UNIVERSITY OF OSLO
Department of Informatics

CacheCast: a system for efficient single source multiple destination data transfer

Ph.D. Thesis

Piotr Srebrny

June 2011

© Piotr Srebrny, 2011

Series of dissertations submitted to the
Faculty of Mathematics and Natural Sciences, University of Oslo
No. 1111

ISSN 1501-7710

All rights reserved. No part of this publication may be
reproduced or transmitted, in any form or by any means, without permission.

Cover: Inger Sandved Anfinsen.
Printed in Norway: AIT Oslo AS.

Produced in co-operation with Unipub.
The thesis is produced by Unipub merely in connection with the
thesis defence. Kindly direct all inquiries regarding the thesis to the copyright
holder or the unit which grants the doctorate.

Acknowledgements

It is simply not possible for me to name every single person who contributed to this thesis, either by offering direct advice on the merits of it, or by rendering personal support, both being immensely important to me during my doctoral studies period. While attempting to name in particular those to whom I owe the greatest debt, I beg your forgiveness if I have omitted including others.

First and foremost, I would like to thank Thomas Plagemann for this great opportunity to embark on a doctorate under his supervision. Thomas is a wonderful teacher and mentor. He is always willing to listen and to try to understand even the most unreasonable and unconventional ideas! It was an excellent lesson for me to be able to witness just how he explains matters during his lectures and various meetings. I found him to be much more than a supervisor. I would also like to thank Vera Goebel who was always ready to lend a helping hand. I deeply appreciated her presence and supervision, especially during the thesis writing challenge, during which I often tangled myself in knots each long day. I am also thankful to Andreas Mauthe who helped me to clarify my ideas, especially during my stay in Lancaster.

I am deeply indebted to my colleagues from the DMMS group who created a very positive environment for research, including: Azadeh, Daniel, Ellen, Francisco, Hans, Katrine, Kennedy, Kjell Åge, Matti, Matija, Morten, Ovidiu, Piotr, Sebastian and Viet. I am especially thankful to Karl-André who inspired me to challenge the problem presented in this thesis. I am grateful to Jarle who was not only my office-mate, but who also guided me through Norwegian language and culture studies during my early days in Norway. I am also indebted to Stein for the numerous interesting and lengthy discussions that we have had.

I would like to thank all of my colleagues from the CONTENT project, with whom I had the immensely pleasurable opportunity of exploring Europe; and also to my Polish friends from Oslo who were excellent company during my period of 'emigration'. I thank Colin Jeneson for his wonderful company and considerable assistance; and my parents for their support from abroad.

Last but not least, I am also thankful to the Holy Spirit for all forms of inspiration, support, and life experiences I received during this time.

Abstract

The basic function of the Internet is to forward messages hop-by-hop towards their destination hosts. A single message has only one destination and the network does not provide a mechanism for delivering a message to multiple hosts. Therefore, in order to transmit the same message to multiple destinations, a host sends the message to each destination separately. This, however, leads to inefficient use of the Internet resources because packets carrying the same message traverse multiple times the same several first hops from the source. In this thesis we propose CacheCast - a system for single source multiple destination data transfer. CacheCast does not change the host-to-host communication model. It is based on a link layer caching technique that removes redundant transfers of the same data. CacheCast consists of two elements: distributed architecture of link caches and server support. The link caches are designed to work independently. A single link cache consists of two elements that operate on the link end-points. The cache management unit located at the link entry removes from packets data that is already present in the cache store unit located at the link exit. The cache store unit reconstructs the packets from the local cache and passes them to a router for further processing. The server support provides a mechanism for an application to transmit the same data over multiple connections in the CacheCast manner. The resulting packets carrying the data are annotated with information that simplifies redundancy detection and removal. This, in turn, greatly reduces the storage and complexity requirements of link caches. The CacheCast system is incrementally deployable. It preserves the end-to-end relationship between communicating hosts thus it can operate with firewalls or NATs. The CacheCast deployment requires minimal changes in the network operation and the minimum amount of resources.

The thesis evaluates three aspects of the CacheCast system. Firstly, it assesses the efficiency in terms of network bandwidth consumption during single source multiple destination transfer. It shows through analysis and simulations that CacheCast achieves near perfect multicast efficiency. Secondly, the thesis studies impact of the link caches on the network traffic. Simulations performed in the *ns-2* network simulator indicate that CacheCast does not violate current understanding of "fairness" in the Internet. Thirdly, the thesis evaluates the computational complexity of the server support and link cache elements. The server support is implemented as a system call in Linux. The detailed measurements of the system call execution show that it outperforms the standard `send` system call when transmitting data to multiple destinations. The link cache is evaluated in the context of the Click router. Even though, the link cache elements consume the router processing capacity, the CacheCast router can forward much larger traffic volumes than a standard router. Finally, the thesis includes an example of a live streaming application that uses the CacheCast system to transmit audio stream to thousands of clients.

Contents

1 Introduction **1**
- 1.1 Problem of single source multiple destination transfers . . . 2
- 1.2 Thesis problem statement . . . 3
- 1.3 Thesis claims . . . 3
- 1.4 Thesis contributions . . . 4
- 1.5 Methods and approach . . . 5
- 1.6 Thesis structure . . . 5

2 Background **7**
- 2.1 Single source multiple destination transfer . . . 7
  - 2.1.1 Explicit Multicast . . . 9
  - 2.1.2 IP Multicast . . . 10
  - 2.1.3 Application Layer Multicast . . . 10
  - 2.1.4 Summary and insights . . . 11
- 2.2 Caching . . . 12
  - 2.2.1 Cache fundamentals . . . 12
  - 2.2.2 Cache applications . . . 14
  - 2.2.3 Summary . . . 17

3 Design **19**
- 3.1 Terms and considerations . . . 19
  - 3.1.1 Link . . . 20
  - 3.1.2 Router . . . 22
- 3.2 Requirements . . . 23
- 3.3 Fundamentals . . . 25
  - 3.3.1 Placement of cache elements . . . 25
  - 3.3.2 Caching aware source . . . 26
- 3.4 Link cache . . . 27
  - 3.4.1 CacheCast header . . . 27
  - 3.4.2 Link cache mechanism . . . 28
  - 3.4.3 Link cache size . . . 28
  - 3.4.4 Memory utilisation of a link cache . . . 30
  - 3.4.5 Configuration of a link cache . . . 31
  - 3.4.6 Payload ID considerations . . . 31
  - 3.4.7 Errors on a link . . . 32

3.5 Server support . . . 33
3.5.1 Application related tasks . . . 33
3.5.2 Operating system related tasks . . . 34
3.5.3 Operating system support for CacheCast . . . 37
3.6 Resilience and operational considerations . . . 40
3.6.1 Attack on the link cache consistency . . . 40
3.6.2 Attack on the link cache efficiency . . . 41
3.7 Summary . . . 42

4 CacheCast efficiency 45
4.1 Efficiency metric . . . 46
4.2 Header transmission costs . . . 47
4.3 Finite cache size . . . 47
4.3.1 Simulation setup . . . 48
4.3.2 Impact of the finite cache size . . . 48
4.4 Incremental deployment . . . 49
4.4.1 Efficiency gains per hop . . . 50
4.4.2 Cacheable and non-cacheable link . . . 51
4.5 Summary . . . 51

