttl > 0){
                  /* Pop route */
                  port = popPort();
                  /* Send to popped port */
                  if(port == 0){
                      /* If outbound, strip IC header */
                      stripICHeader();

                      /* Send to port 0 (external)*/
                      scratch_put_ring((void*)&txReq1,
                                       (void*)(PACKET_TX_SCR_RING_0 << 2),
                                       1,
                                       sig_done,
                                       &sig_scr_put);
                      wait_for_all(&sig_scr_put);
                  }else if(port == 1){
                      scratch_put_ring((void*)&txReq1,
                                       (void*)(PACKET_TX_SCR_RING_1 << 2),
                                       1,
                                       sig_done,
                                       &sig_scr_put);
                      wait_for_all(&sig_scr_put);
                  }else if(port == 2){
                      scratch_put_ring((void*)&txReq1,
                                       (void*)(PACKET_TX_SCR_RING_2 << 2),
                                       1,
                                       sig_done,
                                       &sig_scr_put);
                      wait_for_all(&sig_scr_put);
                  }else if(port == 0xff){
                      /* Forward to XScale */
                      scratch_put_ring(&txReq1,
                                       (void*)(PE_XSCALE_COMM_RING << 2),
                                       1,
                                       sig_done,
                                            &sig_scr_put);
                        wait_for_all(&sig_scr_put);
                        cap_fast_write( 0, csr_thread_interrupt_a);
                    }
                } /* If ttl == 0 -- packet is dropped */
            }
        } /* Packet is dropped */
    }/*if packet*/
  } /*while(1)*/
} /* else */
/* The second context.
   Receives packets from the XScale, and forwards according to metadata output port
   the route has already been popped on the XScale. */
if (ctx() == 2) {
  unsigned int numPackets;
  int i;

  while(1){

    scratch_get_ring(&rx_msg2,
                     (void*)(XSCALE_PE_COMM_RING << 2),
                     1,
                     sig_done,
                     &sig_scr_get);
    wait_for_all(&sig_scr_get);

    numPackets = (rx_msg2 >> 24);

    for(i = 0; i < numPackets; i++){
      txReq2 = rx_msg2 + ( i * 8 );
      if(getOutputPort(rx_msg2) == 0){
        scratch_put_ring((void*)&txReq2,
                         (void*)(PACKET_TX_SCR_RING_0 << 2),
                         1,
                         sig_done,
                         &sig_scr_put);
      }else if(getOutputPort(rx_msg2) == 1){
        scratch_put_ring((void*)&txReq2,
                         (void*)(PACKET_TX_SCR_RING_1 << 2),
                         1,
                         sig_done,
                                             &sig_scr_put);
               }else if(getOutputPort(rx_msg2) == 2){
                  scratch_put_ring((void*)&txReq2,
                                             (void*)(PACKET_TX_SCR_RING_2 << 2),
                                             1,
                                             sig_done,
                                             &sig_scr_put);
               }
               wait_for_all(&sig_scr_put);
            }
         } /*while(1)*/
      }else{
         /* All other contexts are sleeping */
         wait_for_all(&sig_never);
      }
} /* main*/
```

A.2 XScale source

A.2.1 dcube.h

```
#ifndef DCUBE_H
#define DCUBE_H

#include "uclo.h"
#include "hal_mev2.h"
#include "halMev2Api.h"
#include "hal_scratch.h"
#include "hal_sram.h"
#include "hal_dram.h"
#include "ix_macros.h"
#include "ix_ossl.h"
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/time.h>
```

```
#include <sys/ioctl.h>
#include <enpv2_types.h>
#include <sys/mman.h>

/* Socket includes */
#include <sys/socket.h>
#include <sys/types.h>
#include <arpa/inet.h>

#ifdef DEBUG
#define    DBG_MSG(str,args...)                    printf(str,##args)
#else
#define DBG_MSG(str, args...)
#endif

#define VIRT_DRAM_BASE (void *)(ix_uint32)Hal_dram_ch0_virtAddr
#define VIRT_SRAM_BASE (void *)(ix_uint32)Hal_sram_ch0_virtAddr
#define MAX_CONCURRENT_STREAMS 100
#define MULT_FACTOR 6 /* Used to get pointer to data from buf handle */
#define MAX_IC_SESSIONS MAX_CONCURRENT_STREAMS
#define ICH_TTL 4

/* XScale to host socket defines */
#define DEST_PORT 23456
#define DEST_IP "192.168.1.1"

enum {
  RTSP_DESC  =  1,
  RTSP_SETUP
};

typedef union {
  struct {
    unsigned int bufferNext;               /**< Next buffer in the chain */

    unsigned short   bufferSize;               /**< amount of data currently in buffer */
    unsigned short   offset;                           /**< offset in DRAM where data begins */

    unsigned int      packetSize       : 16;    /**< amount of data in the chain of buffers */

    unsigned int      freeListId       : 4;     /**< Free List to which this buffer belongs to */
```

```
        unsigned int    rxStat        : 4;    /**< Receive status */
        unsigned int    headerType    : 8;    /**< HEader Type: IPv4, IPv6 etc */

        unsigned short  inputPort;                  /**< Input port on which packet was received */
        unsigned short  outputPort;                 /**< Output port on which packet to be transmitted */

        unsigned int    nextHopId     : 16;   /**< Nexthop ID */
        unsigned int    fabricPort    : 8;    /**< Blade:Port */
        unsigned int    reserved      : 4;       /**< reserved */
        unsigned int    nhidType      : 4;    /**< nexthop ID type */

        unsigned int    colorId       :4;
        unsigned int    reserved1     :4;
        unsigned int    flowId        :24;    /**< FLow ID */

        unsigned short  classId;                    /**< Class ID */
        unsigned short  reserved2;

        unsigned int    packetNext;                 /**< Next packet in the chain */
        /**< (used only in Hierarchical Queuing) */
    } __attribute__((packed));      // end of struct

    unsigned int value[8];                              /**< aggregate for the above fields */

} dl_meta_t;

enum {
    ICH_FIND_FILE  =  1,
    ICH_ROUTE_FB,
    ICH_CTRL_MSG,
    ICH_CTRL_FB,
    ICH_RTP
};

/* Standard IC header:
     Common for all IC packets. */
typedef struct{
    uint8_t         type; /* Type of IC packet */
    uint8_t         ttl;  /* To avoid circulating packets */
    uint16_t     dataLen; /* Length of IC packet data (excluding the header) */
    uint32_t     route; /* Routing information. Src routing */
```

```
} ICHeader_t __attribute__((packed));

/* IC header extension:
     Used to setup route information when assigning a new stream */
typedef struct{
    uint16_t          sport; /* Client source port */
    uint16_t          dport; /* Server port */

    uint32_t          saddr; /* Client source IP addr. */
    uint32_t          daddr; /* Egress IP addr */

    char            eth_src[6]; /* Ethernet src address */
    char            eth_dst[6]; /* Ethernet src address */
} ICHeader_ext_t __attribute__((packed));

enum {
    ICS_CLOSED = 0,
    ICS_WAIT_ROUTE,
    ICS_ACTIVE,
};

typedef struct{
    uint8_t         status;
    uint32_t        route; /* Routing information. Src routing */
    uint16_t          sport; /* Client source port */
    uint16_t          dport; /* Server port */

    uint32_t          saddr; /* Client source IP addr. */
    uint32_t          daddr; /* Egress IP addr */

    char            eth_src[6]; /* Ethernet src address */
    char            eth_dst[6]; /* Ethernet src address */
} IC_session_t;
/* Structs located at the server that has the relevant file */
IC_session_t serv_ics[MAX_IC_SESSIONS];
/* Structs located at the egress (used for forwarding incoming RTSP and RTCP data) */
IC_session_t egr_ics[MAX_IC_SESSIONS];

extern void StartSpi3br();
extern void StartMacs();

```

```
/* Local includes
   Needs structs above */
#include "bogus_tcp.h"
#include "dcube_utils.h"

#endif /* #ifndef DCUBE_H */
```

A.2.2 dcube.c

```
#include "dcube.h"

Hal_IntrMasks_T intMask;
ix_uint32 cbdata, intThread_A_handle, intThread_B_handle;
const char *pImageName="ICrouter.uof";
int hostSock, errno;


/*/Called when the ICrouter microblock receives a
  packet on port 0 or a packet destined for
  this node is received*/
void intThread_A(){
  unsigned int metaOffset, dataOffset, bufHandle, rtspType;
  dl_meta_t* pMeta;
  void* dataVAddr;

  /* Get packet and vaddr for metadata and data */
  bufHandle = SCRATCH_RING_GET(10);

  DBG_MSG("Got from sr10: %0#10x\n", (int)bufHandle);

  metaOffset = (bufHandle & 0xffffff) << 2;
  dataOffset = (bufHandle & 0xffffff) << (MULT_FACTOR + 2);

  DBG_MSG("metaOffset: %0#10x\n", (int)metaOffset);

  pMeta = VIRT_SRAM_BASE + metaOffset;

  dataVAddr = VIRT_DRAM_BASE + dataOffset + pMeta->offset;

   /* IC packet or from port 0 0? */
```

```
if (pMeta->inputPort == 0){ /* External packet */
   char data[1500];
   int len = 0, idx = 0, status;

   /* Since all other packets are filtered on the uEngines,
    *This has to be a TCP-packet destined for port 9070 */

   /* Port 0:
        Do we have the file?
        Yes: Create session and forward to host.
        No: Create IC header and forward to port 1 and 2 */
   status = tcp_recv(dataVAddr, data, &len, &idx);
   DBG_MSG("status: %i\n", status);
   if((status == B_TCP_DATA)){
      DBG_MSG("tcp_recv returned the following data:\n");

      /* Ensure support for creating more than one stream on each TCP connection */
      rtspType = getRtspType(data);
      if(rtspType == RTSP_DESC){
         DBG_MSG("dcube.c: packet is DESC\n");
         /* Implement later ..*/
      }else if( rtspType == RTSP_SETUP ){
         DBG_MSG("dcube.c: packet is SETUP\n");
         if(localHasFile(data)){
            DBG_MSG("dcube.c: host has file.\n");
            /* Construct IC header and forward packet to host */
            add_IC_hdr(data, len, ICH_FIND_FILE, 0xffffffff);
            add_IC_ext_hdr(data, idx);
            ICPushPort(data, pMeta->inputPort);
            debug_print_ICheader(data);
            idx = newServICSession(data);
            debug_print_servICsession(idx);
            idx = newEgrICSession(data);
            egr_ics[idx].route = 0xffffffff; /* Since this is no feedback message */
            debug_print_egrICsession(idx);
            /* Send to host */
            errno = send(hostSock, data, len+sizeof(ICHeader_t)+sizeof(ICHeader_ext_t), 0);
            if(errno == -1){
               DBG_MSG("send() returned error\n");
            }else{
               DBG_MSG("Successfully sent %i bytes to socket\n",errno);
            }
        }else{
            DBG_MSG("dcube.c: file not found.\n");
            /* Set up IC header and forward packet
                to port 1 and 2 */
            /* Set route to be 0xffffffff. Ports will be pushed
                as the packet is broadcasted through the system */
            add_IC_hdr(data, len, ICH_FIND_FILE, 0xffffffff);
            add_IC_ext_hdr(data, idx);
            ICPushPort(data, pMeta->inputPort);
            ixp_send_packet(data, len+sizeof(ICHeader_t)+sizeof(ICHeader_ext_t), 1);
            ixp_send_packet(data, len+sizeof(ICHeader_t)+sizeof(ICHeader_ext_t), 2);
        }
    }else{
        int icIdx, port;
        /* Data for an existing stream
            if the stream cannot be found,
            discard the packet */
        DBG_MSG("Received RTSP package\n");
        debug_print_egrICsession(0);
        icIdx = egrFindICsess(idx);
        if(icIdx ==-1){
            DBG_MSG("IC session not found\n");
            /* Discard packet */
        }else{
            add_IC_hdr(data, len, ICH_CTRL_MSG, egr_ics[icIdx].route);
            debug_print_buffer32(data, len+sizeof(ICHeader_t));

            port = ICPopPort(data);
            if(port == 0xff){
                /* Send to host */
                errno = send(hostSock, data, len+sizeof(ICHeader_t)+sizeof(ICHeader_ext_t), 0);
                if(errno == -1){
                    DBG_MSG("send() returned error\n");
                }else{
                    DBG_MSG("Successfully sent %i bytes to socket\n",errno);
                }
            }else{
                ixp_send_packet(data, len+sizeof(ICHeader_t), port);
            }
        }
        }
    }else if((status == B_TCP_ERROR)){
        DBG_MSG("tcp_recv returned B_TCP_ERROR \n");
        return;
    }else if((status == B_TCP_NO_DATA)){
        DBG_MSG("tcp_recv returned B_TCP_NO_DATA \n");
        return;
    }

