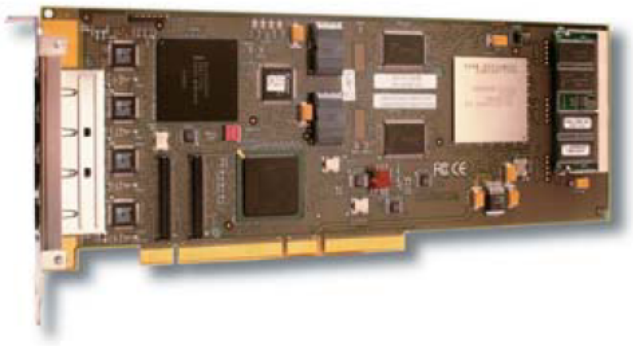
University of Oslo
Department of Informatics



Offloading Multimedia Proxies using Network Processors

Cand. Scient thesis
Øyvind Hvamstad

Published August 2004

Offloading Multimedia Proxies using Network Processors

Øyvind Hvamstad

Published August 2004

Abstract

A Multimedia Proxy aims to reduce the client startup latency, network load and server load. Such a proxy may be subject to many concurrent clients and experience high processing loads due to for example transcoding or protocol translation. At the same time a high load can be experienced when fetching data from the server. In this thesis, we will explore how to offload a multimedia streaming proxy by using network processing technology. We design, implement and evaluate a proxy prototype on the IXP1200 network processor. As a proof-of-concept we show that the prototype sucessfully offloads the proxy host in the data-plane, i.e., no data packets are processed by the host CPU, leaving it free to perform other CPU intensitive tasks. The prototype is able to do application layer forwarding using approximately a thenth of the cycles compared to a traditional architecture, where all pakcets are processed by the host CPU.

Table of Contents

1. Introduction 1
  1.1. Motivation and background 1
  1.2. Problem definition 2
  1.3. Outline of document 3
I. Background 5
  2. Distributed multimedia systems 7
    2.1. General 7
      2.1.1. Classification 7
        2.1.1.1. Taxonomy of applications 7
        2.1.1.2. Synchronicity 8
        2.1.1.3. Interaction 9
        2.1.1.4. Classifying MoD 9
    2.2. Requirements 9
    2.3. MoD access patterns 11
    2.4. Distribution architectures 11
    2.5. Protocol overview 13
      2.5.1. Real Time Streaming Protocol 13
      2.5.2. Real-Time Transport Protocol 15
      2.5.3. Session Description Protocol 15
    2.6. Summary & challenges 15
  3. Proxies 17
    3.1. General 17
    3.2. MoD proxy caches 18
      3.2.1. Properties 19
      3.2.2. Caching strategies 20
    3.3. Cache architectures 22
    3.4. Data flow 23
      3.4.1. Copy operations 24
      3.4.2. The Linux network stack 24
    3.5. Summary 25
  4. Network processors 27
    4.1. General 27
      4.1.1. NIC evolution 30
    4.2. Intel Internet Exchange Architecture 31
    4.3. Intel IXP1200 network processor 32
      4.3.1. The StrongARM core CPU 33
      4.3.2. Microengines 33
      4.3.3. The FBI unit 35
      4.3.4. Memory interfaces 35
    4.4. IXA application development 36
    4.5. ACE run-time framework 39
    4.6. Related work 40
      4.6.1. Network layer 40
      4.6.2. Multimedia streaming 41
    4.7. Summary 41
II. Design, implementation and evaluation 43
  5. Design and implementation 45
    5.1. Design 45
      5.1.1. Design goals 45
      5.1.2. Component overview 45
      5.1.3. Caching 46
      5.1.4. Fast forwarding 47
      5.1.5. Summary 47

5.2. Implementation 47
5.2.1. Limitations and assumptions 47
5.2.2. Overview 48
5.2.3. The RTSP components 49
5.2.3.1. Conformance issues 51
5.2.4. Session management 52
5.2.5. Cache management 52
5.2.6. Packet classification 53
5.2.6.1. Classification rules 53
5.2.6.2. Forwarding table 55
5.2.7. RTP forwarding 58
5.2.8. Summary 59
6. Experiments, results and analysis 61
6.1. Testing environment 61
6.2. Measurements 62
6.3. Results 63
6.3.1. Packet types 64
6.3.2. Ingress to egress 64
6.3.3. Deviation 66
6.3.4. Processing overhead 68
6.3.5. Summary of results 69
6.4. The offloading effect 69
6.5. Prototype performance 71
6.6. Extensions 72
6.6.1. Caching 72
6.6.2. Zero-copy 74
6.7. Summary 75
7. Conclusions and future work 77
7.1. Conclusions 77
7.2. Future work 77
References 79
A. Guide to the attached CD 83
A.1. Directory structure 83
A.2. Building the source 83
A.3. Running the executables 83

List of Figures

2.1. Taxonomy of applications 8
2.2. Distribution architectures 12
2.3. The OSI stack and the Internet protocol stack 13
3.1. Basic proxy architecture 17
3.2. Cache architecture 22
3.3. Proxy data flow 23
3.4. Proxy data-path 24
3.5. Linux 2.0.34 network stack 25
4.1. Cost vs. performance 30
4.2. NPU market shares 30
4.3. Three generations of NICs 31
4.4. The Internet Exchange Architecture 32
4.5. IXP1200 block diagram 33
4.6. Command queues 35
4.7. ACE processing pipeline 37
4.8. Core and microblock components of an ACE processing pipeline 38
4.9. ACE intercommunication 40
5.1. Proxy components 46
5.2. ACE layout 48
5.3. RTSP proxy session 50
5.4. Packet classification 53
5.5. Media Transport Forwarding table 57
6.1. Test configuration 61
6.2. Experiment 1: Cycle probes for packet reception, forwarding and enqueuing 63
6.3. Experiment 2: Cycle probes for packet forward processing 63
6.4. Experiment 3: Cycle probe for packet forward processing and enqueuing 63
6.5. Cycles counted during packet reception, forwarding and enqueuing 65
6.6. Packets 500 through 700 experiment 1 66
6.7. Cycles counted during packet forwarding 67
6.8. Packets 500 through 700 from experiment 2a) 67
6.9. Lazy copy data-structure 74
6.10. Zero copy data-path vs. traditional data-path 75

List of Tables

4.1. Processor hierarchy 29
4.2. IXP1200 memory interfaces 36
6.1. Packet count during session 64
6.2. Ingress to egress statistics 65
6.3. Forwarding statistics 68
6.4. Experienced overhead 69

List of Examples

2.1. Audio data rate and disk usage 9
2.2. DVD data rate and disk usage 10
2.3. DivX data rate and disk usage 10
2.4. Basic RTSP session 14
5.1. RTSP request type 49
5.2. RTSP response type 50
5.3. Session object 52
5.4. Proxy session object 52
5.5. Simple branch classification of Ethernet type 54
5.6. IP address validation and transport protocol classification 55
5.7. Hash function 56
5.8. Incremental checksumming 58
6.1. Microblock probes 62
6.2. Pseudo code for packet copy 73

Chapter 1.

Introduction

1.1. Motivation and background

> People are always blaming circumstances for what they are. I don't believe in circumstances. The people who get on in this world are the people who get up and look for the circumstances they want and if they can't find them, make them.
>
> —George Bernard Shaw

The Internet is growing at an exponential rate. The increasing availability of low-cost bandwidth for regular users enables new applications that have different requirements than the traditional applications. Streaming audio and video, Voice over IP (VoIP), Peer-to-Peer (P2P) networking, and Virtual Private Networks (VPNs) require more bandwidth and computation power than the retrieval of textual data. The ability to download audio and video content is quite common today. However, continuous playback of high-quality video often involves downloading large parts of or the whole object before viewing it. Streaming is to some extent possible, but usually the quality of such streams is low. Increased quality will be available as the bandwidth capabilities evolve in the Internet.

Nevertheless, large-scale deployment of *Media-on-Demand* (MoD) streaming applications in the Internet will introduce challenges. The philosophy on which the *Internet Protocol* (IP) is built, is loose coupling between heterogeneous networks along with end-to-end responsibility [Clark88]. These basic properties complicate the process of exercising *Quality Of Service* (QOS). Most, if not all, of the schemes proposed to guarantee some level of QOS are facing problems with scalability, fulfilling the actual level of guarantee or failing to achieve linear deployability [Xiao99].

The lack of a widely spread framework for handling resource reservation restrains the deployability of MoD. Such applications may require soft real-time guarantees, but will have to run on IP shared mediums without such support. Network effects such as latency and jitter along with possible congestion, contention and physical failures, will therefore reduce the perceived quality of the media and the system in general.

To circumvent the lack of QOS support in the Internet, pragmatic methods are used. Over-provisioning is commonly used by *Internet Service Providers* (ISPs). This basically means that extra resources are put into the network, enabling it to scale. For example, if the network load exceeds a certain percentage, the links are upgraded so that the level of redundant bandwidth is preserved. Further, if a server is experiencing high loads it is common to upgrade it with better hardware, or to put it in a load-balancing cluster, so that it may continue scaling to the number of concurrent clients.

Another pragmatic method is to reduce the resources needed to serve a client. This may involve reviewing and optimizing application protocols, improving the performance of servers [Halvorsen01], or the invention of new technologies that allow resources to be used differently. These efforts are less common. However, they are necessary, since over-provisioning cannot solve all problems related to scalability. Also, using over-provisioning as an indefinite measure might prove impractical and expensive.

A third way is to distribute the server tasks to an infrastructure of proxies. This has been discussed as a schema to help realize MoD in the Internet. Using proxies is also suggested in distributed gaming research. The proxies perform tasks on behalf of the server, e.g., by caching relevant data, shaping and adapting network traffic, etc. Serving clients from the proxy instead of directing them all the way to the server, reduces network load and latency.

A proxy cache, which sits somewhere between the server and the client, may experience a considerable amount of network traffic on both its up and down link. The proxy is therefore subject to the same challenges as the server. It is crucial that the proxy is able to efficiently process information without further adding to the delay. The faster a proxy can process a packet, the better throughput it will achieve. High throughput will allow the proxy to scale to a potentially high number of concurrent clients.

During the last years, a new generation of *Network Interface Cards* (NICs) has evolved. These have onboard *Network Processing Units* (NPUs), with programmable processing power and hardware specialized for network processing. The aim of NPUs is to provide flexibility and speed so that future and present challenges in network development can be met.

This thesis will investigate possible solutions to application specific challenges regarding multimedia proxy nodes by using an NPU, the IXP1200. We will focus on how to offload the proxy and achieving good throughput.

1.2. Problem definition

Multimedia proxies aim to reduce startup latency, network load and server load. An MoD proxy does this mainly by the means of caching. It may, in addition, perform other supplementary tasks to further reduce the network load. For example, it may transcode streams on the fly, translate the protocols used for transport and re-encode a stream to a different format.

As streaming media has relatively high resource demands, and when a proxy must serve many concurrent streams at different rates, a careful allocation of resources is needed to support such a load. The load on the proxy increases further, as transcoding, re-encoding and intelligent caching itself may be highly CPU intensive. We believe that by utilizing the capabilities of NICs with integrated, specialized NPUs we can offload the data-plane network processing to free resources on the proxy host. Further, we believe that an NPU can be efficient when processing the network traffic received at an MoD proxy. Two subjects draw our attention:

1. The caching strategies in multimedia take into account that the size of media objects tend to be too large to be cached in their entirety, unless they are extremely popular. The segments of media objects that are not cached must therefore be fetched from a server and then forwarded through the proxy. We specifically believe that the processing of this network traffic can be offloaded onto an NPU.

2. Network processing in a traditional proxy architecture is slow. *Operating Systems* (OS) provide generalized abstractions and security features, such as a network stack and memory protection boundaries. These features require many operations to be performed in order to process a network packet in the application layer. We believe that an NPU can be used to optimize the network processing needed in an MoD proxy.

To illustrate the feasibility and expediency of an offloaded proxy, we will in this thesis design and implement an MoD proxy prototype on a NIC with an Intel IXP1200 chip [Intel01b]. The implementation will basically consist of two parts, an RTSP proxy and an RTP forwarder. The RTP forwarder will be a proof-of-concept to show application layer forwarding of network traffic. At the same time, the prototype will be the basis of a framework for further work in the area.

1.3. Outline of document

The rest of this document is divided into two parts; Part I, “Background” will describes the theoretical and technological foundation upon which this thesis is built. This is done in Chapter 2-4. Part II, “Design, implementation and evaluation” will describe the contributions made in this thesis in Chapter 5-7.

Chapter 2, *Distributed multimedia systems*. This chapter gives an overview of multimedia applications and essential background on distributed multimedia systems.

Chapter 3, *Proxies*. Here proxies are presented for distributed systems in general and for MoD systems in particular. The properties of an MoD architecture employing proxies are given and different system architectures are presented along with MoD caching strategies.

Chapter 4, *Network processors*. In this chapter a general introduction to NPUs are given before details about the IXP1200 and the Intel's IXA development platform are presented.

Chapter 5, *Design and implementation*. This chapter describes the design considerations and the implementation details of our prototype MoD proxy. It covers the components needed to realize an MoD proxy and the details of how the current implementation works.

Chapter 6, *Experiments, results and analysis*. In this chapter we will give a precise explanation of how we conducted our experiments and what results they produced. Then, the results and contributions are analyzed with respect to our goals and background material.

Chapter 7, *Conclusions and future work*. In the final chapter we summarize our work and describe in what areas we should focus future efforts.

Part I. Background

Chapter 2.

Distributed multimedia systems

> An image, a sound, they are one in the same, just one likes to move and one stays the same.
>
> —Karate

Multimedia allows information to be presented in new ways. It has made its way into regular people's lives and is here to stay. Multimedia distribution across networks is however in its infancy. Several problems must be solved before high quality multimedia can be accessible in large scale packet switched networks. This chapter introduces multimedia. The subjects are: application characteristics, requirements, distribution and protocols.

2.1. General

Multimedia is, in simple terms, the combination of two or more media in relative synchronization. It enables human interaction and presentation of information. Any data consisting of a combination of text, sound, pictures, animation and video is often regarded as multimedia data. Synchronizing the media involves preserving their relationship in time. A movie combining the media of video and sound will synchronize the voices of the actors with their facial expressions. Even old silent movies combine two media, text and motion pictures, though in this case there is no synchronization.

A multimedia system is any combination of computer components that have the ability to present, create or deliver multimedia data. These components may be networks, operating systems and other software. It is the availability of resources in these components that determine the perceived quality of the multimedia. If one component fails to achieve the required performance, it becomes the *bottleneck* of the system, and no matter how well the other components perform, the end result is influenced by this component.

Film is just one real-life application. The range of common multimedia applications also includes video conferencing, gaming, learning software, and reference material such as encyclopedias.

2.1.1. Classification

The multimedia applications above may be classified using several different characteristics. Schemas are available that classify all applications or some subset of them. They are tools that ease the mapping of what requirements the application at hand has.

2.1.1.1. Taxonomy of applications

One classification schema, shown in Figure 2.1[Peterson00], regards all applications as either *elastic* or *real-time*.

The elastic classes of applications has no time bound for the delivery of data. They are however, usu-

ally not loss tolerant, i.e., they require reliable data transfer. Applications that fall into this category are typically text based chat, file transfers and email. These applications can further be classified as respectively interactive, interactive bulk and asynchronous. These applications do not depend on when the data arrives, as long as it arrives. They could benefit from timely delivery of data, but this is no requirement.

Real-time applications do, as opposed to elastic applications, require timely delivery of data. This means that if the data is not received within some time-frame, it is rendered less usable for the application, or not usable at all. For example, continuous media playback is affected by variations in delivery of data. A user will experience much of this jitter as low quality, due to skips in the playback. Other applications have heavier demands on timely delivery, such as industrial assembly lines with synchronization between robot arms. If a message arrives late to one of the robot arms the system will be out of sync.

Real-time applications can further be classified according to their loss tolerance. The assembly line application is an *intolerant* application as loss of data is unacceptable and would wreak havoc on the assembly line. The voice playback application however, allows occasional data loss, and is thus a *tolerant* application.

Another characteristic of real-time applications is their adaptive properties. The intolerant applications can adapt to the rate of data or not adapt at all. Tolerant applications on the other hand may, apart of being *non-adaptive*, adapt to both rate and delay. Video and sound playback may be adaptive applications. Voice is adaptive to changes in delay, i.e., *delay-adaptive*. This means that when playing back speech, the silence between words in a sentence can be shortened or lengthened to some extent, without being noticeable. Many audio encodings can also lower or vary the bit-rate of a recording or a stream by removing frequencies that cannot be heard. Video playback has the same characteristics as audio. The delay between frames can be varied, either by quickening the frame rate or slowing it down. There are also video encodings that can reduce the bit-rate by reducing the quality of each frame. Video and audio are therefore both *delay-adaptive* and *rate-adaptive*.

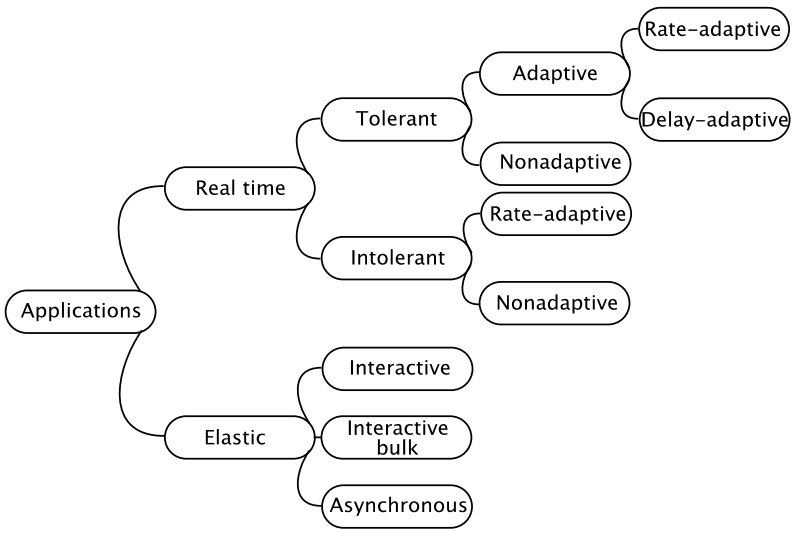


Figure 2.1. Taxonomy of applications

2.1.1.2. Synchronicity

Distributed multimedia applications can, in addition to Peterson's taxonomy, be either synchronous or asynchronous[1]. The distinction is here how the two endpoints communicate. In synchronous communication both parties are using the application simultaneously. Though they may be separated in

[1]*The terms synchronous* and *asynchronous* are used in many research areas within computer science. Such as in programming, I/O systems and in communication.

space, they must both be present at the same time, e.g., in a video conference, when using VoIP or when participating in a real-time game. Asynchronous communication, on the other hand, is indirect and the parties may be separated in both time and space; it allows exchange of information, without the need for every involved user being present. The data is typically stored on a server so that the receiving part may fetch it at a later point in time, e.g., MoD.

2.1.1.3. Interaction

Interaction is a characteristic that further differentiates multimedia applications. In reference material, such as Web pages, it is common to use hyper linking to navigate and switch between different media. Other means of interaction are possible, such as voice control and eye movement. In playback applications interaction, if present, involves navigating back and fourth, pause and stop. Such interactive features are called VCR operations. The number of interactive features, or the level of interaction separates these applications into different classes [Buddhikot98].

2.1.1.4. Classifying MoD

MoD applications, such as *News-on-Demand* (NoD), *Video-on-Demand* (VoD) and *Learning-on-Demand* (LoD), are a set of multimedia streaming applications. MoD is, with respect to the two classification schemes described, a real-time, tolerant, adaptive and asynchronous application. The user expects continuous playback of the media. This requires timely data delivery, which make MoD a real-time application. It is tolerant to the occasional loss of data, although playback quality will be reduced. Depending on the nature of the encoding, a media stream may also reduce the rate at which data is sent. This is rate-adaptive behavior. An MoD application is asynchronous due to the fact that the distributor may upload a media file to a server. This makes the media available for several users at any time.

The classification of a multimedia application will, among other things, dictate the requirements of how it is handled by a multimedia system.

2.2. Requirements

The classification of a multimedia application gives a high-level understanding of what is required by a multimedia system. As most multimedia applications are real time applications they have implicit demands of timely delivery. They usually allow some loss of data and have adaptive properties. If they have interactive characteristics, further demands are put on the system.

Nevertheless, compared to traditional textual data, the digitalized representation of video and sound require more storage space and bandwidth, due to higher data rates. This is because the information needed to encode light and sound waves in a rich way is a lot more than for example the eight bits usually needed to represent a common character. More bits are therefore required to store media files. This remains a fact even though compression is widely used. How many bytes it takes to store a media object depends on the desired quality, the chosen encoding algorithm and the playback length., e.g., see Example 2.1, Example 2.2 and Example 2.3.

Example 2.1. Audio data rate and disk usage

> Audio in pulse code modulation coded *Compact Disk* (CD) quality uses 16 bit samples at 44.1 kHz with two stereo channels [Steinmetz95]. This gives a bandwidth requirement of 1.35 Mbps and a storage requirement of 607.50 MB for one hour of audio.

The bit-rate of media objects requires large disks, but it also requires that the communication system

is able to handle these rates. This includes every component that participate in the communication; the operating system, the entire network, every router and the server and client application.

Example 2.2. DVD data rate and disk usage

A *Digital Versatile Disc* (DVD) of the current standard, using the *Moving Picture Expert Group* (MPEG) codec version 2 has, for video, the maximum video bit rate of 9.8. The average video bit rate is about 3.5 Mbps, but this is dependent on length, quality, amount of audio, etc. After system overhead, the maximum rate of combined elementary streams (audio + video + sub picture) is 10.08 Mbps [Taylor04]. The size of a one hour long video is then 1.575 GB using average bit rate.

The continuous and real-time characteristics of films, speech, music and multimedia in general have other requirements besides storage space and bandwidth. The amount of delay and jitter must be kept as low as possible throughout the system. For example, if any component fails to process video or audio data in time, glitches in the presentation will be the result. The viewer might as a consequence give up or gradually lose interest. Further, if a video is to be experienced as flawless, there can be no more than ±80 ms of skew in lip synchronization. Even higher demands on jitter occurrence are present when two stereo audio streams are played back. A person will detect ±11 µs in this setting, so hardly any jitter at all will ruin the playback [Steinmetz95].

Example 2.3. DivX data rate and disk usage

DivX is a codec derived from the MPEG-4 standard. It has become a very popular codec for exchanging movies over the Internet [Zimmerman03]. DivX is able to reduce the size of an existing MPEG-2 encoded file 6.5 times, e.g., the required playback bandwidth drops from 5.6 Mbps to 870 Kbps. The disk usage is reduced from 423 MB to 63.8 MB. A one hour long video at 870 Kbps requires 393 MB worth of storage.

A real-time system guarantees that deadlines are met while playing back multimedia or other content with real-time requirements. Such systems guarantee QoS. QoS is an abstract term that can be used on many levels. In general, it can be defined in two parts:

1. A set of qualitative and/or quantitative attributes that specify the requirements of an application.
2. A mechanism that ensures that the specified requirements are met by a system running the application.

A user may typically specify a high abstraction of what quality he/she wishes to receive. For example, low, medium or high quality. On a lower level this specification is translated to a more fine-grained set of quantitative attributes, so the system may decide if it is able to meet the requirement or not, e.g., bandwidth, startup latency, jitter and correctness. In a real-time system, the system grants access and then ensures that the requirements are met, if it is able to satisfy them at the time. QoS is therefore also regarded as a contract between the user and the system. The user specifies what he or she wants, and the system provides it. However, real-time systems are rare, and no such support is available in the Internet, unless dedicated resources are set up. However a QoS agreement or contract needs not regard hard deadlines, or require reserved resources. A QoS contract may simply be that certain applications are prioritized. This basically means that throughout the system, applications are handled according to their priority. For example, events, actions and messages from a high prioritized applica-

tion will be handled by the system before those with a lower priority. Thus, the QoS contract is met if the priority scheme works. In the Internet this contract is *best-effort*, i.e., you get what is available, but without any guarantees.

Multimedia systems will need QoS mechanisms to fully support the requirements of multimedia applications. Since such support is inexistent in the core of packet switched networks, other measures have to be considered to improve a multimedia system's ability to provide satisfactory quality when streaming media.

To lower the bandwidth requirement in a system, the data-rate of video streams may be adapted. This further allows heterogeneous networks with different bandwidth capabilities to receive streams of different data-rates. When adapting to jitter, certain codeces, such as MPEG, allow for parts in the video to be discarded. Also, to reduce the effect of jitter, playback buffers are used at the client side so that jitter from the network is masked away. However, this fails if a packet arrives after the playback buffer is empty.

2.3. MoD access patterns

Requests for media objects tend to have certain characteristics. The frequency of access is said to follow the Zipf distribution [Sitaram00]. This means that only a few of the available objects are requested by the majority of the users. As a rule of thumb 10% of the objects are requested 90% of the time. Compared to other web content this is a clear characteristic of media object access, as web content of medium or low popularity usually include 50% of all material accessed [Wang99]. Newly published media objects tend to be the most popular objects, e.g., the top ten most popular videos in rental shops tend to be the newest ones. The access frequency of an object is often reduced over time.

When requested, media objects are usually consumed from start to end. If they are not fully consumed, they are usually canceled at an early stage [Acharya00]. Objects are seldomly accessed by the same person repeatedly.

The access to media objects tend to be “write-once-read-many”, which means that the content rarely changes after being published. Users usually only have read-only interaction, much the same as a video player or music player.

The observed access patterns of media objects may be used to optimize the systems that deliver them. They also have an effect on what architectural choices that are made to build multimedia systems.

2.4. Distribution architectures

A distributed multimedia system can be built with a client, a network and a server [Sitaram00]. The architectural choices of a distributed architecture may have a large impact on how well the system works as there are different pros and cons associated with each architecture. There are three main distribution architectures; *hierarchical*, *distributed* and *hybrid* architectures.

The classic architecture is the hierarchical. It is called the *client-server* architecture and is shown in Figure 2.2 a). This is the model that traditionally has been used in the Internet. It is a simple scheme, where full control is held by the centralized server. The drawbacks of a pure client-server architecture is that the server represents a single point of failure. That is, if the server fails during execution the whole system breaks down. Another drawback is that the server is a potential bottleneck in the system, as all requests must be handled by it. If load exceeds the capacity of the server, then the rest of the system will suffer equally and the service will be degraded. Further, as the central server's geographical location potentially may be far away from the clients, the latency may be high. This will result in clients having to wait a long time before receiving any response from the server. Also, as long distances often imply that every packet must be routed through many nodes, each packet's total time in between the server and the client has a higher probability to vary (jitter).