5 TCP friendliness 53
5.1 Requirement of TCP friendliness . . . 53
5.2 *ns-2* implementation . . . 54
5.2.1 *ns-2* CacheCast header . . . 55
5.2.2 *ns-2* link cache . . . 56
5.2.3 *ns-2* server support . . . 59
5.3 Loosely synchronised streams . . . 61
5.3.1 Effect of increasing the number of receivers . . . 62
5.3.2 Effect of decreased caching efficiency . . . 64
5.4 Tightly synchronised streams . . . 66
5.5 Summary . . . 67

6 Computational complexity - server support 69
6.1 Computational bottlenecks . . . 69
6.2 CacheCast header in the Ethernet networks . . . 70
6.3 Server support in Linux . . . 72
6.3.1 Design . . . 72
6.3.2 Linux networking subsystem . . . 72
6.3.3 Implementation . . . 79
6.4 Micro evaluation . . . 85
6.4.1 Evaluation methodology . . . 85
6.4.2 Costs wrt. group size . . . 86
6.4.3 Costs wrt. payload size . . . 86
6.4.4 Per-byte and per-packet cost contribution to the total cost . . . 89
6.4.5 Memory speed impact . . . 90

6.4.6 Cost of user-kernel mode switch . . . 90
6.5 Testbed Evaluation . . . 91
6.6 Summary . . . 92

7 Computational complexity - link cache **95**
7.1 Router Elements . . . 95
7.1.1 CMU and CSU design . . . 96
7.1.2 Click modular router software . . . 98
7.1.3 CMU and CSU implementation in Click software . . . 100
7.2 Evaluation . . . 107
7.2.1 Router model . . . 107
7.2.2 Trace files . . . 108
7.2.3 Router parameters . . . 108
7.2.4 CacheCast router performance . . . 108
7.3 Summary . . . 110

8 Related work **111**
8.1 Network-wide redundancy elimination . . . 111
8.1.1 Storage space . . . 111
8.1.2 Computational complexity . . . 112
8.1.3 Other considerations . . . 113
8.1.4 SmartRE . . . 113
8.2 Other related work . . . 113
8.2.1 Redundancy elimination in multi-hop wireless networks . . . 114
8.2.2 Point-to-point redundancy removal . . . 114
8.2.3 Redundancy removal by compression . . . 115
8.3 Summary . . . 115

9 Conclusions **117**
9.1 Summary of contributions . . . 117
9.1.1 Principles . . . 118
9.1.2 Feasibility study . . . 118
9.1.3 Environmental impact . . . 119
9.1.4 Other contributions . . . 119
9.2 Critical review of claims . . . 120
9.3 Future work . . . 121
9.3.1 CacheCast elements . . . 121
9.3.2 Future research directions . . . 123

Bibliography **125**

List of Figures

2.1 Generic cache design . . . 12
2.2 Packet caching on point-to-point links . . . 16

3.1 Basic idea of the caching mechanism . . . 20
3.2 The link concept . . . 21
3.3 Ethernet network segment . . . 21
3.4 Conceptual view of a router . . . 22
3.5 Input packet processing per link . . . 22
3.6 The first generation of router architecture . . . 23
3.7 Extended directed link . . . 25
3.8 Caching links and routers . . . 26
3.9 IPv4 Packet with the CacheCast header . . . 27
3.10 Relationship between the CMU table and the CSU . . . 28
3.11 The packet train duration time . . . 29
3.12 Network communication in OS . . . 34
3.13 An application on server S connected to hosts B and C using two sockets . . . 35
3.14 Server S transmitting the same data to machines C, D, and E . . . 38
3.15 An application on a host S communicating with processes on hosts: C, D, and E . 39
3.16 Attack scenario - network topology . . . 41

4.1 Transmission of the same data to three destinations: A, B, and C using (a) unicast and (b) multicast . . . 46
4.2 The efficiency of the link layer caching . . . 49
4.3 Incremental deployment . . . 50

5.1 *ns-2* packet . . . 55
5.2 *ns-2* link . . . 56
5.3 *ns-2* caching link . . . 57
5.4 *ns-2* server support . . . 60
5.5 Single bottleneck scenario . . . 62
5.6 Increasing the amount of TFRC flows on a bottleneck link, the **Low RTT** configuration . . . 63
5.7 Increasing the amount of TFRC flows on a bottleneck link, the **High RTT** configuration . . . 63
5.8 Increasing the amount of TFRC flows on a bottleneck link, the **Different RTT** configuration . . . 64

5.9 Increasing the caching efficiency on a bottleneck link, the **Low RTT** configuration 65
5.10 Increasing the caching efficiency on a bottleneck link, the **High RTT** configuration 65
5.11 Increasing the caching efficiency on a bottleneck link, the **Different RTT** configuration 65
5.12 Increasing streaming rate in the bottleneck link topology 67

6.1 Extended directed link 70
6.2 Use of the packet type field in the CacheCast header 71
6.3 The msend system call design 73
6.4 Application data representation on different layers 75
6.5 Socket buffer - data layout 78
6.6 The msend system call execution flow 81
6.7 The CacheCast kernel module and the neighbouring subsystem 84
6.8 Per packet send time as a function of the group size for four different payload sizes 87
6.9 Per packet send time as a function of the payload size for group size of 100 (the Intel machine) 88
6.10 Real system performance testbed. 91

7.1 CMU and CSU relation 97
7.2 Handling the CacheCast header related information in the Click router 97
7.3 Push-pull example for a Click router configuration 99
7.4 Click CacheCast IP router 101
7.5 The router model 107
7.6 Utilised output capacity of the Click router for three different *packet train* sizes and three different payload sizes 109

List of Tables

3.1 Number of headers in packet train transmitted within a given cache hold time . . 29
3.2 Comparison of transport protocols 36
3.3 Number of CMU entries as a function of link capacity assuming 1500B slots . . . 42

4.1 The size of 10ms packet train and its efficiency as a function of the source uplink speed 48

6.1 Per packet system call cost - the linear function fit results for the Intel machine . . 88
6.2 Per packet system call cost - the linear function fit results for the AMD machine . 89
6.3 Server S streaming 320Kbps audio in 25Mbps network. 92

7.1 The ratio of the effective throughputs of the CacheCast router and the original router 110

Chapter 1

Introduction

The history of the Internet began in the late 60's with a research project called ARPANET founded by Advance Research Projects Agency (ARPA). The goal of the project was to create a network for connecting hosts. At that time, the understanding of networking was mainly based on a telephone network. Therefore, the common wisdom was that in order to enable communication, first an end-to-end channel must be established. This requires allocation of resources along the end-to-end path which are released after the communication is finished. This type of network is called the circuit switched network and it suits very well the human communication pattern. Nonetheless, machines are unlike humans, they communicate using short messages or bursts of messages. This communication pattern does not suit the circuit switched network. A transmission of a single message between hosts would require a costly end-to-end channel setup. Additionally, the delay incurred during the channel setup operation would represent a very slow host-to-host communication. An alternative type of network was necessary which would support the rapid exchange of short messages.