}else{ /* IC packet */
    /* IC-packet*/

    DBG_MSG("Got IC Packet, %i bytes\n", ((ICHeader_t*)dataVAddr)->dataLen);

    switch(*((uint8_t*)dataVAddr)){ /* Read IC header Type */

    case ICH_FIND_FILE: {
        DBG_MSG("ICH_FIND_FILE\n");

        if(localHasFile(dataVAddr+sizeof(ICHeader_t)+sizeof(ICHeader_ext_t))){
            DBG_MSG("dcube.c - IC: host has file.\n");
            /* If this is the first time I got this packet:
                  create stream struct - forward to host
                  If I have seen this packet before:
                  drop packet */
            /* Find index of IC session for stream.
                  If no ssession, create a new one */

            if(findICSession(dataVAddr) >= 0){
                DBG_MSG("ICSession found\n");
                /* We have already received this packet through another route:
                      Drop packet */
            }else{
                int idx, i, fbpSize =sizeof(ICHeader_t) + sizeof(ICHeader_ext_t) + 4 ;
                char *feedback, *rp;
                ICHeader_t *ich;

                feedback = (char*)malloc(fbpSize);
                /* Zero the packet */
                memset(feedback, 0, fbpSize);

                DBG_MSG("Creating new IC session\n");
                debug_print_ICheader(dataVAddr);
                /* Push input port */
                ICPushPort(dataVAddr, pMeta->inputPort);
                idx = newServICSession(dataVAddr);
                debug_print_servICsession(idx);
                /* Send feedback packet to egress to
                      create session struct there */
                memcpy(feedback, dataVAddr, sizeof(ICHeader_t)+sizeof(ICHeader_ext_t));
                ich = (ICHeader_t*)feedback;

                /* Make sure the packet is forwarded to XScale at egress */
                /* Make sub of this ? */
                rp = (uint8_t*)(&ich->route);
                for(i=0; i<4; i++){
                      if(rp[i]!=0xff){
                          rp[i]=0xff;
                          break;
                      }
                }
                /* Copy the route to payload of packet */
                memcpy(feedback+sizeof(ICHeader_t)+sizeof(ICHeader_ext_t), &(ich->route), 4);
                /* Prepare other fields of IC header */
                ich->type=ICH_ROUTE_FB;
                ich->dataLen=4;
                ich->ttl=4;
                DBG_MSG("feedback: ich->route: %x\n", ich->route);

                /* Send feedback packet */
                ixp_send_packet(feedback, fbpSize, ICPopPort(feedback));
                /* Send packet to server */
                /* The program will crash (!!) if the connect was not successful:
                      add handling of this problem */
                errno = send(hostSock, dataVAddr, pMeta->bufferSize, 0);
                if(errno == -1){
                      DBG_MSG("send() returned error\n");
                }else{
                      DBG_MSG("Successfully sent %i bytes to socket\n",errno);
                }
            }
        }else{
            DBG_MSG("dcube.c - IC: file not found.\n");
            /* decrement ttl, forqward to port 1 and 2 */
            if (IC_ttl_dec(dataVAddr)){
               /* Push port on IC header */
               ICPushPort(dataVAddr, pMeta->inputPort);
               /* if ttl=0, drop packet */
               /* Send to the port not received from */
               if(pMeta->inputPort==2)
                  ixp_send_packet(dataVAddr, pMeta->bufferSize, 1);
               else
                  ixp_send_packet(dataVAddr, pMeta->bufferSize, 2);
            }
            DBG_MSG("ttl is now: %i\n", *((uint8_t*)dataVAddr+1));
         }

         break;
      }

      case ICH_CTRL_FB: {
         uint32_t saddr, daddr;
         uint16_t sport, dport;
         int status, size;
         ICHeader_t *ich = (ICHeader_t*)dataVAddr;
         ICHeader_ext_t *iche = (ICHeader_ext_t*)(dataVAddr + sizeof(ICHeader_t));
         DBG_MSG("ICH_CTRL_FB\n");
         debug_print_ICheader(dataVAddr);
         /* Get port data from IC header */
         saddr = iche->saddr;
         daddr = iche->daddr;
         sport = iche->sport;
         dport = iche->dport;
         size = ich->dataLen;
         DBG_MSG("ICH_CTRL_FB: size = %i\n", size);
         /* Strip IC header */
         dataVAddr += sizeof(ICHeader_t) + sizeof(ICHeader_ext_t);
         /* Forward payload to tcp_send */
         status = tcp_send(dataVAddr, size, saddr, daddr, sport, dport);
         if(status == -1){
            DBG_MSG("tcp_send() returned an error.\n");
         }else{
            DBG_MSG("Successfully passed %i bytes to tcp_send.\n",status);
```

```
        }

        break;
    }

    case ICH_CTRL_MSG: {
        DBG_MSG("ICH_CONTROL_MSG\n");
        /* Forward to server */
        errno = send(hostSock, dataVAddr, pMeta->bufferSize, 0);
        if(errno == -1){
            DBG_MSG("send() returned error\n");
        }else{
            DBG_MSG("Successfully sent %i bytes to socket\n",errno);
        }

        break;
    }

    case ICH_ROUTE_FB: {
        int idx;
        DBG_MSG("ICH_ROUTE_FEEDBACK\n");
        debug_print_ICheader(dataVAddr);
        debug_print_buffer32(dataVAddr, pMeta->bufferSize);
        /* Set up egress icsession with reverse route */
        idx = newEgrICSession(dataVAddr);
        debug_print_egrICsession(idx);

        break;
    }

    default: break;
    }

  }

  return;
}

int main(){
  void *ucloHandle;
  int status;
```

```
struct sockaddr_in dest_addr;     // will hold the destination addr

/* init ixa sdk uclo lib */
UcLo_InitLib();

/* init all uengs */
UcLo_InitLibUeng( 0xff );

/* load ueng image into memory */
status = UcLo_LoadObjFile( &ucloHandle, (char*) pImageName );
if( status != UCLO_SUCCESS ) {
   printf("UcLo_LoadObjFile1 failed with status code :%d\n", status);
   return 1;
}

/* write to microengines */
status = UcLo_WriteUimageAll( ucloHandle );
if( status != UCLO_SUCCESS ) {
   printf("UcLo_WriteUimageAll failed\n");
   printf("status = %i\n", status);
   return 1;
}

/* verify uengine 0 for sanity check */
status = UcLo_VerifyUengine( ucloHandle, 0 );
if( status != UCLO_SUCCESS ) {
   printf("UcLo_VerifyUengine 0 failed, status = 0x%x\n", status);
   return 1;
}

printf("verify uengine passed\n");

/* delete object */
status = UcLo_DeleObj( ucloHandle );
if( status != UCLO_SUCCESS ) {
   printf("UcLo_DeleObj failed\n");
   return 1;
}

/* Enable interrupt A */
halMe_IntrEnable(HALME_INTR_THD_A_MASK);
```

```

/* init and start spi3br */
StartSpi3br();

/* init and start macs */
StartMacs();

halMe_Init(0xff);
/*Start microengines */
halMe_Start(0, 0xff);
halMe_Start(1, 0xff);
halMe_Start(2, 0xff);
printf("Microengines started\n");

/* Wait for int A Indicates that uEngines are initialized and ready */
halMe_IntrPoll(HALME_INTR_THD_A_MASK, &intMask);

DBG_MSG("Got int A\n");

/* Spawn callback thread for me-XScale comm */
status = halMe_SpawnIntrCallbackThd( HALME_INTR_THD_A_MASK, intThread_A,
                                        &cbdata, 1, (void*)&intThread_A_handle);
if (status != HALME_SUCCESS){
   printf("Error spawning intThread_A\n");
} else {
   printf("Success spawning intThread_A\n");
}

/* Send signal 13 to me1, ctx 0, 2 and 4 */
status = me_signal( 1, 0, 13);
if (status != HALME_SUCCESS){
   printf("Error signaling me 1, ctx 0\n");
}
status = me_signal( 1, 2, 13);
if (status != HALME_SUCCESS){
   printf("Error signaling me 1, ctx 2\n");
}
status = me_signal( 1, 4, 13);
if (status != HALME_SUCCESS){
   printf("Error signaling me 1, ctx 4\n");
}
```

```

  /* Setup socket to communicate with host */
 hostSock = socket(AF_INET, SOCK_STREAM, 0); // do some error checking!
 if(hostSock == -1){
    printf("socket() returned error\n");
 }else{
    printf("Socket successfully created\n");
 }

 dest_addr.sin_family = AF_INET;
 dest_addr.sin_port = htons(DEST_PORT);
 dest_addr.sin_addr.s_addr = inet_addr(DEST_IP);
 memset(&(dest_addr.sin_zero), '\0', 8);

 errno = connect(hostSock, (struct sockaddr *)&dest_addr, sizeof(struct sockaddr));
 if(errno == -1){
    printf("connect() returned error\n");
 }else{
    printf("Successfully connected to %s, port %i\n",inet_ntoa(dest_addr.sin_addr), DEST_PORT);
 }
 /*Show Spi3br driver information*/
 //ShowSpi3br();

 /*Show mac driver information*/
 //ShowMac();

 while(1) {
    char data[1500], *dp;
    int status, bytecount, dataSize;
    /* Listen to hostSock
         collect data, create Correct headers,
         and forward to uEngines for transmit */

    /* Assumes that IC Header is the first to arrive,
           and that all packets arrive in the order: ICHeader−>Payload−>ICHeader−>Payload etc. */

    /* Read IC header */
    dp = data;
    bytecount = sizeof(ICHeader_t);
    while(bytecount > 0){
       status = recv(hostSock, dp, bytecount, 0);
```

```
        if(status == -1){
            DBG_MSG("recv() returned error\n");
            sleep(1000);
        }else if(status > 0){
            DBG_MSG("Socket: Received IC header, %i bytes\n", status);
            dp += status;
            bytecount -= status;
        }
    }

    /* Get size of data from IC header */
    ICHeader_t *ich = (ICHeader_t*)data;
    int port;
    if((ich->type == ICH_ROUTE_FB) || (ich->type == ICH_CTRL_FB))
        bytecount = ich->dataLen + sizeof(ICHeader_ext_t);
    else
        bytecount = ich->dataLen;

    dataSize = bytecount;
    DBG_MSG("dataSize: %i\n", dataSize);
    DBG_MSG("bytecount: %i\n", bytecount);
    while(bytecount > 0){
        status = recv(hostSock, dp, bytecount, 0);
        if(status == -1){
            DBG_MSG("recv() returned error\n");
            sleep(1000);
        }else if(status > 0){
            DBG_MSG("Socket: Received %i bytes of data\n", status);
            dp += status;
            bytecount -= status;
        }
    }
    dataSize+=sizeof(ICHeader_t);

    debug_print_buffer32(data, dataSize);
    port = ICPopPort(data);
    if(port == 0xff){ /* This is an RTSP message on egress machine */
        ICHeader_ext_t *iche = (ICHeader_ext_t*)(data+sizeof(ICHeader_t));

        status = tcp_send(data+sizeof(ICHeader_t)+sizeof(ICHeader_ext_t), ich->dataLen,
                          iche->saddr, iche->daddr, iche->sport, iche->dport);
            if(status == -1){
               DBG_MSG("tcp_send() returned an error.\n");
            }else{
               DBG_MSG("Successfully passed %i bytes to tcp_send.\n",status);
            }
         }else{
            dp = data;
            if(port == 0){ /* If data is at egress, strip IC header */
               dp += sizeof(ICHeader_t);
               dataSize -= sizeof(ICHeader_t);
            }
            ixp_send_packet(dp, dataSize, port);
         }
      }

   return 0;
}











```

A.2.3 dcube_utils.h

```
#ifndef _DCUBE_UTILS_H
#define _DCUBE_UTILS_H

#include "dcube.h"

void tcp_udp_checksum(void*, int);
void ip_checksum(void*);
ix_error me_signal( ix_uint32, ix_uint32, ix_uint32);
void ixp_send_packet(uint8_t*, int, uint16_t);
void debug_print_metadata(dl_meta_t*);
void debug_print_buffer32(void*, int);
int getRtspType(char *data);
int localHasFile (char *descString);
void add_IC_hdr(char *data, int len, int type, uint32_t route);
void add_IC_ext_hdr(char *data, int idx);
int IC_ttl_dec(void* dataVAddr);
int findICSession(void* ICpacket);
int newServICSession(void* ICpacket);
int newEgrICSession(void* ICpacket);
void debug_print_egrICsession(int idx);
void debug_print_servICsession(int idx);
void debug_print_ICheader(void* ICpacket);
void ICPushPort(void* ICpacket, uint8_t port);
int ICPopPort(void* ICpacket);
int egrFindICsess(int idx);
int timeval_subtract (struct timeval *result, struct timeval *x, struct timeval *y);

#endif /* #ifndef _DCUBE_UTILS_H */
```

A.2.4 dcube_utils.c

```
#include "dcube_utils.h"

#define IX_RM_SAME_ME_SIGNAL_OFFSET            0x108
#define VIRT_DRAM_BASE (void *)(ix_uint32)Hal_dram_ch0_virtAddr
#define VIRT_SRAM_BASE (void *)(ix_uint32)Hal_sram_ch0_virtAddr

int bufCount = 0; /* what buffer handle number is currently "active */
uint8_t packetcount = 0; /* How many packets currently scheduled for batck send */
int bufBase; /* The base buffer handle for the next batch */
struct timeval curTime, lastBatch;

ix_error me_signal( ix_uint32 arg_MENumber,
                    ix_uint32 arg_ContextNumber,
                    ix_uint32 arg_SignalNumber ) {
  ix_error err = IX_SUCCESS;