The benefit of the client-server architectures is, as mentioned, that the server has control of the distribution of information to the clients. It is easy to administer such a scenario, both when publishing new content, charging for use and updating software. The main reason why it is so, is that data does not need to be replicated. Everything is centralized. Charging for services is easily managed by a centralized server and it is also simple to manage any state information as it needs not be replicated. Having a single copy of every accessible object also reduces the impact on the system when an object changes.

The opposite of the client-server model is a distributed architecture called the *peer-to-peer* model, see Figure 2.2 b). This is a collaborative model where every node is connected to every other node, and they all have the same amount of control, i.e., they are "peers". The P2P architecture has the advantage that it makes use of the resources that are available on the connected nodes. This distribution of computation removes the bottleneck represented by the server in a client-server architecture. The P2P architecture does not have a single point of failure, as every node has the same amount of control. If some nodes fail, the rest will still be able to function without them.

The P2P model is more complex than the client-server model, and no simple access control may be implemented as control is shared by all nodes. This makes charging for the use of services harder. Resources will typically be replicated across several nodes for reliability, making consistent updates of changes harder. Another issue with the complexity of P2P architectures is the need for a substantial amount of communication control messages between the peers. These messages are used to exchange synchronization and meta data. This may flood the network and lead to scalability problems.

The *hybrid* architecture is literally a hybrid of the hierarchical, i.e., client-sever and the distributed model, i.e., the P2P model. It incorporates characteristics from both, by for instance having a centralized meta-data server, while storage is distributed. Many variations can be implemented and these architectures may remove some of the drawbacks while keeping most benefits of an architecture.

Existing video and audio streaming solutions tend to choose the client-server architecture[2]. The client-server architecture has also been the preferred architecture for commercial game vendors because of the benefits described above, but it is suggested that a hybrid architecture might be better suited [Mauve02], while still preserving the needs of commercial vendors.

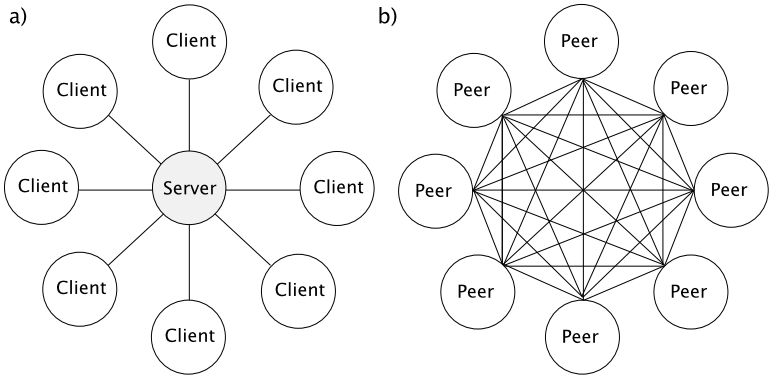


Figure 2.2. Distribution architectures

Regardless of the distribution architecture, there are more ways of distributing the packets that flow from a server or in between peers. In the Internet there are three ways to do this. Packets may be sent to everybody with *broadcast*, to a set or a group with *multicast* or to one receiver with *unicast*.

Television uses broadcasting networks to distribute their media. This means that everyone that listens on a certain channel, will receive what is on at that particular moment. MoD, however, must be able to serve a client whenever it requests to view a certain media object. Therefore, broadcasting is a less suitable distribution mechanism. Many variations of broadcasting have been created to adapt to an

[2]Streaming solutions such as the Darwin Streaming Server [Apple04], Helix DNA Server [Real04a] and KOMSSYS [Griwodz03] all use a hierarchical architecture.

MoD scenario, but have been unable to provide *true-MoD*, i.e., that a user can request a media object at any time.

Most applications in the Internet usually use *unicast* communication. This means that there is only one source and one destination participating in the data transfer. If there are more than one receiver, one copy must be sent to each of them. This is a scalability problem as each unique packet sent to the network must be copied to a multiple number of receivers.

To solve this *multicast* was purposed. Multicast may also be called group-cast, as clients must join a group in order to receive the information that is sent out. In contrary to unicast, only one packet is sent out from a sender and it propagates the network as one copy. Along the network path, the packets will be duplicated if there are clients that have joined from different directions. While multicast is supported by the IP, it is usually blocked by *Internet Service Providers* (ISPs). This may be because multicast makes it hard to account for network usage and it is is vulnerable to *Distributed Denial of Service* (DDoS) attacks.

2.5. Protocol overview

Protocols define the format and order of messages exchanged between two or more communicating entities, as well as actions taken on the transmission and reception of messages or other events [Kurose04]. A multimedia streaming system makes extensive use of protocols. Protocols are needed to control interaction, setup and transfer the data. Protocols also exist to retrieve metadata about streaming sessions, e.g., the *Session Description Protocol* (SDP) [Handley98].

Figure 2.3 shows the *Open Systems Interconnect* (OSI) model alongside the Internet TCP/IP protocol stack [Socolofsky91]. Multimedia applications may employ the whole range of protocols, depending on the characteristics of the application.

The protocol pair that is commonly used for streaming is the combination of the *Real Time Streaming Protocol* (RTSP) [Schulzrinne98] and the *Real-time Protocol* (RTP) [Schulzrinne96]. Both these are standards defined by IETF and are widely supported by streaming solutions. In this section, we describe those that are relevant for this thesis.

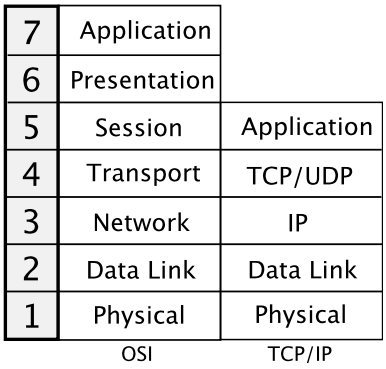


Figure 2.3. The OSI stack and the Internet protocol stack

2.5.1. Real Time Streaming Protocol

RTSP [Schulzrinne98] is an application-level control protocol for streaming. It resembles and has overlapping features with the *Hypertext Transfer Protocol* (HTTP) version 1.1 [Fielding99].

The main purpose of RTSP is to establish and control streams of continuous media. Usually, the actual data-transfer is performed out-of-band using RTP, but RTSP is not bound to any specific transport protocol. Data may even be interleaved within RTSP messages. To control a session, RTSP maintains

session state for each stream.

RTSP supports both multicast and unicast streaming. This may be both playback or recording of such, but the standard does not require an implementation to support recording.

To establish and control streams several methods are defined in the standard. Those *required* (OPTIONS, SETUP, PLAY and TEARDOWN) by an implementation are shown in Example 2.4.

Other methods are defined as well, such as DESCRIBE, PAUSE, ANNOUNCE, GET_PARAMETER, RECORD, REDIRECT and SET_PARAMETER. Not all of these methods change the state of a RTSP session, only SETUP, PLAY, RECORD, PAUSE and TEARDOWN do. SETUP starts a session and allocates resources for a stream. PLAY and RECORD start data transmission on a stream that has been set up. PAUSE temporarily halts a stream without releasing its resources. Finally, TEARDOWN will halt the stream and free the resources allocated by the session.

The first thing that happens when a client initiates a session, is that it sends a SETUP request to the server. The central part of this request is the transport specification. In Example 2.4, the client tells the server that it wants to use RTP unicast to transport the data out of band on port 1004. The stream could also be sent over TCP, interleaved in RTSP messages or to a multicast group. The server replies with a status message verifying the request of the client. It also adds the server port pair, where it enables data to be received. Note that the first number in the client and the server port pair is for data retrieval and the second is for any control data on the stream. The data port always is even, and that the control port is always odd, and one higher than the data port. The server has now set up a session on its side and the session number can be used to identify subsequent RTSP requests. The PLAY request from the client therefore adds this number to the session header. The client wants to play the whole length of the stream, so it sends a range header indicating this. The server sends a status message, and the streaming starts on the channel setup by the transport specification. When the object is played back, the client sends a TEARDOWN request and the session is removed.

Example 2.4. Basic RTSP session

```
C->S  SETUP rtsp://10.0.0.2:554/ex.mpeg RTSP/1.0
      CSeq: 1
      Transport: RTP/AVP;unicast;client_port=1004-1005

S->C  RTSP/1.0 200 OK
      CSeq: 1
      Session: 8
      Transport: RTP/AVP;unicast;client_port=1004-1005;server_port=1008-1009

C->S  PLAY rtsp://10.0.0.2:554/ex.mpeg RTSP/1.0
      CSeq: 2
      Session: 8
      Range: npt=0-

S->C  RTSP/1.0 200 OK
      CSeq: 2
      Session: 8
      Date: 25 Feb 2004 18:07:02 GMT

C->S  TEARDOWN rtsp://10.0.0.2:554/ex.mpeg RTSP/1.0
      CSeq: 3
      Session: 8

S->C  RTSP/1.0 200 OK
      CSeq: 3
```

2.5.2. Real-Time Transport Protocol

RTP [Schulzrinne96] is the Internet-standard protocol for the transport of real-time data, including audio and video. It can be used for media-on-demand as well as interactive services such as Internet telephony, real-time control and distributed simulation. RTP consists of a data part and a control part. The latter is called the *RTP Control Protocol* (RTCP).

The data part of RTP is a thin protocol providing support for applications with real-time properties such as continuous media, e.g., audio and video, including timing reconstruction, loss detection, security and content identification. It also features negotiation of what encoding to use.

RTCP provides support for real-time conferencing of groups of any size within the Internet. This support includes source identification and support for gateways like audio and video bridges as well as multicast-to-unicast translators. It offers QoS feedback from receivers to the multicast group as well as support for the synchronization of different media streams.

While UDP/IP is RTPs initial target networking environment, efforts have been made to make RTP transport-independent so that it could be used, say, over TCP, Novells IPX protocol or other protocols. RTP does not address the issue of resource reservation or QoS control; instead, it relies on resource reservation protocols such as RSVP.

2.5.3. Session Description Protocol

SDP describes multimedia sessions for the purpose of session announcement, session invitation and other forms of multimedia session initiation.

Session directories assist the advertisement of conference sessions and communicate the relevant conference setup information to prospective participants. SDP is designed to convey such information to recipients. SDP is purely a format for session description - it does not incorporate a transport protocol, and is intended to use different transport protocols as appropriate. Example transport protocols for SDP include the *Session Announcement Protocol* (SAP), *Session Initiation Protocol* (SIP), RTSP, electronic mail using the MIME extensions, and HTTP.

SDP is intended to be general purpose so that it can be used for other network environments and applications than just multicast session directories. However, it is not intended to support negotiation of session content or media encodings.

The SDP communicates the existence of a session and conveys sufficient information to enable participation in the session. In this thesis we omit the use of this protocol when implementing RTSP. The session information is fetched manually.

2.6. Summary & challenges

When running distributed multimedia applications in the Internet, the main problem is getting time sensitive data to its destination in the appropriate period of time. The challenges are especially demanding on the server-side and in the network. Streamed data needs a relatively continuous flow of information in order to maintain its integrity. There is no support for this in IP networks, without dedicating a route for a particular stream.

IP networks work well for streaming if every packet spends the same amount of time reaching their fellow destination. However, this only applies in a perfect network where no congestion occurs at any routers and where all packets follow the same route or a different route with the same latency. Congestion is a result of buffering within the router. The router must buffer packets if the capacity of one output is exceeded, i.e., too many of the incoming packets are destined to the same output port or if it experiences bursts on input ports. Packets must therefore wait in the buffer or they are dropped because the buffer fills up. Packets may also take different routes through IP networks, as routers may appear or disappear. Combining congestion with alternative routes results in packets arriving at their

destination out of order and the time it takes them to reach the destination may vary.

Another challenge is the latency itself. Synchronous streams in opposite directions, e.g., live conversation, must be transmitted through the net before the receiving side can reply. This will, if the latency is high, result in long gaps from when you end a sentence until you receive a reply. Also in asynchronous streaming, e.g., MoD, a client might have to wait quite a bit before the stream can start playback. This start-up delay is further lengthened if the client has to buffer data to alleviate jitter in the network.

Multimedia applications may also occupy a lot of bandwidth. The high bit-rate puts load on the network and makes multimedia servers highly I/O centric. On the server-side this represents a scalability problem as resources are bound to one stream for a long period of time.

In the next chapter, we describe a pragmatic solution suggested to improve this situation: the proxy.

Chapter 3.

Proxies

> Every peer... may make another lord of parliament his proxy, to vote for him in his absence.
>
> —Blackstone

Proxies are network nodes in the vicinity of clients, performing tasks on behalf of a server. They traditionally perform caching, to reduce latency for clients, reduce network load and offload servers. These nodes also have other uses depending on the context in which they run. For instance in an MoD setting, they may perform re-encoding, encryption and adaption of multimedia streams. They may also have general features such as access control, re-routing and simple QOS mechanisms.

3.1. General

A proxy is someone that acts on the behalf of others. In politics, a proxy can be given the authority to vote on the behalf of someone else. In computer science, the proxy may be a process or an object acting on behalf of a server process or a remote object[3]. Whether the proxy runs as a stand-alone process on a separate host or is embedded in a local process running on the client host, it is located closer to the client, along the network path, compared to the server process or remote object. This is shown in Figure 3.1. Every remote call the client makes is intercepted by the proxy so that certain operations can be performed. These operations vary as to which context the proxy runs in.

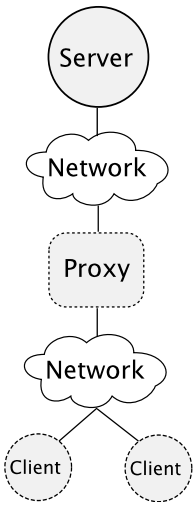


Figure 3.1. Basic proxy architecture

[3]To be precise, the role of a proxy is twofold. It may retrieve information on behalf of a client or deliver information on behalf of a server.

Proxies[4] were first introduced as a way for machines behind firewalls to access web content [Wang99]. This was achieved by placing a host machine on the outer rim of the network with access to the Internet. The observation was made that the users within such firewalled corporate networks would access much of the same content. By having the proxies cache text and images, the users would get faster access to it. The central server running the service would not have to process these requests and fewer packets would be sent to the outside. Today, it is fairly common for *Internet Service Provider* (ISP) and *Local Area Network* (LAN) administrators to deploy proxies to ensure quick response when users load web content, by caching images and text. In HTTP, a client may be configured to send every request via a proxy towards the server. The main purpose of the proxy is then basically to cache any content the client may want to receive and only re-retrieve it if it has been altered on the server. The proxy will then be able to serve the cached content on behalf of the server. If additional content is wanted, it will forward such requests to the server, and cache the response before sending it back to the client. An HTTP proxy may also log web activity and block certain content if so desired. The HTTP contains specific fields concerning proxy operations, such as forcing reload or specifying that the content cannot be cached due to dynamically changing content.

In MoD, proxies are used in the same way as in web caching. However, larger objects, different access patterns and application requirements are characteristics that must be considered. These characteristics are explained in Section 3.2.

Proxies are also present in distributed gaming research. A Generic Proxy System for Networked Games [Mauve02], suggests that a number of interconnected proxies be deployed at the edge of the network to help minimize the drawbacks, but also keep the benefits of a client-server architecture. The proxies may then perform various actions on behalf of the server and also send messages among themselves in order to relieve the server.

Another suggestion for distributed games [Bauer02] is to use network aware *booster boxes*. “Booster boxes acquire network awareness by monitoring traffic, measuring network parameters such as delay, by participating in routing exchanges, or by acting as an application-layer proxy server. Each booster box serves a number of clients in its network vicinity and performs caching, filtering, forwarding and redirecting of game events in a game-specific way.”

3.2. MoD proxy caches

As with HTTP proxies, the MoD proxy's prime operation is to cache objects. However, other application specific considerations will have to be made in MoD proxy caches than in HTTP proxy implementations. The prime difference is how objects are cached and how they are replaced. MoD proxies have other application specific uses besides caching. They may perform dynamic transcoding, i.e., change the bit-rate at which a stream is sent. They may also support changing the encoding on the fly as well as transport protocol translation, i.e., using UDP in a Wide-area network and TCP in a access network [Griwodz04]. It is therefore hard to use one single proxy implementation to handle both classic web objects and media objects.

Two primary observations motivate the caching of media files in an MoD system. First, an application needs to play back large amounts of data with timely delivery. The continuous media makes the application sensitive to jitter. To reduce this, the client usually buffers a certain amount of data before starting playback. This means that media playback cannot be started until this buffer is filled up. As mentioned in Section 2.4, the distance from client to server also is a variable that regulates latency. A proxy cache can alleviate this startup-latency by being deployed in the vicinity of the client. Media files also consume a lot of bandwidth due to their high bit-rate. An early cache hit would therefore reduce the load on the core of the network.

[4]Proxies are also used in *Remote Procedure Calls* (RPC) and *Inter Process Communication* (IPC) though the operations performed differs from those in a HTTP proxy. In IPC, the operations mainly consist of marshalling and UN-marshaling the messages that are sent to and received from the network. Proxies are sometimes called stubs in these contexts.

Secondly, media objects in MoD seldom change, and the “write-once-read-many” pattern is a strong argument for caching these objects in contrast to regular web-objects. There is therefore a low administrative overhead to keep caches up to date. The main problem when caching media files is the storage requirement, see Example 2.2. It is impossible to cache the immense amounts data that a collection of media files represent entirely in the memory of a host. It is even hard to cache all available media objects on disk. The Zipf distribution [Sitaram00], however, indicates that a good cache hit prognosis is to cache the 10% most popular films. The hard part is then to know what films are the most popular. Many caching strategies exist to deal with the problem of huge media objects. The solution is to partition an object and only cache parts of it. Examples of such algorithms follow in Section 3.2.2.

An MoD system must support the heterogeneity in the Internet. Proxies have to accommodate the requirements of different clients with respect to streaming rates and encoding formats. In commercial streaming systems, this has been solved by replicating objects at different rates and formats. This approach has prohibitively high demands on storage and bandwidth. To avoid this, proxies may perform on-demand transcoding of the streams. The intensive computation overhead of transcoding however prevents a proxy from supporting a large client population [Liu03].

Bommaiah et al [Bommaiah00] design and implement a proxy using the Internet-standard protocols RTSP and RTP. They use their prototype to show the performance gains by inserting proxies in a streaming environment. Their results, using a single proxy, show that by caching the 50 first seconds of a 1 minute media object, they are able to reduce network and server loads by 75%. They also show a 50% improvement in startup latency, by streaming the first three seconds ten times faster than the streaming rate.

Another proxy scheme presented by Acharya and Smith [Acharya00], called MiddleMan looked at the caching of movies within a campus LAN. While having proxies cooperate when caching objects, they achieved a high aggregate storage space. Their proxies cooperate through a coordinator process that held information about where elements were located. They found that the effective bandwidth of their entire streaming system increased by a factor between three and ten.

3.2.1. Properties

To serve their purpose, proxy cache systems should fulfill certain qualities. A number of these are discussed for Web caching[Wang99] and include fast access, robustness, transparency, scalability, efficiency, adaptability, stability, load balancing, dealing with heterogeneity and simplicity. These properties also, as described below, apply to MoD proxy cache system.

- *Responsiveness*: Users expect applications to respond quickly to commands. If not, they would grow inpatient and maybe stop using the application. This property is important in an MoD cache. In a distributed system without real-time support, a multimedia player uses jitter avoidance buffering to “ensure” continuous playback. The startup latency will therefore to a higher extent be noticeable to a user compared to streaming a text document. So, a replica of the data with shorter transfer time to the client, would increase the responsiveness of the application, in respect to both startup latency and interaction latency.

- *Robustness*: How robust a caching system is to failures is a property that for a user is experienced as availability of the service. Having a single point of failure in an MoD caching schema should be avoided. If one proxy crashes, the system should not be rendered inaccessible to clients. The system should provide fallback mechanisms if one or a few nodes fall out. Proxies may form an overlay network by being connected to each other in order to provide such robustness.

- *Transparency*[5]: A system should appear as a whole and conceal details of the underlying components from the user. This would imply that inserting a cache in the network should not have implications for the user, other than the improvements of the service. The user should not need to be aware of whether the content is coming from the server or the proxy, nor of how the possible per-

[5]There are several categories of transparency such as, access, location, concurrency, replication, failure, mobility, performance - and scaling transparency [Coulouris01].

formance gain is achieved.

- *Scalability*: Scalability is how well a solution to some problem will work when the size of the problem increases. In a server or proxy system, the increase of resources used should be constant to the increase in usage. A caching scheme must therefore not impose heavy resource demands due to changes in density, usage or size of a network. Inserting proxies in a system should not increase resource demands. This means that any inter-cache protocols should be lightweight and not add numerous packets in the network. Also, every proxy must carefully allocate its resources so it may serve many clients without degrading its performance.

- *Efficiency*: The efficiency of a system is dependent on the throughput of it. The time spent by processing or communicating in an MoD caching system should therefore impose a minimal addition to delay on a stream. MoD cache should process as close to link speeds as possible and send few control packets into the net to achieve this. This is closely related to scalability.

- *Adaptability*: Environmental changes often mean that behavior must be altered as well. A cache should be able to adapt to changes in access patterns and proxy placement. It should either be aware of architectural changes or be configurable to adapt to such changes. It should also be able to adapt to changes in network load, by altering the data-rate in an MoD stream.

- *Load Balancing*: This property means to distribute the computation and network load. A caching scheme should avoid hot-spotting certain parts of a system. It should have mechanisms to distribute load on the network as well as the server.

- *Heterogeneity*: The Internet is built up using a collection of different computer and network technologies. This heterogeneity is challenging in many areas, and must be considered when designing distributed systems. A caching scheme must be able to handle the differences in the link -and network layers. MoD caches must in particular be able to deal with different formats for representing media. Such formats are MPEG-2, AVI, WMA, etc. Also an MoD proxy should be able to deliver different quality streams depending on the clients requirements.

- *Simplicity*: Keeping the scheme simple is important in order to easily implement and deploy it in an already existing system.

These properties have implications on the design of a proxy system because they dictate the goals of such a design. If a design holds efficiency as a prime concern, then adaption and transparency may get a lower priority. However, efficiency may also imply that the system should be simplistic. Whether the properties converge or diverge they impose tradeoffs in a design. There are other possible properties in such a system as well, security being one and extendibility another. These properties are often in conflict with scalability and efficiency, as they require resources and time consuming processing.

The properties above applies both to the internals of a proxy host and the surrounding system. The considerations are different if the proxy must cooperate with other proxies than if it is a stand-alone proxy. A design should not restrict itself, but allow extension so that the adaptability of the proxy to the surroundings can be met.

3.2.2. Caching strategies

A well known fact is that the optimal way to perform caching is to always have the object that will be accessed next ready in the cache. Also, when the cache is full, the objects best to replace are the ones that will not be accessed for the longest time. If a caching scheme could predict this perfectly a perfect cache hit rate would be achieved. However, predicting the future is as impossible in computer science as in life. Caching algorithms try their best to predict what to place in the cache and what to replace when needed, based on typical application access patterns. In MoD, there are numerous such algorithms. Liu and Xu [Liu03] divide them into four categories for homogeneous clients, i.e., for clients that have identical requirements on data-rate, etc.:

- *Sliding-interval caching* caches a sliding interval of a media object to exploit sequential access of streaming media. If two consecutive requests for the same object occurs, the first will fetch the object from the server and incrementally store it in the cache of the proxy. The second request may then access the cached portion and release it when fetched. If several requests for the same object arrive close in time, the last request will be the one releasing the object from the cache. This algorithm can be very effective in saving network bandwidth and start-up latency when subsequent requests for the same object happen within close time intervals.

- *Prefix caching* caches the start of an object, called the prefix [Sen99]. When serving a request the proxy immediately deliverers the prefix to the client and, meanwhile, fetches the rest of the object, called the *suffix* from the server. When using prefix caching the size of the prefix is chosen based on available resources and optimization objectives. It can significantly reduce the start-up latency of a stream, but it does not reduce the network load or server load to the same extent.

- *Segment caching* is a generalization of prefix-caching. It partitions a media object into a series of segments and set different caching values on each segment. Usually, the first few segments are viewed as more important than those at the end, as they may be pre-fetched. However, if the media object is popular enough, it should also be possible to cache the later segments in a stream. If the popularity of an object decreases the proxy may discard large chunks of the object, but it has the possibility of leaving the first segments, e.g., the prefix of the object. This algorithm has about the same savings on start-up latency and network load as prefix caching.

- *Rate-split caching* partitions the data horizontally along the rate axis, instead of vertically along the time axis as the previous three algorithms do. The upper part of the data is cached in the proxy. This algorithm is attractive when the data is variable-bit-rate media. In a non-QoS network it improves the network load and start-up latency moderately.

The above strategies all implicitly expects the remaining non-cached data to be fetched from the server in order to satisfy homogeneous clients. However, to serve heterogeneity, layered caching is suggested. This method of caching requires a layered encoding of the data, i.e., the data is split into several layers: the most significant layer, called the *base layer*, contains data representing the most important features of the object, while additional layers, called *enhancement layers*, contain data that refine quality. The higher the layer is from the base, the more fine grained is the quality. Layered caching usually caches the lower layers, as they are relevant to all clients requesting the object. On cache replacement a layer is identified based on popularity, and its cached segments are removed from the tail until sufficient space is obtained.

A proxy cache also needs a strategy for replacing objects when the cache fills up. Several methods exist to do this. Podlipnig and Böszörmenyi [Podlipnig03] give a survey of such strategies. They classify them as follows:

- Recency-based strategies: These strategies take into account how much time has passed since the object was last referenced. These strategies are usually variants of the *Last Recently Used* (LRU) strategy, which simply replaces the object that has been in the cache the longest without any accesses.

- Frequency-based strategies: These strategies take into account how often a object is accessed, i.e., the frequency of references decides what to remove. These strategies are usually extensions to the *Least Frequently Used* (LFU) strategy. Objects that are seldomly accessed are removed.

- Recency/frequency-based strategies: These strategies combine recency and frequency factors to make caching decisions.

- Function-based strategies: These strategies use a function to calculate the value of an object. They use different weighting factors in the functions to calculate an appropriate value, e.g., age, size, cost of fetching the object, etc.

- Randomized strategies: These strategies decides what to replace in a randomized way.

In multimedia caching, the access patterns are a vital piece of information in order to make caching decisions. A movie that is accessed often can be regarded as popular and should probably remain in a cache. The *Eternal history, Conditional replacement and Temporal gap size* (ECT) [Griwodz00], is loosely based on the LFU strategy[6]. It accounts for all requests to an element, and stores this information in an eight entry history per element. Each entry in the history is the gap in time between two consecutive hits on a stored element. The ECT strategy does not remove the history of an element after it is removed, so elements have an eternal history. It is conditional as all first time elements are cached, but after being removed the history of the element will still exist and is used for future cache decisions. This strategy shows good simulation results when caching media objects in a large population system with many movies.