ARPANET was based on the packet switching paradigm which was a new approach to communication at that time. A packet switched network does not provide a fixed end-to-end path, rather it forwards messages on a hop-by-hop basis. Network nodes cooperate to build a consistent view of a network topology and maintain this knowledge locally. Thus, each node can evaluate the destination address of a message and forward it to the next hop towards the final destination. The packet switched network matches the requirements of the host communication. A host can transmit a message immediately without any channel setup and resource allocation. This, however, has consequences for the message delivery. Since there are no preallocated resources, there is no guarantee that a message will be delivered.

The work on ARPANET resulted in the development of the Internet Protocol (IP) and the Transmission Control Protocol (TCP) which together are considered the foundation of the Internet. The Internet Protocol (IP) assumes that a network is unreliable, thus, a message can be corrupted, lost, delivered out of order, or even duplicated. The protocol defines the structure of a message together with the addressing scheme and defines mechanisms to detect the message corruption and erroneous behaviour of the network. TCP provides the abstraction of an end-to-end connection for processes and ensures reliable message transfer. The full transition to the new protocols was conducted in 1983. The TCP/IP protocol suit was quickly adopted for other networks which were gradually connected to ARPANET thus establishing the Internet.

Over the last forty years the Internet evolved from a small project to a world wide network

providing connectivity for all types of parties, ranging from individual users to large corporations, governments, banks and institutions. The original concept permitted gradual attachment of new networks to the base inter-network infrastructure. With its growth the Internet provided increasingly more content and services. However, the distributed character of these resources required new approaches which gave birth to all types of overlay systems. At present, the Internet is a platform that provides host-to-host datagram transmission, while the services and content are accessible by a significant variety of distributed systems.

1.1 Problem of single source multiple destination transfers

The original design of the Internet architecture did not consider the single source multiple destination datagram transmission, namely *multicast*. However, already in the middle 80's it was recognized that this functionality is necessary [1, 2] for applications such as distributed databases, distributed computation, or teleconferencing, which require efficient multiple destination data delivery. With content delivery over the Internet being more prominent nowadays multicast is increasingly becoming a necessary feature. Unfortunately, the Internet still lacks widely deployed multicast services.

The first proposal to introduce multicast in the Internet [1] assumed that a source knows all destinations. Thus, whenever it sends a packet it lists all of the destinations in the packet header. This, however, severely limits the number of receivers, since a packet is of fixed size and has only limited header space. Initially this proposal was rejected by the community but it has recently been revised [3] and has been found useful as a complement to IP Multicast for small groups.

The second proposal by Cheriton and Deering [2] is IP Multicast. It assumes that a source does not know the receivers of a datagram, rather it sends a packet to a group of hosts identified by an IP multicast address. Therefore, the network is responsible for the delivery of a datagram to all group members; hence it is also responsible for group management. However, due to many technical and commercial issues it was not widely deployed. According to AmericaFree.TV[1] from 2007 the IP multicast penetration to the Internet was 2.2% measured as a ratio of the number of IP multicast enabled systems to the total number of autonomous systems.

The aforementioned issues related to network layer multicast, and difficulties in the deployment of this class of services, led researchers to address the problem at the application layer. The application layer multicast (ALM) quickly became very popular not only as a research topic but also as a commercial product. But even though ALM can provide users with full multicast functionality it *"introduces duplicate packets on physical links and incurs larger end-to-end delays than IP Multicast"* as Yang-Chu et al. [4] point out. ALM distributes the burden of data transmission across data receivers; however, it does not address the root cause of the problem which is the lack of the network layer mechanism for single source multiple destination data transfers.

At present, the only way to transmit the same data to multiple destinations which are scattered across many autonomous systems is to use multiple unicast connections. However, this results in numerous packets carrying the same payload crossing the same path for many hops. This redundant payload transmission wastes Internet resources and the waste is particularly severe near to sources streaming popular content, like IPTV or IP radio servers.

[1] http://www.multicasttech.com/status/

1.2 Thesis problem statement

The host-to-host communication paradigm, which the Internet is based on, is decreasing in importance and is gradually being replaced by the content centric paradigm, where content – instead of a host – is in the centre [5, 6]. In this model, the main transport mechanism is single source multiple destination transfer, while point-to-point transfer appears as a special case where only a single user expresses an interest in content.

At present, the Internet does not provide efficient mechanisms for single source multiple destination transfer. The network layer multicast does not work on the large scale and is inherently insecure. The overlay solutions do not address this problem at the root, but provide workaround methods where transmission costs are distributed among participating parties. This leaves us with unicast as the only method for data transfer; a consequence of which the amount of redundancy in the Internet increases.

The goal of this work is to remove the Internet redundancy in a minimal invasive way, i.e., using the minimum amount of resources and with minimal changes in the network operation.

1.3 Thesis claims

This thesis explores the feasibility of suppression of the redundant payload transmissions by applying a new caching technique on the Internet links. The thesis studies are based on the CacheCast system, our reference system for the link layer redundancy elimination. The claims of this thesis are the following:

Claim 1: *A system of link caches can achieve near multicast bandwidth savings for a superposition of unicast connections.* The main benefit of using IP Multicast for data delivery is efficient utilisation of network bandwidth. However, this is achieved at the cost of a huge management burden in the network. We claim that a system of link caches can achieve similar utilisation of network bandwidth without the management burden in the network.

Claim 2: *A system of link caches requires server support in order to be feasible.* In the context of single source multiple destination data transfers, a source has all information on the data that it generates and it also controls the data transmission. Thus, the source can support link caches by providing information on packet redundancy and by transmitting the same data within the minimum amount of time. We claim that this support reduces link cache requirements to the point where it is feasible to implement them on all Internet links.

Claim 3: *A system of link caches is incrementally deployable.* In a large scale system it is highly recommended that new functionality is incrementally deployable. Functionality which is not incrementally deployable will not yield benefits until fully deployed, e.g. IP Multicast. In the case of the Internet, which is owned by multiple parties, common agreement between parties must be achieved in order to enable this functionality and to obtain benefits. This poses additional difficulties for attempts to achieve full deployment, since it greatly increases costs of initial investment. In contrast, an incrementally deployable functionality provides immediate benefits at deployment of the first element. We claim that a system of link caches is incrementally deployable and yields benefits at the first link cache deployment.

Claim 4: *A system of link caches maintains fairness with respect to bandwidth sharing on a bottleneck link.* The Internet traffic consists of flows which are controlled by protocols. A protocol instance runs on the flow end points and adjusts the flow throughput to meet the network capacity. If the network becomes congested the protocol instance reduces the flow throughput to ensure equal bandwidth sharing, i.e. "fair sharing", with other flows. The protocol instance infers the network condition based on the behaviour of packets which compose the flow. We claim that link caches do not disturb the operation of current protocols. The protocols can ensure "fair sharing" of the network resources in the presence of link caches.