  WRITE_LWORD(((ix_uint32)Hal_cap_me_local_csr_virtAddr
             + (arg_MENumber << 10) + IX_RM_SAME_ME_SIGNAL_OFFSET),
                        ((arg_SignalNumber << 3) + arg_ContextNumber));

    return err;
}

/**************************************************
 * To calculate tcp checksum :
 *  - Construct pseudo header:
 *       +--------+--------+--------+--------+
 *       |           Source Address          |
 *       +--------+--------+--------+--------+
 *       |        Destination Address        |
 *       +--------+--------+--------+--------+
 *       |  zero  |  PTCL  |    TCP Length   |
 *       +--------+--------+--------+--------+
 *  - Zero out checksum field
 *  - Pad one zeroed byte if length is an odd number
 *  - Do a ones complement sum of the whole thing
 *        -If a carry occur, add one, and return
 *  - Return the ones complement of the sum
 **************************************************/

/* Calculates the 16 bit ones-complement sum of a given
     buffer. Pads the last byte with 0 if odd size
     Input: data: Pointer to buffer start
                  len: length of data */
uint32_t partial_csum(uint8_t* data, int len) {
    uint32_t sum = 0;
    uint16_t last = 0;
    int odd = 0;

    if(len & 1)
        odd = 1;

    len >>= 1;
    while (len > 0) {
        sum += *((uint16_t*)data);
        data += sizeof(uint16_t);
        len--;
    }
    /* odd len */
    if (odd){
        last = *data << 8;
        sum += last;
    }

    return sum ;
}

/* Calculate checksum for TCP or UDP header, and
   write it to the packet
   Input: ipHdrStart - Pointer to beginning of IP header
              size - total size of packet */
void tcp_udp_checksum(void* ipHdrStart, int size){
    uint32_t sum = 0;
    uint16_t finalsum;
    uint8_t protocol;
    uint32_t zero_ptcl_tcpsize;

    /* Extract protocol type from IP header */
    memcpy(&protocol, ipHdrStart+9, 1);
    zero_ptcl_tcpsize= (protocol << 16) | ((size-20) & 0xffff);

    /* zero out checksum field */
    memset(ipHdrStart+36, 0, 2);
    /* Pseudo header calculations */
    /* Calculate checksum of src & dest */
    sum = partial_csum(ipHdrStart+12, 8);
    /* Calculate checksum of protocol nr and TCP size*/
    sum += partial_csum((uint8_t*)&zero_ptcl_tcpsize, 4);

    /* Calculate checksum of tcp header and data*/
    sum += partial_csum(ipHdrStart+20, size-20);

    /* Add carries */
    while (sum & 0xffff0000)
        sum = (sum >> 16) + (sum & 0xffff);

    /* Write the one's complement of the sum to
       the correct spot in the TCP header */
    finalsum = ~sum;
    memcpy(ipHdrStart+36, &finalsum, 2);
```

```
}

/* Calculate checksum for IP header, and
   write it to the packet
   Input: Pointer to beginning of IP header */
void ip_checksum(void* ipHdrStart){
  uint32_t sum;
  uint16_t finalsum;

  memset(ipHdrStart+10, 0, 2);
  sum = partial_csum(ipHdrStart, 20);

  /* Add carries */
  while (sum & 0xffff0000)
    sum = (sum >> 16) + (sum & 0xffff);

  finalsum = ~sum;
  memcpy(ipHdrStart+10, &finalsum, 2);
}

/* Send packet buffer to port specified
   todo: implement batch sending and sending to
   different ports */
void ixp_send_packet(uint8_t* buffer, int size, uint16_t port){
  int curBufHandle, bufHandle, metaOffset, dataOffset, status;
  struct timeval timediff;
  void* dataVAddr;
  dl_meta_t* pMeta;
  DBG_MSG("Send packet of %i bytes to port %i\n", size, port);

  packetcount++;
  curBufHandle = 0x20000 + (bufCount * 8);
  /* If we're at the beginning of a new batch */
  if( packetcount == 1){
    bufBase = curBufHandle;
  }

  /* Prepare the current packet */
  bufHandle = curBufHandle;
  metaOffset = (bufHandle & 0xffffff) << 2;
  dataOffset = (bufHandle & 0xffffff) << (MULT_FACTOR + 2);
```

```
pMeta = VIRT_SRAM_BASE + metaOffset;
/* Zero metadata */
memset(pMeta, 0, sizeof(dl_meta_t));

pMeta->bufferNext =0xff;
pMeta->value[2] = 0x00001001;
pMeta->bufferSize = size;
pMeta->packetSize = size;
pMeta->nextHopId = 0xff;
pMeta->offset = 0x100;
pMeta->outputPort = port;

dataVAddr = VIRT_DRAM_BASE + dataOffset + pMeta->offset;

memcpy(dataVAddr, buffer, size);

/* Increment or wrap buffer count */
if(bufCount == 1024){
   bufCount = 0;
} else
   bufCount++;

status = gettimeofday(&curTime, NULL);
if(status == -1){
   DBG_MSG("gettimeofday returned an error\n");
}

timeval_subtract (&timediff, &curTime, &lastBatch);
DBG_MSG("timediff.tv_sec: %u - timediff.tv_usec: %u\n", timediff.tv_sec, timediff.tv_usec);

//if( timediff.tv_sec > 0 || timediff.tv_usec > 500 || packetcount >= 50){
   /*
      |    31 - 24    | 23    -    0 |
          num packets      bufHandle
   */
bufHandle = (packetcount << 24) | curBufHandle;
DBG_MSG("bufHandle : %0#10x\n", bufHandle);
SCRATCH_RING_PUT(11, bufHandle);
/* Reset batch packet counter */
packetcount = 0;
status = gettimeofday(&lastBatch, NULL);
```

```
    if(status == -1){
        DBG_MSG("gettimeofday returned an error\n");
    }
    //}

    //status = me_signal( 1, 2, 14);
    //if (status != HALME_SUCCESS){
    //   DBG_MSG("Error signaling me 1, ctx 2\n");
    // }
}

/* Print relevant metadata */
void debug_print_metadata(dl_meta_t* pMeta){
    DBG_MSG("bufferNext: %0#10x\n", pMeta->bufferNext);
    DBG_MSG("bufferSize: %0#10x\n", pMeta->bufferSize);
    DBG_MSG("offset: %0#10x\n", pMeta->offset);
    DBG_MSG("packetSize: %0#10x\n", pMeta->packetSize);
    DBG_MSG("freeListId: %0#10x\n", pMeta->freeListId);
    DBG_MSG("rxStat: %0#10x\n", pMeta->rxStat);
    DBG_MSG("headerType:: %0#10x\n",pMeta->headerType);
    DBG_MSG("inputPort: %0#10x\n", pMeta->inputPort);
    DBG_MSG("outputPort: %0#10x\n", pMeta->outputPort);
    DBG_MSG("nextHopId: %0#10x\n", pMeta->nextHopId);
    DBG_MSG("fabricPort: %0#10x\n", pMeta->fabricPort);
    DBG_MSG("flowId: %0#10x\n", pMeta->flowId);
    DBG_MSG("classId: %0#10x\n", pMeta->classId);
    DBG_MSG("classId: %0#10x\n\n", pMeta->packetNext);
}

/* Print buffer in 32 bit words hex in byte order*/
void debug_print_buffer32(void* buf, int size){
    int i;

    for(i=0; i < size; i++){
        if(i % 4==0)
            DBG_MSG("\nbuf[%i]: 0x", i);

        if(*((uint8_t*)(buf+i)) == 0)
            DBG_MSG("00");
        else if(*((uint8_t*)(buf+i)) < 15)
            DBG_MSG("0%x", *((uint8_t*)(buf+i)));
        else
            DBG_MSG("%x", *((uint8_t*)(buf+i)));
    }
    DBG_MSG("\n");
}

/* Check if the payload is a SETUP packet
  Implement check for other types when needed */
int getRtspType(char *data){
    char *setupStr = "SETUP";
    if(strncmp(data, setupStr, strlen(setupStr)) == 0)
        return RTSP_SETUP;

    return 0;
}

/* Check if the requested file exists on the local server */
int localHasFile (char *descString){
    char fname[50] = "/opt/storage/\0";
    char *start;
    FILE *status;

    start = strstr(descString, "9070");
    if(start == NULL)
        return 0;
    start += 5;

    strncat(fname, start, strchr(start, ' ')-start);
    DBG_MSG("fname = %s\n", fname);

    status = fopen (fname, "rb");
    if(status == NULL)
        return 0;
    else
        fclose(status);
    return 1;
}

/* Input: data: pointer to buffer containing payload
                len: pointer to length of payload
                idx: index of tcp stream in the ts-array
                Prepends the payload with an IC header */
void add_IC_hdr(char *data, int len, int type, uint32_t route){
    ICHeader_t ich;
    /* Zero IC header */
    memset(&ich, 0, sizeof(ICHeader_t));
    ich.type = type;
    ich.ttl = ICH_TTL;
    ich.dataLen = len;
    ich.route = route;
    memmove(data+sizeof(ICHeader_t), data, len);
    memcpy(data, &ich, sizeof(ICHeader_t));
}

/* Assumes that an IC header already has been added */
void add_IC_ext_hdr(char *data, int idx){
    ICHeader_t *ich = (ICHeader_t*)data;
    ICHeader_ext_t iche;
    /* Zero IC header ext */
    memset(&iche, 0, sizeof(ICHeader_ext_t));
    iche.sport = ts[idx].sport;
    iche.dport = ts[idx].dport;
    iche.saddr = ts[idx].saddr;
    iche.daddr = ts[idx].daddr;
    memcpy(&iche.eth_src, &ts[idx].eth_src, 6);
    memcpy(&iche.eth_dst, &ts[idx].eth_dst, 6);
    memmove(data+sizeof(ICHeader_t)+sizeof(ICHeader_ext_t), data+sizeof(ICHeader_t), ich->dataLen);
    memcpy(data+sizeof(ICHeader_t), &iche, sizeof(ICHeader_ext_t));
}

/*Input: pointer to beginning of IC packet
    Decrements IC ttl by one, and returns the value */
int IC_ttl_dec(void* dataVAddr){
    return --(*((uint8_t*)dataVAddr+1));
}

/* Check if IC session exists.
    Input: pointer to start of IC packet
    Output: postion of IC session (or -1 if it does not exist) */
int findICSession(void* ICpacket){
    ICHeader_ext_t *iche = (ICHeader_ext_t*)(ICpacket+sizeof(ICHeader_t));
    int i, pos = -1;
```

```

    DBG_MSG("findICSession: sport= %i\n", iche->sport);

    for (i=0; i<MAX_IC_SESSIONS; i++) {
        if(serv_ics[i].status != ICS_CLOSED && serv_ics[i].saddr==iche->saddr &&
                serv_ics[i].daddr==iche->daddr && serv_ics[i].sport == iche->sport && serv_ics[i].dport == iche->dport) {
            /* We have found an active session that corresponds to the ic header */
            pos = i;
            DBG_MSG("ICSession: found session\n");
            break;
        }
    }
    return pos;
}

uint32_t routeInverse(uint32_t route){
    uint32_t inv_route;
    uint8_t *rp= (uint8_t*)&route, *irp=(uint8_t*)&inv_route;
    int i, j=3;

    for(i=0; i < 4; i++){
        if(*(rp+i)==0xff)
            *(irp+i) = 0xff;
        else
            *(irp+(j--)) = *(rp+i);
    }
    return inv_route;
}

/* Initialize IC session from data
    Input: Pointer to IC packet
    Output: index of new IC session (-1 if no free slot)
*/
int newEgrICSession(void* ICpacket){
    ICHeader_ext_t *iche = (ICHeader_ext_t*)(ICpacket + sizeof(ICHeader_t));
    /* The payload of IC_ROUTE_FB is rout to server with the file */
    uint32_t route_data = *(uint32_t*)(ICpacket+sizeof(ICHeader_t)+sizeof(ICHeader_ext_t));
    DBG_MSG("newEgrICSession: route_data: %x\n", route_data);
    int i, pos = -1;
    /* locate first free slot */
    for (i=0; i<MAX_IC_SESSIONS; i++) {
        if(egr_ics[i].status == ICS_CLOSED){
            pos = i;
            DBG_MSG("newEgrICSession: first free: %i\n", pos);
            break;
        }
    }

    if(pos != -1){
        egr_ics[pos].status = ICS_ACTIVE;
        egr_ics[pos].route = routeInverse(route_data);
        egr_ics[pos].sport = iche->sport;
        egr_ics[pos].dport = iche->dport;
        egr_ics[pos].saddr = iche->saddr;
        egr_ics[pos].daddr = iche->daddr;
        memcpy(&egr_ics[pos].eth_src, &iche->eth_src, 6);
        memcpy(&egr_ics[pos].eth_dst, &iche->eth_dst, 6);
    }
    return pos;
}