3.3. Cache architectures

As shown in Figure 3.1 the proxy is an hierarchical extension to the client-server architecture. To realize many of the properties in Section 3.2.1, proxy designs usually include the possibility for proxies to interconnect. When a number of caches are inserted into a network, the probability of an object already being cached increases, i.e., a increased hit-probability [Acharya00]. What differentiates the available architectures is how information is exchanged between the nodes and how they are laid out in the network.

A *Hierarchical architecture*, as shown in Figure 3.2 a), divides caching in levels. A request will traverse up the hierarchy until the requested content is found, the response will then traverse back down the hierarchy, leaving a copy of the content at each subsequent level along the path. The drawbacks of this architectural scheme its that many replicas of objects are left down through the hierarchy. Also the high level caches become key access points that might go down or become bottlenecks. Lastly, a negative effect is that processing delay is added at each level of the hierarchy.

Distributed architectures or cooperative caching, as shown in Figure 3.2 b), only operate with one level of caches. Theses caches distribute meta-information on where to fetch cache misses. The distribution of the meta-information may be hierarchical, but fetching a cached object is only done within the same level. This model may experience high connection times and high bandwidth usage when used in large scale. Also, as with P2P, administrative issues exist.

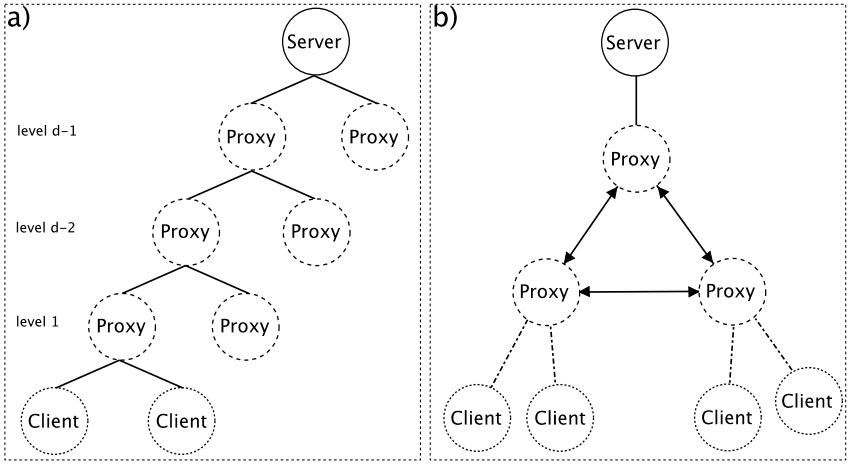


Figure 3.2. Cache architecture

[6]In the reference it says that ECT is based on the *Inter-Reference Gap* (IRG) algorithm, but personal correspondence with the author suggests that this is wrong.

The *Hybrid* proxy architecture is not shown in the figure, but it incorporates features from both the hierarchical architecture and the distributed architecture. A hybrid have caches cooperating on the same level, or in between levels. The requested cached object is fetched from the parent of neighboring cache with the lowest *round trip time* (RTT).

3.4. Data flow

The flow of packets in a proxy can typically be divided into a control-plane and a data-plane. In HTTP proxies, this distinction is not as clear as in MoD proxies. The HTTP messages hold both control data and content, while for instance, using RTSP and RTP separates the two[7]. When not interleaving data, RTSP messages follow the control-plane and RTP packets flow in the data-plane. The separation of the two planes is shown in Figure 3.3. Both the data-plane and the control-plane are bidirectional. The control-plane experiences traffic in both directions during a RTSP session, but in the data-plane, traffic only flows in one direction, either from client to server or from server to client. The recording feature of RTSP is an example of what enables data to flow from client to server.

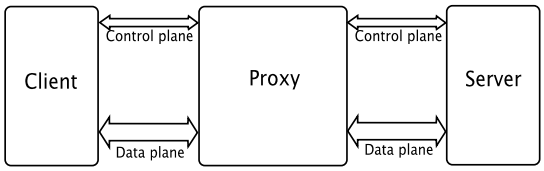


Figure 3.3. Proxy data flow

On a high-level, a proxy cache could be implemented as a process, or a set of communicating processes, running on a machine within the network. If one assumes that it is one process, this would act as both server and client. The server part of the process would handle incoming requests and serve these if able. If it is unable to respond to a request immediately, it would use the client-side of the proxy and connect to another proxy or directly to a server. When the requested object is found through the control-plane it would be transfered back to the proxy through the data-plane. The proxy would receive the object, cache it and send it further down the request path. As seen in Figure 3.3 the control-plane is assumed to be a lot thinner than the data-plane. This illustrates that the requests to establish a stream do not take up as much resources as the actual streaming.

The internals of a proxy cache depend on what platform it runs on and how it is implemented. However, it would normally use a socket interface to communicate with the network. Every incoming packet on a network interface would find its way to the correct socket. Whenever a packet is received on the network interface card, it will emit an interrupt. This interrupt will normally activate the interrupt handler within the *Operating System* (OS) kernel. This results in a change of context in order to process the packet. The payload of the packet will then be transferred into kernel memory and will traverse up the network stack. Whenever application level data is received, the data must be copied to the memory of the user space application. Then, the proxy may deal with this data according to any application level protocol, like saving the data to disk, before any response is sent down the stack and transmitted by the NIC.

[7]Although RTSP allows for interleaving the data-plane, this is usually only done when circumventing firewalls.

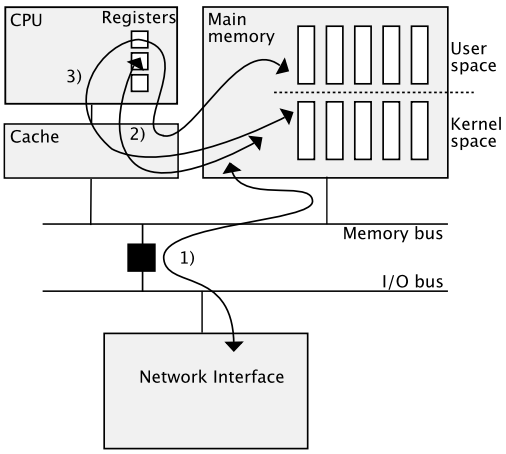


Figure 3.4. Proxy data-path

The data-flow is illustrated in Figure 3.4. An interrupt and a bus transfer (1) is needed to get the packet into kernel memory. While being processed by the network stack, elements in memory will pass through the cache and the CPU registers (2). Then, when reaching the application, the packet payload must be copied to user space memory, and every memory element will be moved by the CPU to the user space memory location (3). If the application then needs to write data to disk, or send new data back out, additional copying and CPU processing is needed.

3.4.1. Copy operations

As explained above, every incoming payload of a packet must be copied to the applications memory location in user-space through the CPU. Several experiments show that copy operations represents a major bottleneck in high throughput systems, such as a multimedia server, or proxy. For example in [Halvorsen01] a series of experiments using different copy operations on a NetBSD platform, are described. When copying data into user-space using the *copyout()* function a 1024 byte block of memory was copied in an average of 0.48 µs, i.e., about 447 cycles with interrupts turned off. Similar overhead is shown when copying from user space to kernel memory and also internal copying within kernel memory, regardless of whether interrupts were turned on or off. These tests were performed using an Intel Pentium II Xeon 933 MHz processor with a 256 PC800 RDRAM chip.

3.4.2. The Linux network stack

The Linux network stack [Beck99] is an implementation of the BSD socket interface. Although Linux provides support for other protocols, such as AppleTalk and IPX, the most commonly used is the TCP/IP collection of protocols. This includes TCP, UDP and IP. The data link layer focuses on Ethernet, but Linux also support other link layers. For instance, the *Serial Line Interface Protocol* (SLIP) and the *Parallel Line Interface Protocol* (PLIP). Figure 3.5 shows how packets traverse the stack from the physical interface until it reaches the application through the appropriate socket. The figure uses UDP as the transport protocol, but the same applies to TCP.

In their research, Guo and Zheng analyzed and evaluated the network stack performance of the GNU/Linux 2.0.34 kernel [Guo00]. They concluded that a pair of Linux hosts were able to communicate at near the max of the capability of a 100 Mbps NIC. The setting in which they ran their tests was with both hosts having 128 MB of RAM and a Pentium II with a clock-rate of 350 MHz. The hosts ran in multiuser mode, but only a minimum of processes were started.

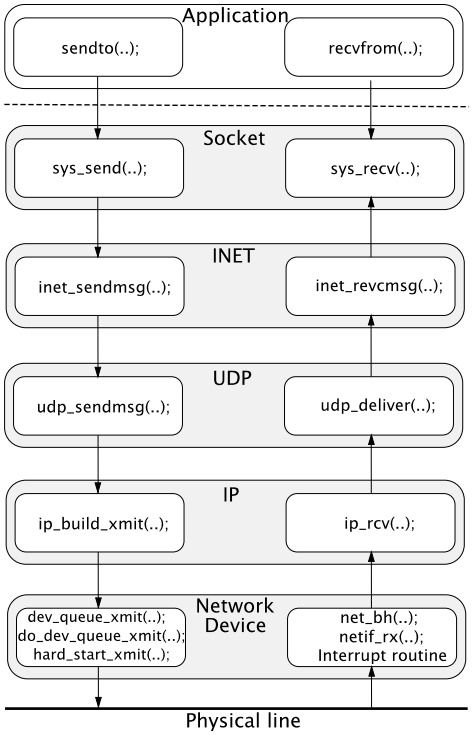


Figure 3.5. Linux 2.0.34 network stack

They also performed a different set of tests, in order to show the total capacity of the stack. To do this they counted the processing time of each layer and also the time it took to transfer a packet to and from memory. In total, the stack spent 18.9 µs to send a UDP packet, without calculating the checksum. The total processing time needed to receive a UDP packet was 35 µs. The difference between sending and receiving a packet was due to the fact that receiving a packet was more complex than sending a packet. Receiving a packet from the NIC required a CPU driven transfer of the data, while sending employed DMA. The DMA transfer when sending took 2 µs and the CPU transfer when receiving took 4 µs. This only make up for 2µs, and the difference between sending and receiving is still 14.1 µs. The rest of the difference comes from the fact that the subsequent layers on the receive side is more complex. Their measurements showed that the total sending capability using UDP was 433 Mbps. When receiving UDP, the total capability was 234 Mbps. For TCP these numbers were slightly lower; 364 Mbps while sending and 234 Mbps while receiving.

Their measurements were made with packets of the same size; 1024 bytes. The results would have been different if the packet sizes were larger, as the CPU would have to move more data and the checksums might also take longer.

Context switches and memory copy operations makes the Linux kernel spend much time processing network packets. In order for this architecture to support applications that employ a fully utilized 1 Gbps network, a 1.5 GHz CPU clock would have been needed. The Linux kernel has in the time of writing reached version 2.6.7 [Kernel04], so different approaches might have been implemented. Nevertheless, if a proxy were to be implemented on a Linux platform with the resources of the above testbed, it would not be able to receive more than an aggregate bit-rate of 234 Mbps and send 433 Mbps.

3.5. Summary

Proxies are deployed in the vicinity of clients to help deploy applications in the Internet. They reduce startup-latency, network load and server load. The proxy approach shows a good potential in state-of-the-art multimedia research [Acharya00], and have also been adopted in commercial streaming products [Real04b].

However, in a large-scale multimedia system, a proxy might need to handle heterogeneous requirements, i.e., data transcoding, high aggregate network loads and many concurrent clients. It may also have to perform protocol translation, encryption and other tasks. Therefore, it is subject to scalability issues, as it needs to be able to process high rates of network traffic with real-time requirements and also perform CPU insensitive tasks as dynamic transcoding or re-encoding. Also, MoD caching strategies mostly cache only some segments of a media object and require that non-cached data is retrieved from the server.

In order for a multimedia proxy to meet the demands of high data-rates and at the same time be flexible enough to perform application layer operations, we suggest that it can make use of NPUs to offload the data-plane. In the next chapter, we therefore present an overview of the NPU technology we will use for this purpose.

Chapter 4.

Network processors

> God turns you from one feeling to another and teaches by means of opposites, so that you will have two wings to fly, not one
>
> —Mevlana Rumi

Traditional network processing is facing challenges. It must keep up with the increasing bandwidth and new types of network traffic. Every packet flows through several layers of processing and is sometimes replicated in memory. Calculating checksums and copying memory blocks add overhead to the time required to finish processing a network packet. These facts have motivated innovations within network processing. Hardware accelerated processing helps performance and scalability, but is restricted to change. An increasing demand for new support in networks has therefore led to a technology that intermingles programmability and hardware acceleration: the *Network Processing Unit* (NPU).

NPUs have the ability to process network traffic aided by hardware and software. They usually have special network-task instruction sets for hardware accelerated programmable units. These units perform wire speed processing. The NPU also allows for general purpose programming for exceptional cases and applications.

Following in this chapter is a general overview of NPUs and an in-depth description of the Intel produced IXP1200 processor chip [Intel01b].

4.1. General

An NPU may be defined as a system of interconnected programmable devices performing wire speed processing of *Protocol Data Units* (PDUs). The processing performed can be split into the following main functions:

- *Classification*, which establishes a relationship between a PDU and one or many members of a finite set of classes. A class is an abstraction that can be expressed as an action or as a number. For example, related to QoS a class may describe the the absolute requirements of the PDU or a class may simply be the protocol as protocols are treated differently throughout the system. Classification is the driver of the other network processing functions as it dictates what to be modified, managed or encrypted;

- *Modification*, which changes the contents of a PDU. By modification we mean PDU header editing and sometimes payload editing. Header editing involves insertion, removal and replacement of header fields, and payload editing may involve adaption or re-encoding of the data contained in the PDU;

- *Traffic Management*, which shapes PDU traffic. This involves queuing PDUs for transmission with potential discarding of PDUs based on a predefined discard policy; and

- *Encryption*, which encrypts and decrypts the PDU payload.

Traditionally, core network nodes have been processing the low-end of the network stack, i.e., layers 1-3 in Figure 2.3. Apart from physically transmitting packets, tasks within these layers are network layer routing and data link switching. An end node is therefore responsible for processing the high-end of the stack. As mentioned in Section 1.1, the philosophy of the IP relies on keeping the network core as lightweight as possible, and this may be the reason for much of its success. It has been shown that performing complex processing and keeping much state in the core network will deteriorate scalability. An example of this is the QoS architecture *Integrated Services* (IntServ) which was abandoned in favor of *Differentiated Services* (DiffServ)[8]. The difference between the two illustrates the point of keeping the core lightweight. DiffServ relies on complex processing to be done at the boundary of the network. The core is simply forwarding packets on what they call a *Per Hop Behavior*. This is a simple classification scheme that decides the priority of the packet. IntServ, on the other hand holds unique states for every flow of packets in every network node. This is in IntServ done regardless of where the node is situated; in the core or at the boundary of the network. Even though DiffServ keeps the core lightweight, it is not widely deployed. This may be due to many factors, but one thing might be that it requires changes in the network nodes. These changes can be applied linearly and do therefore not need all nodes to be upgraded at the same time, but if processing functionality is implemented in hardware, it may be expensive to extend them to support DiffServ.

In all networks, the processing needs to meet the demands of new classes of network traffic, such as multimedia and VPN, that are becoming common due to increases in available bandwidth. The traditional general-purpose processor architecture is no longer sufficiently able to scale to such a scenario. *Application-Specific Integrated Circuits* (ASICs) are commonly used to solve this issue by offloading the data-plane processing to dedicated hardware. By providing wider memory buses and faster execution times, they scale well for this purpose. The original processor will only process the control-plane. However, ASICs have a long development cycle and lack the flexibility of re-programmability. This means that whenever a bug is detected or whenever extensions are needed to support new classes of traffic, ASIC designers must enter another development cycle. Re-implementing a whole hardware chip is a time-consuming and costly process.

NPUs emerged as an alternative solution to meet the increasing demands of network processing. To be an acceptable replacement for ASICs and to meet future demands, NPUs should have certain characteristics. Though these can be met using different approaches, they have some general aspects:

- *Relatively low cost*: The key of NPU design is the combination of flexibility and speed, e.g., programmability and hardware. This is what achieves lower cost compared to ASICs. As NPUs also target the entire network processing field, one device may be used in a wide range of network solutions while ASICs are specialized and must be tossed away when their functionality becomes obsolete. Money is therefore saved when purchasing an NPU as it may be reused in different networking contexts by changing software. Also, the number of potential purchasers is larger as the same device can be used in several areas, and re-usage of software libraries and components lower the cost and the “Time-to-Market”.

- *Straightforward hardware interface*: In order to offer a straightforward hardware interface, standards are necessary. By complying to standards such as *Peripheral Component Interconnect* (PCI) and *Direct Memory Access* (DMA), it is easier for OSes to supply means of communication, i.e., device drivers to NPUs.

- *Programmability*: Programmability may be supported differently on different NPUs. Douglas E. Comer describes the hierarchy of processing units that are used in network processing [Comer03]. As shown in Table 4.1, this ranges from *General Purpose Processors* (GPPs) to low level physical transmitters. The GPPs, the Embedded processors and the I/O processors, i.e., the three upper ones, are programmable. Each level in the hierarchy performs and is programmed according to the tasks it is meant for. These tasks are not strict, but a rough outline follows. The GPP typically per-

[8]IntServ and DiffServ are extensively documented on www.faqs.org. IntServ components are described in RFC 2210 through 2216 and DiffServ is documented in RFC 2474, 2475, 2597, 2983, 3140, 3246, 3247, 3248, 3260, 3289 and 3290.

forms overall system control and management, complex protocols and a possible administrative interface. The Embedded processor processes the high-layer protocols along with reassembly and traffic shaping. It also controls the I/O processors and handles exceptions and errors. Finally, the I/O processors basically perform reception and sending along with classification and forwarding of packets.

Table 4.1. Processor hierarchy [Comer03]

Level	Processor Type	Programmable?	On Chip?
8	General purpose CPU	yes	possibly
7	Embedded processor	yes	possibly
6	I/O processor	yes	typically
5	Co-processor	no	typically
4	Fabric interface	no	typically
3	Data transfer unit	no	typically
2	Framer	no	possibly
1	Physical transmitter	no	possibly

- *Ability to scale to high data and packet rates*: Without scaling to high data and packet rates, NPUs will be less liable as an alternative to ASICs. In order to scale, NPU must either have processing units with high processing rate, handle packets in parallel or have many types of processors in the hierarchy of processing units. The lower level processing units are expected to handle many packets without sending them to the higher levels. The high level processors are expected to handle few packets. For example, the GPPs handle few packets, while the I/O processors handle many. To make the I/O processors scale, they must, as said, either have a high processing rate or execute in parallel. No software control the threads running in parallel on the I/O processors. They are switched and controlled through hardware. This is referred to as hardware supported parallelism. Parallel execution is often supported through hardware by allowing thread communication mechanisms, i.e., passing messages in between threads is done by hardware.

  Another scalability issue is the memory interfaces and their delay. If the processing units were made to scale by increasing their processing rate, this would not be successful if they had to block while waiting for a memory request to finish. Types of memory differ in access latency. Comparing the two basic *Random Access Memory* (RAM) types, *Static RAM* (SRAM) and *Dynamic RAM* (DRAM) shows this difference. SRAM has a typical latency of 1.5-10 ns, while DRAM has 50-70 ns. In addition, the bus transfer speed is critical. This is because every request to memory must be transfered back to the processor that issued the request. Asynchronous memory access may mask the access and bus latency, but this requires processors to process in parallel.

Figure 4.1 shows where NPUs lie in reference to cost and performance, compared to general purpose and ASIC processing. NPU technology is expected to be cheaper than ASICs, but are more expensive than a general purpose software solution. Performance wise, it is expected to be slower than ASICs, but faster than pure software processing. Again, this is due to the combination of programmability and hardware. The question-marks denote that how cheap and fast NPUs are is unknown and depend on model, vendor, etc. Different designs may perform and scale better, while others may prove to be cheaper. These characteristics depend largely on design tradeoffs.

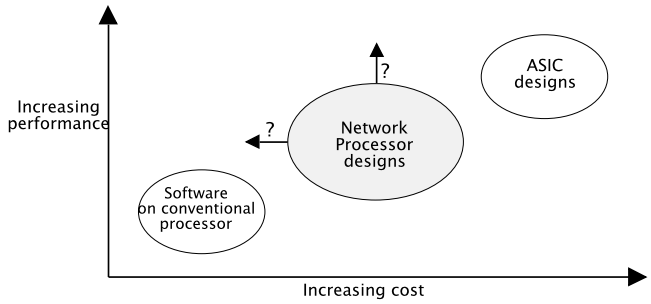


Figure 4.1. Cost vs. performance [Comer03]

Several vendors, such as Intel[9] and Motorola[10] design and produce NPUs [Shah01]. While the market value was about $60 million in 2003, it is expected that the NPU market is to out-pace the rest of the silicon markets for the next 5 years [InStat04]. Figure 4.2 shows the main contenders and their market shares.

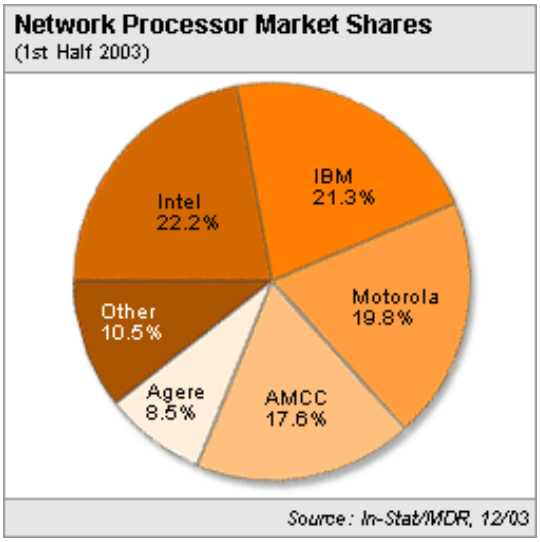


Figure 4.2. NPU market shares [InStat04]

4.1.1. NIC evolution

Figure 4.3 shows a rough overview of how the hardware architecture of NICs has evolved. To illustrate the differences, three network routers are displayed. Each router has two network interfaces, or “blades” connected to a GPP and other possible router hardware, such as switch fabrics and buses. The reason why a router is used as an example, is because this is one of the most central and relevant applications for networking, and is also one of the primary applications that NPUs are meant for.

[9]An overview of the available NPUs from Intel can be found here: http://www.intel.com/design/network/products/processors.htm
[10] Motorola's NPUs can be found here: http://www.freescale.com/webapp/sps/site/homepage.jsp?nodeId=02VS0l

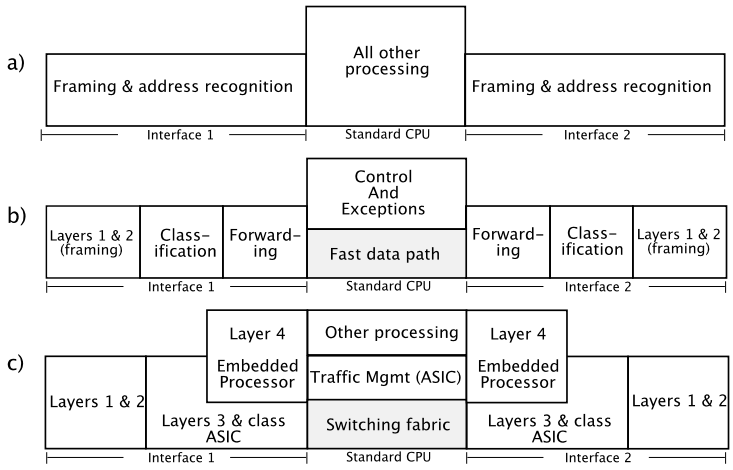


Figure 4.3. Three generations of NICs [Comer03]

Figure 4.3 a) is the traditional software router, where all processing except *Media Access Control* (MAC) functionality is done by the standard CPU. The chief advantages of a software router is low cost. The main disadvantage is low processing speed. In a software router, the GPP will quickly become the bottleneck, as it has to process the *aggregate data-rate* of all the interfaces connected to it. The aggregated data-rate is defined to be the sum of the rates at which traffic enters or leaves a system. For example, if both the interfaces of Figure 4.3 a) is able to handle 100 (Mbps), then the aggregate data-rate that the whole router must handle is 200 Mbps. Therefore the software router is unable to scale well as bandwidth in networks increase, e.g., managing several of the 1-10 Gbps NICs available today.

To remedy this fact, much effort has been put into finding out how to relieve the GPP from having to do all the work. As seen in Figure 4.3 b), more functionality is offloaded into the NIC. These devices perform classification, checksumming and forwarding in addition to layer one and two functionality. To achieve such complex functionality onboard devices, two approaches are common; Special purpose coprocessors (ASICs) and Embedded RISC hardware. Using these approaches are costly, but yields high performance.

Figure 4.3 c) depicts the NPU. It is the third generation of *Network Interface Cards* (NIC). This evolutionary process is as the second generation, moving towards decentralization of protocol processing in order to reduce the load on a central CPU. This approach is, however supposed to be flexible, yield high performance and be cheap.

In this thesis we have chosen to make use of the Intel IXP1200 [Intel01b] NPU, which will be described next.

4.2. Intel® Internet Exchange Architecture

The Intel® Internet eXchange Processor 1200 (IXP1200) [Intel01b] chip is a part of the Intel® Internet Exchange Architecture (IXA) network processor family. The Intel® IXA summarizes the basic components for network processing [Intel02a]:

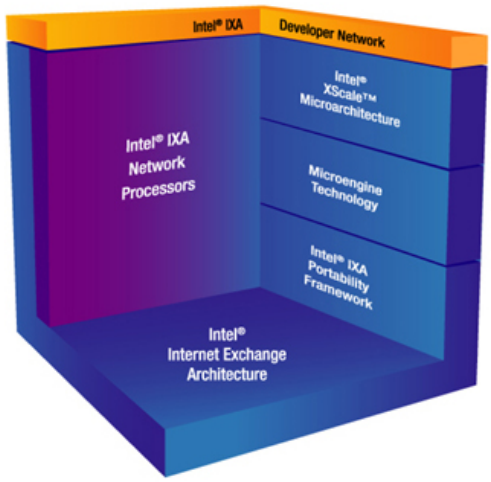


Figure 4.4. The Internet Exchange Architecture

- "Microengine technology -- a subsystem of programmable, multi-threaded RISC microengines that enable high-performance packet processing in the data-plane through Intel® Hyper Task Chaining[11]. This multi-processing technology features software pipelining and low-latency sequence management hardware."
- "Intel®XScale™ technology -- providing the highest performance-to-power ratio in the industry, with performance up to 1000 MIPS and power consumption as low as 10mW, for low-power, high-density processing of control-plane applications."
- "The Intel IXA Portability Framework -- an easy-to-use modular programming framework providing the advantages of software investment protection and faster time-to-market through code portability and reuse between network processor-based projects, in addition to future generations of Intel IXA network processors."