1.4 Thesis contributions

The main contributions of this thesis include:

Principles

The first key contribution consists of a set of principles for the design and implementation of a link layer caching technique. The principles are established based on thorough analysis of (1) network element capabilities and characteristics, (2) former and currently established systems for single source multiple destination transfers, and (3) requirements for this type of system. We conclude that a caching technique which operates on network links must be supported by sources of redundant data. A source transmitting the same data to multiple destinations should batch the transmissions and annotate the data with information that simplifies redundancy elimination. In this way, the link cache complexity is significantly reduced. Based on these principles we have designed and implemented CacheCast - a system comprising unique architecture that consists of a distributed system of link caches and server support.

Feasibility study

The second key contribution is a feasibility study of the proposed principles. The study is based on the CacheCast system. In order to measure the system efficiency, we use a simple model capturing only basic characteristics of the CacheCast system, such as finite cache size, or packet header transmission. The measurement results obtained confirm the feasibility of this approach for suppressing redundancy introduced by single source multiple destination transfers. We also evaluate the CacheCast system with respect to computational complexity. The server support part is implemented in a Linux operating system and the link cache element is implemented as part of a software router. Finally, in order to prove the applicability of this approach we adapt an open source live streaming application to the CacheCast system. We show that the modified live streaming software can serve more clients than the original software, by a significant order of magnitude.

Environmental impact

The third key contribution is a study of the environmental impact of the CacheCast system. Internet fairness is based on the assumption that different packet flows should obtain the same share of a bottleneck link. However, when a link cache operates on a bottleneck link, packet flows carrying redundant data obtain much less of the bottleneck link capacity. Using a network simulator, we study this issue in a small network possessing a bottleneck link topology.

1.5 Methods and approach

The applicability of a caching technique for removing redundant payload transmission requires thorough analysis. A broad range of aspects of the system design is reflected in a range of methods which we use during the analysis. In the following points, we briefly discuss the problems addressed and the methods which we use to approach them.

- First, we establish where the caching technique should be applied in order to achieve the goal of the redundancy elimination. This involves studies of two basic elements of the Internet infrastructure, namely a link and a router. The study covers both the functional side and the implementation side of the elements and results in the design of a link cache.

- Given a link cache we perform the initial assessment of bandwidth savings that can be obtained when deploying a system of link caches in the Internet. The system is compared with a "perfect" multicast, i.e., a multicast which avoids any redundant transmission and does not incur management overheads. We use graphical analyses to determine the differences in bandwidth savings between the systems and to understand the impact of the cache size.

- It is not given how end-to-end protocols will react in the presence of link caches. Congestion control mechanisms built in the protocols estimate the throughput of a flow based on factors such as packet size, packet arrival frequency, or end-to-end delay. However, these factors may be affected in the presence of link caches. In order to understand the protocol behaviour we use the network simulator *ns-2* and perform simulations in a typical bottleneck link topology, where the bottleneck link is a caching link.

- The efficiency of a link cache model depends solely on the cache size. However, when implemented the link cache will not achieve the model efficiency. To understand how fast it can operate, how much resources it consumes, and what are its bottlenecks we implement the link cache in the Click modular router software and evaluate it in the context of network operation.

- A system of link caches requires support from a server. However, the additional load on a server related to the link cache support may reduce the benefits of using link caches. Additionally, it is unknown whether the support can be integrated into the server operating system. We address these concerns by a clean design of the server support in the Linux operating system and a thorough evaluation of the server support implementation.

1.6 Thesis structure

In this chapter we provide a brief history of the Internet and introduce the problem of single source multiple destination transfers. Chapter 2 provides the background for this thesis which consists of two parts. In the first part, we elaborate the issue of single source multiple destination transfers, and in the second part, we describe different caching techniques. We focus especially on the point-to-point redundancy suppression techniques which are closely related solutions to CacheCast.

Chapter 3 presents the design of the CacheCast system. Firstly, we introduce the CacheCast idea. Then, we analyse requirements and discuss properties of basic network elements. Based on

these inputs, we give a rationale for our design decisions to distribute the caching burden between a server and link caches. Next, we present the detailed design of the CacheCast elements, viz., link cache and server support. Finally, we consider resilience and operational aspects of the system.

The next four chapters describe evaluation results of different aspects of the CacheCast system. Chapter 4 presents the system efficiency with regard to the amount of removed redundancy from network links during single source multiple destination data transfers. Additionally, it assesses the benefits of incremental deployment. Chapter 5 presents the link cache impact on congestion control mechanisms. The measurements are performed in the network simulator *ns-2*. We also provide a description of the *ns-2* CacheCast implementation. Chapters 6 and 7 assess the computational complexity aspect of the CacheCast system. Chapter 6 presents the design of the server support and discusses issues related to the server support implementation in the Linux operating system. The implementation is thoroughly evaluated with regard to computational complexity. Additionally, the chapter presents an example of CacheCast enabled audio streaming in a testbed network. Chapter 7 covers the design and evaluation of the link cache elements. The elements are implemented using the Click modular router software and the evaluation is performed in the context of router operation.

In Chapter 8 we compare the CacheCast system design with a closely related system for network-wide redundancy elimination. We also present other related works to give a wider picture of redundancy removal techniques. Finally, Chapter 9 presents the thesis summary. We evaluate the thesis contributions, and claims, and also discuss future work, both short term and long term.

Chapter 2

Background

In the previous chapter we have introduced the problem of single source multiple destination transfer in the Internet. Despite a considerable research effort to enable network layer multicast in the Internet, at present the service is still not available for an average user due to numerous problems. Thus, the only method to transmit the same data to multiple destinations is to use multiple unicast connections. However, this approach creates an unnecessary redundancy in the network traffic.

This chapter provides the background knowledge necessary to understand the motivation for the thesis and to position the proposed system among other related solutions. The chapter is divided into two parts. In the first part we give a thorough description of the problem of single source multiple destination transfer in the Internet and in the second part we describe caching techniques. Single source multiple destination transfer can be realised at the network layer or at the application layer. At the network layer, multicast is approached using two different techniques, viz., explicit multicast and IP multicast; however, both techniques have not been successfully deployed in the Internet. We describe these two techniques and discuss reasons for the failure in deployment. Application layer multicast operates only on end systems therefore it does not require any changes in the network infrastructure. This ease of deployment contributed to quick development of application multicast solutions and resulted in a great number of systems. We describe principles of application layer multicast and the basic designs. To conclude the first part of the background, we compare the network layer and the application layer approaches to multicast and derive insights for our system design.

This thesis presents a system of link caches and therefore, in the second part of the background, we introduce the caching technology. The technology has a broad range of applications which we illustrate with a few examples. Finally, caching has already been employed on network links in a similar manner to that presented in this thesis. We describe these related works in brief and we compare them in detail with our system towards the end of the thesis, since this requires a thorough knowledge of our system.

2.1 Single source multiple destination transfer

Single source multiple destination transfer can be understood in different ways within the context of data transmission in a network. In this thesis, we use this expression to refer to an act of synchronous transmission of the same data to multiple receivers by means of an application. We do not use the word *synchronous* in an absolute form but rather as an effort of an application to achieve

simultaneous transmission using available means. We use the expressions *single source multiple destination transfer* and *multicast* interchangeably except when referring to a specific technology like IP Multicast, or Application Layer Multicast. To illustrate this definition, we give three examples. Let us consider an IP network providing both unicast and IP multicast transport methods. The network connects a server and a set of client machines. We regard the following scenarios as examples of single source multiple destination transfers:

1. Several clients request the same content from the server. The server batches all requests and transfers the content to the clients using multicast. If the requested content does not fit a transfer unit, it is divided into chunks and the server transfers successively all chunks to the clients.