/* Initialize IC session from data
     Input: Pointer to IC packet
     Output: index of new IC session (-1 if no free slot)
*/
int newServICSession(void* ICpacket){
    ICHeader_t *ich = (ICHeader_t*)ICpacket;
    ICHeader_ext_t *iche = (ICHeader_ext_t*)(ICpacket+sizeof(ICHeader_t));
    int i, pos = -1;
    /* locate first free slot */
    for (i=0; i<MAX_IC_SESSIONS; i++) {
        if(serv_ics[i].status == ICS_CLOSED){
            pos = i;
            DBG_MSG("newServICSession: first free: %i\n", pos);
            break;
        }
    }

    if(pos != -1){
        serv_ics[pos].status = ICS_ACTIVE;
        serv_ics[pos].route = ich->route;
        serv_ics[pos].sport = iche->sport;
        serv_ics[pos].dport = iche->dport;
        serv_ics[pos].saddr = iche->saddr;
        serv_ics[pos].daddr = iche->daddr;
        memcpy(&serv_ics[pos].eth_src, &iche->eth_src, 6);
        memcpy(&serv_ics[pos].eth_dst, &iche->eth_dst, 6);
    }
    return pos;
}

void debug_print_servICsession(int idx){
    if(idx < MAX_IC_SESSIONS){
        DBG_MSG("IC-session:\n");
        DBG_MSG("index: %i\n", idx);
        DBG_MSG("status: %u\n", serv_ics[idx].status);
        DBG_MSG("route: %x\n", serv_ics[idx].route);
        DBG_MSG("sport: %u\n", serv_ics[idx].sport);
        DBG_MSG("dport: %u\n", serv_ics[idx].dport);
        DBG_MSG("saddr: %x\n", serv_ics[idx].saddr);
        DBG_MSG("daddr: %x\n", serv_ics[idx].daddr);
        DBG_MSG("eth_src: ");
        debug_print_buffer32(serv_ics[idx].eth_src, 6);
        DBG_MSG("eth_dst: ");
        debug_print_buffer32(serv_ics[idx].eth_dst, 6);
    }
}

void debug_print_egrICsession(int idx){
    if(idx < MAX_IC_SESSIONS){
        DBG_MSG("IC-session:\n");
        DBG_MSG("index: %i\n", idx);
        DBG_MSG("status: %u\n", egr_ics[idx].status);
        DBG_MSG("route: %x\n", egr_ics[idx].route);
        DBG_MSG("sport: %u\n", egr_ics[idx].sport);
        DBG_MSG("dport: %u\n", egr_ics[idx].dport);
        DBG_MSG("saddr: %x\n", egr_ics[idx].saddr);
        DBG_MSG("daddr: %x\n", egr_ics[idx].daddr);
        DBG_MSG("eth_src: ");
        debug_print_buffer32(egr_ics[idx].eth_src, 6);
        DBG_MSG("eth_dst: ");
        debug_print_buffer32(egr_ics[idx].eth_dst, 6);
    }
```

```
}

void debug_print_ICheader(void* ICpacket){
   ICHeader_t *ich = (ICHeader_t*)ICpacket;

   DBG_MSG("ICheader:\n");
   DBG_MSG("type: %u\n", ich->type);
   DBG_MSG("ttl: %u\n", ich->ttl);
   DBG_MSG("dataLen: %u\n", ich->dataLen);
   DBG_MSG("route: %x\n", ich->route);

   if((ich->type == ICH_FIND_FILE) || (ich->type == ICH_ROUTE_FB) || (ich->type == ICH_CTRL_FB)){
      ICHeader_ext_t *iche = (ICHeader_ext_t*)(ICpacket+sizeof(ICHeader_t));
      DBG_MSG("sport: %u\n", iche->sport);
      DBG_MSG("dport: %u\n", iche->dport);
      DBG_MSG("saddr: %x\n", iche->saddr);
      DBG_MSG("daddr: %x\n", iche->daddr);
      DBG_MSG("eth_src: ");
      debug_print_buffer32(iche->eth_src, 6);
      DBG_MSG("eth_dst: ");
      debug_print_buffer32(iche->eth_dst, 6);
   }
}

/* Input: ICpacket: Pointer to IC packet
           port: input port number
    Pushes the given port number on the right side of the 4 byte route */
void ICPushPort(void* ICpacket, uint8_t port){
   ICHeader_t *ich = (ICHeader_t*)ICpacket;

   ich->route <<= 8;
   ich->route |= port;
   DBG_MSG("ICPushPort: ich->route: %x\n", ich->route);
}

/* Input: ICpacket: Pointer to IC packet
    Pops a port from the IC header route, and returns it. */
int ICPopPort(void* ICpacket){
   ICHeader_t *ich = (ICHeader_t*)ICpacket;
   int port;

```

```
    port = ich->route & 0xff;
    ich->route >>= 8;
    ich->route |= 0xff000000;
    DBG_MSG("ICPopPort: ich->route: %x\n", ich->route);
    DBG_MSG("ICPopPort: popped port: %u\n", port);
    return port;
}

/* Input: idx: Index of TCP session
   Output: index of IC session (or -1 if not found) */
int egrFindICsess(int idx){
    int i;

    for (i=0; i<MAX_IC_SESSIONS; i++) {
        if(egr_ics[i].status != ICS_CLOSED && egr_ics[i].saddr==ts[idx].saddr &&
             egr_ics[i].daddr==ts[idx].daddr && egr_ics[i].sport == ts[idx].sport &&
             egr_ics[i].dport == ts[idx].dport){
            /* We have found an active IC session that corresponds to the received packet */
            return i;
            DBG_MSG("egrFindICsess: found session\n");
        }
    }
    return -1;
}

/* Subtract the `struct timeval' values X and Y,
   storing the result in RESULT.
   Return 1 if the difference is negative, otherwise 0.  */
int timeval_subtract (struct timeval *result,
                                  struct timeval *x, struct timeval *y){
    /* Perform the carry for the later subtraction by updating y. */
    if (x->tv_usec < y->tv_usec) {
        int nsec = (y->tv_usec - x->tv_usec) / 1000000 + 1;
        y->tv_usec -= 1000000 * nsec;
        y->tv_sec += nsec;
    }
    if (x->tv_usec - y->tv_usec > 1000000) {
        int nsec = (x->tv_usec - y->tv_usec) / 1000000;
        y->tv_usec += 1000000 * nsec;
        y->tv_sec -= nsec;
    }

    /* Compute the time remaining to wait.
          tv_usec is certainly positive. */
    result->tv_sec = x->tv_sec - y->tv_sec;
    result->tv_usec = x->tv_usec - y->tv_usec;

    /* Return 1 if result is negative. */
    return x->tv_sec < y->tv_sec;
}

/* To do:
     Make linked lists of TCP structs and IC session structs */
```

A.2.5 bogus_tcp.h

```
#ifndef _BOGUS_TCP_H
#define _BOGUS_TCP_H

#include "dcube.h"

#define MAX_TCP_SESSIONS              MAX_CONCURRENT_STREAMS
#define ETH_HDR_SIZE                  14
#define ETH_TYPE_IP                 0x0800
#define IP_HDR_SIZE                   20
#define TCP_HDR_SIZE                  20

/* Return states */
#define B_TCP_ERROR       0
#define B_TCP_NO_DATA     1
#define B_TCP_DATA        2

/* TCP FLAGS */
#define TCP_FLAGS_FIN 1
#define TCP_FLAGS_SYN 1<<1
#define TCP_FLAGS_RST 1<<2
#define TCP_FLAGS_PSH 1<<3
#define TCP_FLAGS_ACK 1<<4
#define TCP_FLAGS_URG 1<<5

/* TCP states */
```

```
enum {
    TCP_CLOSED = 0,
    TCP_SYN_RCVD,
    TCP_ESTABLISHED,
    TCP_CLOSE_WAIT,
    TCP_FIN_WAIT,
    TCP_TIME_WAIT
};

/* TCP Change options */
enum {
    TCP_NO_CHANGE = 0,
    TCP_RST,
    TCP_FIN
};

typedef struct{
    uint8_t             status;
    uint32_t            saddr;
    uint32_t            daddr;
    uint16_t            sport;
    uint16_t            dport;
    uint32_t            local_num;
    uint32_t            remote_num;
    time_t              last_used;
    char               eth_dst[6];
    char               eth_src[6];
} tcp_session_s;
tcp_session_s ts[MAX_TCP_SESSIONS];

/* 14 bytes Ethernet header */
typedef struct{
    char         dst[6];
    char        src[6];
    uint16_t type;
} eth_hdr __attribute__((packed));

/* 20 bytes IP Header */
typedef struct{
    uint8_t version    : 4 ;           /* Version */
    uint8_t hlen         : 4 ;           /* Header length */
    uint8_t tos;                    /* Type of service */
    uint16_t length;                /* Total length */
    uint16_t ident;                 /* Identification */
    uint16_t flags    : 3 ;         /* Flags */
    uint16_t offset   : 13;         /* Fragment offset */
    uint8_t ttl;                    /* Time to live */
    uint8_t protocol;               /* Protocol */
    uint16_t checksum;              /* Header checksum */
    uint32_t  src;                  /* Source address */
    uint32_t  dst;                  /* Destination address */
}ip_hdr __attribute__((packed));

/* 20 bytes TCP Header */
typedef struct{
    uint16_t sport;           /* Source port */
    uint16_t dport;           /* Destination port */
    uint32_t seq;             /* Sequence Number */
    uint32_t ack;             /* Acknowledgement number */
    uint16_t hdrlen   : 4;    /* TCP Header length */
    uint16_t reserved : 6;    /* Reserverd – Zero */
    uint16_t flags    : 6;    /* Flags */
    uint16_t win;             /* Window size */
    uint16_t checksum;        /* Header Checksum */
    uint16_t urgptr;          /* Urgent pointer */
}tcp_hdr __attribute__((packed));

/* Exported methods */
int tcp_recv(void *dataVAddr, char *data, int *len, int *idx);
int tcp_send(void *data, int size, uint32_t saddr, uint32_t daddr, uint16_t sport, uint16_t dport);
void set_tcp_data(int ts_id, uint32_t sa, uint32_t da, uint16_t sp, uint16_t dp, uint32_t seq, uint8_t flags);
int find_create_tcp(uint32_t sa, uint32_t da, uint32_t sp, uint16_t dp, uint32_t seq, uint8_t flags);
int find_tcp(uint32_t sa, uint32_t da, uint32_t sp, uint16_t dp);
void get_eth_src(void *dataVAddr, char *eth_src);
void get_eth_dst(void *dataVAddr, char *eth_dst);
void get_ip_hlen(void *dataVAddr, uint8_t *ip_hlen);
void get_ip_data_len(void *dataVAddr, uint16_t *datagram_len);
void get_ip_src(void* dataVAddr, uint32_t *ip_sa);
void get_ip_dst(void* dataVAddr, uint32_t *ip_da);
void get_tcp_sport(void* dataVAddr, uint8_t ip_hlen, uint16_t *tcp_sp);
void get_tcp_dport(void* dataVAddr, uint8_t ip_hlen, uint16_t *tcp_dp);
void get_tcp_seq(void* dataVAddr, uint8_t ip_hlen, uint32_t *tcp_seq);
void get_tcp_ack(void* dataVAddr, uint8_t ip_hlen, uint32_t *tcp_ack);
void get_tcp_flags(void* dataVAddr, uint8_t ip_hlen, uint8_t *tcp_flags);
void get_tcp_hlen(void* dataVAddr, uint8_t ip_hlen, uint8_t *tcp_hlen);
void prepare_iph(ip_hdr *iph);
void prepare_tcph(tcp_hdr *tcph);
void debug_print_TCPsess(tcp_session_s tcps);

#endif /* #ifndef _BOGUS_TCP_H */
```

A.2.6 bogus_tcp.c

```
#include "bogus_tcp.h"

/*
 * (C)2003, Matija Puzar <matija@ifi.uio.no>
 * Adapted by Andreas Petlund 2005
 */

/* Input: ipData: Pointer to beginning of IP packet
   Returns: B_TCP_ERROR if error, B_TCP_NO_DATA if no data, B_TCP_DATA if data is received.
   data: pointer to packet payload (should be at least 1500 bytes long).
   len: length of packet payload.
   idx: index into tcp-struct-array (to get port and IP data for IC sessions)*/
int tcp_recv(void *ipData, char *data, int *len, int *idx){
  /* We already know this is a TCP packet destined for port 9070 */

  char     eth_src[6], eth_dst[6], resp_data[54];
  uint8_t            tcp_flags, new_tcp_flags, ip_hlen, tcp_hlen;
  uint16_t           tcp_sp, tcp_dp, datagram_len, tcp_len, data_len;
  uint32_t           ip_sa, ip_da, tcp_seq, tcp_ack;
  int                ts_id, tcp_ack_incr = 0, fin_incr = 0;

  *idx = -1;
  /* Get necessary data */
  /* Get eth src and dst addr */
  get_eth_src(ipData, eth_src);
  debug_print_buffer32(eth_src, 6);
  get_eth_dst(ipData, eth_dst);
  debug_print_buffer32(eth_dst, 6);

```

```
/* Get necessary data from IP header */
get_ip_hlen(ipData, &ip_hlen);
get_ip_data_len(ipData, &datagram_len);
get_ip_src(ipData, &ip_sa);
get_ip_dst(ipData, &ip_da);