As shown in Figure 4.4 and described above, the IXA contains all the components that make up the network processing architecture of Intel. It includes the hardware processing units, and the framework to develop applications. The IXA architecture divides the levels of processing into a data and control-plane. It consists of "Microengine technology" for *fast-path* processing in the data-plane and a core CPU for *slow-path* processing in the control-plane. Above the Intel XScale processor is mentioned, but the IXP1200 is in fact equipped with a StrongARM processor. Most other and newer NPUs from Intel have a XScale processor[12]. The next sections will give further details about the components for the IXP1200 NPU.

4.3. Intel® IXP1200 network processor

To our knowledge the Intel® IXP1200 was the first official NPU released by Intel, code named the "Evaluation Board". The card acquired for this thesis, is the ENP-2505, produced by Radisys. It has a later version of the IXP1200 chip. The card is announced to be suitable for applications running on OC-3 to OC-12 lines, which equals speeds of 155-622 Mbps.

Figure 4.5 shows the internal and external units of the IXP1200 processor. Internally or on-chip, it has a 232MHz embedded StrongARM core and 6, 232 MHz programmable I/O processors. In addition, it

[11]Hyper Task Chaining is a set of hardware features ranging from thread-to-thread communication, memory read-modify-write operations and buffers between pipelines/stages.
[12]Cards equipped with the XScale processor range from the IXP400 to the IXP2850.

has other functional units that primarily allow the chip to interface external units. These are the *First In First Out* (FIFO) *Bus Interface* (FBI), *Synchronous Random Access Memory* (SRAM) Interface, *Synchronous Dynamic Random Access Memory* (SDRAM) Interface and the PCI Bus Interface. They are described in more detail below.

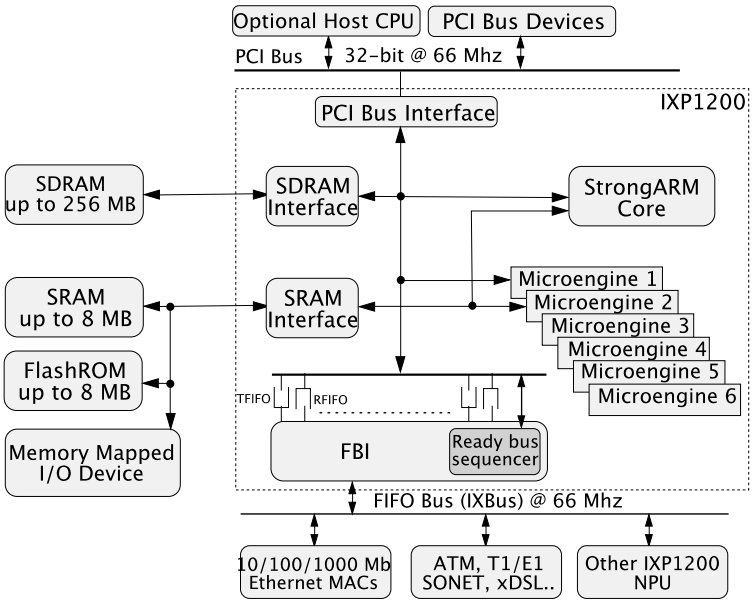


Figure 4.5. IXP1200 block diagram

4.3.1. The StrongARM core CPU

The StrongARM core is a general purpose CPU that implements the 32-bit ARM V4 architecture. Known to be used in hand-held devices, it has a tradeoff between computational capabilities and power consumption. The StrongARM is supported to run the GNU[13]/Linux kernel [Kernel04] and the real-time operating system VxWorks [WindRiver04]. Cross-compiler tool chains from the GNU [Gnu04] project also exist in order to create executables for the ARM instruction set.

On the IXP1200 board, the StrongARM governs the initialization of the microengines, and it is also used for packet processing and exception handling. How this works is further described in Section 4.4.

4.3.2. Microengines

Intel® has equipped the IXP1200 with six 32 bit RISC processors. These I/O processors, called Microengines are optimized for fast-path packet processing. Their instruction set is tuned for processing network data. This involves reception, inspection, manipulation and transfer of packets. Together with the FBI, described later, the microengines establishes the *fast path* of the IXP1200.

Each microengine has an instruction store of 8192 bytes. Every instruction takes up one long-word (32-bits) of control store space. This allows for a total of 2048 instructions to be stored. Once the microengines are running and the StrongARM has loaded the instruction store, execution starts in a five-stage pipeline. When the pipeline is full, every instruction is completed in an average of one cycle. If instructions block to access memory or other external devices, the pipeline must be flushed, and the number of cycles needed per instruction will be higher than one.

The Microengines support what is called *Hardware Multi-threading* which gives them four program counters and thus also the ability to run four threads each. In theory, this means that a total of 24 threads may run concurrently. Threading is supported in hardware through *non-preemptive* signals

[13]GNU is a recursive acronym and stands for *GNU is Not UNIX*.

that are controlled by a threading arbiter unit. What this means is that in order to change from one thread context to another, the current running thread must yield control of its operations. In contrary to *preemptive* thread scheduling where a timer will interrupt thread execution, threads in *non-preemptive* scheduling must be programmed to give up control. One thread may potentially run forever, if so desired. When or if a thread yields, it gives control to the threading arbiter. This unit will swap between threads in a *round robin* fashion, only activating threads that are ready to run. The scheduling is *round robin* because the arbiter keeps track of the ready threads and searches in thread-identifier order to find the next to run.

Switching between threads on the microengines does not require the registers to be flushed to memory. On most general purpose processors this must be done to ensure that the context is the same when the thread is rescheduled. The microengines enables the registers to stay untouched by equally dividing them between each thread. This means that fewer registers are available to each thread, but it also means that low overhead is associated with context switches. The overhead is the same as when an instruction causes the pipeline to abort the current execution, because the execution pipeline must be flushed. The number of cycles needed to perform a switch between one thread to another is about three [Johnson02].

Typically, a thread would yield while waiting for data from a functional unit, e.g., the SDRAM unit. When the functional unit is done, the thread is signaled and put back in ready state and may be scheduled by the arbiter unit. This allows for asynchronous memory access which is a key difference between microengines and general purpose processors.

Synchronization is simpler in non-preemptive scheduling as the programmer just need to assure that he does not yield while inside a critical section of the code. It is important to remember yielding at some point though, or else the thread will run forever.

The microengines have three kinds of registers. These include general purpose, SRAM transfer and SDRAM transfer registers:

- *General purpose registers* (GPRs): Each microengine has 128, 32-bit GPRs. These are split into two 64-register banks, the A and the B bank. Any instruction that allows two GPRs as input requires that they come from distinct banks; one from the A bank and the other from the B bank.

  GPRs may be accessed in thread-local or absolute mode. In thread-local mode, each thread has an unique set of 32 GPRs, 16 in the A bank and 16 in the B bank. For threads to intercommunicate, they may use GPRs in absolute mode, making them accessible to all threads running on the designated microengine.

- *SRAM transfer registers*: SRAM transfer registers are used to read and write to all functional units with the exception of the SDRAM unit. They are used to transfer data to and from the SRAM unit, the IX Bus and the PCI interface.

  Each microengine also has 64 SRAM transfer registers. These may, as GPRs, be accessed in thread-local or absolute mode. Each thread has 16 transfer registers at its disposal.

  Half of the SRAM transfer registers are write-only and the other half is read-only, i.e., 8 write-only and 8 read-only per thread context. In order to write to SRAM, the microengine must put the data in a write-only register, and when reading, the data must be put in a read-only register. This is the primary mechanism for asynchronous memory operations. The transfer registers act as a intermediate storage, allowing execution to continue regardless of the data transfer, making is asynchronous. The distinction between read-only and write-only registers is transparent for the programmer though, because they share the same names in microcode. Reading from the named register accesses the read-only register and writing to the named register accesses the write-only register.

- *SDRAM transfer registers*: A microengine has the same number of SDRAM transfer registers as it has SRAM registers, 64 in total and 16 per microengine thread. These registers are exclusively

used to read and write to the SDRAM unit. They have the same separation between read-only and write-only registers as SRAM does.

Whenever any of the six microengines issues a request to access any of the functional units, e.g., SRAM, SDRAM or the FBI, the command is put in a FIFO queue internal to the microengine. The chip-wide *Command Bus Arbiter* then chooses what microengine that is to be allowed access to the shared command bus next. After being transferred across the bus, commands are put into a set of hardware command queues shown in Figure 4.6. There are three different queues, the ordered queue, the priority queue and a set of optimize queues. Unless specified otherwise, all commands are put in the ordered queue. This way, every command is issued in order. This is useful to ensure that a memory location is not modified out of the expected order. Commands specified to go through the priority queue, will be issued before any other commands regardless of starvation. Finally, the optimize queues can be chosen by software to have hardware rearrange the order of commands to take advantage of physical characteristics. In this way, SRAM requests can be separated in multiple reads and then multiple writes instead of alternating reads and writes. However, caution must be exercised when using this functionality, as hazards can occur, such as reading a value that should have been written first, i.e., *read-after-write* (RAW) or when writing a value in the wrong order, i.e., *write-after-write* (WAW).

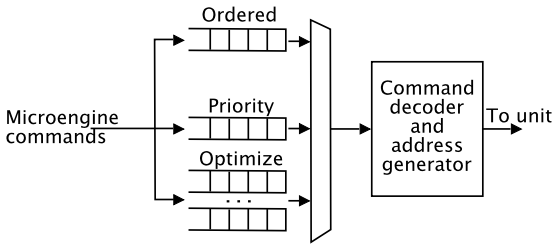


Figure 4.6. Command queues

4.3.3. The FBI unit

The main purpose of the FBI unit is to provide a high-speed interface for receiving and transmitting packets. It is, as shown in Figure 4.5, directly connected to the Ix Bus where all network packets enter and leave through the *Media Access Controls* (MACs). In addition, the FBI unit has several chip-wide *Control Status Registers* (CSRs), a small on-chip SRAM called Scratchpad memory and a little programmable processor called the *Ready Bus Sequencer*.

When receiving data, the physical MAC interfaces external to the IXP1200 chip will receive Ethernet frames with a *Maximum Transfer Unit* (MTU) of 1500 bytes. These frames are then broken into 64-byte chunks, called mpackets. The Ready Bus Sequencer will periodically poll the MAC devices to see if any mpackets are present. Whenever 64 bytes are available, the *Receive Ready* CSR registers are set. The microengines must then be programmed to poll the CSRs and if a packet is ready, instruct the FBI to pull it into one of the *Receive FIFO* (RFIFO) queues. Then, the Ethernet frame can be directly reassembled in SDRAM.

To transmit packets an egress ACE would have to reserve space in one of the *Transmit FIFO* (TFIFO) queues. Then, it must create a mpacket with data from SDRAM and copy this to the TFIFO. Finally, to allow hardware to send the packet, the code must set the valid flag in a CSR.

4.3.4. Memory interfaces

The IXP1200 provides three different memory interfaces, Scratchpad, SRAM and SDRAM. These units have different limitations in functionality, size and speed as shown in Table 4.2.

Table 4.2. IXP1200 memory interfaces [Johnson02]

Memory Interface	Minimum Addressable Unit	Size	Size(bytes max)	Approx. Unloaded Latency (clks)	Special Operations
Scratchpad	4	1K	4K	12-14	Atomic increment, bit test-and-set, bit test-and-clear
SRAM	4	2M	8M	16-20	CAM lock, bit test-and-set, bit test-and-clear, push-pop queues
SDRAM	8	32M	256M	33-40	Direct path to and from the FBI which allows data to be moved between the two without first going through one of the processors

- *Scratchpad*: Scratchpad memory is a low-latency on-chip memory interface to all of the microengines. As shown in Table 4.2, the Scratchpad is 1024 long-words (32-bit) in size, this equals 4KB. The Scratchpad is as mentioned, physically located on the FBI unit. It provides atomic bit-test-and-set, bit-test-and-clear and increment operations, as well as random read and write operations. Many accesses to Scratchpad memory should be avoided as the FBI unit is responsible for many tasks. The FBI might become a bottleneck and degrade overall performance if Scratchpad is heavily used. However, the Scratchpad might in some cases be useful to efficiently communicate with the StrongARM as it has low latency and locking functionality. The atomic increment operation is also useful for counters shared by several threads across many microengines.

- *SRAM*: The SRAM is an off-chip unit that provides an interface for medium latency memory. Its size is maximum 2048K long-words, which is a total of 8MB. It has, like Scratchpad memory, bit-test-and-set and bit-test-and-clear operations. In addition, it supports eight *Last In First Out* (LIFO) linked lists. These can be used as stacks with atomic push and pop operations across multiple threads. It also has a generic synchronization primitive for locking and unlocking memory. Locking can be combined with a read operation, and unlocking can be combined with a write operation.

- *SDRAM*: The SDRAM is also off-chip. The unit does not provide any atomic bit operations, but it has the unique possibility to move data to and from the FBI, without going through the microengines. It has a maximum of 32M quad-words (64-bit), i.e., a total of 256MB of memory. What is special about the SDRAM is that it must be addressed one quad-word at the time.

As the latency, bandwidth and hardware supported operations differ depending on the memory interface, they are also suited for different usages. Scratchpad, as mentioned is good to use for storing few bytes that are accessed often or shared by threads. SRAM is also good for sharing somewhat larger amounts of data, e.g., lookup tables. SDRAM is typically where packets are stored, as it has the capability to directly move incoming and outgoing packets to and from the FBI unit.

4.4. IXA application development

Developing applications for the IXA environment is significantly different from general purpose programming, due to the Microengine architecture. The special instruction set, the threading policy and the memory interfaces mean that the developer cannot abstract himself from the hardware architecture. Another distinction is the need for asynchronous behavior throughout the system. Sequential programming with blocking I/O requests and blocking intercommunication, will quickly degrade performance, due to memory latencies. Therefore, communication between components use asynchron-

ous message queues and threads yield when accessing memory.

There are two programming languages available when developing applications for the IXA, the microcode assembly [Intel02b] and the microengine C [Intel02c]. The microcode is plain instruction coding with some handy preprocessor constructions. For instance, while clauses and inline macros are available to ease the development process. Microblocks are low-level with full control of hardware and fast execution. The microengine C, or Micro C, is on the other hand a high-level C-like language. Micro C is not as fast as microcode, but require less in-depth knowledge of the hardware. Both languages have a set of library functions supplied by Intel, but their support for Micro C is, at the time of writing, limited to certain platforms and the library contains less functionality than for microblocks.

Intel supply a *Software Development Kit* (SDK)[14] to aid the development of applications for the IXP1200 platform. The SDK from Intel [Intel01a] contains compilers, libraries, different tools to manage bootstrapping and deployment and a run-time framework. Their libraries hold functions to ease common network processing tasks, such as checksum calculation. Also, they advertise to have been designed for portability between different IXP processors as their instruction sets might differ slightly.

The *Active Computing Element* (ACE)[15] model is a central abstraction within the IXA Framework. It is the base of every IXP application using the Intel framework. ACEs can be used to realize processing element pipelines. Either as a high-level model or as implemented components. Each ACE is a single instance in a processing pipeline. ACEs are interconnected to each other to complete pipelines. The pipelines may have several branches, which means that an ACE may have several outputs. Packet processing is henceforth performed in each ACE. This is illustrated in Figure 4.7. The ingress ACE receives the packet. The processing ACE may classify and modify before the egress ACE transfers the packet to the a designated output port.

Figure 4.7. ACE processing pipeline

Figure 4.7 is a simplified model of an ACE pipeline. The model gives no information of how the ACEs are realized. The following defines the different types of Intel ACEs:

- *Conventional ACE*: This type of ACE, also called *standard ACE,* is not using hardware to accelerate its performance but runs solely on the StrongARM.
- *Accelerated ACE*: An accelerated ACE, also called *microACE*, employs hardware to accelerate its performance. The microACE uses the microengines as means to accelerate themselves. They are split into two components: a core and a microblock.
- *Library ACE*: This kind of ACE usually have a well-defined interface. Other ACEs may use this interface. Library ACEs may be both accelerated by hardware or not.

A microACEs is constructed by two components; a core component and a microblock component. Figure 4.8 shows the two tiers and the possible communication paths. The core component of an ACE runs as a user-space process on the StrongARM CPU. It typically administers the microblock, by loading it into the instruction store and by handling runtime exceptions. It may also take part in the packet processing. The microblock component is running on one or more of the microengines, together with other microblock components or alone. It performs one or several parts of the networking processing tasks, e.g., classification or modification.

[14]Other SDKs exist, e.g., from Radisys, but this is actually just an extension of the original SDK from Intel.
[15]This acronym is sometimes expanded as *Action/Classification Engine* or even *Active Computing Engine*.

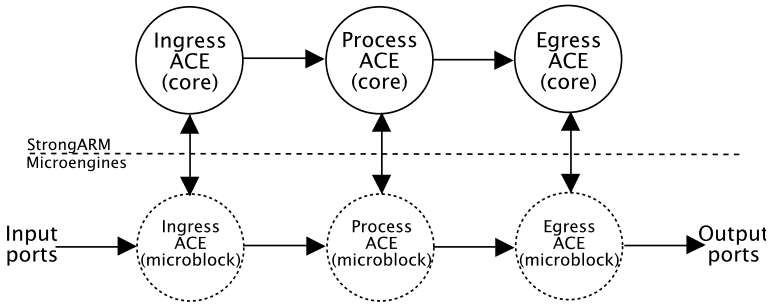


Figure 4.8. Core and microblock components of an ACE processing pipeline

To be run, microblocks must be assigned to a microengine. As mentioned, it may occupy the engine alone or share it with other microblocks. The process of deciding what microblock is to reside on what microengine is called partitioning. One partition is called a *microblock group* in the Intel documentation. They may also be replicated across more than one microengine, e.g., the same microblock group may be run by more than one microengine.

Partitioning which threads to run on which microengines is a challenge that makes programming NPUs an art. Kulkarni et al [Kulkarni03] measured that inefficient partitioning could impact the throughput by more that 30%; a better allocation of components and arbitration of resources can increase the throughput by at least 10%. They say: “Partitioning of functionality onto processors and threads is an important first step towards achieving an implementation that meets the required performance metrics. The programmer has to match properties of the application with characteristics of the underlying hardware.” This statement further implies that hardware knowledge is imperative in order to develop efficient programs for NPUs. They also claim that few tools exist to support simulation of partitioning. The IXA SDK [Intel01a] comes with a workbench that have such a simulator, and it is possible to verify how threads and microengines interact[16]. Other challenges presented by Kulkarni et al is for example what the programming effort is to reach a certain performance criteria. For the IXP1200, 32% of their code deals with internal communication, such as bus and memory access, while 63% is calculations.

ACEs can be independently developed and are bound together at run-time, i.e., *late binding*. What late binding means in this context is that nothing is known about the neighboring ACEs at compile time, but these targets are resolved when the whole application starts.

The Intel IXA SDK comes with tools that automatically load the microcode images onto the appropriate microengines. This configuration tool is largely restricted by the fact that only two microcode images can be loaded, one on the receiving side and one on the transmitting side. The receiving side is what Intel calls the ingress and the transmitting side is the egress.

In order for ACEs to communicate with other programs, a mechanism called crosscalls is available for the Intel IXA SDK. This is a IPC mechanism, that allows core ACEs to export interfaces that may be called by other applications or core ACEs. Other applications or ACEs may also register to receive event notifications. The crosscall mechanism is a handy way to configure ACEs at run-time and to share state between applications and the underlying ACE framework. Crosscalls are very similar to OMG's CORBA standard [OMG04], but it is in no way compatible with it. The main similarity is that crosscalls also use the *Interface Definition Language* (IDL) to specify the interfaces that ACEs are to export. The IDL code is handled by a stub and skeleton generator to create the client and server code.

Packet reception in the IXA SDK is performed by a set of microACEs. The realization of a microACE needs an initialization microblock and a processing microblock and Intel has developed what they call the Ingress ACE for this purpose. As the IXP1200 card used in our case has four Ethernet interfaces the Ingress microblock to use is the EthernetIngress. Regardless of what ingress implementation a solution chooses to use, packet reception follows the same pattern.

[16]The implementation done in this thesis has not utilized this tool as of yet. It might be useful later, to find optimizations for our solution.

4.5. ACE run-time framework

As one of the goals of the IXA application development process is to allow re-usage of code, ACEs can be independently developed. This, as mentioned, requires a run-time where they can be bound together. Also, as microACEs have one component running on the StrongARM and a microblock running on one or several microengines, support for loading the microcode and initializing the ACEs must be given.

The Intel IXA SDK supplies such a run-time framework for deployment of ACEs. It consists of user-space programs, kernel modules and run-time libraries. The four main components are:

- The *Object Management System* (OMS) component is a set of user space processes, that allow binding and resolving names to the underlying system as well as a IPC mechanism called *crosscalls*. The OMS has three processes: the resolver, the name server and the communications manager.

  The resolver is used to create ACEs, delete ACEs and bind named targets in an ACE to specific destinations. This is the binding of input from one ACE to another. This is indicated by the arrows in Figure 4.7. Basically, this means the resolver sets up the data flow through the core components of the IXA application.

  The name server manages the correspondence between names and the internal representation of an ACE. A programmer may name an ACE and the name server will store the relation between the string and the handle to the object.

  The *Communications manager* manages the crosscall mechanism the distributed objects use to communicate. This allows core ACE components to communicate with regular programs that run on the StrongARM. Say a configuration application or a process that retrieves information from the ACE at run-time.

- The *Resource Manager* is a loadable kernel module that handles communication between the microengines and the StrongARM. It loads software into each microengine at startup and manages the message queues when messages propagate from the StrongARM component to the microblock component or vice versa. As the microengines and the StrongARM use different addressing schemes the resources manager also provides an API to map between such addresses.

- *Operating System Specific Library* (OSSL) is a shared library that supports portability. Instead of using calls to, for example Linux specific functions, calls are made through the OSSL. It is basically an adapter so that code may run on other operating systems as well.

- The *Action Services Library* (ASL) is a kernel module. It provides run-time support for core components such as an event processing loop and network libraries. The library defines an API to provide entry points to initialize and start the ACEs. It provides finalize primitives to terminate and clean up when execution halts.

Figure 4.9 shows three micro ACEs bound toghether at run-time. Two microblock groups are loaded through resource manager API and are the assigned to one microengine each. The core ACEs are loaded on the StrongARM as user processes within the ACE framework, and communication paths are setup towards the microblocks through the resource manager. The intercomminucation between the core ACEs are are set up by the OMS reslover.

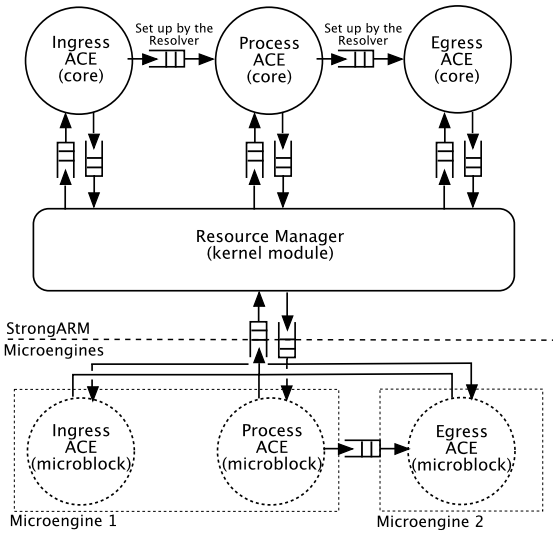


Figure 4.9. ACE intercommunication

4.6. Related work

Even though the NPU is a relatively new technology, there has already been much research in the field (see for example www.ixaedu.com/resources). Most of the effort has been looking into how well NPUs may support router applications (see below). Other application areas are also suggested and tested, such as on-the-fly Video-quality adjustment [Yamada02], active networking [Kind03] and use of NPUs in conjunction with proxies to support distributed gaming [Bauer02]. The research we have reviewed is mostly based on the IXP1200, but some deal with network processors in general, or are applicable to NPUs in general. In the early work of this thesis, we also reviewed some work done on older NPUs.

Some research is dedicated to how well a programmer can allocate resources within an NPU for specific applications. I/O engine allocation, also called partition [Kulkarni03] is, as mentioned, a challenge for NPU programmers. Also, comparisons between different hardware architectures within NPUs is also a subject that is critical to understand in order to choose the right NPU architecture for an application. The different *Processing Element Topologies*[Gries03] deals with how many stages processing is pipelined in to complete a set of functionality. It is shown that depending on what the processing does, different topologies have different effects.

In [Mackenzie03] the four 100 Mbps ports on an IXP1200 NIC was exposed to either the host kernel or to user level applications by implementing firmware on the IXP1200. They estimate that only 40% of the resources was utilized on the IXP1200 board, and that this enables application specific processing in addition to their solution.

4.6.1. Network layer

Relative to the subjects we have reviewed, a significant amount of research have been made on network layer applications, i.e., router applications. Ranging from how well hardware-software routers scale to their aggregate data-rate to how routers may be extended and fit into active networks.

A group of scientists at Princeton have done much work in this area. In [Spalink01], a hardware accelerated software router was implemented and evaluated using the IXP1200 Evaluation Board. Utilizing a three-level processor hierarchy in their solution, the GPP, the StrongARM and the microengines, they achieved extensibility, a high forwarding rate and robustness. The router could be exposed to new functionality at any of the three layers, without degrading robustness or forwarding performance.

This work resulted in an extension to the router architecture, called the *Virtual Extensible Router Architecture* (VERA) [Karlin01]. The aims of the architecture is to bridge the gap between new services for IP routers and the underlying hardware. VERA defines three layers to provide the necessary abstractions. The upper layer, called the router abstraction, is where extending features can be plugged into the system. The lower layer is the hardware abstraction, that hides details about the hardware and finally a middle layer is a distributed OS that ties the two other tiers together.

VERA was also used in another project [Shalaby02] to make an extensible router for *Active Networks* (AN). In ANs, packets are not passively forwarded, but code carried in packets can actively influence the forwarding process in routers. The main addition to VERA in this paper, was how the control-plane was handled in order to support active injection of arbitrary code.

AN is also discussed in [Kind03], where they argue that the performance of secure and manageable AN approaches can be addressed by NPUs. Their prime argument is that injection of code requires significant data-plane processing, which suits the design of NPUs. They also offload AN functionality onto an NPU, and claim that it is done without sacrificing the safety of traditional IP networking.