2. Several clients request the same content from the server. The server batches all requests and transfers the content successively to all clients using unicast. If the content does not fit a transfer unit, it is divided into chunks and the server transfers the subsequent chunks to all clients using unicast.

3. The server streams live content using unicast. The clients request the stream from the server at different points in time. However, since the stream carries live content, data transfer is synchronous, i.e, the clients receive the same data (e.g. a new audio sample in the case of live audio streaming).

As can be derived from our three examples, in the centre of our definition is the synchronous transmission. In the first two examples the clients initiate data transfer; thus, the server must batch these requests in order to perform synchronous transfer. In the third example, the server initiates transfer whenever it has a new sample for transmission; thus, the transfer to all clients is synchronised *per se*. To give an example of what we do not regard as single source multiple destination transfer, let us consider the following scenario. Clients request content from a server. However, the server does not batch the requests but rather it starts to transfer the content at the time a client requests it. In this example the server does not attempt to synchronise the transfers but sends the content to each client individually; therefore, we say that the server delivers the content using single source single destination transfers.

In order to transfer the same data to multiple destinations using multicast, a server must make an effort to synchronise transmissions to individual destinations. This effort varies depending on the transmitted data type. In this context we give two examples of data type: live-data and on-demand-data. The live-data is valuable at the time when it is being created (e.g. a transmission of a football match or a video conference) therefore it is immediately sent to clients who are interested in it. This property makes live-data suitable for multicast transmission without additional processing, the content is synchronised *per se*. In the case of on-demand-data, clients request it at different points in time and download it at different speeds. Thus, in order to deliver on-demand-data using multicast, a server must employ advanced techniques to synchronise transmissions to individual clients. However, these can consume a considerable amount of the server CPU power.

Different data types require different adaptation techniques to make the data amenable for multicast transmission. These techniques, in turn, consume additional CPU power of a server. Therefore, the relevant question is: What is the benefit of using *single source multiple destination*

transfer when compared to multiple *single source single destination* transfers? According to our definition multicast transfer is a synchronous transmission of the same data to multiple receivers by an application. This synchronous transmission can be exploited by an operating system (OS) or a network mechanism to handle efficiently the data transfer. At present, there is a broad range of systems for efficient handling of multicast transfer. We focus our discussion on the following three solutions:

1. Explicit Multicast
2. IP Multicast
3. Application Layer Multicast (ALM)

The list of systems for multicast transfer is not complete but it contains the key systems. We do not include in the list a class of content distribution networks (CDNs), since these solutions (when considering multicast transfers) are similar to ALM. The common idea is to distribute the server load. For this purpose ALM utilises end-host resources and CDNs rely on a dedicated infrastructure.

2.1.1 Explicit Multicast

The necessity to support single source multiple destination data transfers in the Internet was recognised already in the middle 80's and the problem was approached from two directions. The first direction was set by Aguilar in [1]. He proposed modifying the structure of the IP header. The standard IP header contains a pair of source-destination addresses which reflect the *unicast* character of a packet. Aguilar suggested that a *multicast* packet should contain a list of destination addresses instead of a single destination address. Thus, when a server transmits the same data to multiple destinations, it lists all destination addresses in the packet header. This new IP packet is processed on a router in the following way: first, a router evaluates all destination addresses from the list and determines the next hops; second, it splits the destination list into sub-lists which contain destinations reachable via the same hop; third, it creates new packets for each hop with the list of destinations; and finally, the new packets are forwarded to the next hops. This process is performed by each router on a packet route to destinations, until the resulting packets have only one destination address when they become standard unicast packets. Please note that in this process a router forwards packets according to the unicast routing table and there is no necessity for multicast routing.

While Aguilar's approach to multicast is very simple, it does not scale well with the growing number of destinations. Considering the limited size of a packet, the more destinations are addressed the less space is left for data. Moreover, Explicit Multicast does not completely eliminate redundancy from single source multiple destination data transfers. When a destination list does not fit a single IP header, it must be split into sub-lists. Consequently, a new packet carrying the same data is created for each sub-list. The approach was deemed to be unscalable by the networking research community and it was dismissed soon after Aguilar proposed it. Nonetheless, a decade later it was revisited in [7, 8] and it was found useful as a complement to IP multicast. At present, it is known as Small Group Multicast (SGM) or XCast (eXplicit Multicast) and is standardized in [9]; however, since it is not deployed in the Internet, it still remains a research concept.

2.1.2 IP Multicast

The second approach to *multicast* was proposed by Cheriton and Deering in [2] and it is known as IP Multicast. While Aguilar's approach focused only on the multicast transfer mechanism, Cheriton and Deering proposed a more comprehensive approach to the problem. IP Multicast is based on a host group model where a set of hosts is identified with a single multicast IP address. The Internet maintains the knowledge of the group members and actively constructs routes connecting them. If a host wants to join or leave a group, it notifies a network using the Internet Group Management Protocol (IGMP) [10]. Each membership change propagates through the network and the multicast routes are updated. In order to transmit data to a group, a server using the group address sends the data to the network. The data is forwarded and replicated along the multicast routes of that group. The original IP Multicast model permits any source to transmit data to a group, even if a source is not a member of the group; thus, it is called Any-Source-Multicast (ASM).

As Diot et al. point out in [11] the open multicast model is prone to malicious attacks, it requires global address allocation, it does not provide any billing model, and it reveals serious scalability issues due to group management in the network. To address some of these problems, Holbrook and Cheriton proposed an extension to multicast called EXPRESS [12] – EXPlicitly REquested Single-Source multicast. EXPRESS uses the semantics of a multicast channel, instead of a multicast group. A multicast channel has only one source and is identified by a tuple (S,E) where S is the source address and E is the channel number. This solves the problem of global address allocation and the shortage of multicast groups. Moreover, since only a source S can transmit to channels (S,*) no malicious source can pollute the channels. A source can secure a channel with a key; thus, only hosts that present the key can join the channel. Considering the billing model for multicast, EXPRESS provides a service for counting channel subscribers. Since a channel has only one source in the EXPRESS model, it is easy to charge the source based on the number of subscribers.

At present, IP Multicast still lacks vital services like reliable receiver authorization, authentication, and also accounting (AAA) in order to be accepted by content providers and Internet Service Providers (ISP). Moreover, IP multicast requires per group state on forwarding nodes which poses heavy burden on backbone routers. Another issue is the difficulty to construct the multicast congestion control [13]. The solutions for handling heterogeneous receivers are based on layered transmission which in turn requires allocation of expensive multicast channels. Finally, IP Multicast breaks the end-to-end relationship between communicating hosts, thus rendering communication inherently insecure. Altogether, these issues account for the lack of IP multicast services in the Internet.