/* Let's get all the necessary data from the TCP header */
get_tcp_sport(ipData, ip_hlen, &tcp_sp);
get_tcp_dport(ipData, ip_hlen, &tcp_dp);
get_tcp_seq(ipData, ip_hlen, &tcp_seq);
get_tcp_ack(ipData, ip_hlen, &tcp_ack);
get_tcp_flags(ipData, ip_hlen, &tcp_flags);
get_tcp_hlen(ipData, ip_hlen, &tcp_hlen);
tcp_len = datagram_len - (ip_hlen * 4); /* hlen is given in 32 bit words */
data_len = tcp_len - (tcp_hlen * 4); /* hlen is given in 32 bit words */

/* Let's find the TCP session which corresponds to the given data (or create a new one) */
ts_id = find_create_tcp(ip_sa, ip_da, tcp_sp, tcp_dp, tcp_seq, tcp_flags);
/* Copy MAC addresses into struct
    --- Do this only once (to be implemented :) --- */
memcpy(ts[ts_id].eth_src, eth_src, 6);
memcpy(ts[ts_id].eth_dst, eth_dst, 6);

/* Now, let's handle the received flags and set the sending ones.
 * By default, we set the ACK flag
 * tcp_ack_incr is set to 1 if we got a SYN or FIN flag (each of them "consumes" 1 sequence number)
 */
new_tcp_flags = TCP_FLAGS_ACK;
tcp_ack_incr = 0;

/* If we got a SYN packet, we will respond also with a SYN packet */
if (tcp_flags & TCP_FLAGS_SYN) {
   new_tcp_flags |= TCP_FLAGS_SYN;
   tcp_ack_incr = 1;
}

/* By default we push all received data immediately, so here we don't do anything */
if (tcp_flags & TCP_FLAGS_PSH) {
   DBG_MSG("TCP_FLAGS_PSH\n");
}

```

```
/* If we get an ACK, let's check if we have to switch to a different state */
if (tcp_flags & TCP_FLAGS_ACK) {
  switch (ts[ts_id].status) {

    /* We received the third TCP packet (last in the 3-way handshake operation) */
  case TCP_SYN_RCVD: {
    //fprintf(stderr, "Received 3hs ACK, dropping packet\n");
    ts[ts_id].status = TCP_ESTABLISHED;
    return B_TCP_NO_DATA;
  }

    /* We received the last ACK, after already closing the connection */
  case TCP_CLOSE_WAIT: {
    //fprintf(stderr, "Received last ACK, dropping packet\n");
    ts[ts_id].status = TCP_CLOSED;
    return B_TCP_NO_DATA;
  }

    /* We received the last ACK but still need to send ours */
  case TCP_FIN_WAIT: {
    ts[ts_id].status = TCP_TIME_WAIT;
    break;
  }

  default: break;
  }
}

/* If we get a FIN, we should respond with an ACK and, if we didn't initiate
 * a FIN before, we will do it now and pass to the CLOSE_WAIT state
 */
if (tcp_flags & TCP_FLAGS_FIN) {
  if (ts[ts_id].status == TCP_ESTABLISHED) {
    new_tcp_flags |= TCP_FLAGS_FIN;
    ts[ts_id].status = TCP_CLOSE_WAIT;
  }
  tcp_ack_incr = 1;
}

/* If we get a seq. number that we already ACK-ed, we don't do it again */
if (ts[ts_id].status == TCP_ESTABLISHED  &&  data_len == 0  &&  tcp_seq == ts[ts_id].remote_num) {
        //fprintf(stderr, "Already ACK-ed, dropping packet\n");
        return B_TCP_NO_DATA;
    }

    /* If any payload, copy it into buffer pointed to by data */
    if ((data_len > 0) && (ts[ts_id].status != TCP_FIN_WAIT)) {
        memcpy(data, ipData+ETH_HDR_SIZE+(ip_hlen*4)+(tcp_hlen*4), data_len);
        *len = data_len;
        *idx = ts_id;
    }

    /* Now, let's send a response */
    memset(resp_data, 0, sizeof(resp_data));
    if (tcp_seq == ts[ts_id].remote_num  &&  ts[ts_id].status != TCP_CLOSED) {

        /* Prepare ethernet header */
        eth_hdr ethh;
        memset(&ethh, 0, sizeof(ethh)); /* make sure struct is zeroed */
        memcpy(&ethh.dst, eth_src, sizeof(eth_src));
        memcpy(&ethh.src, eth_dst, sizeof(eth_dst));
        ethh.type = ETH_TYPE_IP;

        /* Let's prepare the IP header as well */
        ip_hdr iph;
        prepare_iph(&iph); /* Fill in predefined fields */
        iph.src = ip_da;
        iph.dst = ip_sa;
        iph.length = sizeof(resp_data)-ETH_HDR_SIZE;

        /* remote_num is the remote sequence number we expect next */
        ts[ts_id].remote_num = tcp_seq + data_len + tcp_ack_incr;

        /* Now, we prepare the TCP header as well */
        tcp_hdr tcph;
        prepare_tcph(&tcph); /* Fill in predefined fields */
        iph.src = ip_da;
        tcph.sport = tcp_dp;
        tcph.dport = tcp_sp;
        tcph.seq = ts[ts_id].local_num;
        tcph.ack = ts[ts_id].remote_num;
        tcph.flags = new_tcp_flags;

        /* Copy headers to buffer */
        memcpy(resp_data, &ethh, sizeof(ethh));
        memcpy(resp_data+ETH_HDR_SIZE, &iph, IP_HDR_SIZE);
        memcpy(resp_data+ETH_HDR_SIZE+IP_HDR_SIZE, &tcph, TCP_HDR_SIZE);

        /* Calculate checksums */
        ip_checksum(resp_data + ETH_HDR_SIZE);
        tcp_udp_checksum(resp_data + ETH_HDR_SIZE, IP_HDR_SIZE + TCP_HDR_SIZE);

        /* local_num has the next sequence number on our side */
        ts[ts_id].local_num += tcp_ack_incr + fin_incr;

    } else {
        /* Something went wrong */
        DBG_MSG("B_TCP_ERROR\n");
        return B_TCP_ERROR;
    }

    /* In this app, only port 0 (external) will need TCP */
    ixp_send_packet(resp_data, sizeof(resp_data), 0);

    /* If we were in the TIME_WAIT state and sent our last ACK,
     * we close the connection totally */
    if (ts[ts_id].status == TCP_TIME_WAIT)
        ts[ts_id].status = TCP_CLOSED;

    /* If we get a RST flag, we reset the connection and discard the packet */
    if (tcp_flags & TCP_FLAGS_RST) {
        ts[ts_id].status = TCP_CLOSED;
    }

    /* Give proper return value */
    if(data_len > 0)
        return B_TCP_DATA;
    else
        return B_TCP_NO_DATA;
}

/* Input: data:    Pointer to payload.
             size:    Size of payload in bytes.
```

```
                saddr:   Source IP address.
                daddr:   Destination IP address.
                sport:   Source port.
                dport:   Destination port.
                change: Status change (i.e. FIN or RST) (0 if no change)
   Output: Nuber of bytes sent, −1 if error. */
int tcp_send(void *data, int size, uint32_t saddr, uint32_t daddr, uint16_t sport, uint16_t dport){
  int tcpIdx, psize = size + TCP_HDR_SIZE + IP_HDR_SIZE + ETH_HDR_SIZE;
  char *packet = malloc(psize);

  tcpIdx = find_tcp(saddr, daddr, sport, dport);
  if(tcpIdx == −1)
    return tcpIdx;

  DBG_MSG("tcp_send(): Found TCP stream. Index: %i\n", tcpIdx);

  /* Prepare ethernet header */
  eth_hdr eh;
  memcpy(&eh.dst, &ts[tcpIdx].eth_src, 6);
  memcpy(&eh.src, &ts[tcpIdx].eth_dst, 6);
  eh.type = ETH_TYPE_IP;

  DBG_MSG("After ETH header prepare");

  /* Let's prepare the IP header as well */
  ip_hdr iph;
  prepare_iph(&iph); /* Fill in predefined fields */
  iph.src = daddr;
  iph.dst = saddr;
  iph.length = (uint16_t)(psize–ETH_HDR_SIZE);

  DBG_MSG("After IP header prepare");

  /* Now, we prepare the TCP header as well */
  tcp_hdr tcph;
  prepare_tcph(&tcph); /* Fill in predefined fields */
  tcph.sport = ts[tcpIdx].dport;
  tcph.dport = ts[tcpIdx].sport;
  tcph.seq = ts[tcpIdx].local_num;
  tcph.ack = ts[tcpIdx].remote_num;
  tcph.flags = TCP_FLAGS_PSH | TCP_FLAGS_ACK;
```

```

    DBG_MSG("After TCP header prepare");

    /* Increment local_num with number of bytes sent */
    ts[tcpIdx].local_num = ts[tcpIdx].local_num + size;

    /* Copy headers to packet */
    memcpy(packet, &eh, ETH_HDR_SIZE);
    memcpy(packet+ETH_HDR_SIZE, &iph, ETH_HDR_SIZE+IP_HDR_SIZE);
    memcpy(packet+ETH_HDR_SIZE+IP_HDR_SIZE, &tcph, ETH_HDR_SIZE+IP_HDR_SIZE+TCP_HDR_SIZE);
    /* Copy payload to packet */
    memcpy(packet+ETH_HDR_SIZE+IP_HDR_SIZE+TCP_HDR_SIZE, data, size);

    debug_print_buffer32(packet, psize);
    /* Calculate checksums */
    ip_checksum(packet + ETH_HDR_SIZE);
    tcp_udp_checksum(packet + ETH_HDR_SIZE, size + IP_HDR_SIZE + TCP_HDR_SIZE);

    ixp_send_packet(packet, psize, 0);
    free(packet);

    return size;
}
```

A.2.7 bogus_tcp_utils.c

```
#include "bogus_tcp.h"

/* Look up our table of established TCP connections – if found, return the index in the table
 * If not, create a new entry */
int find_create_tcp(uint32_t sa, uint32_t da, uint32_t sp, uint16_t dp, uint32_t seq, uint8_t flags){
    int i, pos = -1, min_time_id = 0, first_free = -1;
    time_t min_time;

    min_time = time(NULL);

    for (i=0; i<MAX_TCP_SESSIONS; i++) {
        if(ts[i].status != TCP_CLOSED && ts[i].saddr==sa && ts[i].daddr==da && ts[i].sport == sp && ts[i].dport ==
            /* We have found an active stream that corresponds to the packet */
            pos = i;
            DBG_MSG("b_tcp: found stream\n");
            break;
        } else if(ts[i].status != TCP_CLOSED){
            /* This is an active stream that does not correspond to the packet.
                 Check when it was last used */
            if (ts[i].last_used < min_time) {
                min_time = ts[i].last_used;
                min_time_id = i;
            }
        } else if(first_free == −1){
            /* First free slot */
            first_free = i;
        }
    }

    /* Use the first free slot, if none, reuse the least recently used */
    if (pos == −1){
        if(first_free != −1){
            pos = first_free;
            DBG_MSG("b_tcp: using first free\n");
        }else{
            pos = min_time_id;
            DBG_MSG("b_tcp: using lru\n");
        }
    }

    set_tcp_data(pos, sa, da, sp, dp, seq, flags);

    return pos;
}

/* Look up our table of established TCP connections.
     If found, return the index in the table.
     If not found, return −1 */
int find_tcp(uint32_t sa, uint32_t da, uint32_t sp, uint16_t dp){
    int i;

    for (i=0; i < MAX_TCP_SESSIONS; i++) {
        if(ts[i].status != TCP_CLOSED && ts[i].saddr == sa
              && ts[i].daddr == da && ts[i].sport == sp && ts[i].dport == dp) {
            /* We have found an active stream that corresponds to the packet */
        DBG_MSG("b_tcp: found stream\n");
        return i;
      }
    }
    DBG_MSG("find_tcp: No matching stream found\n");
    return -1;
}

void set_tcp_data(int ts_id, uint32_t sa, uint32_t da, uint16_t sp, uint16_t dp, uint32_t seq, uint8_t flags){
    if (flags & TCP_FLAGS_SYN){
          ts[ts_id].remote_num = seq;
          ts[ts_id].local_num = 1;
          ts[ts_id].status = TCP_SYN_RCVD;
      }
      if (ts[ts_id].status == TCP_SYN_RCVD){
          ts[ts_id].saddr = sa;
          ts[ts_id].daddr = da;
          ts[ts_id].sport = sp;
          ts[ts_id].dport = dp;
      }
      ts[ts_id].last_used = time(NULL);
  }

/* Input: Pointer to start of eth packet
      Output: Eth src - 6 bytes */
void get_eth_src(void *dataVAddr, char *eth_src){
    memcpy(eth_src, dataVAddr+6, 6);
}

/* Input: Pointer to start of eth packet
      Output: Eth dst - 6 bytes */
void get_eth_dst(void *dataVAddr, char *eth_dst){
    memcpy(eth_dst, dataVAddr, 6);
}

/* Input: Pointer to start of eth packet
      Output: ip src - 4 bytes */
void get_ip_src(void* dataVAddr, uint32_t *ip_sa){
    memcpy(ip_sa, dataVAddr+ETH_HDR_SIZE+12, 4);
}

```