An example of router functionality loaded onto an NPU is the implementation of a DiffServ edge router on the IXP1200 [Lin03]. Here, a set of benchmarks were run to determine bottlenecks in the implementation. It was shown, that under different conditions the implementation would experience possible bottlenecks both in SRAM access and in microengine performance. To solve these issues they suggested changes to the hardware, such as faster SRAM units.

Research efforts have been made on how to share load between several NPUs distributed within a router [Kencl02]. This was done using a feedback control mechanism to prevent processor overload and adaptive scheduling of the incoming traffic. Another area traffic policing for servers and routers in order to stop misbehaving clients and *Distributed Denial of Service* (DDoS) Attacks [Mirkoviç02].

4.6.2. Multimedia streaming

As the subject of this thesis is offloading multimedia proxies, it was interesting to find other work in this area. To our knowledge no work exists, that overlap with ours, but suggestions are made that NPUs can be used to offload multimedia applications in VoD servers, high-definition video clients and in multimedia editors. For example, an NPU was used to support a high-definition video client [Fiuczynski98]. This solution had the NIC process video data and transfer it directly into a framebuffer device. This way the host was offloaded while the client viewed the data.

Also, NPUs have been evaluated in a on-the-fly adaption of MPEG-2 streaming scenario [Yamada02]. By implementing a low-pass-filter on an IXP1200 chip, utilizing the microengines and the StrongARM to do the work, Yamada et al showed that when using all 6 microengines for video filtering, they were able to process streams up to a rate of 1.35 Mbps.

4.7. Summary

In this chapter we have given an introduction to NPU technology and a specifically described details about the IXP1200 NPU. NPUs provide the network programmer with a mix of speed and flexibility. To program the IXP1200 knowledge about hardware is needed and tasks must be carefully allocated to the appropriate processing unit.

Next in this thesis, we will make use of the IXP1200 chip to design, implement and evaluate a prototype that aim to offload the control and the data forwarding of an MoD proxy.

Part II. Design, implementation and evaluation

Chapter 5.

Design and implementation

> Everything should be made as simple as possible, but not simpler.
>
> —Albert Einstein

> Power at its best is love implementing the demands of justice. Justice at its best is love correcting everything that stands against love.
>
> —Dr. Martin Luther King Jr.

5.1. Design

When designing an MoD Proxy, the things to consider are the properties of caching, the architecture the proxy is to be inserted into, and the caching strategy to use. This also includes the replacement algorithm of the cache. The data path through the proxy must also be considered, so that good throughput can be achieved. Other considerations have to be made if the proxy is to perform other tasks for the server or the client. Possible tasks include re-encoding, adapting or encrypting the stream passing through the proxy. Access control, overlay-multicast, traffic engineering and protocol translation are also mechanisms that may be performed by a proxy. Also, when designing a proxy for a platform, the characteristics of that specific platform must be considered. This chapter will describe the design choices of our prototype.

5.1.1. Design goals

The goal of this design is to enable the implementation of our prototype for our specific architecture, namely the Intel IXP1200 NPU [Intel01b]. We will also attempt to design the proxy in a manner so that it easily may be used for testing and comparison on other platforms. The design is founded on a component view of the proxy. This should give us the choice of how tightly bound every component will be in the resulting system. What this means, is that no hard separation of the components are required. Also, components can be used to extend the design at a later point in time. In this way, the design will provide a framework for further work. The design should enable taking advantage of the properties that the IXP1200 has, i.e., it should enable us to utilize the processor hierarchy. Thus, the functionality can be split up and be run on the appropriate processing unit, ranging from the host CPU down to the microengines of the IXP1200. The design should lastly allow the appropriate communication channels between the separated layers.

5.1.2. Component overview

To utilize the processor hierarchy of the IXP1200, we have split the proxy into several logical components. These, as shown in Figure 5.1, are separated in two main layers; the control-plane and the data-plane. This decision is founded on the observation that the MoD proxy employing RTSP/RTP has a distinct control-plane and data-plane (see Section 3.4). The separation of the two layers is

needed for other reasons as well. The data-plane in an MoD schema utilizes RTP over UDP, so it must either be embedded asynchronously in the RTSP component, or it must run as an independent thread or process. This makes the design valid on a regular platform as well as on the IXP1200. The separation is even more distinct on the IXP1200, as we will run the RTP components bound together in an ACE to make use of the processor hierarchy. Not shown in the figure is the classifier component. This is a special component needed to direct packets that are received by the microengines. The functionality of the classifier component would be handled by the network stack on a more conventional platform, such as Linux or BSD.

The figure also shows a caching manager and a session manager component. These components are basically components that add MoD functionality. The session manager is needed to store sessions and communicate this information between the control-plane and the data-plane. The cache manager component will be storing media that flows through the data-plane. A signaling mechanism must also be present between the cache manager and the session manager to know what data to cache and not. It should also handle cache replacement.

There are several other components that would fit into the design of an MoD proxy. Re-encoding, adaption or encryption components are possible relevant extensions. What characterizes these components is that they require significant processing power. Implementing them onboard the IXP1200 would probably deplete the resources available to process other traffic at high speeds, even though it is shown that the IXP1200 was able to adapt a video stream using the StrongARM and the microengines [Yamada02]. An option is to allow this kind of functionality to be allocated to the host CPU as it usually has more cycles to spare, if successfully offloaded by an IXP card.

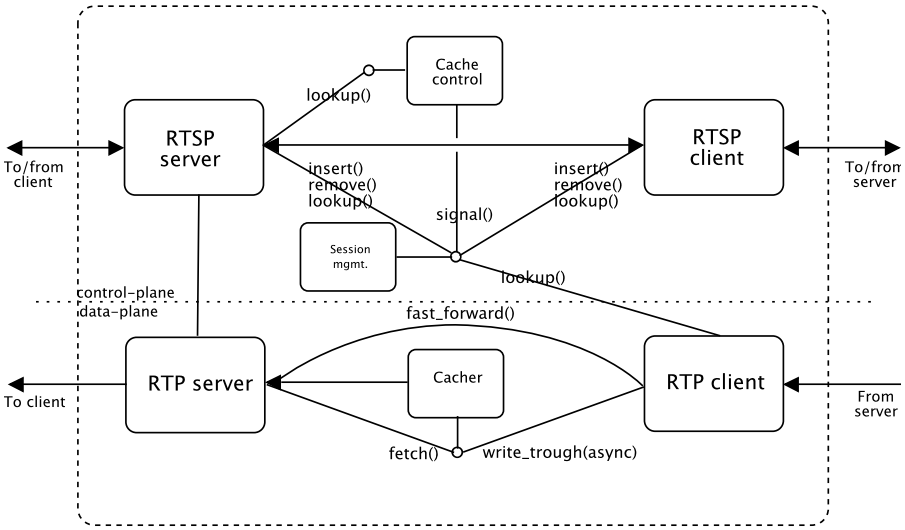


Figure 5.1. Proxy components

In addition to the control-plane and the data-plane, there is also a management-plane. This plane consists of the Intel configuration tools and our own management tool. What the management plane does is basically conveying the information needed at startup for the other planes.

5.1.3. Caching

The caching is designed to be independent of any replacement algorithms or caching strategies. The design is more concerned with enabling an implementation to efficiently transfer the data that will be cached. This means that the design is not focusing on the cache mechanisms, but on the ability to efficiently cache data on the fly or quickly fetch data from the cache.

The caching component is partly in the control-plane and partly in the data-plane as shown in Figure 5.1. The RTSP server will do lookups through the cache control to see if requests can be served entirely or partially from the cache. Also, the cache control must be able to signal the session manager

about when to start caching or stop caching of the data transfered in the session. Any replacement logics and strategy logics will be in the cache control. This logic could be embedded in pluggable subcomponents so that caching strategies can easily be changed.

The data-plane part of the cache component is where data flowing through a stream will be written to the cache storage and at the same time forwarded to a client. Achieving good throughput is important; unnecessary delay on the stream must be avoided and resource usage must be kept low within the proxy. Stream data that is not going to be cached will simply be fast forwarded or buffered if it has been prefetched.

5.1.4. Fast forwarding

Whenever the cache control is satisfied, i.e., it has cached the amount of data that is required by the cache strategy (see Section 3.2.2), the remaining data in the stream will be forwarded to the client without being sent to the cache storage. This also applies when the cache control has decided not to store any data elements of a stream. As illustrated on Figure 5.1, the cache is bypassed in these cases and the packet is forwarded directly.

5.1.5. Summary

We have described a basic MoD proxy design. It includes the components needed to give the proxy the most common functionality. Next, we will describe how we have implemented these components in our prototype.

5.2. Implementation

The prototype implements all the components of the design described in the previous chapter, with the exception of the caching component. This includes the classifier, the RTP and RTSP components along with the session manager. The prototype is implemented as a set of inter-operating components using the Intel IXA framework and the embedded Linux operating system.

The design is realized by developing an RTSP proxy and a Classifier/RTP-forwarder ACE. These executables are then assembled together with Intel ACEs to realize the prototype.

5.2.1. Limitations and assumptions

The implementation of the prototype is largely limited by the time frame of the project. The prototype implements the most basic functionality so that results can be collected, and so that the proof-of-concept can be shown. However, the prototype is far from being a fully fledged system, and we have made the following limitations and assumptions:

- The implementation is limited to deal with performance issues and will focus on using the abilities that NPUs like the IXP1200 have to enhance these properties. In particular, the prototype aims to offload the proxy host in the forwarding of RTP/UDP packets and thus reduce the networking processing needed by the proxy host.

- The prototype will neither perform caching, i.e., transfer of data to the host for storage, nor does it implement cache replacement algorithms, but the proxy is prepared for a cache extension. Caching objects is regarded as the prime functionality of an MoD proxy, so we will discuss how it can be implemented in an efficient manner later (see Section 6.6.1).

- No strong assumptions are made about the caching strategy, but for a system streaming large media objects with high data-rates, it is unlikely that the cache will hold any of the objects in their entirety. Therefore, strategies like prefix, segment or layered caching are considered as the most likely ones (see Section 3.2.2).

- The implementation assumes that only one data source exists. This means that it does not support any distribution mechanisms to inter-operate with other proxies on their same level, i.e., meta-information distribution between proxies or overlay network routing. It may, however, be part of a hierarchical distribution architecture where the data source assigned to it may be another proxy (see Section 3.3).

- Although the proxy is implemented to support multiple streams in parallel, no tests have been made to verify the actual support.

5.2.2. Overview

The design described in the previous chapter may be realized in different ways. As the components of the proxy have different tasks to solve, their respective implementations are based on the characteristics of these tasks. The control-plane tasks need relatively more complex processing compared to the data-plane tasks. Therefore, the RTSP processing is implemented as a regular executable in the StrongARM GNU/Linux environment. The data-plane processing is quite simple, but as the data-rates are potentially high, it needs to perform well to yield high throughput. The RTP/UDP packet processing are therefore implemented as an ACE using the Intel IXA framework.

The separation between the control-plane and the data-plane is again shown in Figure 5.2. The figure shows an outline of how the packets will flow through the proxy. Every packet will, after they are received, be classified. A simplified view of this classification will basically result in one of two possible actions. The packet will, if classified as a TCP packet, be sent to the *slow path*[17] as a potential RTSP packet.

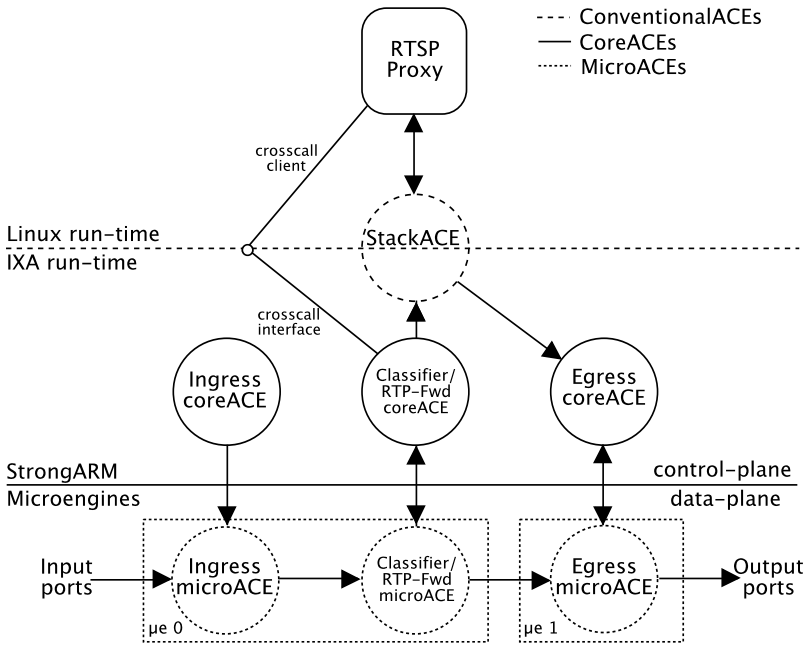


Figure 5.2. ACE layout

If it is classified as a UDP packet, a lookup will decide if it is a RTP packet that should be further processed by the microengines, e.g., *fast path* processing. The Egress ACE will be responsible for moving packets to the physical layer. The classification process is fully explained in Section 5.2.6.

As explained in Section 4.4, the IXA platform makes use of ACEs to create processing pipelines. Figure 5.2 also shows this concept with regards to our prototype. The figure is not entirely correct in this

[17]As explained, the *slow path* is processing performed by the core ACEs or other programs running on the StrongARM.

sense, as the Intel Ingress ACE only controls the input/output interfaces, by configuring their settings, e.g., MAC and IP addresses. The Ingress ACE is thus not involved in the processing of packets. Nevertheless, the classification and forwarding ACE implements the functionality to classify every packet, and also the RTP forwarder.

Finally, the figure shows the Intel stack ACE. This ACE is used to make all the network interfaces of the IXP1200 accessible from the StrongARM through BSD sockets.

5.2.3. The RTSP components

To simplify the inter-operation between the RTSP server and client proxy-components, they are implemented as a set of C-functions that are compiled to one executable. This means that they are only separated logically, but share the same address space in memory. This is done due to the fact that whether a host will act as a proxy or as a server will most likely be a build-time decision, especially since the proxy, to fully utilize this solution, also needs an NPU. It simplifies the synchronization needed when accessing shared memory. For instance, if using mailboxes for message exchange, the access to these must be synchronized to guarantee consistency. It also minimizes the processing overhead compared to having any other IPC mechanism between them, e.g., CORBA. The logical separation between them could enable re-usage of the components to build a stand-alone server or a client.

A session and cache management component is also linked together with the RTSP components. These two implement common interfaces so that their behavior may be changed. How they work in our case is described later.

The RTSP proxy uses a traditional socket interface to the network. First an attempt was made to implement TCP functionality inside an ACE, but this became unnecessarily complicated and too time consuming. The only benefit of such an approach would be to lessen the inspection of headers in the packets that are propagated through the stack. This could have implications on a solution that relies heavily on TCP as a data transport protocol, but in a case such as this, where the majority of the data transfer is handled using UDP, this overhead is negligible.

The proxy will handle any request classified (see: Section 5.2.6) as an RTSP packet in a separate thread of control. This is to ensure a non blocking service, so that several requests may be processed in parallel. This involves synchronization when inserting, updating or removing values in shared memory and is handled by a simple mutual exclusion from the shared data. This is not considered to be too complex compared to the benefits of enabling multiple requests being processed in parallel.

Although the server and client component are separated logically, the only time the proxy needs to use the client side, will be when forwarding an RTSP request. For simplicity, this forwarding process is done in sequence, meaning that the proxy worker thread will block while waiting for the server to reply.

In general, every request and response to the RTSP proxy is a character stream that first is parsed to a memory representation. Every supported RTSP request method is represented the same way as shown in Example 5.1.

Example 5.1. RTSP request type

```
typedef struct
{
    char *method;
    Url_t *url;
    char *version;
    Dict_t *headers;
} RtspRequest_t;
```

Every response is represented by the type shown in Example 5.2. As the number of headers may vary in every method, they are put into a lightweight dictionary. The dictionary allows inserting, looking up, removing and copying header items. After successfully being parsed into a memory representation, the request is processed by the appropriate method handler.

Example 5.2. RTSP response type

```
typedef struct
{
    char *version;
    int status;
    Dict_t *headers;
} RtspResponse_t;
```

Figure 5.3, shows the sequence of events during an RTSP session involving a client, proxy and server. As shown, the following steps are performed by the proxy during the session:

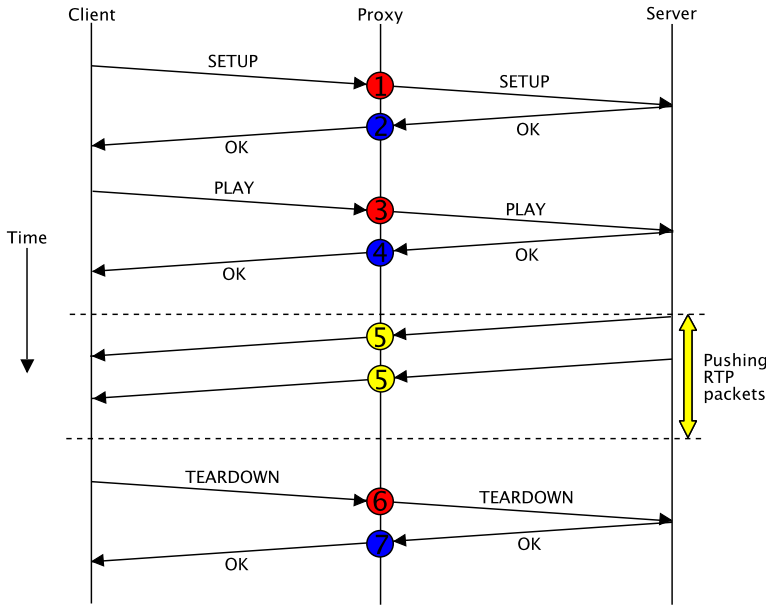


Figure 5.3. RTSP proxy session

1. After parsing a SETUP message received from a client, the proxy will consult its cache manager component to see what action to take. If the request can be satisfied directly, i.e., the whole object is present in the cache the proxy will act as the server and no forwarding of messages will be needed throughout the session. How the proxy is to obtain knowledge about whether it holds the whole object or not can be obtained through fields in the DESCRIBE message. Another way could be to delay the SETUP towards the server until a PLAY request comes to the proxy. The savings of doing this is minimal, so it might be better to always forward the SETUP message to the server. Initially, the proxy will have no objects in the cache so the request must be forwarded to the server. If the request could be partially served from the cache, the request will also be forwarded, as a range of data is likely to be fetched from the server later on. Before the proxy can forward the request to a data source, certain transport parameters must be set. The proxy must tell the server on which port it wishes to receive the stream. To know this it must consult the un-

derlying layer to allocate a free port. If the port is successfully allocated, the proxy needs only to set it in the transport header of the RTSP request along with the correct server *Uniform Resource Locator* (URL). The proxy then blocks while awaiting the server response.

2. When the proxy receives a verification message for the forwarded SETUP message, it means that the server itself has set up a session to hold the state between the proxy and the server. A proxy session is created to hold the relation between the server side session and the client side session. A session corresponding to the server can be set up after the response from the server is parsed and the session corresponding to the client is created with the client session through the session management interface, described in Section 5.2.4. A server side session is also created to hold the state between the client and the proxy. The proxy session will set the “forward” flag before a response is built and sent to the client.

3. When the client requests to play back a range of an object, the RTSP component will first locate the proxy session that the message belongs to through the session manager interface. If a session exists, the previously set forward flag will be checked to determine whether the PLAY message must be forwarded or not. If the flag is set, the message is simply forwarded to the data source. As when forwarding the SETUP message, the proxy will also block while awaiting the PLAY response from the server.

4. If the server successfully responds to the PLAY request forwarded from the proxy, a response will be built based on this reply and it is then sent directly back to the client.

5. The session now enters the playing state, and the server starts pushing RTP packets. This stream is handed to the RTP forwarder running on the microengines. The implementation of this functionality is described in Section 5.2.7, but what happens is basically fast forwarding of the packets towards the client.

6. When the stream is finished, the client will initiate a TEARDOWN of the session. This message will cause the proxy to do a lookup on the session. In this way it can locate what session corresponds to the server. It will then send a TEARDOWN to the server.

7. When the client session from the proxy to the server is torn down, the server side session is removed through the session manager interface, and a response is sent to the client.

5.2.3.1. Conformance issues

The RTSP standard [Schulzrinne98] is a detailed specification, and it is not possible, due to time constraints, to make the prototype 100% conformable. It only implements the methods described as *required* by the standard, i.e., OPTIONS, SETUP, PLAY and TEARDOWN. The methods described as *recommended* or *optional* like DESCRIBE, PAUSE or RECORD, are not supported. Other details are also left out, as described below.

For the actual data transfer the implementation only supports unicast RTP. No support for split audio and video streams are implemented. Further, the raising of error messages to the client must be revised and sanity checks must be added to the parsing of RTSP messages.

The implementation only supports full playback of a range, this means that PLAY requests are not queued as required by the standard. Instead, if a client attempts to gradually playback ranges, as in, say a presentation, the error message “Method not supported in this state” will be sent back. Not only is this due too the fact that it is too much work within our time frame to implement this feature, but also because this partly is irrelevant when streaming movies. This is based on the assumption that people watch movies sequentially, from the beginning to the end. If PLAY requests were to be queued many “unnecessary” RTSP packets would have to be transfered through the network.

5.2.4. Session management

The session management component is constructed to internally handle the relationship between the client and the server. The proxy effectively needs two sessions, one to the client and one to the server. Each session, as shown in Example 5.3, contains the information needed to correctly fulfill the communication throughout the lifetime of the session. Some of the fields of the RTSP session are needed for RTSP control communication along, i.e., id, state, cseq and the socket descriptor. The other fields are used by the classifier and the RTP packet forwarding.

Example 5.3. Session object

```
typedef struct
{
        uint64_t id; // Session id
        char state; // Is INIT, READY or PLAYING
        uint32_t cseq; // Current sequence number
        uint32_t ip; // Ip address of server or client
        uint16_t client_port, server_port; // UDP transport
        int socket; // Connection handle
} RtspSession_t;
```

Whenever a session is initiated by a client that needs data from a server, the proxy will map the two sessions together in a proxy session. This session looks like the type defined in Example 5.4. The id of the proxy session object is the same as that of the session to the client. This is a slight restriction to how session management is handled, as there is no relation between the server messages and a proxy session. This is why the proxy blocks when sending messages to the server. The restriction is easily solved by allowing lookup on both client and server session identifiers.

Example 5.4. Proxy session object

```
typedef struct
{
        uint64_t id;
        unsigned char cache, forward;
        RtspSession_t *client, *server;
} ProxySession_t;
```

The session manager is responsible for conveying information to the classifier/forwarder microACE about active sessions. It does this through a crosscall interface (see Figure 5.2) which is exported by the core component of the classifier/forwarder ACE. The same interface allows for removals of sessions when they are torn down. The server side of the crosscall will manage the allocation and set values on inserts and free memory on removals.

5.2.5. Cache management

The cache management component is only a dummy component in the system. It implements an empty interface, so that the RTSP server component may perform the logic required to support caching in the future. For details on how to implement caching in the data-plane see Section 6.6.1

5.2.6. Packet classification

In order to control the flow of incoming network packets that arrive at the proxy, they must be classified. To do this, different rules are defined, so that there is a precise way to handle each packet quickly. In this section, we describe these rules and how they are implemented.

The classifier is the driver of the proxy, every packet destined for any of the four configured MAC addresses will arrive here. These packets are then classified and forwarded internally within the proxy, according to certain rules. The classification process is implemented as a part of an ACE. This ACE has a microblock component and a core component working together. The microblock is responsible for the main processing. It also performs forwarding of packets, either to another microblock or to its core component by emitting exceptions. The core component then examines the exception code to determine what the next target is.

In order not to impose huge restrictions on our solution, TCP and UDP packets that fail to classify as RTP are not simply dropped, but they are forwarded to stack ACE. This allows running other services outside the scope of this application, and allows the proxy to run helper daemons. For instance, an inter-proxy daemon could then be loaded in Linux, without having to change the classifier. Currently the classifier only attempts to classify UDP packets as RTP, so in order to support interleaved streams in the future the classifier must be extended. There are problems related to this though. Other applications using UDP or TCP can then accidently be bound to the same port numbers that are defined as RTP ports. To avoid such conflicts, the RTSP proxy must bind the ports through the stack as well, so that they are off limits to other processes. At the same time it will also check it the port already is used by another process.

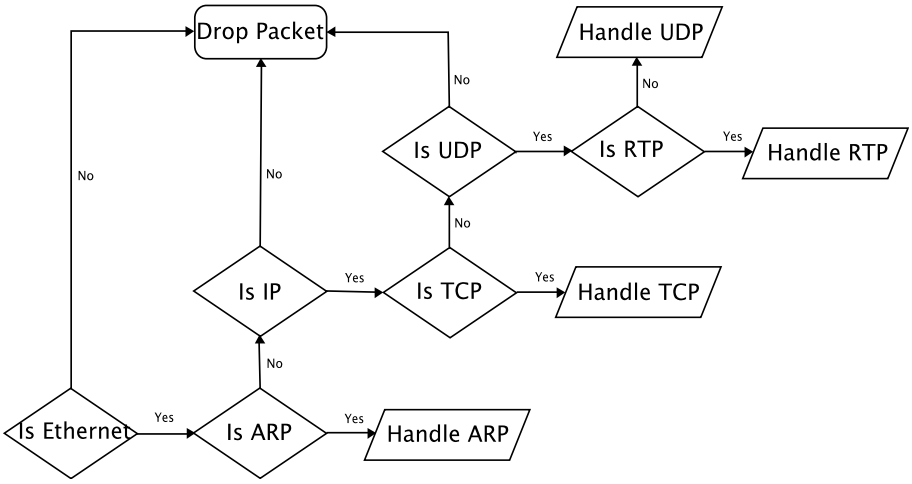


Figure 5.4. Packet classification

The flow chart in Figure 5.4 shows the destinations for packets received by the prototype. The only application layer classification is that of RTP packets. RTP packets are to be further handled by the RTP forwarder and any future cache ACE. Other UDP packets, TCP packets and ARP packets are sent directly to the stack through the core component. This happens without any application layer classification. ARP packets are handled directly by the Linux kernel, while TCP and UDP packets are handled by services listening to their respective ports or by the kernel, e.g., RTSP packets are handled by the RTSP proxy and TCP acknowledgments are handled by the Linux kernel.