2.1.3 Application Layer Multicast

The two aforementioned approaches to multicast address the problem at the network layer. Explicit Multicast proposes to extend the IP header with additional destination addresses. This new IP header requires a different type of processing on the Internet routers. Similarly, IP Multicast requires updating routers. Routers must run the distributed group management, co-operate to build multicast routes between group members, and forward and replicate IP Multicast packets. The requirement of router level deployment in the whole network greatly contributes to the failure of these solutions. Routers are updated very rarely, typically when old routers are replaced by new

ones. This, in turn, raises an issue of interoperability and increases costs of initial investment.

The reluctance to deploy and enable network layer multicast, directed the research effort to application layer solutions that require only unicast transfers. The complete functionality of Application Layer Multicast is implemented at the hosts that are part of a multicast group. This accounts for rapid development of this class of solutions, since prototypes can be tested immediately in the real world scenarios. The earliest examples of ALM from the research community include systems like: Narada [4], NICE [14], Overcast [15], or Splitstream [16]. It was quickly recognised that ALM is capable of efficiently delivering data to a large number of clients and the ALM technology was adopted for commercial purposes, primarily to stream television channels. At present, systems like PPLive [17], PPStream [18], Sopcast [19], or QQLive [20] attract a vast number of listeners. For instance, the largest ALM PPLive claims to have more than 100 million users[1].

Although the different ALM systems vary in their architecture, they share the common idea of distributing the server load during data transfer among clients. This is achieved by arranging clients in a tree structure where a server is at the root of the tree and the clients form subsequent levels of the tree. The clients are in the parent-child relation, where a parent transfers data to children and the first parent is the server. A parent has a limited amount of children and this number is called *degree*. The degree parameter controls the tree shape. If the degree is high, the tree is short and therefore the latency between the server and last client is low; however, the client load is high. Reversely, if the degree is low, the tree is long and therefore the transmission latency is high; however, the client load is low. In practice, the distribution tree is unbalanced - a few powerful hosts serve many poorly connected clients (a host with an asymmetric connection, like ADSL, often cannot serve even a single child due to the low uplink bandwidth).

The main challenge in ALM is to optimise the delivery tree structure under continuous changes in network conditions and churn of group members. The amount of redundancy generated by an ALM solution is smaller than the amount generated by purely a server-client model. ALM exploits clients locality to efficiently deliver data. It can cluster clients by network location, thus, a server sends data to only one client from a cluster which then distributes it to its neighbours. Nonetheless, an ALM client that actively participates in the distribution tree, transfers the same data to its children using unicast, thereby creating redundancy at the network layer. Moreover, even the optimal ALM solutions incur additional delays during data transfer.

2.1.4 Summary and insights

Explicit Multicast and IP Multicast provide the mechanism for single source multiple destination transfer at the network layer. Explicit Multicast implements only the transfer mechanism and a server is responsible for group management. Therefore, an Explicit Multicast packet carries a list of group member addresses. To overcome this limitation, IP Multicast combines the transfer mechanism together with a group management system. The destination of an IP Multicast packet is a group of hosts identified with a single address. While IP Multicast is scalable regarding group size, it does not scale well regarding group number. The cost of the group management in the network is difficult to justify with the degree of efficiency obtainable. Furthermore, the host group model breaks the end-to-end relationship between communicating parties, causing a number of security issues.

[1] http://download.pptv.com/en/about.html

Even though the host group model provides a clean solution for the multicast problem, it does not suit well the Internet architecture. The Internet design decisions follow primarily the end-to-end arguments of Saltzer et al. [21]. Saltzer argues that the core network should provide only basic functionality upon which higher layers can build diverse services and applications. Application layer multicast indicates that group management can be successfully implemented at the higher layers; however, it requires a network primitive to support single source multiple destination transfer.

2.2 Caching

Caching makes access to data faster by adding transparently a small secondary storage unit to the system. The secondary unit is either faster, i.e., within a computer - cheap and slow storage units are paired with faster and more expensive storage units to improve access time; or this secondary storage unit is located closer to the process that requested the data. In this case redundant transfers over networks are avoided.

Caching effectiveness follows from a phenomenon that most of process requests can be served from a relatively small storage unit. This is because in most systems the request pattern exhibits temporal and spatial locality of reference. The temporal locality of reference occurs when a process requests the same data in short time periods. For instance, CPU often executes the same instructions due to frequent loops in computer programs, or users often access the same favorite web-sites. The spatial locality of reference occurs when a process within a short time period requests nearby data elements. For instance, most CPU instructions are executed sequentially, or within data structures related data elements are located nearby.

2.2.1 Cache fundamentals

A cache is a small storage unit that keeps a subset of data from the primary storage unit. As it is depicted in Figure 2.1, a cache is organised as a pool of slots containing copies of data from the primary storage unit. Each copy of data stored in a slot is identified with a tag. The same tag is used to identify the original of this datum in the primary storage unit. In order to access an element in either of the storage units, a system uses a tag to access the datum containing the element and an offset to access the element in the datum. For instance, in a computer main memory an instruction address determines both tag and offset of the instruction. With the increasing cache granularity, more bits of the instruction address line are allocated for the tag part and less for the offset part.

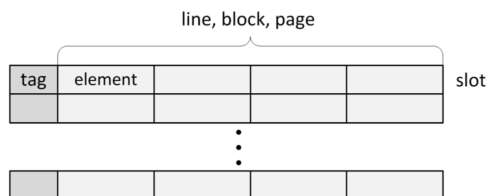


Figure 2.1: Generic cache design

The cache assisted read operation is performed in two steps. When a process requests an element, first, the cache is queried using the element tag. If the tag is found in the cache, the associated

element is immediately served from the cache and we say that a *cache hit* occurred. If the tag is not found in the cache, the element is served from the primary storage unit incurring high access latency and long transfer time. In this case, we say that a *cache miss* occurred. In the second step during *cache miss* a datum containing the requested element is inserted into the cache. Since the cache space is limited, the new datum replaces the datum chosen from the cache according to the *replacement policy*. The task of the cache replacement policy is to point to a datum that is most unlikely to be referenced in the future, thereby maximising cache utilisation.

During the cache assisted write operation, a system does not write an element directly to the primary storage unit, but transfers it to the cache. Then, the cache is responsible for writing the modified datum to the primary unit to maintain coherency. In general, there are two policies to handle the cache write operation: write-through and write-back. A cache with the write-through policy immediately transfers a modified datum to the primary unit. A cache with the write-back policy delays the transfer to the primary unit until the modified datum is not evicted from the cache. This policy can yield better performance, since other elements in a single datum may be accessed and modified by the system multiple times before the datum is evicted. However, the gain in performance is achieved with temporal inconsistency.

Replacement policy

A good replacement policy should guarantee efficient use of the cache space. The cache efficiency is usually measured by the cache hit ratio therefore the goal of the replacement algorithms is to maximise this parameter. Obviously, it is not possible to provide the optimal replacement algorithm, since this requires knowledge of future access requests. Therefore, based on the previous request pattern the replacement algorithm approximates the future behaviour.