```
/* Input: Pointer to start of eth packet
     Output: ip dst – 4 bytes */
void get_ip_dst(void* dataVAddr, uint32_t *ip_da){
   memcpy(ip_da, dataVAddr+ETH_HDR_SIZE+16, 4);
}

/* Input: Pointer to start of eth packet
     Output: IP Hlen – 4 bit */
void get_ip_hlen(void *dataVAddr, uint8_t *ip_hlen){
   memcpy(ip_hlen, dataVAddr+ETH_HDR_SIZE, 1);
   *ip_hlen &= 0xf; /* Hlen is the 4 lsb */
}

/* Input: Pointer to start of eth packet
     Output: IP length – 2 bytes */
void get_ip_data_len(void *dataVAddr, uint16_t *datagram_len){
   memcpy(datagram_len, dataVAddr+ETH_HDR_SIZE+2, 2);
}

/* Input: dataVAddr: Pointer to start of eth packet
     ip_hlen: length of IP header in 32 bit words
     Output: TCP source port – 2 bytes */
void get_tcp_sport(void* dataVAddr, uint8_t ip_hlen, uint16_t *tcp_sp){
   int tcp_hdr_off = ip_hlen * 4; /* ip_hlen is given in 32 bit words */
   memcpy(tcp_sp, dataVAddr+ETH_HDR_SIZE+tcp_hdr_off, 2);
}

/* Input: dataVAddr: Pointer to start of eth packet
     ip_hlen: length of IP header in 32 bit words
     Output: TCP dest port – 2 bytes */
void get_tcp_dport(void* dataVAddr, uint8_t ip_hlen, uint16_t *tcp_dp){
   int tcp_hdr_off = ip_hlen * 4; /* ip_hlen is given in 32 bit words */
   memcpy(tcp_dp, dataVAddr+ETH_HDR_SIZE+tcp_hdr_off+2, 2);
}

/* Input: dataVAddr: Pointer to start of eth packet
     ip_hlen: length of IP header in 32 bit words
     Output: TCP seq  – 4 bytes */
void get_tcp_seq(void* dataVAddr, uint8_t ip_hlen, uint32_t *tcp_seq){
   int tcp_hdr_off = ip_hlen * 4; /* ip_hlen is given in 32 bit words */
   memcpy( tcp_seq, dataVAddr+ETH_HDR_SIZE+tcp_hdr_off+4, 4);
```

```
}

/* Input: dataVAddr: Pointer to start of eth packet
   ip_hlen: length of IP header in 32 bit words
   Output: TCP ack - 4 bytes */
void get_tcp_ack(void* dataVAddr, uint8_t ip_hlen, uint32_t *tcp_ack){
  int tcp_hdr_off = ip_hlen * 4; /* ip_hlen is given in 32 bit words */
  memcpy( tcp_ack, dataVAddr+ETH_HDR_SIZE+tcp_hdr_off+8, 4);
}

/* Input: dataVAddr: Pointer to start of eth packet
   ip_hlen: length of IP header in 32 bit words
   Output: TCP flags - 6 bits */
void get_tcp_flags(void* dataVAddr, uint8_t ip_hlen, uint8_t *tcp_flags){
  int tcp_hdr_off = ip_hlen * 4; /* ip_hlen is given in 32 bit words */
  memcpy( tcp_flags, dataVAddr+ETH_HDR_SIZE+tcp_hdr_off+13, 1);
  *tcp_flags &= 0x3F; /* all but the 6 lsb's */
}

/* Input: dataVAddr: Pointer to start of eth packet
   ip_hlen: length of IP header in 32 bit words
   Output: TCP header length  - 4 bits */
void get_tcp_hlen(void* dataVAddr, uint8_t ip_hlen, uint8_t *tcp_hlen){
  int tcp_hdr_off = ip_hlen * 4; /* ip_hlen is given in 32 bit words */
  memcpy( tcp_hlen, dataVAddr+ETH_HDR_SIZE+tcp_hdr_off+12, 1);
  *tcp_hlen >>= 4; /* Hlen is in the 4 msb's */
}

void prepare_iph(ip_hdr *iph){
  memset(iph, 0, sizeof(ip_hdr)); /* make sure struct is zeroed */
  iph->ttl = 64;
  iph->version = 4;
  iph->hlen = (sizeof(ip_hdr) / 4 ); /* hlen is given in 32 bit words */
  iph->flags = 2;
  iph->protocol = 6;
}

void prepare_tcph(tcp_hdr *tcph){
  memset(tcph, 0, sizeof(tcp_hdr)); /* make sure struct is zeroed */
  tcph->hdrlen = (sizeof(tcp_hdr) / 4); /* hlen is given in 32 bit words */
  tcph->win = 0x16d0;
```

```
}

void debug_print_TCPsess(tcp_session_s tcps){
   DBG_MSG("TCP session:\n");
   DBG_MSG("tcps.status: %i\n", tcps.status);
   DBG_MSG("tcps.saddr: %x\n", tcps.saddr);
   DBG_MSG("tcps.daddr: %x\n", tcps.daddr);
   DBG_MSG("tcps.sport: %i\n", tcps.sport);
   DBG_MSG("tcps.dport: %i\n", tcps.dport);
   DBG_MSG("tcps.local_num: %x\n", tcps.local_num);
   DBG_MSG("tcps.remote_num: %x\n", tcps.remote_num);
   DBG_MSG("tcps.last_used: %x\n", (uint32_t)tcps.last_used);
   DBG_MSG("eth_dst: 0x%x\n", *((uint32_t*)&(tcps.eth_dst)));
   DBG_MSG("eth_src: 0x%x\n", *((uint32_t*)&(tcps.eth_src)));
}
```

A.3 Host source

A.3.1 srtsp.h

```
#ifndef _SRTSP_H
#define _SRTSP_H

/* felles */
#include <pthread.h>
#include <stdio.h>
#include <linux/types.h>
#include <time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <sys/sendfile.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <errno.h>
#include <ctype.h>
#include <netdb.h>
#include <string.h>
#include <arpa/inet.h>
```

```
#include <fcntl.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <ctype.h>
#include <sys/resource.h>
#include <sys/mman.h>
#include <unistd.h>
#include <linux/unistd.h>
#include <syscall.h>

/* "Local" includes */
#include "dbg_msg.h"
#include "rtp.h"
#include "ic.h"


#define UDP_CORK 1
#define NSEC_PER_USEC (1000L)

#define MYPORT 23456      // the port users will be connecting to
#define BACKLOG 5         // how many pending connections queue will hold
#define MAX_RTP_STREAMS 100
#define ETH_TYPE_IP                     0x0800
#define BIT_WRAP 0xffff

#define MEDIASTREAM_PLAY                  1
#define MEDIASTREAM_PAUSE                 2
#define MEDIASTREAM_FF                    3
#define MEDIASTREAM_REWIND                4
#define MEDIASTREAM_TEARDOWN              5
#define MEDIASTREAM_STATISTICS_INIT   6
#define MEDIASTREAM_STATISTICS          7
#define MEDIASTREAM_STATUS                8

#define SESSION_STATUS_INIT           201
#define SESSION_STATUS_RUNNING        202
#define SESSION_STATUS_PAUSED         203
#define SESSION_STATUS_FF_REWIND    204
#define SESSION_STATUS_STOPED         205


```

```
typedef struct {
    uint32_t ipsrc;
    uint32_t ipdst;
    uint8_t notused;/* always zero */
    uint8_t proto;/* protocol used */
    uint16_t len;/* UDP len */
} pseudo_udp __attribute__((packed));

/* 14 bytes Ethernet header */
typedef struct{
    char        dst[6];
    char       src[6];
    uint16_t type;
} eth_hdr __attribute__((packed));

/* 20 bytes IP Header */
typedef struct{
    uint8_t version    : 4 ;            /* Version */
    uint8_t hlen          : 4 ;            /* Header length */
    uint8_t tos;                                /* Type of service */
    uint16_t length;                         /* Total length */
    uint16_t ident;                           /* Identification */
    uint16_t flags      : 3 ;             /* Flags */
    uint16_t offset     : 13;            /* Fragment offset */
    uint8_t ttl;                                 /* Time to live */
    uint8_t protocol;                       /* Protocol */
    uint16_t checksum;                     /* Header checksum */
    uint32_t   src;                            /* Source address */
    uint32_t   dst;                            /* Destination address */
}ip_hdr __attribute__((packed));

/* 8 bytes UDP Header */
typedef struct{
    uint16_t sport;                 /* Source port */
    uint16_t dport;                 /* Destination port */
    uint16_t len;                     /* Length */
    uint16_t csum;                     /* Checksum */
}udp_hdr __attribute__((packed));

typedef struct {
    uint32_t secs;
    uint32_t frac;
} ntp64_t;

/* supported RTSP message types: */
enum {
    RTSP_SETUP = 1,
    RTSP_PLAY,
    RTSP_UNKNOWN
};

struct playinfo {
    pid_t pid;               /* initielt 0, settes etter retur fra første play kall. */
    int in_fd;               /* fil som skal streames. */
    int out_fd;              /* socket. */
    loff_t start_offset;     /* start offset i in_fd. */
    loff_t stop_offset;      /* stop offset i in_fd. */
    long trans_interval;     /* transmission interval:
                                  - mpeg ts: antall usec mellom hver pakke.
                                  - mpeg es: antall usec mellom hver frame.
                              */
    uint16_t max_pkt_len;    /* max pakke lengde:
                                  - mpeg ts payload:
                                  -   max_pkt_len-rtp_hdr_size.
                                  -   max_pkt_len må være (188*n)+rtp_hdr_size.
                                  - mpeg es payload:
                                  -   payload > 1 && payload < max_pkt_len-rtp_hdr_size-mpeg_hdr_size.
                              */
    uint32_t tsinc;          /* timestamp increment:
                                  - mpeg ts: increment per pakke
                                  - mpeg es: increment per frame
                              */
    uint8_t type;            /* type media */
    /*RTP*/
    uint32_t ssrc;           /* synchronization source */

    unsigned long head;
    unsigned long tail;
    unsigned long datasize;
};


```

```
struct ff_rewind_info {
    loff_t start_offset;
    loff_t stop_offset;
};

struct ses_stat {
    ntp64_t ntp_timestamp;
    uint32_t rtp_timestamp;
    uint32_t packet_count;
    uint32_t octet_count;
    uint16_t rtp_seq;
};

struct ses_stat_first {
    uint32_t rtp_timestamp;
    ntp64_t ntp_timestamp;
};

typedef struct {
    //struct list_head list;
    uint32_t sid;
    pid_t pid;
    uint8_t status;

    int in_fd;
    int out_fd;
    loff_t start_offset;
    loff_t stop_offset;
    unsigned long buffaddr;

    long trans_interval; //i usec
    uint32_t tsinc;
    uint16_t max_pkt_len;

    /*RTP*/
    uint16_t rtp_seq;
    uint32_t rtp_timestamp;
    uint32_t ntp_timestamp;
    uint32_t ssrc;

    /* The value of the RTP timestamp when the session opened (chosen randomly)
    */
  uint16_t init_RTP_seq;
  uint32_t init_RTP_timestamp;
  ntp64_t  init_NTP_timestamp;

  struct timeval time_start; // settes til timeofday etter pause
  /* current rtp timestamp offset*/
  struct timeval time_elapsed; // settes til 0 ved pause

  //DISSE NULLES UT FOR HVER GANG STAT KALLES
  uint32_t packet_count;
  uint32_t octet_count;

  unsigned long head;
  unsigned long tail;
  unsigned long datasize;

} rtp_session_info_t;

enum {
  RTPS_CLOSED = 0,
  RTPS_ACTIVE
};

enum {
  R_EGRESS = 0,
  R_OUTBOUND
};

typedef struct{
  uint8_t status;
  int thdId; /* Id of streaming thread */
  int client_rtp_port;
  int client_rtcp_port;
  int server_rtp_port;
  int server_rtcp_port;
  int icIdx; /* Index of corresponding IC session */
  char filename[100];
} RTP_stream_t;
RTP_stream_t rtps[MAX_RTP_STREAMS];

```

```
/* Socket descriptor for comm with XScale */
int xscale_sock;


void debug_print_ICheader(void* ICpacket);
void debug_print_buffer32(void* buf, int size);
void debug_print_ICsession(int idx);
struct timeval usec2tv(long usec);
struct timeval AddTimes(struct timeval *time1, struct timeval *time2);
struct timeval SubTimes(struct timeval *time1, struct timeval *time2);
int getRtspType(char* data);
int getRtspSession(char *data);
void getFileName(char *data, char* fn);
int newICSession(void* ICheader, void* icHdrExt);
int newRtpStream(int icIdx, ICHeader_t *ich, char *data);
void getClientPorts(char *data, int *c_rtp_p, int *c_rtcp_p);
void createRtspReply(int ses, char *data, int type, char *reply);
int addICheader(IC_session_t ics, char *data, int size, int ICtype, int routeType);
void addICExtHeader(IC_session_t ics, char *data, int size);
void sendToXScale(char *reply, int size);
uint32_t egressRoute(uint32_t route);

void partial_csum(uint32_t *sum, void* data, int len);
void udp_checksum(void* ipHdrStart, int size);
void ip_checksum(void* ipHdrStart);