5.2.6.1. Classification rules

The implementation is restricted to working with frames of *Ethernet* packets. No support for other link-layers such as, *Asynchronous Transfer Mode* (ATM), are implemented. This is due to the fact that the IXP1200 NIC, the ENP-2505 only has four *Ethernet* ports, other cards however may provide ATM or even optical interfaces. In order to handle the *Ethernet* packets, the microblock part of the

classifier relies on macros provided by the Intel SDK. To support different link-layers one would need *dispatch loop* macros to handle them. These would probably be supplied by the vendor, or they would have to be written. These macros would then have to be compiled together with the ACE that wishes to retrieve such link-layer frames.

Since we already know that we are receiving *Ethernet* frames, the first check that must be done is whether we have an ARP or an IP payload. As shown in Example 5.5, this is done by checking the *frame type* field of the Ethernet header. If it has a value of 0x0806, we have an ARP packet. The classifier then sets the appropriate exception code and forwards the packet buffer to its core component. First, a dedicated target was implemented as the handler for ARP packets. However, as the decision was made to make use of the stack ACE provided by Intel, ARP packets are now forwarded there. If the classifier fails to recognize the packet as an ARP packet, it continues execution by checking the same field of the *Ethernet* header to see if it is an IP packet. The *frame type* of an IP packet is 0x0800, if this comparison fails as well, the packet is dropped, thus restricting support for any other network layer protocols. This behavior is not regarded as a restriction, as IP is the main network protocol in todays Internet.

Example 5.5. Simple branch classification of Ethernet type

```
.local etype sd_base
  ;; Load 0x0806 in the GPR etype
  immed[etype, ETH_ARP]

  ;; The value of the packet header from SDRAM
  ;; is shifted 16 bits, then subtracted from etype.
  alu_shf[--, etype, -, $$hdr3, >>16]

  ;; If the result of the alu operation is 0.
  ;; branch to the label ARP_Exception
  br=0[ARP_Exception#]

  ;; Load 0x0800 in the etype general purpose register.
  immed[etype, ETH_IP]

  ;; Shift and subtract
  alu_shf[--, etype, -, $$hdr3, >>16]

  ;; If not equal, branch and drop the packet.
  br!=0[Drop#]
.endlocal

;; Continue with more IP classification
```

When the classifier detects an IP datagram, it first verifies that the destination address is equal to the configured address of the interface that received the packet. If not, the packet is dropped.

In order to verify the IP address, the microcode has to access shared memory where the configured addresses reside. Runtime patching of symbols gives the base address to the memory location from where to search for the address. Each interface has a defined number of 32 bit fields in memory where MAC and IP addresses are written on startup, or when changed by the configuration application. What address to compare with is uniquely identified by the port on which the packet arrived. This number is then multiplied with the size of the interface info-block and added to the base address.

Next, if the IP address matches, the transport layer protocol must be resolved. This is done by checking the *type* field of the IP header. If it matches the well-known bit-mask of TCP then a stack excep-

tion is set, and the packet is directed towards the core component of the classifier. The core component then directs the packet to the stack ACE for further application-layer processing.

Example 5.6. IP address validation and transport protocol classification

```
.local ipt pa
  ;; Read the ip type from the header
  alu[ipt, 0, +, $$hdr5]

  ;; Read more of the IP header from SDRAM
  alu[boff, off, +, 24] ; Offset from where to read
  alu_shf[boff, --, B, boff, >>3] ; Convert to Q-Word addr
  sdram[read, $$hdr0, base, boff, 2], ctx_swap

  ;; Check that the destination ip address
  ;; is the same as that of the input
  ;; port; to verify that this packet is for the proxy
  PolPortIp[pa]

  ;; Get the upper 16 bits of the IP destination address
  ld_field_w_clr[ip_da, 1100, $$hdr1, <<16]
  ;; Get the lower 16 bits of the IP destination address
  ld_field[ip_da, 0011, $$hdr2, >>16]
  alu[--, pa, -, ip_da]
  ;; If not IP, then drop
  br!=0[Drop#]

  ;; If TCP set exception and send to core.
  br=byte[ipt, 0, IPT_TCP, TCP_Exception#] ;; Got a TCP packet
  ;; If UDP continue processing.
  br=byte[ipt, 0, IPT_UDP, ChkPrt#] ;; Got a UDP packet
.endlocal
```

If the packet is not a TCP packet, a check is performed to see if a UDP packet has arrived. Whenever this test succeeds, the microblock will attempt to decide if the packet carries data that is part of a media stream that should be forwarded and/or cached. This decision will be based on the result of a lookup in what we have called “The Media Transport Forwarding Table”, described in the next section. If such a lookup fails, the default fall back mechanism is to send the packet up the stack to be processed without being subject to forwarding.

5.2.6.2. Forwarding table

The “Media Transport Forwarding Table” is used both by the classifier and by the RTP-forwarding components. It enables the classifier to decide which UDP packets to forward and which to send to the stack ACE. The forwarder component uses the information stored in the table to modify the packets as required.

The forwarding table is implemented as a hash table, as shown in Figure 5.5, where the port number is used as the key. There are several reasons to use the port number as the key. First, it enables the forwarder to be independent of the application layer media transport protocol, e.g., RTP, by not using any fields in that header. Also, it is more efficient not having to read the RTP header to determine the type of the packet, as this would involve more processing and memory accesses. The port number is also located at the same place in both UDP and TCP, so this will make any extension to support interleaved streaming easier.

The hash table has the final capacity of *N* entries. N is a prime that is selected to get a spread in the

hash-values and thus avoid collisions when hashing. The session manager will also try to avoid collisions by choosing transport ports wisely, so that we have a good probability of achieving perfect hashing. Collisions will though be resolved, if any, by inserting keys that hash to the same value in a linked list.

This table is used to lookup forward-information for the majority of packets that are expected to flow through the proxy. Therefore it is subject to certain performance requirements. It must therefore not add unnecessary delay by slow lookups. This is why a hash table was chosen as the primary data structure for the forwarding table.

Lookups on a “key” should yield an answer in O(1), due to a potential high number of lookups during a session. This means that a direct lookup on that key must be possible. Also, the hashing function must be as simple and fast as possible. Inserts and removals are not that critical as they will be performed more seldom compared to lookups, i.e., only when a RTSP session is successfully set up or torn down.

When choosing the hash function, we realized many ways to optimize it. First, the session manager will be able to wisely choose ports, by having knowledge about what ports that already have been assigned to other sessions. Also, since transport ports in RTSP are evenly allocated, i.e., only even numbers are used for the transport of data, while any data control ports must be odd, we were able to create a simple hashing function that was easily implemented in microcode. Example 5.7 shows this function.

Example 5.7. Hash function

```
;; Calculate the modulo of an integer in_x
;; and put the result in out_z
#macro Mod[out_z, in_x, in_y]
.local xm
        alu[xm, --, B, in_x]
Loop#:
        alu[out_z, xm, -, in_y]
        br<0[End#]
        alu[xm, --, B, out_z]
        alu[--, xm, -, in_y]
        br>=0[Loop#]
End#:
        alu[out_z, --, B, xm]
.endlocal
#endm

;; Specialized hash function macro to generate the
;;  index in the forwarding table
#macro Hash[out_z, in_x, in_seed]
.local x start
        immed[start, START_CHANNEL]
        alu[x, in_x, -, start]          ; x = x - start
        alu_shf[x, --, B, x, >>1]        ; x = x / 2
        Mod[out_z, x, in_seed]          ; z = x % seed
.endlocal
#endm

;; C-equivalent
hash = ((port - START_CHANNEL)/2) % MAX_SESSIONS
```

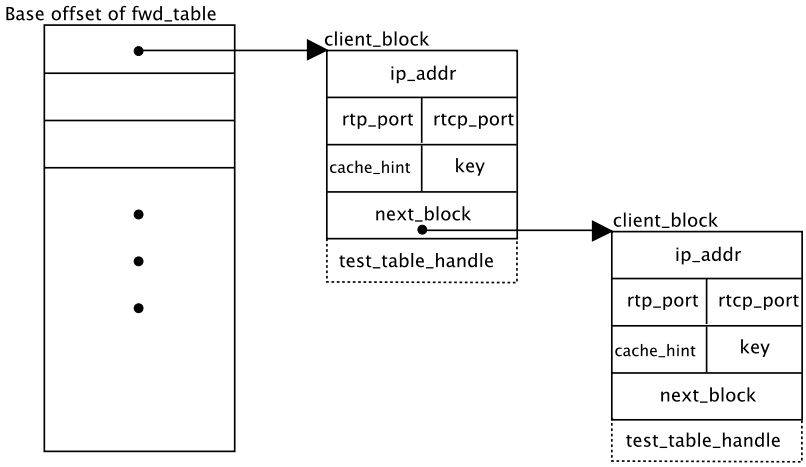


Figure 5.5. Media Transport Forwarding table

The fields in each client block hold the values which will be used to classify and forward each packet. The classifier makes use of the *key* and the *cache_hint* fields to decide the data flow. The *key* is used to verify the correctness of the hash, as values potentially may be hashed to the same index. The *cache_hint* field is used to decide whether to send the packet to the cacher or not. If this field is set it will be sent to the cacher and if not it will bypass the cache and is forwarded directly.

Lookup operations. Any incoming packet from the "data source" will be classified according to these rules: If the packet is a UDP (or TCP, depending on retransmission strategy), check the forwarding table for an entry. If an entry exists, i.e., there is an address != 0 at the hashed index + base address, go to this address and check if the key is equal to the destination port of the incoming packet. If not, a possible collision has occurred, or faulty behavior is detected. Two options to handle this are available. One could simply give up, or resolve the collision by traversing the next address until a match is found in the list. If the end of the list is reached (i.e., the next address equals zero), an error has occurred. This could be a random host sending packets into the oblivion or a malicious DDoS attack. Checking the IP to match any of those it is expected to be should resolve this by quickly dropping the packet if a mismatch is found. This relates to how many servers a proxy is helping. If there is a match, then a number copies of the packet will be made, depending on the number of clients having requested the file/stream/object, also, if the cache hint is set, then a copy should be sent to the core of the classifier for further treatment. The number of copies made is the same as the number of subsequent key matches found in the list of client blocks. The copies will be altered according to the information in the client block and then sent to the egress ACE which puts it on the correct interface.

Insert operations. When a *SETUP* message is received by the RTSP proxy, it must, if the whole object is not present in the cache, forward the request to (the server or neighboring proxy) a source holding the whole object. When doing this, it must record the client port it will use, along with the IP and whether the object should be cached or not. It will also have to insert the port that it expects incoming RTP packets on. This is not known, until the server replies to the SETUP message that is forwarded. However, in the forwarded SETUP message the proxy may announce the client port of its wish. Assuming that this wish is granted, the proxy should choose the port based on the next free address in the forwarding table. This will allow ports to be assigned without creating collisions in the hash table. If the server chooses another port that collides with an already set port, the insert operation will be slower, as the end of the list must be found before inserting the block. If other clients already have requested the data from this server a traversal of the list is also needed, before inserting the block.

Remove operations. Removing a "client block" from the forwarding table will depend on the conflicts within it. If no conflicts are present, any removal will be fast. This operation will occur every time a session is torn down or when timeouts are reached. These removals will occur in the same rate as inserts, so they need not be as fast as lookups.

5.2.7. RTP forwarding

The RTP forwarding functionality is implemented in the same ACE as the classification. This is done to simplify the code and to lower any inter-microengine communication. The forwarding can be implemented as a standalone microcode executable, and in a fully fledged system where the proxy also performs caching this might be needed. This will depend on how the caching functionality is implemented. However, when we perform this in the same microblock as the classification, we can more easily perform measurements on the performance.

The forwarding code basically uses the lookup information from the forwarding table described in Section 5.2.6.2. The close integration with the classifier, allows this to happen in just one memory access, i.e., the values needed for header modification are stored in local registers after the classifier has made the lookup. The values retrieved on a lookup are those shown in Figure 5.5. They include the client IP address and the pair of ports that are set up in the session, i.e., the RTP and RTCP ports announced by the client. Also, a cache flag is retrieved in order to know if packets are to be written to the cache.

The information needed to forward a packet includes more attributes than the information retrieved from the forwarding table. The code needs access to the MAC destination address. As our test setup is known, this information is hard coded in the prototype, but it should be implemented as an ARP-cache in a fully fledged system. This can be done by storing the destination ARP from the incoming RTSP requests in a shared table. By hashing the destination IP address to the indexes in the table, one would get a fast way to retrieve a MAC address based on an IP address. If no mapping is found in the table, an exception can be set, and the packet can be handled by the StrongARM core component, by issuing an ARP request. Another way to solve this could be by exposing the kernel ARP-cache to the microengines. Doing this would require less replication of data in memory, and would better utilize the finite memory resource.

When the forwarder has all the information needed to forward a packet, it first modifies the Ethernet header and then the IP and UDP header. The existing destination IP is written to the source IP field and the destination field is overwritten with the IP retrieved from the forwarding table. Then, after writing the new values to SDRAM, the IP checksum is updated to reflect the changes. This is done incrementally according to RFC 1624 [Rijsinghani94] and is implemented in microcode as shown in Example 5.8. What this means is basically that for every modified 16-bit field in the IP or UDP header, the old value is removed from the checksum and the new value is added. These calculations are performed using one-compliment arithmetics.

Example 5.8. Incremental checksumming

```
#macro IncrementCksum[out_newsum, oldsum, old, new]
.local sum tmp mask
        immed[mask, 0xffff]
        alu[sum, --, ~B, oldsum]        ; sum = ~oldsum;
        alu[sum, sum, -, old]           ; sum = sum - old;
        alu[sum, sum, +, new]           ; sum = sum + new;
        alu[tmp, sum, AND, mask]        ; tmp = sum & 0xffff
        alu_shf[sum, tmp, +, sum, >>16] ; sum = tmp + (sum >> 16);
        alu[tmp, sum, AND, mask]        ; tmp = sum & 0xffff;
        alu_shf[sum, tmp, +, sum, >>16] ; sum = tmp + (sum >> 16);
        alu[sum, --, ~B, sum]           ; sum = ~sum;
        alu[out_newsum, sum, AND, mask] ; return (sum & 0xffff);
.endlocal
#endm
```

The UDP destination port field is also modified to reflect the entry in the forwarding table, and the source port field is copied from the existing destination port in the header. However, the destination port is not used by the server in this case, as RTP packets are only flowing in one direction. The UDP checksum is for now set to zero in the prototype, as our point of reference in Chapter 6 also does this.

When all the fields are modified and written to the packet in SDRAM, the ACE enqueues the packet for transmission by the egress ACE.

5.2.8. Summary

Our prototype implements basic proxy functionality to show that application layer forwarding can be offloaded on to the IXP1200. Though it is not a fully fledged system for production use, it implements enough functionality to verify its usefulness and give a proof-of-concept. It successfully sets up sessions in the control-plane, it conveys information needed to perform forwarding in the fast-path of the IXP1200 and implements the forwarding functionality needed to collect measurements. These will be described and analyzed in the next chapter, where we conduct some controlled experiments on our prototype and analyze these with respect to our goals.

Chapter 6.

Experiments, results and analysis

> It is inexcusable for scientists to torture animals; let them make their experiments on journalists and politicians.
>
> —Henrik Ibsen

Our experiments will first and foremost time the processing done by the classifier/forwarder ACE in the implementation. We attempt to accurately measure the time spent by every RTP/UDP packet within the forwarding procedure. This chapter will also analyze the work done in this thesis. It will do so with respect to the information in Part I, and to the results gained from our experiments.

6.1. Testing environment

Our testing environment, shown in Figure 6.1, consists of one IXP1200, ENP-2505 [Radisys03] card that is connected to the PCI bus of an Intel Pentium 4 2.0 GHz, Dell Optiplex GX260. Since all experiments are conducted within the IXP1200 card, the host machine is free to run the Darwin Streaming Server [Apple04] and the client without affecting the results.

The Darwin Streaming Server is an open source version of the proprietary Quick Time Streaming Server. It claims to be compliant to the Internet standards RTSP, RTP and SDP.

The client is a dummy python program that communicates with the proxy through the socket interface. It only sets up the RTSP session and receives the stream of incoming RTP packets in a *select* loop.

One of the four 10/100 Mbps Ethernet interfaces on the IXP1200 is connected to a switch, as is the host. All network communication will flow back and forth through the switch as shown in Figure 6.1.

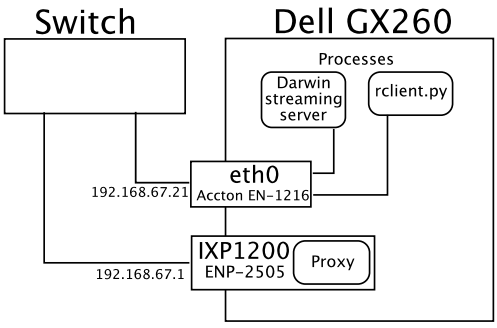


Figure 6.1. Test configuration

6.2. Measurements

We conducted two sets of measurements during a streaming session. We measured the cycles spent processing in the data-plane for every RTP/UDP packet, and counted the number of unique packet types received by the proxy. The latter numbers are used to validate our repeatedly used assumption that more packets flow through the data-plane than through the control-plane in scenarios like MoD.

To accurately measure the data-plane processing, we created a microblock probe (see Example 6.1) which utilizes the CSR, CYCLE_CNT register [Intel02b]. This register, located on the FBI, holds a 64 bit counter that is incremented on every core clock frequency, i.e., at the rate of a 232 MHz clock. The probe reads the value at the point where we want to start measuring and rereads it when the functionality to measure completes. Then, it stores the difference between the cycle counts in a table allocated in SRAM. The difference also includes the cycles needed to execute the probes. This includes reading the CYCLE_CNT register twice and calculating the difference. The number of cycles it takes to read the cycle count from the FBI is not documented, but we assume that it takes about the same number of cycles as accessing the Scratchpad memory, i.e., 12-14 cycles.

Example 6.1. Microblock probes

```
; Read the cycle count register, and wait for signal
csr[read, $misc[0], CYCLE_CNT], sig_done ; Probe
ctx_arb[fbi]

; Insert code to measure here
[....]

; Reread the cycle count and store the difference.
.local start end pkg_cnt
  alu[start, --, B, $misc[0]]
  csr[read, $misc[0], CYCLE_CNT], sig_done
  ctx_arb[fbi]

  sram[read, $misc[1], diff_base, 0, 1], sig_done
  ctx_arb[sram]

  alu[pkg_cnt, --, B, $misc[1]]
  alu[$misc[1], $misc[0], -, start]
  sram[write, $misc[1], diff_base, pkg_cnt, 1], sig_done
  ctx_arb[sram]

  alu[$misc[1], pkg_cnt, +, 0x1]
  sram[write, $misc[1], diff_base, 0, 1], ctx_swap
.endlocal
```

The table, in which the results are stored, is allocated on a per-session basis. This ensures that no values are overwritten, and the probe should also be useable when testing several streams in parallel. The handle to the table is stored in the forwarding table along with the forwarding information as shown in Figure 5.5. When the session is torn down, the table is flushed to file, and memory is freed for future use.

When measuring the cycles spent processing in the data-plane, we inserted the probes at different locations. Figure 6.2 shows the probes when inserted before packet reception and after enqueuing the packet for transmission. Thus, this probe measured the cycles spent to retrieve the packet, process it and send it to the egress ACE. We will refer to this as *experiment 1*.

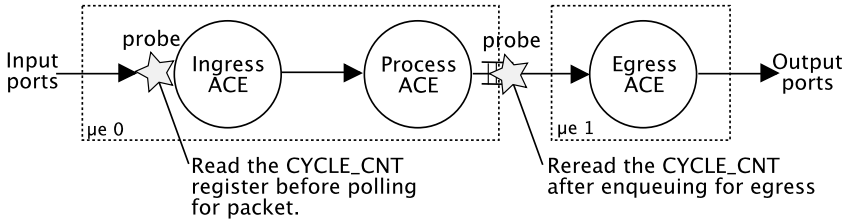


Figure 6.2. Experiment 1: Cycle probes for packet reception, forwarding and enqueuing

Figure 6.3 shows experiment 2. The probes were now inserted before starting packet classification, header modification and checksumming and before enqueuing the packet for transmission.

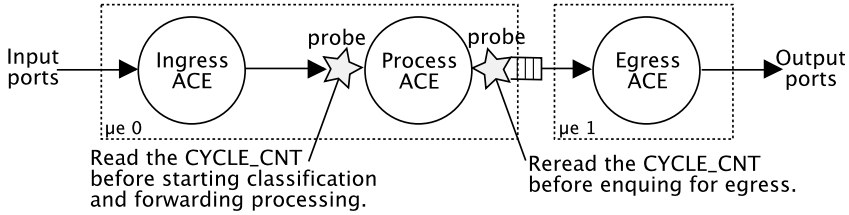


Figure 6.3. Experiment 2: Cycle probes for packet forward processing

Finally, in Figure 6.4, we show the third experiment that was conducted to measure the cycles spent forwarding and enqueuing the packets. This was basically done to find the total time needed for ingress processing.

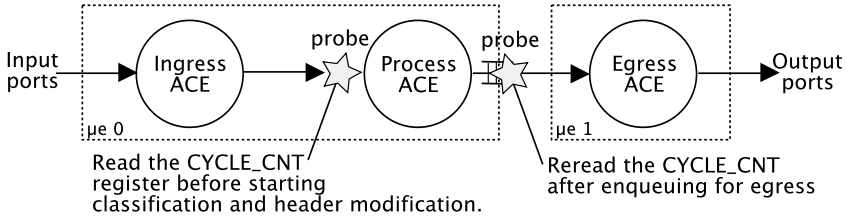


Figure 6.4. Experiment 3: Cycle probe for packet forward processing and enqueuing

During every run of our cycle count tests, the client requests a media object that has been previously hinted, i.e., knowledge about what to request in the RTSP SETUP message has been retrieved manually. The object requested is a 300 Kbps media file, encoded in the Quick Time movie format, stored on the host by the Darwin server. The Darwin server sends RTP packets with a constant size of 1024 bytes.

6.3. Results

In this section, the results of our measurements are presented and explained. First, we present the range of packet types received by the proxy, and then, results and graphs of the data-plane processing are shown and discussed.

6.3.1. Packet types

Table 6.1 shows the distribution of packets received at the proxy during the streaming of the test object. In this case over 99% of the traffic received is stream data, while under one percent is control messages. A similar distribution could be expected in MoD systems in general. For larger and longer media objects, the number of RTP/UDP packets would probably be higher, and the number of RTCP packets would also be higher as the duration of the session would be longer. Note that the client does not send any RTCP messages, they are only received from the server. In a real system, the client would also be sending these, thus increasing the number of RTCP packets further.

Nevertheless, the conclusion from this packet count experiment is that, as expected, far more packets are processed in the data-plane compared to the control-plane. Thus, the data-plane is the place to optimize first to gain as high performance improvement as possible.

Table 6.1. Packet count during session

Packet type	Count	Percentage	Comment
TCP	8	0.25%	RTSP, handshake and acknowledgments
RTP/UDP	3131	99.2%	
RTCP/UDP	17	0.53%	Only sent by the server
Other	1	0.03%	ARP packets and packets with bad checksums

6.3.2. Ingress to egress

In experiment 1, shown in Figure 6.2, we measure the cycle count for ingress to egress processing. The results of this experiment are shown Figure 6.5. It shows the number of cycles needed to processes every incoming RTP packet at the proxy. The processing includes packet reception, packet classification, checksumming, header modification and enqueuing. Table 6.2 shows statistics derived from these measurements.

As seen in the figure, most packets are processed in about 1325 cycles. In fact, calculations show that 88.2% of all packets are processed within a range of 1320 and 1330 cycles. This is also indicated by the median value of 1326. However, some packets deviate quite a bit from the main distribution. The difference between the maximum and the minimum value is 356 cycles, and the unbiased standard deviation of the distribution is 36.3. The deviation will be further discussed in Section 6.3.3

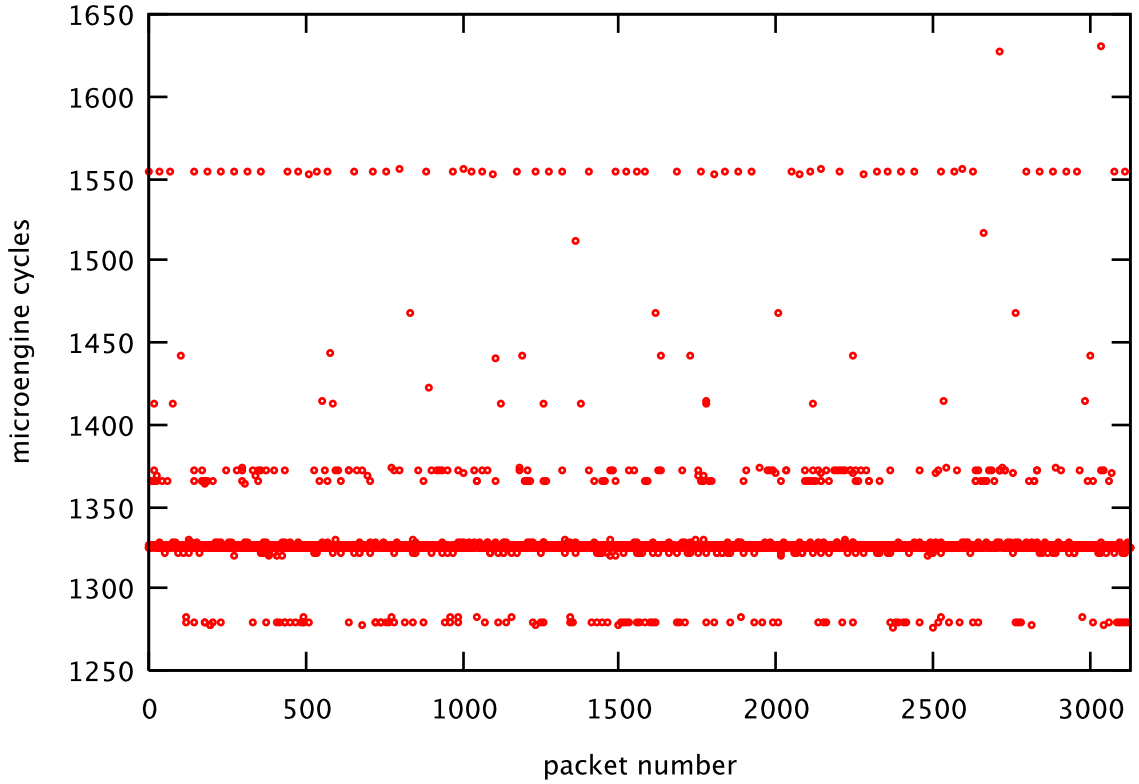


Figure 6.5. Cycles counted during packet reception, forwarding and enqueuing

Table 6.2. Ingress to egress statistics

Measurement	Cycles
Maximum	1630
Minimum	1276
Average	1331
Median	1326
Unbiased Standard Deviation	36.3

To further visualize the results, a snapshot of packets 500 through 700 from Figure 6.5 is shown in Figure 6.6. We observe that although most packets need about 1325 cycles to be received, processed and enqueued, every now and then a packet deviates from the pattern.