One of the most popular cache replacement policies is the Least Recently Used (LRU) policy. LRU evicts the least recently used element from the cache. Therefore, the elements that are accessed frequently are unlikely to be evicted. The policy yields high hit ratio for most of the caching systems. However, in order to achieve better cache performance, the replacement policy should match the system request pattern. Thus, it must be designed to match the content characteristic and the system behaviour. For example, in [22] Robinson and Devarakonda show that a frequency-based replacement policy outperforms the LRU policy when managing caches used for disk blocks by a file system, a database management system, or a disk control unit.

Optimisation of the cache replacement algorithm is not always beneficial. With the growing complexity of the replacement algorithm the cache response time decreases. Complex algorithms require more processing power and local storage space which are not always available. For example, the CPU cache operates in a very constrained environment where the replacement algorithm is implemented in hardware. Moreover, to decrease lookup time in the cache a single line can only be mapped to a subset of possible cache slots. Therefore, this type of cache implements only a limited version of the LRU policy. Virtual memory management systems also do not implement the basic LRU policy for disk cache, but use the less complex clock replacement algorithm [23, 24]. The algorithm does not achieve the LRU efficiency, but provides faster response times.

Cache Coherence

A cache holds a subset of elements from a primary storage unit for fast access. These copies of elements should reflect the original elements, and only then, can we say that the cache is coherent with the primary unit. A cache may become incoherent in two situations: (1) a process modifies elements in the cache, or (2) an external process modifies originals of the cached elements in the primary storage unit. We have already discussed the first situation with the cache write policies. To recap, the write-through policy provides strict coherence while the write-back policy permits temporal incoherence for the sake of performance. The second situation, whereby an external process modifies elements in the primary storage unit, occurs often in environments where multiple systems access the same storage unit like a distributed file system, multi-core processors, or World Wide Web (WWW) services.

The importance of cache coherence depends on the system characteristic and it is not always crucial. Usually, systems that do not involve humans also do not tolerate inconsistencies. For example, multi-core processors demand strict coherency between caches and a main memory. Deviations from coherent state may cause system crash. Therefore, additional mechanisms are employed to ensure the coherent state – both in hardware and software. Other considerations are required for WWW proxy caches. Users accessing a web-page are often willing to get a stale web-page if they do not have to wait a long time [25]. Hence, a proxy cache often serves stale web-pages as long as their expiration time has not elapsed. Only then, will the proxy cache fetch the web-page from the server.

2.2.2 Cache applications

Caches are ubiquitous in computer systems. We have already mentioned some of the cache applications, in order to support our description of the cache mechanism. In the following paragraphs we extend these examples and discuss application specific cache design considerations. Finally, we describe a cache for point-to-point data transfers in the Internet. This type of cache in many aspects resembles our solution, however, here we describe only principles, providing a detailed comparison with our system towards the end of the dissertation.

CPU caches

From the very beginning of computer systems, the instruction and data transfer from the main memory to CPU posed a bottleneck for system performance. The CPU executes instructions an order of magnitude faster than the speed instructions can be fetched from the main memory. To bridge this gap, already in the early 80's most large and moderate computer systems were equipped with cache memories [26]. CPU cache has multiple design considerations, however, here we mention only two: access time and hit ratio.

To provide the shortest access latency, modern CPU caches are integrated in the same circuit as the processing unit. In addition to the cache placement, the cache lookup time constitutes a considerable part of the access time. In order to minimise the lookup time a single memory line can be placed only in a limited number of cache slots. Thus, based on the line tag the cache knows immediately which set of slots to query. A directly mapped cache permits a single line to occupy only one location in the cache. A less strict scheme - a set associative cache - permits a single line to

be stored in alternative locations. While the flexibility increases the cache hit ratio, it also extends the lookup time.

The cache hit ratio is directly proportional to the cache size. The larger the cache, the higher the hit ratio. However, increasing cache size also results in increased access latency, which is undesirable. One solution to this problem is a multi-level cache hierarchy [27]. By providing a second level cache, we can efficiently handle first level cache misses. Therefore, the second level cache allows reduction in the size of the first level cache, thereby increasing its responsiveness.

Disk caches

In computer systems, disks provide an inexpensive storage space for large amounts of non-volatile data. However, disks are mechanical devices and thus have a high access latency due to disk head positioning. Moreover, advances in disk technology do not help to reduce the gap between the disk access time and the CPU speed which is instead increasing. Therefore, processes performing input/output (IO) operations are often blocked for a long period of time while waiting until the requested data element is fetched to the main memory. To overcome the IO bottleneck limitation, disk caches were proposed as a solution [28].

Disk caches use a part of the main memory to store frequently accessed blocks of data. This improves the computer system performance in two ways. Firstly, the disk cache moves large blocks of data, which is much more efficient than moving small data elements upon each IO request. Hence, when a process performs multiple short IO operations on a file, most of the operations are transparently handled by the disk cache. Secondly, the disk cache eliminates many IO write operations, since data elements are only written to blocks in the main memory. The modified data blocks are marked as dirty blocks and are written back to the disk either when evicted from the disk cache or upon explicit system request (like the `sync` command in Unix systems).

In most operating systems, the disk cache functionality is implemented by the virtual memory management system. When a file is open, it is mapped to a memory address space which is done implicitly by a process loader, or explicitly with for example the `mmap` system command in Unix operating systems.

WWW proxy caches

The first caches were primarily used to accelerate CPU access to main memory, and to improve IO throughput when accessing disks. With the advent of the Internet, the caching technique was also applied to web services. The idea of WWW proxy caches arose as an extension to proxy servers. A proxy server originally provided access to web services for users located in a subnet protected with a firewall. The proxy server was installed at the gateway and it forwarded all HTTP requests generated by the subnet users to web servers. Since the subnet users often share common interest, the proxy server was an excellent place to build a web-page cache [29].

A proxy cache reduces network traffic generated by users that are located in the same subnet, since it eliminates redundant transfers of the same web-page from a remote web server to different users. At the same time, the proxy cache cuts down the access latency to cached web-pages. However, due to performance issues, proxy caches do not maintain strict coherency between cached documents and their primary copies on web servers. Cached documents have an assigned expiration time which is usually a fixed time period for all documents. The proxy cache does not check the

validity of documents until the time expires. Therefore, proxy caches often serve stale documents.

While proxy caches remove a part of redundant transfers in web traffic, in practice, it appears that it is only a small fraction. Users demanding fresh documents force caches to always fetch a web-page from a server, regardless of the age of the document copy. Moreover, a proxy cache cannot handle requests for content that: requires authorisation, is generated dynamically, or is personalised. It also does not recognise the same content mirrored on a different server, since it caches content by the URL address.

Point to point link caches

To overcome the shortcomings of a proxy cache Spring and Wetherall proposed in [30] a packet level redundancy suppression. The technique is applied on a point-to-point bandwidth constraint channel like for example: an access link, a wireless link, or a path between a server and a client. The algorithm requires two caches that are located at the channel entry and at the channel exit (see Figure 2.2). The packet transfer is handled in the following way. Before a packet enters the channel, it is compared against the cache content and if any substring of the packet is found in the cache, it is substituted with a tag. Thus, packets traversing the channel consist of unique byte strings and short tags replacing redundant information. On the channel exit, the packet is reconstructed from the cache by replacing the tags with data. To ensure the correct operation, the caches must be consistent.