#endif /* #ifndef _SRTSP_H */
```

A.3.2 srtsp.c

```
#include "srtsp.h"

/* Will send the whole file with packets in a predefined interval */
void *mpeg_ps(RTP_stream_t *streamInfo) {
  char *path = "/opt/hardhat/previewkit/arm/xscale_be/target/opt/storage/", fullpath[100];
  char *packet = malloc(1500); //char packet[1500];
  int hdrs_size= sizeof(ICHeader_t) + sizeof(eth_hdr) + sizeof(ip_hdr) + sizeof(udp_hdr) + sizeof(rtp_hdr_t);
  ICHeader_t *ich=(ICHeader_t*)packet;
  eth_hdr *eth=(eth_hdr*)(packet+sizeof(ICHeader_t));
  ip_hdr *iph=(ip_hdr*)(packet+sizeof(ICHeader_t) + sizeof(eth_hdr));
```

```
udp_hdr *udph=(udp_hdr*)(packet+sizeof(ICHeader_t) + sizeof(eth_hdr) + sizeof(ip_hdr));
rtp_hdr_t *rtph = (rtp_hdr_t*)(packet+sizeof(ICHeader_t) + sizeof(eth_hdr) + sizeof(ip_hdr)+sizeof(udp_hdr));
uint32_t new_rtp_seq, new_rtp_ts;
int fd, bytes_read;

DBG_MSG("***********NEW THREAD - ID: %i ************\n", (int)pthread_self());

/* Generate full path to file */
sprintf(fullpath, "%s%s", path, streamInfo->filename);

/* Open file */
fd = open(fullpath, O_RDONLY);
if(fd == -1){
   DBG_MSG("open() failed\n");
   DBG_MSG("Couldn't open file. error: %s. \n", strerror (errno));
   goto out;
}else{
   DBG_MSG("Successfully opened %s, fd: %i\n", fullpath, (int)fd);
}
/* Init IC header */
memset(ich, 0, sizeof(ICHeader_t));
ich->type = ICH_RTP;
ich->ttl = 4;
ich->route = htonl(ics[streamInfo->icIdx].route);

/* Init ETH header */
memset(eth, 0, sizeof(eth_hdr));
memcpy(eth->dst, ics[streamInfo->icIdx].eth_src, 6);
memcpy(eth->src, ics[streamInfo->icIdx].eth_dst, 6);
eth->type = htons(ETH_TYPE_IP);

/* Init IP header */
memset(iph, 0, sizeof(ip_hdr));
//iph->version = 4;
//iph->hlen = 5;
*((uint8_t*)iph) = 0x45;
//iph->flags = 2;
*((uint8_t*)iph+6) = 0x40;

iph->ttl = 64;
iph->protocol = 0x11;
```

```
iph->src = htonl(ics[streamInfo->icIdx].daddr);
iph->dst = htonl(ics[streamInfo->icIdx].saddr);

/* Init UDP header */
memset(udph, 0, sizeof(udp_hdr));
udph->sport = htons(streamInfo->server_rtp_port);
udph->dport = htons(streamInfo->client_rtp_port);
udph->csum   = 0xffff;

/* Init RTP header */
memset(rtph, 0, sizeof(rtp_hdr_t));
rtph->version = RTP_VERSION;
//rtp_hdr.p = 0;
//rtp_hdr.x = 0;
//rtp_hdr.cc = 0;
//rtp_hdr.m = 0;
rtph->pt = PT_MP2P;
rtph->seq = htons((uint16_t)rand());
rtph->ts = htonl(rand());
rtph->ssrc = htonl(rand());

/****************************************/
/*                STREAM FILE                 */
/****************************************/
while(streamInfo->status==RTPS_ACTIVE) {

   /* Read file data into buffer+headers */
   bytes_read = read( fd, packet + hdrs_size, 1400);
   if(bytes_read > 0){

      /* Update IC header */
      ich->dataLen = htons(bytes_read + sizeof(eth_hdr) + sizeof(ip_hdr) + sizeof(udp_hdr) + sizeof(rtp_hdr_t));
      /* Update IP header */
      iph->length = htons(bytes_read + sizeof(ip_hdr) + sizeof(udp_hdr) + sizeof(rtp_hdr_t));
      /* Update UDP header */
      udph->len = htons(bytes_read + sizeof(udp_hdr) + sizeof(rtp_hdr_t));

      /* Update RTP header */
      new_rtp_seq = ntohs(rtph->seq);
      rtph->seq = htons(new_rtp_seq++);
      new_rtp_ts = ntohl(rtph->ts);
```

```
            rtph->ts = htonl(new_rtp_ts++); /* Update this to correct values */

            /* Calculate checksums */
            ip_checksum(iph);
            udp_checksum(iph, bytes_read + sizeof(rtp_hdr_t)+sizeof(udp_hdr));

            /* Send packet to socket */
            sendToXScale(packet, bytes_read + hdrs_size);

            /* Wait appropriate amount of time */
            usleep(2000);
        }else{
            close(fd);
            DBG_MSG("*********************BREAK***************************\n");
            break;
        }
    }
  out:
    DBG_MSG("********** EXIT THREAD *********\n");
    free(packet);
    pthread_exit(0);
}

int main(){

    int sockfd;   // listen on sock_fd, new connection on xscale_sock
    struct sockaddr_in my_addr;      // my address information
    struct sockaddr_in their_addr; // connector's address information
    int sin_size, dataLen, rtspType, icType;
    char data[1500], icheader[20];
    int status, recv_size, threadcount=0;
    pthread_t thread[100];

    sockfd = socket(AF_INET, SOCK_STREAM, 0); // do some error checking!
    if(sockfd == -1){
      printf("main(): socket returned error\n");
    }else{
      DBG_MSG("main(): Socket successfully created\n");
    }

    my_addr.sin_family = AF_INET;          // host byte order
    my_addr.sin_port = htons(MYPORT);     // short, network byte order
    my_addr.sin_addr.s_addr = INADDR_ANY; // auto-fill with my IP --endre til 192.168.1.1
    memset(&(my_addr.sin_zero), '\0', 8); // zero the rest of the struct

    status = bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));
    if(status == -1){
      printf("main(): bind returned error\n");
      exit(1);
    }else{
      printf("main(): Port %i successfully bound\n", MYPORT);
    }

    status = listen(sockfd, BACKLOG);
    if(status == -1){
      printf("main(): listen() returned error\n");
    }else{
      printf("main(): Listening on port %i\n", MYPORT);
    }

    sin_size = sizeof(struct sockaddr_in);

    status = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size);
    if(status == -1){
      printf("main(): accept() returned error\n");
    }else{
      printf("main(): accepting connection on port %i\n", MYPORT);
      xscale_sock = status;
    }

    /* Close listen-socket (will only need one connection) */
    close(sockfd);

    while(1){
      /* Assumes that IC Header is the first to arrive,
         and that all packets arrive in the order: ICHeader->Payload->ICHeader->Payload etc. */

      /* Read IC header */
      status = recv(xscale_sock, icheader, sizeof(ICHeader_t), 0);
      if(status == -1){
        DBG_MSG("main(): recv() returned error\n");
      }else if(status > 0){
        DBG_MSG("main(): Received IC header, %i bytes\n", status);
      }

      /* Get size of data from IC header
         If ICH type is ICH_FIND_FILE, we also have to
         receive an IC header extension with address-info */
      ICHeader_t *ich = (ICHeader_t*)icheader;
      icType = ich->type;
      dataLen = ntohs(ich->dataLen);
      if((icType == ICH_FIND_FILE))
        recv_size = dataLen + sizeof(ICHeader_ext_t);
      else
        recv_size = dataLen;

      status = recv(xscale_sock, data, recv_size, 0);
      if(status == -1){
        DBG_MSG("main(): recv() returned error\n");
      }else if(status > 0){
        DBG_MSG("main(): Received %i bytes of data.\n", status);
      }

      if(icType == ICH_FIND_FILE)
        DBG_MSG("main(): Data: %s\n", data+sizeof(ICHeader_ext_t));
      else
        DBG_MSG("main(): Data: %s\n", data);


      /* Get RTSP message type
         take appropriate action
         IF type==ICH_FIND_FILE data resides after IC header ext*/
      if(icType == ICH_FIND_FILE){
        DBG_MSG("ICH_FIND_FILE\n");
        rtspType = getRtspType(data + sizeof(ICHeader_ext_t));
      }else{
        rtspType = getRtspType(data);
      }
      if(rtspType == RTSP_SETUP){
        int rtpIdx, icIdx, packetSize, replysize; /* RTP-session index  and IC-session index*/
        char reply[200];
        DBG_MSG("main(): Received RTSP_SETUP\n");

        /* Setup icsession struct */
        icIdx = newICSession(ich, data);
        /* Create RTP stream struct */
        rtpIdx = newRtpStream(icIdx, ich, data+sizeof(ICHeader_ext_t));

        /* Give RTSP feedback */
        createRtspReply(rtpIdx, (char*)(data+sizeof(ICHeader_ext_t)), RTSP_SETUP, reply);
        replysize = strlen(reply);
        packetSize = addICheader(ics[icIdx], reply, replysize, ICH_CTRL_FB, R_EGRESS);
        addICExtHeader(ics[icIdx], reply, replysize);
        packetSize+=sizeof(ICHeader_ext_t);
        sendToXScale(reply, packetSize);

      }else if(rtspType == RTSP_PLAY){
        int rtpIdx, packetSize, replysize;
        char reply[200];
        DBG_MSG("main(): Received RTSP PLAY\n");

        /* Give rtsp feedback */
        rtpIdx = getRtspSession(data);

        createRtspReply(rtpIdx, (char*)data, RTSP_PLAY, reply);
        replysize = strlen(reply);
        packetSize = addICheader(ics[rtps[rtpIdx].icIdx], reply, replysize, ICH_CTRL_FB, R_EGRESS);
        addICExtHeader(ics[rtps[rtpIdx].icIdx], reply, replysize);
        packetSize+=sizeof(ICHeader_ext_t);

        sendToXScale(reply, packetSize);

        /* Create rtp stream thread */
        /* Implement handling of thread_exit */
        pthread_create( &(thread[threadcount++]), NULL, (void*)mpeg_ps, &rtps[rtpIdx]);
        //threadcount++;

      }else if(rtspType == RTSP_UNKNOWN){
        DBG_MSG("main(): Unknown RTSP type received, packet discarded.\n");
      }
    }
 }
```

A.3.3 srtsp_utils.h

```
#ifndef _SRTPS_UTILS_H
#define _SRTPS_UTILS_H

#include <netinet/in.h>
#include <string.h>
#include <sys/time.h>
#include <time.h>
#include "dbg_msg.h"
#include "ic.h"

void debug_print_ICheader(void* ICpacket);
void debug_print_buffer32(void* buf, int size);

#ifndef _RTSP_TYPE
#define _RTSP_TYPE
enum {
   RTSP_SETUP = 1,
   RTSP_PLAY,
   RTSP_UNKNOWN
};
#endif

struct timeval usec2tv(long usec);
struct timeval AddTimes(struct timeval *time1, struct timeval *time2);
struct timeval SubTimes(struct timeval *time1, struct timeval *time2);
int getRTSPType(char* data);

#endif /* #ifndef _SRTPS_UTILS_H */
```

A.3.4 srtsp_utils.c

```
#include "srtsp.h"

pthread_mutex_t socklock = PTHREAD_MUTEX_INITIALIZER;

void debug_print_ICheader(void* ICpacket){
   ICHeader_t *ich = (ICHeader_t*)ICpacket;

```

```
    DBG_MSG("ICheader:\n");
    DBG_MSG("type: %u\n", ich->type);
    DBG_MSG("ttl: %u\n", ich->ttl);
    DBG_MSG("dataLen: %u\n", ich->dataLen);
    DBG_MSG("route: %x\n", ich->route);

    if((ich->type == ICH_FIND_FILE) || (ich->type == ICH_ROUTE_FB) || (ich->type == ICH_CTRL_FB)){
        ICHeader_ext_t *iche = (ICHeader_ext_t*)(ICpacket+sizeof(ICHeader_t));
        DBG_MSG("sport: %u\n", iche->sport);
        DBG_MSG("dport: %u\n", iche->dport);
        DBG_MSG("saddr: %u\n", iche->saddr);
        DBG_MSG("daddr: %u\n", iche->daddr);
        DBG_MSG("eth_src: ");
        debug_print_buffer32(iche->eth_src, 6);
        DBG_MSG("eth_dst: ");
        debug_print_buffer32(iche->eth_dst, 6);
    }
}

void debug_print_ICsession(int idx){
    if(idx < MAX_IC_SESSIONS){
        DBG_MSG("IC-session:\n");
        DBG_MSG("index: %i\n", idx);
        DBG_MSG("status: %u\n", ics[idx].status);
        DBG_MSG("route: %x\n", ics[idx].route);
        DBG_MSG("sport: %u\n", ics[idx].sport);
        DBG_MSG("dport: %u\n", ics[idx].dport);
        DBG_MSG("saddr: %x\n", ics[idx].saddr);
        DBG_MSG("daddr: %x\n", ics[idx].daddr);
        DBG_MSG("eth_src: ");
        debug_print_buffer32(ics[idx].eth_src, 6);
        DBG_MSG("eth_dst: ");
        debug_print_buffer32(ics[idx].eth_dst, 6);