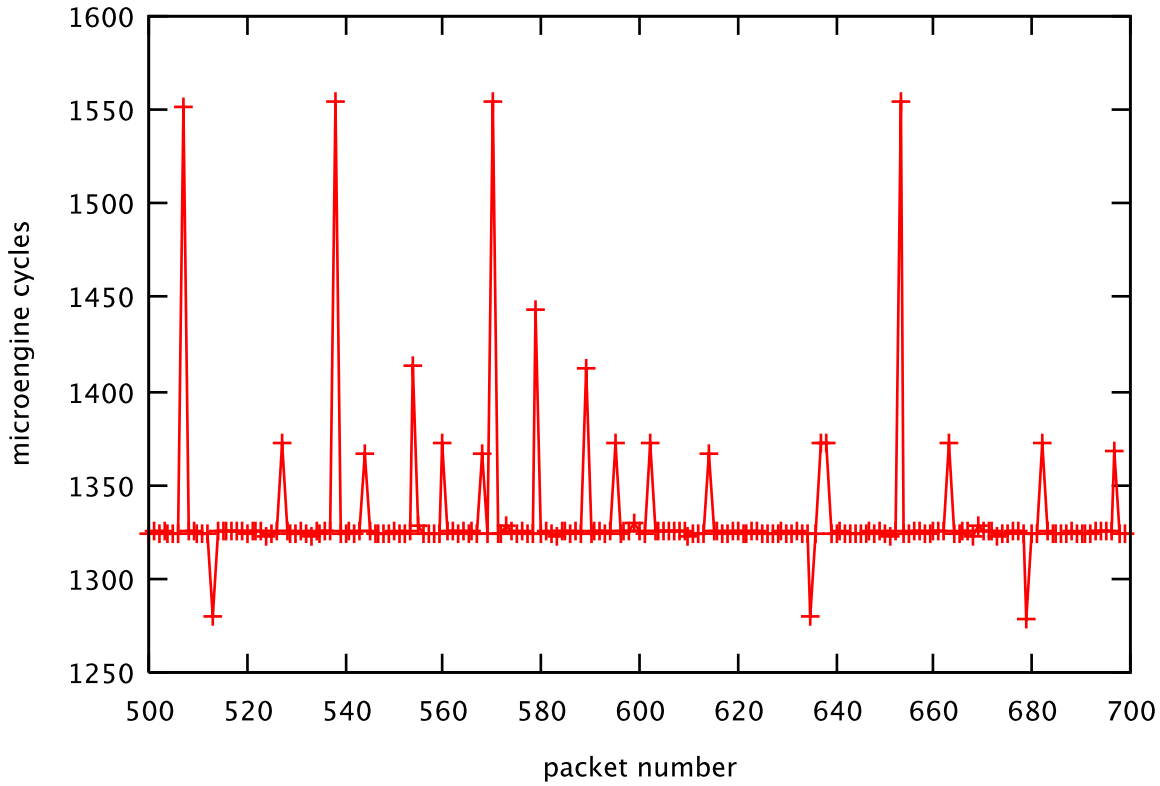


Figure 6.6. Packets 500 through 700 experiment 1

6.3.3. Deviation

We have, in experiment 1, shown the total number of cycles needed to receive, process and enqueue a packet through the prototype. We have noticed that a few percent of the packets deviate quite a bit from the rest. Analytically, we would however expect that the processing cost of our classification and modification of the packet in experiment 2 (see Figure 6.3) would be close to a constant cycle count. This is based on the fact that every packet we send through our microblock follows the same flow and requires the same amount of processing. Figure 6.7 a) shows the number of cycles spent performing the functionality in our second experiment. Surprisingly, a few packets are still deviating from the general pattern. Though the measurements stabilize somewhat, the difference between the maximum and the minimum is still large, i.e., 204 cycles, and the standard deviation is 22.5.

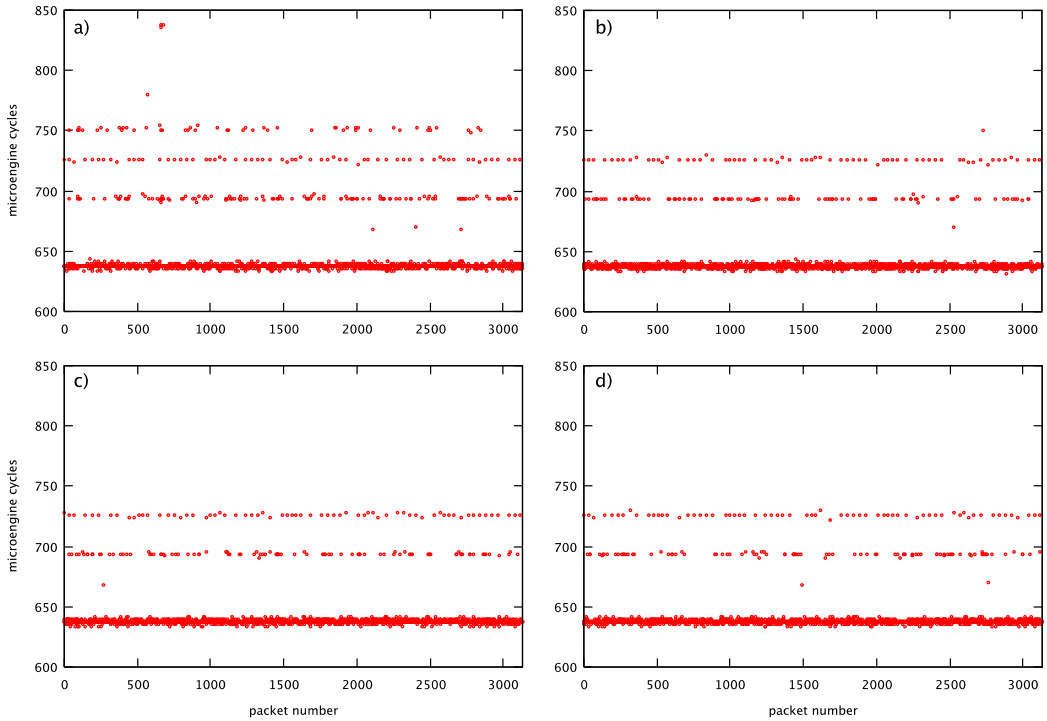


Figure 6.7. Cycles counted during packet forwarding

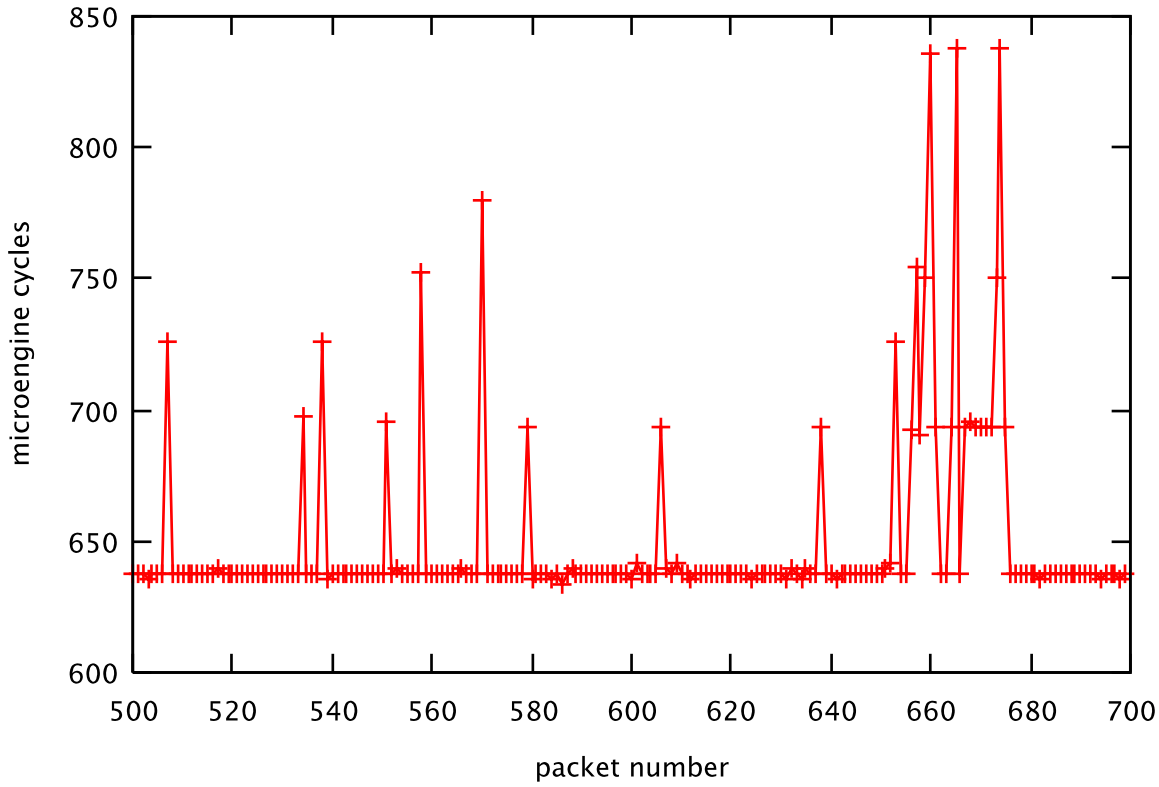


Figure 6.8. Packets 500 through 700 from experiment 2a)

The results shown in Figure 6.8 are packets 500 through 700 from our second experiment. The figure shows a similar spread of values as in the previous closeup in Figure 6.6. In this case, the most extreme values are inside the range, so we get the outliers around packet 660.

To understand why some of the packets require more cycles to be processed, we conducted three variations of experiment 2. There are a number of things that could interfere with the processing of packets. In the microblock that performed classification and forwarding, we yielded on every memory access. This allows other threads to run, if they are in ready state. Also, the microengines all share the command bus as shown in Figure 4.6. Because they share this bus, they also contend for it. This contention can make some memory requests wait longer, as other requests might already be in the queue. Also, to send a request to one of the three command queues, the command bus arbiter must grant access to the bus, and finally, there are variations in memory latency as shown in Table 4.2.

We wanted to assure that no scheduling would occur while we did the cycle count. To achieve this, we altered the code so that our thread would stay in control of the microengine even if it was idle waiting for a memory access, i.e., the processing was done as an atomic operation. We ran the atomic operation in three “modes”. First, all memory accesses were ordered, then we allowed the hardware to optimize them, and finally, we sent them all through the prioritized command queue. This exercise was done to see if there was any contention from other microengines, i.e., those running the egress ACE. If there was, the optimized and prioritized memory accesses should yield less deviation. The results of the three scenarios are shown in Figure 6.7 b-d) and statistics are shown in Table 6.3. Even though they show a slightly better distribution, there are marginal differences between the three memory access “modes”. What is clearly seen, is that several measurements are stabilized when comparing a) and b) on Figure 6.7. One plausible explanation for this is that four threads are running on the microengine, and whenever the microblock in the non-atomic ordered scenario makes a memory request, it swaps out, to allow other threads to do their work. This means that the threading arbiter unit will schedule the other threads in a round-robin fashion, and if any of them are ready, they will get control of the microengine. As all threads run the same code, and we only have one connected interface, none of the other threads should be ready. Otherwise, having them poll the *Ready Receive* bits should fail and they swap out again. The latter is not very likely, as it would probably give a higher overhead than measured. However, there seems to be some activity unaccounted for, as the deviation on Figure 6.7 a) is far greater on some of the packets than what is documented to be the cost of a context-switch, i.e., about three cycles.

Table 6.3. Forwarding statistics

Experiment scenario	Max	Min	Average	Median	Std. deviation	% in range 632-644
1. Non-atomic ordered forwarding	838	634	644	638	22.48	91.9%
2. Atomic ordered forwarding	750	632	641	638	15.98	94.7%
3. Atomic optimized forwarding	728	634	641	638	15.93	94.6%
4. Atomic prioritized forwarding	730	634	641	638	15.70	94.8%

The best explanation we could find for why certain packets need more cycles than others, is the memory latency differences. Our microblock makes eight memory access while processing the packet, 3 SRAM read operations, 3 SDRAM read operations and 2 SDRAM write operations. If we take the maximum variation in delay, this gives us 47 cycles. This does however only add up to about half of the deviation experienced in the three atomic measurements and even less in the non-atomic scenario!?

6.3.4. Processing overhead

In the two first experiments, we measured the processing overhead from ingress to egress and how

much of this overhead was added within our forwarding procedure. To complete our range of possible experiments, we did, as our final experiment, measure the overhead of forwarding and enqueuing as shown in Figure 6.4. The results from experiment 3 gives us the opportunity to show how much overhead is added in specific parts. An approximate average enqueuing delay can be calculated by subtracting the average from the non-atomic and ordered scenario in experiment 2 from the average of experiment 3. The same goes for ingress processing. The separation of where processing delay is added is shown in Table 6.4.

We see, from the table, that most of the processing overhead is added in our classification and forwarding ACE.

Table 6.4. Experienced overhead

Functionality	Average overhead in cycles	Average overhead in µs
Ingress processing	493	2.13
Classification and forwarding	644	2.78
Enqueuing for egress	194	0.83

6.3.5. Summary of results

The implemented prototype offloads basic RTSP/RTP proxy functionality onto the IXP1200 NPU. It offloads the host machine by processing all network traffic flowing through the proxy, both in the control-plane and in the data-plane. This leaves all the resources in the carrier machine unused and free for other purposes, e.g., transcoding and protocol translation.

In the control-plane, the prototype performs basic RTSP forwarding and data-plane control. It does so by utilizing the embedded StrongARM processor for packet processing and setting up shared data structures that are later used by the microengines, when they process packets in the data-plane.

In the data-plane, the prototype shows that one microengine thread is able to perform forwarding of RTP/UDP packets. This includes packet reception, classification, checksum validating, header modification and checksum recalculation.

The prototype is able to process every RTP/UDP packet in an average of 1331 cycles, with not counting the cycles needed for the egress ACE to pull data from SDRAM to the TFIFO queues. We have shown that that packets are not processed at a constant cycle rate. We have suggested that scheduling overhead and memory latency are probable causes for this variation.

6.4. The offloading effect

One interesting aspect is how well the prototype offloads the host carrier. As a point of reference, we use the results found by Guo and Zheng [Guo00], explained in Section 3.4.2. Their measurements show that the reception of a 1024 byte large UDP packet in the GNU/Linux 2.0.34 kernel takes 35 µs. Converted to cycles using a 350 MHz clock the reception takes 12250 cycles. Sending a packet of the same size requires less processing and employ DMA. It takes 18.9 µs to send a UDP packet without calculating the checksum, i.e., it takes 6615 cycles.

The prototype handles all the network traffic sent to the host, so it effectively offloads 100% of the network processing from the host CPU. This means that a sum of 18865 cycles are free to be utilized differently by the host every time a 1024 byte packet is received and handled by the prototype. In addition, any application layer processing is also offloaded to the NPU.

The offloading effect will largely depend on the network load on the proxy host. The distribution of packets shown in Table 6.1 suggest that the network load is significant in the data-plane. Although all

traffic is offloaded by the prototype, 99% of the traffic flows in this data-plane, and this is where the offloading will have real effect. The host carrier would have no problems processing the control-plane, as this would only require processing a mere 1% of the traffic or less.

Now, this scenario seems very promising, but it is a highly simplified view of an MoD proxy-cache. As explained in Section 3.2.2, the algorithms developed for MoD caching try to reduce the network load. They do this by relieving the network between the proxy and the server. This means that the traffic flowing through the proxy also is reduced, as every cached segment of data will be sent directly from the proxy. The amount of traffic that a caching strategy reduces, depends on how well it predicts what to keep in the cache, and what to replace. It also depends on how much of each object the proxy-cache is able to store.

Some MoD caching replacement algorithms take popularity in account when deciding what objects to replace or keep. The Zipf distribution explained in Section 2.3, says that 10% of the movies are accessed 90% of the time, due to their popularity. In a caching algorithm that replaces objects based on popularity, we suggest three coarse grained offloading scenarios to be discussed based on the Zipf distribution. These include *best case Zipf, worst case Zipf* and a middle ground.

- *Best case Zipf caching*: In the case where a caching strategy is able to perfectly cache according to the Zipf distribution, 90% of all accesses can be served directly from the proxy-cache, given that the proxy has enough space to store 10% of all media objects. This means that the offloading potential is only 10% using our prototype, i.e., 9.9% in the data-plane processing.

- *Worst case Zipf caching*: In this case, the cache algorithm totally fails in following the Zipf distribution and all data must be fetched from the server and be forwarded through the proxy, i.e., 100% of all traffic. This means that the offloading potential is 99% using the prototype.

- *Middle ground*: This scenario would give a caching hit-rate in between the worst case and the best case scenarios. To serve the middle ground scenario, we suggest using an intelligent replacement algorithm like ECT, explained in Section 3.2.2. ECT shows good simulation results when serving large client populations and many movies.

If we assume that a cache achieves the best case cache hit-rate, the offloading potential is only 9.9% and all resources should be allocated to help pushing data from the cache as 90% of the requests can be served directly. However, a perfect Zipf distribution is unlikely to occur. It is hard to model the theoretical Zipf distribution accurately in a caching scenario. The popularity of movies change dynamically, and fluctuations may occur based on lurker variables, e.g., the release of sequels to a movie can increase the popularity of the original. Also, the Zipf distribution is claimed to be inaccurate when dealing with large client populations and when the number of media objects is high. [Griwodz00] Even if the best case is achieved, most media caching schemes assume that only a part of the object is stored in the cache due to the high storage requirement (see Section 3.2.2). Therefore, the missing parts of a stream must be fetched from the server. These data must be forwarded by the proxy and can effectively be offloaded by our prototype.

The worst case Zipf is also an unlikely scenario, as most caching algorithms, no matter how simple, are able to preserve some cache hits. In shorter periods it might, however, be that cache misses are repeatedly experienced. In such a period, the offload effect would be significant.

The middle ground scenario is assumed to be most likely. An intelligent caching algorithm can achieve near perfect hit rates in periods, but periods with massively cache misses can also be experienced. Nevertheless, caches are not able to store all the media objects entirely and fluctuations in the access patterns will require that much data is fetched from the server.

Determining quantitative measures of how much traffic in the data-plane must be retrieved from the server is dependent of several variables, e.g., the number of concurrent streams that are flowing through the proxy and the bit-rate at which they are sent. Large-scale experiments or simulations are needed to get empirical data on this. The data-rate is of course limited by the available downstream

bandwidth. With the IXP1200 NIC used in this thesis, the maximum aggregate bit-rate is 400 Mbps, but newer cards support 3 Gbps and more.

The caching of media objects and offloading of fast forwarding of stream data are clearly orthogonal methods to improve an MoD system. The caching reduces startup latency, network load and server load while the offloading alleviates network processing within the proxy. The offloading effect is in no way dependent on how well the offloaded solution performs. The number of freed cycles will be available to the proxy host, no matter how long it takes the prototype to process the same packets. The performance of the prototype is still important, but it does not have any impact on the offloading effect.

6.5. Prototype performance

The performance of the prototype can be compared to that of a traditional OS architecture, where a socket interface is employed to handle network traffic. We will compare our performance to that of such an architecture, i.e., the Linux stack. As no measurements of the egress cycle count was performed, due to time constraints, we will only compare the reception of packets. This should be enough to prove the concept.

The data in Section 3.4.2 shows that the Linux OS is, with an Intel Express 100pro NIC, able to receive 1024 byte large UDP datagrams using 12250 cycles per packet. This processing overhead does not show any application layer processing. Our results also show how many cycles it takes to receive a packet, but they also show the cycles needed to perform application layer RTP forwarding. This means that the comparison is slightly biased towards the Linux cycle count, due to the coarse granularity in our measurements.

Still, we see that our prototype performs reception functionality at a much lower cycle count. The prototype uses an average of 1137 cycles (493 + 644) versus 12250 cycles in the Linux stack. Of course, a NIC with support for DMA on the receiver side in the same Linux environment would probably perform slightly better. But, the origin of the main addition to the delay is not in the transfer from the NIC, but higher up in the stack, as explained in [Guo00].

It takes our prototype less than one tenth of the cycles to receive a 1024 byte UDP packet, compared to a traditional architecture. The implication of this, is that the total delay of every packet flowing through the system is reduced. In the Linux case, every packet being forwarded would be delayed more than 81.3 µs[18], as application layer processing overhead also would have to be considered. If we assume that the egress ACE in the prototype is able to send packets using approximately the same amount of cycles as the ingress[19], i.e., 493, the prototype overhead amounts to a total of 7.9 µs per packet. This means that the delay of each packet is reduced by 73.4 µs.

The four 100 Mbps interfaces on the IXP1200 NIC can sustain a maximum aggregated data-rate of 400 Mbps. Further, the maximum throughput of a forwarder using 7.9 µs per 1024 byte packet is 1036 Mbps. This is a good indication of that there are plenty of resources left for other processing within our microblock, such as caching. This is even more evident, as we neither employ parallelism, nor utilize more than 2 microengines; one for ingress, classification and forwarding and one for egress.

In an MoD scenario, buffering is used at the end nodes of the system to compensate for jitter. This means that reducing the delay in a proxy has little impact on the user experience of a streamed movie. However, the forwarding functionality of the prototype might be more applicable in a real-time streaming scenario, e.g., in a distributed game or a video conference, were no buffering is used. Imagine a real-time game, where the contestants continuously send events that update the game state. Also, they can team up, and have real-time communication while playing. Proxies have been proposed as a way to offload servers in network gaming [Mauve02]. If the real-time traffic must pass through

[18]This number is scaled down to a 232 MHz processor. The 350 MHz CPU used in [Guo00] would use 53.9 µs plus application layer processing.
[19]Studying the Intel source code suggest that egress processing requires less processing per packet than ingress processing, as ingress ACE first moves mpackets into the RFIFO, and then to SDRAM, while the egress ACE only transfer a mpacket from SDRAM to the TFIFO queues. Also, the results from [Spalink01] show that their egress uses 109 cycles.

the proxy, it becomes more important to reduce the delay of every packet. Our prototype proves the concept of such reduction. Also, it might be practical in real-time video conferencing. In a video conference with three participants, a proxy could perform overlay multicast of the streams flowing back and fourth, so that the number of unicast streams can be reduced. Adding as little delay as possible would be beneficial in this scenario as well.

The clock speeds of the processing units used by the IXP1200 and the Linux test machine were respectively 232 MHz and 350 MHz. In order for the Linux architecture to receive the packet and add less delay than the NPU, it will need a CPU with a clock speed over 1.570 GHz. 232 MHz for the NPU, and 350 MHZ for the GPP are, however, not representative for the technology available at the time of writing. GPPs have reached 3.2 GHz frequencies, and microengines from Intel have reached 1.4 GHz. If we recalculate the number of cycles and compare the two, we see that a 3.2 GHz GPP will execute the cycles needed for reception in 3.8 µs and the NPU will use 0.8 µs. This implies that the NPU still would be 4.75 times faster, even though the comparison still is biased towards the GPP.

The prototype forwarding functionality seems to perform well, as was expected, due to the use of specialized hardware for network processing. The processing also becomes far less complicated when implemented on an NPU, compared to a fully fledged socket interface, such as the Linux stack. The prototype does, however, have the potential to perform even better. The implemented forwarding functionality is not fully optimized, and ingress and egress processing may be improved. We perform some unnecessary validity checks in our code, such as verifying the IP address of the packets received. The ingress does the same check, based on the MAC addresses, so this could be removed. Further, the prototype does not make use of parallelism, which is one of the primary benefits of the IXP1200.

6.6. Extensions

In addition to optimizations, several extensions in functionality are needed to make the prototype a fully fledged MoD proxy. Possible extensions include caching, caching algorithms, zero-copy from network to disk, zero-copy from disk to network, prefetching from server and efficient communication with host applications.

6.6.1. Caching

The primary functionality of an MoD proxy is caching. In other multimedia applications, such as distributed gaming, proxies would typically not cache media data, but rather game state. In this section we will describe how caching segments of data in the MoD proxy prototype could be performed. The description will deal with the implementation details for the microengines.

The basic functionality needed in order to implement caching functionality running on the microengines, involves memory copying of RTP packets. This is because we would like to minimize the processing done on the microengines and leave the actual disk writes and stripping of headers to processing units higher up in the hierarchy, e.g., the StrongARM or the host CPU. Therefore, a copy of the packet must be made so that it can be both forwarded and cached. This process should add as little delay to the packets as possible, to preserve the throughput of the prototype and because every packet in a stream will be copied to the cache until the caching strategy is satisfied. The number of packets copied will of course depend on the caching strategy (see Section 3.2.2).

We suggest two possible ways to go about implementing the “multicast” of packets. Both approaches involve changes to the Intel IXA SDK, but one more than the other. The easiest way would only require changes to the configuration tools. The other would require touching several core components, e.g., ingress, egress and the resource manager, but the configuration tool could stay the same.

The most intuitive way to implement copying on the microengines, is to dedicate one microengine to the task. Again, two options are available: four threads running on the microengine can copy one package each, or one thread can copy several packages. The former can use every thread to dequeue one packet handle each, sequentially read from it, and write to an allocated buffer. By yielding on

every read and write, some memory latency is hidden, as every thread may process while the others are waiting for a memory request. A possible way to implement the latter approach is shown in Example 6.2. It is essential that packet copying hide as much memory latency as possible, so that the processing resources are utilized. The pseudo code in the example does this by allowing several continuous reads, followed by several writes. The writes will have to wait for the reads, so more latency will be hidden when there are several packets to copy.

Example 6.2. Pseudo code for packet copy

```
while(1) {
  packetReady = checkQueue();

  if (!hasFreeCopySlot() && !packetReady)
    goto copy;

  freeSlot.handle = dequeue();
  freeSlot.size = getSize(freeSlot.handle);
  freeSlot.duplicate = allocate(size);
  slots.add(freeSlot);

copy:
  for(everySlot)
    slot.bytes = read(&sdramRegs, slot.handle);


  for(everySlot) {
    write(sdramRegs, slot.duplicate);
    slot.size -= bytes;
    if(slot.size == 0) {
      signal(ingress);
      sendTo(strongARM, slot.duplicate)
    }
  }
}
```

The other way to multicast packets is to have every duplicate of the packet share the payload, and only copy the headers when they are modified. Using this “lazy copy” strategy requires more logic in the buffer management, and the code must be modified in the ingress, egress and the resource manager components. We suggest that a data structure like the one in Figure 6.9 can be used for this purpose. Every packet is represented by a pointer to its header and a pointer to its payload structure. The payload structure holds a reference counter for every copy of the header, i.e., the number of packet copies. Every time a copy of the packet is needed, the reference count is increased. The first modification of a value in a packet results in the header being copied to a free memory location, but as no modification is done to the payload, every copy can share this. Whenever a component is done with its copy, the reference count is lowered and any modified header is freed. When the reference count reaches zero, the payload is also freed.