    }
}

/* Print buffer in 32 bit words hex in byte order*/
void debug_print_buffer32(void* buf, int size){
    int i;

    for(i=0; i < size; i++){
        if(i % 4 == 0)
            DBG_MSG("\nbuf[%i]: 0x", i);

        if(*((uint8_t*)(buf+i)) == 0)
            DBG_MSG("00");
        else if(*((uint8_t*)(buf+i)) < 15)
            DBG_MSG("0%x", *((uint8_t*)(buf+i)));
        else
            DBG_MSG("%x", *((uint8_t*)(buf+i)));
    }
    DBG_MSG("\n");
}

/* Return RTSP message type, based on the first characters in the packet */
int getRtspType(char* data){
    char *setup = "SETUP";
    char *play = "PLAY";

    if(strncmp(data, setup, strlen(setup)) == 0)
        return RTSP_SETUP;
    else if(strncmp(data, play, strlen(play)) == 0)
        return RTSP_PLAY;
    else
        return RTSP_UNKNOWN;

    return 0;
}

/* Initialize IC session from data
     Input: Pointer to IC header
     Output: index of new IC session (-1 if no free slot)
*/
int newICSession(void* ICheader, void* icHdrExt){
    ICHeader_t *ich = (ICHeader_t*)ICheader;
    ICHeader_ext_t *iche = (ICHeader_ext_t*)icHdrExt;
    int i, pos = -1;
    /* locate first free slot */
    for (i=0; i<MAX_IC_SESSIONS; i++) {
        if(ics[i].status == ICS_CLOSED){
            pos = i;
            DBG_MSG("newICSession: first free: %i\n", pos);
            break;
        }
    }

    if(pos != -1){
        ics[pos].status = ICS_ACTIVE;
        ics[pos].route = ntohl(ich->route);
        ics[pos].sport = ntohs(iche->sport);
        ics[pos].dport = ntohs(iche->dport);
        ics[pos].saddr = ntohl(iche->saddr);
        ics[pos].daddr = ntohl(iche->daddr);
        memcpy(&ics[pos].eth_src, &iche->eth_src, 6);
        memcpy(&ics[pos].eth_dst, &iche->eth_dst, 6);
    }
    return pos;
}


/*Input: icIdx: index int array of ic-session structs
              ich: pointer to start of IC header
              data: pointer to packet data

              Allocate RTP-stream struct
              return index (session nr */
int newRtpStream(int icIdx, ICHeader_t *ich, char *data){
    int i, pos=-1, c_rtp_p, c_rtcp_p;
    char fn[100];

    /* locate first free slot */
    for (i=0; i<MAX_RTP_STREAMS; i++) {
        if(rtps[i].status == RTPS_CLOSED){
            pos = i;
            DBG_MSG("newRTPStream: first free: %i\n", pos);
            break;
        }
    }

    getClientPorts(data, &c_rtp_p, &c_rtcp_p);
    getFileName(data, fn);

    if(pos != −1){
        rtps[pos].status = RTPS_ACTIVE;
        rtps[pos].client_rtp_port = c_rtp_p;
        rtps[pos].client_rtcp_port = c_rtcp_p;
        rtps[pos].server_rtp_port = 7654; /* bogus value */
        rtps[pos].server_rtcp_port = 7655; /* bogus value */
        rtps[pos].icIdx = icIdx;
        strcpy(rtps[pos].filename, fn);
    }
    return pos;
}

/* Input: data: Pointer to RTSP SETUP message
     Output: c_rtp_p: Client RTP port
                  c_rtcp_p: Client RTCP port
*/
void getClientPorts(char *data, int *c_rtp_p, int *c_rtcp_p){
    char *sp, *ep;

    sp = strstr(data, "client_port=");
    sp+=12; /* Move pointer to beginning of port number */
    *c_rtp_p = strtol(sp, &ep, 10);
    *c_rtcp_p = strtol(ep+1, NULL, 10);
}

/* Input: data: Pointer to RTSP SETUP message
     Output: fn pointer to filename (\0 terminated)
*/
void getFileName(char *data, char* fn){
    char *sp, *ep;
    int len;

    sp = strstr(data, "9070/");
    sp+=5;
    ep = strstr(sp, "RTSP/1.0");
    ep−=1;
    len = ep−sp;
    strncpy(fn, sp, len);
    memset(fn+len, 0, 1); /* Zero−terminate string */
}

```

```
void createRtspReply(int ses, char *data, int type, char *reply){
   char *cseq;

   switch(type){

   case RTSP_SETUP: {
      cseq= strstr(data, "Cseq: ");
      cseq+= 6;
      sprintf(reply, "RTSP/1.0 200 OK\nCseq: %i\nSession: %i\nTransport: RTP/AVP;unicast;client_port=%i−%
                  (int)strtol(cseq, NULL, 10), ses, rtps[ses].client_rtp_port, rtps[ses].client_rtcp_port,
                  rtps[ses].server_rtp_port, rtps[ses].server_rtcp_port);

      break;
   }

      /* May supply more info here */
   case RTSP_PLAY: {
      cseq= strstr(data, "Cseq: ");
      cseq+= 6;
      sprintf(reply, "RTSP/1.0 200 OK\nCseq: %i", (int)strtol(cseq, NULL, 10));

      break;
   }

   default: break;

   }
}

/* Assumes there is room in data buffer for header
   Adds values in network byte order */
int addICheader(IC_session_t ics, char *data, int size, int ICtype, int routeType){
   ICHeader_t *ich;

   /* Make space for header */
   memmove(data + sizeof(ICHeader_t), data, size);
   ich = (ICHeader_t*)data;
   /* Zero IC header */
   memset(ich, 0, sizeof(ICHeader_t));

   ich−>type = ICtype;
```

```
    ich->ttl = 4;
    ich->dataLen = htons(size);
    if (routeType == R_EGRESS)
        ich->route = htonl(egressRoute(ics.route));
    else
        ich->route = htonl(ics.route);


    return size + sizeof(ICHeader_t);
}

void addICExtHeader(IC_session_t ics, char *data, int size){
    ICHeader_ext_t *iche;

    /* Make space for header */
    memmove(data + sizeof(ICHeader_t) + sizeof(ICHeader_ext_t), data+sizeof(ICHeader_t), size);
    iche = (ICHeader_ext_t*)(data + sizeof(ICHeader_t));
     /* Zero ICExt header */
    memset(iche, 0, sizeof(ICHeader_ext_t));
    iche->sport = htons(ics.sport);
    iche->dport = htons(ics.dport);
    iche->saddr = htonl(ics.saddr);
    iche->daddr = htonl(ics.daddr);
    memcpy(&iche->eth_src, &ics.eth_src, 6);
    memcpy(&iche->eth_dst, &ics.eth_dst, 6);
}

void sendToXScale(char *data, int size){

    pthread_mutex_lock( &socklock );
    errno = send(xscale_sock, data, size, 0);
    pthread_mutex_unlock( &socklock );
    if(errno == -1){
        DBG_MSG("send() returned error\n");
    }else{
        //DBG_MSG("Successfully sent %i bytes to socket\n", errno);
    }
}

/* Removes the outgoing port from a route to make
     the packet go to the XScale on the egress node
```

```
   to be transmitted with in the correct TCP stream */
uint32_t egressRoute(uint32_t route){
  uint8_t *rp;
  int i;

  rp = (uint8_t*)(&route);
  for(i=3; i>=0; i--){
    if(rp[i]!=0xff && rp[i]==0){
      rp[i]=0xff;
      break;
    }
  }
  return route;
}

/* Input: pointer to rtsp packet
   Output: Session number for This RTP stream. */
int getRtspSession(char *data){
  int ses;
  char *sp;
  sp = strstr(data, "Session: ");
  sp+=9; /* Move pointer to beginning of session number */
  ses = strtol(sp, NULL, 10);

  return ses;
}

/* Calculates the 16 bit ones-complement sum of a given
   buffer. Pads the last byte with 0 if odd size
   Input: data: Pointer to buffer start
          len: length of data */
void partial_csum(uint32_t *sum, void* data, int len) {
  int i;

  for(i = len>>1; i>0; i--){
    *sum += *((uint16_t*)data);
    data += sizeof(uint16_t);
    /* Add carries */
    *sum = (*sum >> 16) + (*sum & 0xffff);
    *sum += *sum >> 16;
  }

}

void udp_checksum(void* ipHdrStart, int size){
  uint32_t sum = 0;
  uint16_t finalsum;
  pseudo_udp ph;

  /* Construct pseudo header */
  memcpy(&ph.ipsrc, ipHdrStart+12, 4);
  //ph.ipsrc = ntohl(ph.ipsrc);
  memcpy(&ph.ipdst, ipHdrStart+16, 4);
  //ph.ipdst = ntohl(ph.ipdst);
  ph.notused = 0;
  ph.proto = 0x11;
  ph.len = htons(size);

  /* Calculate checksum of pseudo header */
  partial_csum(&sum, &ph, 12);
  /* Zero checksum field */
  memset(ipHdrStart+26, 0, 2);
  /* Calculate checksum of udp header and data*/
  partial_csum(&sum, ipHdrStart+20, size);

  /* Write the one's complement of the sum to
        the correct spot in the TCP header */
  finalsum = ~sum;
  memcpy(ipHdrStart+26, &finalsum, 2);
}

void ip_checksum(void* ipHdrStart){
  uint32_t sum=0;
  uint16_t finalsum;

  memset(ipHdrStart+10, 0, 2);
  partial_csum(&sum, ipHdrStart, 20);

  finalsum = ~sum;
  memcpy(ipHdrStart+10, &finalsum, 2);
}
```

A.3.5 ic.h

```
#ifndef _IC_H
#define _IC_H

#include <inttypes.h>

#define MAX_IC_SESSIONS 100

enum {
    ICH_FIND_FILE = 1, /* Packets used to locate file */
    ICH_ROUTE_FB, /* Used to setup egress IC sessions */
    ICH_CTRL_MSG, /* RTSP messages (except DESC and SETUP */
    ICH_CTRL_FB, /* RTSP replies */
    ICH_RTP /* RTP data Packets */
};

/* Standard IC header:
     Common for all IC packets. */
typedef struct{
    uint8_t           type; /* Type of IC packet */
    uint8_t           ttl;  /* To avoid circulating packets */
    uint16_t       dataLen; /* Length of IC packet data (excluding the header) */
    uint32_t       route; /* Routing information. Src routing */
} ICHeader_t __attribute__((packed));

/* IC header extension:
     Used to setup route information when assigning a new stream */
typedef struct{
    uint16_t          sport; /* Client source port */
    uint16_t          dport; /* Server port */

    uint32_t          saddr; /* Client source IP addr. */
    uint32_t          daddr; /* Egress IP addr */

    char            eth_src[6]; /* Ethernet src address */
    char            eth_dst[6]; /* Ethernet src address */
} ICHeader_ext_t __attribute__((packed));

enum {
    ICS_CLOSED = 0,
    ICS_WAIT_ROUTE,
    ICS_ACTIVE,
};

typedef struct{
    uint8_t         status;
    uint32_t        route; /* Routing information. Src routing */
    uint16_t            sport; /* Client source port */
    uint16_t            dport; /* Server port */

    uint32_t            saddr; /* Client source IP addr. */
    uint32_t            daddr; /* Egress IP addr */

    char              eth_src[6]; /* Ethernet src address */
    char              eth_dst[6]; /* Ethernet src address */
} IC_session_t;

/* Information about the IC sessions */
IC_session_t ics[MAX_IC_SESSIONS];

#endif /* #ifndef _IC_H */
```

A.3.6 rtp.h

```
#ifndef _RTP_H
#define _RTP_H

#include <inttypes.h>

#define RTP_VERSION 2
#define PT_MP2T                         33 /* MPEG2 TS A/V */
#define PT_MP2P                         567 /* MPEG2 PS A/V */

/* RTP data header */
typedef struct {
    uint16_t version:2;         /* protocol version */
    uint16_t p:1;                  /* padding flag */
    uint16_t x:1;                  /* header extension flag */
    uint16_t cc:4;                /* CSRC count */
    uint16_t m:1;                   /* marker bit */
    uint16_t  pt:7;                    /* payload  type  */
    uint16_t  seq;                          /* sequence  number  */
    uint32_t  ts;                           /* timestamp  */
    uint32_t  ssrc;                        /* synchronization  source  */
    //u_int32  csrc[1];                   /* optional  CSRC  list  */
}  rtp_hdr_t  __attribute((packed));

#endif  /* #ifndef  _RTP_H  */
```

A.3.7 dbg_msg.h

```
#ifndef  _DBG_MSG_H
#define  _DBG_MSG_H

#define  DEBUG

#include  <stdio.h>

#ifdef  DEBUG
#define     DBG_MSG(str,args...)                          printf(str,##args)
#else
#define  DBG_MSG(str,  args...)
#endif

#endif  /* #ifndef  _DBG_MSG_H  */
```