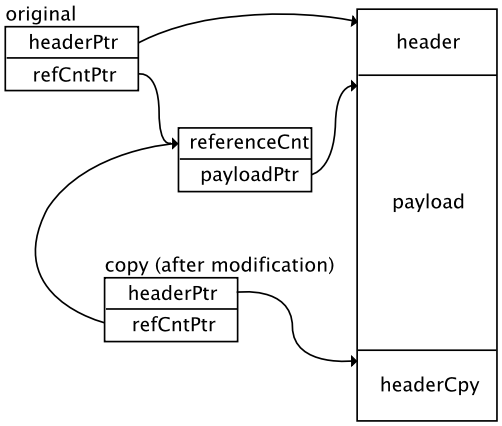


Figure 6.9. Lazy copy data-structure

The lazy copy strategy is by far the best of the two, as it requires a smaller portion of the packet to be copied. If a 1024 byte large packet were to be copied in the first solution, it would require sixteen 64-byte chunks to be read, and then written. In memory latency alone, this would amount to about 1600 cycles, or 6.8 µs, on a 232 MHz microengine. Some of this latency would be hidden, but would still represent a significant increase of the total delay added to every packet flowing through the prototype. Performing lazy copying would make it far more efficient to copy packets. It only requires the headers, e.g., 42 bytes for an UDP packet, to be copied and this would only happen if it was modified. Further, when combining copying with forwarding all modifications can be written directly, thus reducing the number of memory writes.

The main reason for not implementing caching in the existing prototype was time. We realized that changing several components in the SDK is too extensive for our time frame. The caching functionality is needed to get further data on how well an NPU is suited for offloading an MoD proxy cache. Also, as the microengines will only perform a packet copy, and further caching logic is left to be handled by a processing unit higher up in the hierarchy, the packet copy is, as mentioned, actually a multicaster of the packet. This can also be used in situations where IP multicast is unavailable, to create overlay multicast.

6.6.2. Zero-copy

Another interesting extension to the prototype is to investigate how to make zero-copy data paths from the network to disk and from the network to the application. This is relevant to caching, but can also be applicable in other cases, e.g., when a server stores a live recording from a client, or if payload data must be modified by a proxy performing transcoding, application layer encryption or re-encoding. We suggest how a zero-copy data-path, from network to disk, may be designed for our prototype.

Figure 6.10 shows the traditional data-path from network to disk and a zero-copy from the network to the disk. The traditional data-path requires data to be moved from the NIC to a kernel memory buffer, either by a DMA transfer or by a CPU driven transfer. On a regular NIC, this happens every time a packet is received. Then, packet data must be copied to an application buffer, due to memory protection, before the application can issue a disk request. For every disk request, the data is again copied to kernel memory. Usually, the kernel will buffer several requests, before flushing them to disk. As we see, the traditional data-path is slow, due to many interrupts and memory operations on packets and data.

The figure also shows a zero-copy path from an NPU, into kernel memory and down to disk. The idea is to reduce the number of memory operations, interrupts to the kernel and small disk requests. A possible way to solve these three goals is to make use of NPU features and memory resources. Packets received by the NPU can be batched in memory, so that a certain amount of packets are buffered. When

the set amount of packets are received, the NPU will issue a scatter-gather DMA[20] transfer of the batched packet payloads into a continuous block of kernel memory. As several packets worth of data are transferred on every turn, the 0number of interrupts to the kernel can be dramatically reduced, depending on how many packets it is expedient to batch. Since the different payload segments are gathered during the transfer, no internal kernel copy operations are needed.

As disk requests are very slow, i.e., typically one order of magnitude slower than memory access, they should be few and large. Therefore, the kernel I/O system should collect several batches of payloads and send them to disk in one large disk-write.

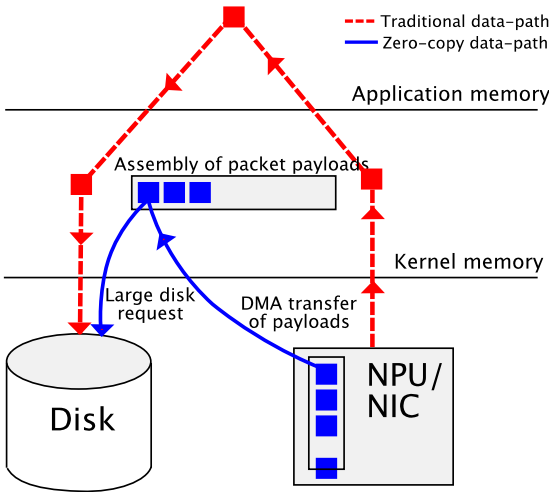


Figure 6.10. Zero copy data-path vs. traditional data-path

We suggest a similar approach for a zero-copy data-path from the network to the application. The difference is that the batched payloads, or rather the whole packets, should be transfered directly to application memory, so that the packet may be processed and then effectively sent back to the card. This is possible only if the NPU does all the TCP/IP processing, which it can. Otherwise, the kernel must also be regarded, which will complicate the situation.

6.7. Summary

This chapter has dealt with the measurements, the results and the analysis of our offloaded RTSP/RTP proxy prototype. To summarize, we will discuss our findings with respect to our goals.

As our primary goal in this thesis, see Section 1.2, we wanted to explore the feasibility and the expediency of offloading MoD proxy features on to an NPU. Specifically, we wanted to look at two subjects: Offloading the forwarding of data that had to be fetched by the proxy from the server, and how such processing could be optimized by implementing it on an NPU compared to a traditional architecture where data is processed by the host CPU. We therefore designed our prototype to enable such an offloaded solution (see Section 5.1.1). Our design goal was to make the appropriate separation of concerns within the proxy, i.e., control-plane and data-plane, and enable portability of code. We also wanted the prototype to be a basis for further work in the area of offloading multimedia proxies, in particular MoD proxies.

Our prototype shows that we have successfully identified the separation of concern within an MoD proxy by splitting its processing up in the control-plane and data-plane. Though this is an obvious observation, our measurements verify it, as 99% of all packets received at a proxy are in the data-plane when streaming a non-cached object. Further, the design has been successful in identifying the intercommunication needed between the two planes, at least for the implemented functionality, so that the

[20]We are unaware of any scatte-gather support on the IXP1200, but instead a DMA transfer could be issued per packet payload.

offloaded solution may operate correctly. However, the implementation is not fully portable to another platform, as the RTP forwarder is highly IXP1200 specific. A dedicated codebase will be needed to port the prototype to a different platform. The RTSP proxy, on the other hand, can be directly built on any Linux platform.

We further argue that it is expedient to offload the processing of the data that is retrieved by the proxy from the server, because even though caching reduces this traffic, it is unrealistic that it is removed altogether. We have shown that our prototype can offload all processing of RTP forwarding and RTSP session control. In a best case, having perfect Zipf behaviour where 90% of the requests are for cached objects, our method would offload 9.9% of the data-plane processing and in a worst case cache 99% if no cache hits are acheived. Our method serves a purpose in any of these cases. We also realize that our RTP forwarder can be useful in other real-time streaming applications that might need a proxy to relay data, e.g., distributed interactive games or video conferencing.

We have also shown strong indications that the forwarding of RTP packets, using the IXP1200, is highly optimized compared to a traditional architecture, as every forwarded RTP packet is processed at approximately a tenth of the time. Also, as our prototype does not fully utilize the resources of the two microengines (out of six), it should be possible to implement the proposed extensions, without deteriorating the performance. This should be true even at high aggregate data-rates. The low delay added to all the RTP packets flowing through our prototype further strengthens the applicability to use this approach for other real-time applications that make use of proxies.

Finally, the relevance and performance of the RTP forwarder suggests that it is an orthogonal method for improving the internal architecture of the MoD proxy, and thus the MoD system in general. The prototype is also ready to be extended with *fast-path* cache or “multicast” functionality, which should give a highly efficient proxy architecture, as the host is fully offloaded with respect to forwarding and the gathering of small dataobjects might be done during the bus transfer. Conclusively, such an architecture reduces the overall resource consumption per packet and additionally offloads tha host CPU. Therefore, we argue that it founds the basis for further fork in the area.

Chapter 7.

Conclusions and future work

> Genius - To know without having learned; to draw just conclusions from unknown premises; to discern the soul of things
>
> —Ambrose Gwinett Bierce

This thesis makes use of new technology in order to perform application specific low-level forwarding. We have designed and implemented offloaded RTSP control and RTP forwarding onan NPU, the IXP1200 chip. This prototype MoD proxy has been evaluated and the results analyzed. This chapter will summarize our findings and describe where to focus future efforts in the area.

7.1. Conclusions

Our focus in this thesis has been on how to offload an MoD proxy. We have specifically investigated RTP forwarding through the proxy data-plane, but we have also shown that the RTSP control-plane can be offloaded successfully, and thereby show the feasibility of the prototype.

We have shown that the prototype is expedient and relevant in an MoD setting, because of two properties:

1. There will always be a certain amount of data that must be fetched by an MoD proxy from a server. Thus, the RTP packet forwarding will serve its purpose as an offloaded solution.

2. The RTP forwarder performs well. It uses approximately one tenth of the cycles to process a packet, compared to a traditional network processing architecture. The low resource usage of the forwarder also allows adding functionality to further complement its usefulness.

These two properties make the prototype an ortogonal method when improving the architecture of an MoD proxy and thus to improve an MoD system in general. The prototype is ready for extensions and may work as a foundation for future work in the area of MoD proxy offloading. It can also be ported to other platforms with little effort, and the RTP forwarding is applicable to any proxy in a multimedia context.

7.2. Future work

There are many possible and interesting areas in which to continue the work done in this thesis. The use of network processors in a multimedia context is not extensively explored, and in MoD alone there are several possibilities. For example, there is work in progress at the department to implement n-cube[21] like load-sharing functionality using the IXP2400 chip onboard a NIC with three optical 1 Gbps interfaces.

[21]Information about n-cube can be found here: http://www.ncube.com/

On our proxy prototype, there are many subjects to look into apart from the caching and zero-copy extensions described in Section 6.6:

- Streaming directly from the cache requires low-latency fetching of cached data from disk, and the creation of RTP packets, in order to scale to many clients. This can for example be done by integrating the INSTANCE OS enhancements [Halvorsen01] with an NPU. At the same time, the NPU could handle the prefetching of data from the server if this is required by the cache strategy.

- Efficient communication between the NIC and the host over the PCI bus is essential to make use of the processing resources available there, both in the application layer and in the kernel. Again, if transcoding of a stream flowing through the proxy is to be performed, the delay and jitter of the packets that are sent to the host should be kept low. Work similar to this suggestion has been done in relation to cluster computing applications [Mackenzie03].

- The current prototype currently only supports a single server. A future version should be extended to support multiple incoming channels from many different sources, as this would also make the prototype useful in other distribution architectures, such as cooperating proxies, multilevel proxies and P2P architectures.

Apart from functionality, other issues should be looked at. The proxy should be tested in an environment with a large client population. In this context, it would be interesting to explore scalability and parallelization. High aggregate data-rates and many concurrent clients would give a more realistic view of how important offloading could be. How to best partition the workload between threads and microengines will be a central part of such work. The same can be said for how to decide what functionality is to run at what level in the processor hierarchy. Resource consumption is always a crucial consideration when developing high-speed scalable network applications as over utilizing resources creates bottlenecks.

Porting the prototype to run on new NPUs, such as the IXP2400, is also interesting to gain up-to-date empirical data and also see if the prototype can scale to higher data-rates than what is possible on the ENP-2505. Also, up-to-date experiments and comparisons to newer Linux kernels should be conducted. Work is in progress at the department to analyze the Linux 2.6 and the NetBSD network stacks, and the results from this, could give more precise comparisons as the proxy prototype could be ported and run on any of these architectures to also get the application layer processing cost.

Finally, optimizing the prototype forwarding ACE, along with the ingress and egress ACEs could give better and more precise measurements. This is especially true for egress processing as no results was gained about that in this thesis. Application specific optimizations for the ingress and egress processing, such as prioritized scheduling of different application data might also be considered.

In this thesis we have designed, implemented and evaluated an MoD proxy prototype. We have shown the feasibility, relevance and expediency of such functionality. We further have good indications of that offloading multimedia proxies is an orthogonal method in order to improve the internal architecture of the host running proxy applications. The potential of NPUs is not fully explored, and we believe that more can be gained in performance and scalability by conducting future studies.

References

[Acharya00] Soam Acharya and Brian Smith. *"MiddleMan: A Video Caching Proxy Server"*. Proceedings of Network and Operating System Support for Digital Audio and Video (NOSSDAV). Chapel Hill, NC, USA. June 2000. http://www.nossdav.org/2000/abstracts/16.html.

[Apple04] Apple Computer, Inc.. *"Darwin Streaming Server"*. http://developer.apple.com/darwin/projects/streaming/.

[Bauer02] Daniel Bauer, Sean Rooney, and Paolo Scotton. *"Network infrastructure for massively distributed games"*. Proceedings of NetGames2002. Braunschweig, Germany. April 2002. pp. 36-43.

[Beck99] Michael Beck, Harald Böhme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus, and Dirk Verworner. *"Linux Kernel Internals"*. Addison Wesley Longman. 1999.

[Bommaiah00] Ethendranath Bommaiah, Katherine Guo, Markus Hofmann, and Sanjoy Paul. *"Design and Implementation of a Caching System for Streaming Media over the Internet"*. Proceedings of the Sixth IEEE Real Time Technology and Applications Symposium (RTAS). Washington DC, USA. May/June 2000. p. 111.

[Buddhikot98] Milind M. Buddhikot. *"Project MARS: Scalable, High Performance, Web Based Multimedia-on-Demand"*. Phd. Thesis. Department of Computer Sicence, Washington University, St. Louis, MO, USA. June/July 1998.

[Clark88] David D. Clark. *"The Design Philosophy of the DARPA Internet Protocols"*. Proceegings of ACM Special Interest Group on Data Communications (SIGCOMM). Stanford, CA. August 1988. pp. 106-114.

[Comer03] Douglas E. Comer. *"Network Systems Design Using Network Processors"*. Pearson Education, Inc. 2003.

[Coulouris01] George Coulouris, Jean Dollimore, and Tim Kindberg. *"Distributed Systems, Concepts and Design"*. 3rd Edition. Addison-Wesley Publishers Ltd.. 2001.

[Fielding99] R Fielding, J Gettys, J Mogul, H Frystyk, L Masinter, and T Derners-Lee. *"Hypertext Transfer Protocol -- HTTP/1.1"*. Internet Request For Comments 2616. June 1999.

[Fiuczynski98] Marc E Fiuczynski, Richard P Martin, Tsutomu Owa, and Brian N Bersahd. *"On Using Intelligent Network Interface Cards to support Multimedia Applications"*. In Proceedings of the 8th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV). Cambridge, UK. July 1998. pp. 95-98.

[Gnu04] The GNU Project. *"The GNU Project and The Free Software Foundation"*. http://www.gnu.org/.

[Gries03] Matthias Gries, Chidamber Kulkarni, Christian Sauer, and Kurt Keutzer. *"Exploring Tradeoffs in Performance and Programmability of Processing Element Topologies for Network Processors"*. 2nd Workshop on Network Processors (NP2) at the 9th International Symposium on High Performance Computer Architecture (HPCA9). Anaheim CA, USA. February 2003.

[Griwodz00] Carsten Griwodz. *"Wide-area True Video-on-Demand by a Decentralized Cache-based Distribution Infrastructure"*. Phd. Thesis. Darmstadt, Germany. April 2000.

[Griwodz03] Carsten Griwodz. *"KOM(S) Streaming System"*. http://sourceforge.net/projects/komssys.

[Griwodz04] Carsten Griwodz, Steffen Fiksdal, and Pål Halvorsen. *"Translating Scalable Video Streams form Wide-area to Access Networks"*. Internatinal Network Conference (INC04). Plymouth, UK. July 2004.

[Guo00] Chuanxiong Guo and Shaoren Zheng. *"Analysis and Evaluation of the TCP/IP Protocol Stack of LINUX"*. IEEE International Conference on Communication Technology (WCC-ICCT). Aug 2000. pp. 444-453.

[Halvorsen01] Pål Halvorsen. *"Improving I/O Performance of Multimedia Servers"*. Phd. Thesis. Unipub. Oslo, Norway. August 2001.

[Handley98] M Handley and V Jacobson. *"SDP: Session Description Protocol"*. Internet Request For Comments 2327. April 1998.

[InStat04] In Stat Press Room. *"NPU Market Gets Reincarnated - Intel Now True Market Share Leader"*. http://www.instat.com/press.asp?ID=863. January 2004.

[Intel01a] Intel Corporation. *Intel® IXA Programming Framework*. "SDK 2.01 Reference". December 2001.

[Intel01b] Intel Corporation. *Intel® IXP1200 Network Processor Family*. "Hardware Reference Manual". http://www.intel.com/design/network/manuals/278303.htm. December 2001.

[Intel02a] Intel Corporation. *"Intel® Internet Exchange Architecture"*. Programmable Network Processors for Today's Modular Networks. http://www.intel.com/design/network/papers/intelixa.htm. Jauary 2004.

[Intel02b] Intel Corporation. *Intel® IXP1200 Network Processor Family*. "Microcode Programmer's Reference Manual". http://www.intel.com/design/network/manuals/IXP1200_prog.htm. March 2002.

[Intel02c] Intel Corporation. *Intel® Microengine C Compiler Language Support*. "Reference Manual". March 2002.

[Johnson02] Erik J. Johnson and Aron R. Kunze. *"IXP1200 Programming"*. Intel Press. 2002.

[Karlin01] Scott Karlin and Larry Peterson. *"VERA: An Extensible Router Architecture"*. Proceedings of the 4th IEEE International Conference on Open Architectures and Network Programming (OPENARCH) . Anchorage, AK, USA. April 2001. pp. 3-14.

[Kind03] Andreas Kind, Roman Pletka, and Marcel Waldvogel. *"The Role of Network Processors in Active Networks"*. Proceedings of IWAN 2003. Kyoto, Japan. 2003. pp. 18-29.

[Kencl02] Kencl and Jean-Yves Le Boudec. *"Adaptive Load Sharing for Network Processors"*. Proceedings of Infocom 2002. New York, USA. 2002. pp. 545-554.

[Kernel04] Kernel.org. *"The Linux Kernel Archives"*. http://www.kernel.org/.

[Kulkarni03] Chidamber Kulkarni, Matthias Gries, Christian Sauer, and Kurt Keutzer. *"Programming Challenges in Network Processor Deploymnet"*. Proceedings of the international conference on Compilers, architectures and synthesis for embedded systems. San Jose, California, USA. 2003. pp. 178-187.

[Kurose04] James F Kurose and Keith W Ross. *"Computer Networking: A Top-Down Approach Featuring The Internet"*. Pearson Education, Inc.

[Lin03] Ying-Dar Lin, Yi-Neng Lin, Shun-Chin Yang, and Yu-Sheng Lin. *"DiffServ over Network Processors: Implementation and Evaluation"*. IEEE Network. Network Processors Vol. 17 No. 4. July 2003.

[Liu03] Jiangchuan Liu and Jianliang Xu. *"A Survey of Streaming Media Caching"*. Technical Report . http://cchen1.et.ntust.edu.tw/mm/Liu-streamCaching.pdf. December 2003.

[Mackenzie03] Kenneth Mackenzie, Weidong Shi, Austen McDonald, and Ivan Ganev. *"An Intel IXP1200-based Network Interface"*. 2nd Annual Workshop on Novel Uses of Systems Area Networks (SAN-2) at the 9th International Symposium on High Performance Computer Architecture (HPCA9). Anaheim, California. February, 2003.

[Mauve02] Martin Mauve, Stefan Fischer, and Jörg Widmer. *"A Generic Proxy System for Networked Computer Games"*. Proceedings of NetGames2002. Braunschweig, Germany. April 2002. pp. 25-28.

[Mirkoviç02] Jelena Mirkoviç, Gregory Prier, and Peter Reiher. *"Attacking DDos at the Source"*. Proceedings of the 10th IEEE International Conference on Network Protocols (ICNP). Paris, France. November 2002. pp. 312-321.

[OMG04] Object Management Group. *"Object Management Group Web-site"*. http://www.omg.org/.

[Peterson00] Larry L. Peterson and Bruce S. Davie. *"Computer Networks"*. 2nd Edition. Morgan Kaufmann Publishers. 2000.

[Podlipnig03] Stefan Podlipnig and Laszlo Böszörmenyi. *"A Survey of Web Cache Replacement Strategies"*. ACM Computing Surveys, Vol 35, No 4. December 2003. pp. 374-398.

[Radisys03] Radisys Corporation. *"ENP-2505/2506 Data Sheet"*. http://www.radisys.com/files/1160_04_1202.2.pdf. January 2003.

[Real04a] RealNetworks, Inc.. *"Helix DNA Server"*. https://helix-server.helixcommunity.org/.

[Real04b] RealNetworks, Inc.. *"Helix Proxy"*. http://www.realnetworks.com/products/proxy [http://www.realnetworks.com/products/proxy/].

[Rijsinghani94] A Rijsinghani. *"Computation of the Internet Checksum via Incremental Update"*. Internet Request For Comments 1624. May 1994.

[Schulzrinne96] Henning Schulzrinne, Stephen Casner, Ron Frederick, and Van Jacobson. *"RTP: A transport protocol for real-time applications"*. Internet Request For Comments 1889. January 1996.

[Schulzrinne98] Henning Schulzrinne, Stephen Casner, Ron Frederick, and Van Jacobson. *"Real time streaming protocol"*. Internet Request For Comments 2326. April 1998.

[Sen99] Subhabrata Sen, Jennifer Rexford, and Donald Towsley. *"Proxy Prefix Caching for Multimedia Streams"*. Proceedings of the IEEE Infocom. New York, NY, USA. April 1999. pp. 1310-1319.

[Shah01] Niraj Shah Shah. *"Understanding Network Processors"*. University of California, Berkeley. September 2001.

[Shalaby02] Nadia Shalaby, Larry Perterson, Andy Bavier, Yitzchak Gottlieb, Scott Karlin, Aki Nakao, Tammo Spalink, and Mike Wawrzoniak. *"Extensible Routers for Active Networks"*. DARPA Active Netwoks Conference and Exposition (DANCE). San Francisco, CA, USA. May 2002. pp. 92-117.

[Sitaram00] Dinkar Sitaram and Asit Dan. *"Multimedia Server - Applications, Environments and Design"*. Morgan Kaufman Publishers. 2000.

[Socolofsky91] T Socolofsky and C Kale. *"A TCP/IP Tutorial"*. Internet Request For Comments 1180. January 1991.

[Spalink01] Tammo Spalink, Scott Kalin, Larry Peterson, and Yitzchak Gottlieb. *"Building a Robust Software-Based Router Using Network Processors"*. Symposium on Operating Systems Principles (SOSP). Banff, Alberta, Canada. 2001. pp. 216-229.

[Steinmetz95] Ralf Steinmetz and Klara Nahrstedt. *"Multimedia: Computing, Communications & Applications"*. Prentice Hall.

[Taylor04] Jim Taylor. *"DVD Frequently Asked Questions"*. http://dvddemystified.com/dvdfaq.html.

[Wang99] Jia Wang. *"A survey of Caching Schemes for the Internet"*. ACM Computer Communication Review Vol. 25 No. 9. 1999. pp. 36-46.

[WindRiver04] Wind River. *"Device Software Optimization: Integrated Embedded Platform and Technology"*. http://www.windriver.com/.

[Xiao99] Xipeng Xiao and Lionel M. Ni. *"Internet QoS: A Big Picture"*. IEEE Network Magazine Vol. 13 No. 2. March 1999. pp. 8-18.

[Yamada02] Tatsuya Yamada, Naoki Wakamiya, Masayuki Murata, and Hideo Miyahara. *"Implementation and Evaluation of Video-Quality Adjustment for heterogeneous Video Multicast"*. Proceedings of The 8th Asia-Pacific Conference on Communications. Bandung. Sept 2002. pp. 454-457.

[Zimmerman03] Roger Zimmerman. *"Implementation and Evaluation of Video-Quality Adjustment for heterogeneous Video Multicast"*. Proceedings of the 2003 ACM symposium on Applied computing. Melbourne, Florida, USA. March 2003. pp. 979-982.

Appendix A. Guide to the attached CD

The attached CD contains all the text, figures and source code produced during the work on the thesis. Also, there are stylesheets used to create html or pdf targets from the docbook xml. The source code consists of about 5000 lines ANSI C and about 500 lines of microcode assembler.

A.1. Directory structure

```
.                    -- Base directory with docbook xml
|-- figures          -- SVG figures and bitmaps
|-- source
|   |-- client       -- The python client
|   |   `-- rtsp
|   |-- configurator -- Our configuration tool
|   |-- data         -- Graphs and data from measurements
|   |-- pol          -- The core classifier/forwarder ACE
|   |   `-- ucbuild  -- The microblocks
|   `-- proxy        -- The RTSP proxy code
`-- tools            -- Tools needed to build the docbook xml
    |-- fonts
    |-- xsl-fo
    `-- xsl-html
```

A.2. Building the source

There are four components that must be built in order to run the prototype proxy: The configurator, the proxy, the core ACE and the microblocks. There are makefiles in every source directory. To build, say the RTSP proxy, you first need the Intel IXA SDK and the location to the SDK root must be exported to the `IXROOT` environment variable. Then, you must make a link to the approperiate makefile, e.g., ixa.mk to build for the strongARM. Simply type **make** and the source will compile and the executable will be copied to the `$IXROOT/bin/arm-be/` directory. The same procedure must be repeated for the remaining components.

A.3. Running the executables

To run the executables compiled for the strongARM you need a properly set up and booted Linux kernel (consult the Intel/Radisys documents on how to do this). Then, simply **telnet** to the card and change directory to wherever the `$IXROOT/bin/arm-be/` directory is mounted. Start the ACE by running **ixstart cfg/ixsys.config-pol**. When it is fully initialized, start the **./proxy**.

The proxy listens to TCP port 10002 for incoming RTSP connections, and the server it tries to connect to is hard coded to `192.168.67.21`. An RTSP server must be running on port 445 at this address to test the proxy. We did not spend time making this configurable, but it will be done, as the code is to be used for a course at the department.