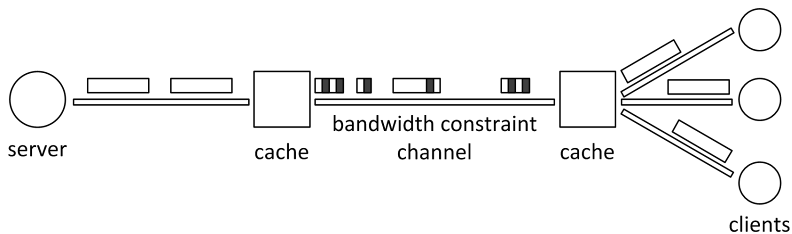


Figure 2.2: Packet caching on point-to-point links

The packet level redundancy suppression is protocol independent. Therefore, it can eliminate redundancy regardless of the HTTP semantics. Since it operates at the level of byte strings, it can detect redundancy in personalised or dynamically generated content. Furthermore, it does not need to be updated to support new types of protocols or content. These distinct features led to fast commercialisation and the technique became a fundamental part of WAN optimisers [31–34]. Recently, Aggarwal et al. [35] proposed to incorporate the end-to-end redundancy suppression technique as a part of a network stack.

In [36], Anand et al. propose to deploy the redundancy suppression technique as a universal primitive on all Internet links. The network-wide redundancy elimination would considerably decrease load on links and make networks resilient to flash crowds. As the authors argue, even greater improvements can be obtained by modifying forwarding routes to extend a common path of packets carrying redundant content. The link layer redundancy elimination requires the installation of a pair of caches between a router egress and a downstream router ingress ports for each router link. Caches installed at router egress ports remove redundancy from outgoing packets and downstream caches reconstruct the incoming packets.

As the authors point out in the followup [37] this naive approach does not take into consideration the limited amount of memory available on routers and the memory throughput bounds that renders the solution infeasible. The new design addresses these constraints with a distributed architecture. To reduce computational effort, packets are not cached on the hop-by-hop basis. Rather, an encoded packet may be reconstructed by a cache located a few routers downstream. This greatly increases complexity of the architecture, since it requires coordination between different link caches which raises difficulty in inter-cache consistency.

2.2.3 Summary

Caching became a standard element in computer architectures that transparently accelerate system operation. It is present on all levels of the computer system architecture. It cuts down access latency and transfer time of small portions of data such as single instructions in the CPU to large transfers of content from web-servers to clients. Caching also eliminates redundant transfers of the same data over the same channel. In this context, it trades memory and computational power for channel bandwidth.

Chapter 3

Design

In this chapter we present CacheCast - a link layer caching mechanism that removes redundancy from single source multiple destination transfers. The basic idea of CacheCast is presented in Figure 3.1. To ilustrate the idea, we consider two consecutive packets that originate from the same source, traverse a few hops over a common path and carry the same content but have different destinations. The first packet traverses the path hop-by-hop. Each hop along the path caches the packet payload and records the output link for the packet (steps (1) and (2)). When the second packet enters the path, the first hop determines the output link for the packet (steps (3) and (4)). It recognizes that it has sent the first packet over that output link with the same payload. Since the payload is already in the next hop cache, it sends the packet header. The same occurs on each hop until the last hop of the common path is reached. The last hop determining the output link for the second packet recognizes that it will travel a different path than the first packet. Thus, the payload cannot be present in the next hop cache. The last hop attaches the payload from its cache to the second packet header, sends the entire packet and records the output link.

The CacheCast system operates on the link layer and its goal is to remove redundant payload transmission from links. In order to find an instantiation of CacheCast for packet switch networks such as the Internet, firstly, in Section 3.1 we provide basic terms and considerations for network elements. Secondly, in Section 3.2 we present a list of requirements that CacheCast should satisfy. Thirdly, in Section 3.3 we discuss two fundamental elements of the CacheCast system, i.e. link cache and server support. The detailed designs of these elements are described in Section 3.4 and Section 3.5. Finally, in Section 3.6 we consider resilience and operational issues related to the CacheCast design and we conclude the chapter in Section 3.7.

3.1 Terms and considerations

Packet-switched networks such as the Internet consist of links and routers which provide end-to-end packet delivery. In the following we use the Internet as an example to demonstrate how CacheCast works. In the Internet links are communication channels that transport packets over a distance, while routers connect multiple links and forward packets towards their destinations. Since the meaning of the link and router terms is broad and depends on the context, in this section we provide our terminology to avoid ambiguities. Furthermore, we discuss assumptions related to the elements and analyse to which extent these assumptions hold.

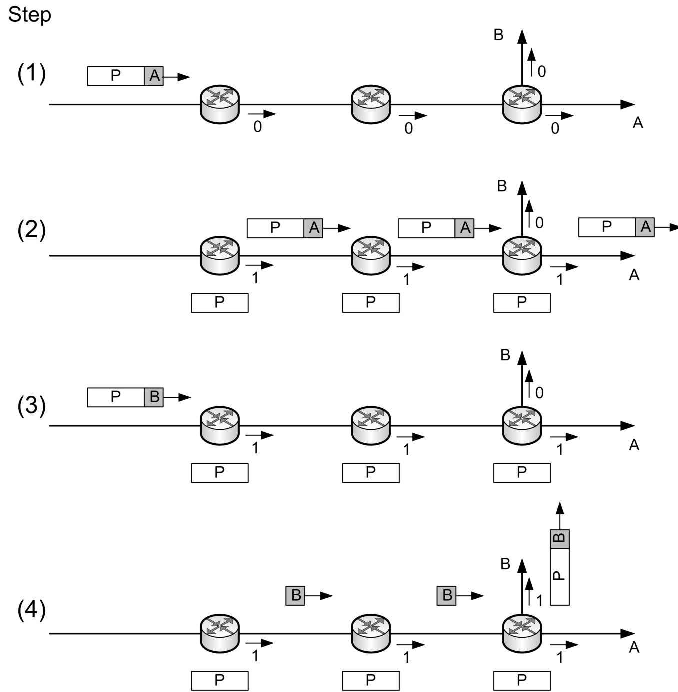


Figure 3.1: Basic idea of the caching mechanism

3.1.1 Link

A link is a point-to-point communication channel that transports packets over a distance. The link abstraction can be applied in the context of both the end host communication and the single hop communication. However, in this thesis we use it always in the context of the single hop communication. A link should not be associated with the physical medium. It is rather implemented on top of other networks (see Figure 3.2). At present, Internet links are mainly based on switched networks such as switched Ethernet or ATM which are used to access end hosts, or dedicated lines which constitute the Internet backbone. Despite the variety of link technologies, a link has three general properties that are of key importance in the CacheCast design.

(1) Packet sequence: Even though this is not required, in most cases links preserve the order of packets. This is due to the link technology. Backbone links are mainly created using dedicated lines, or circuit switched networks, where packet reordering does not occur. In case of access networks which are dominated by switched Ethernet, the order of packets is preserved due to single path routing inside a network.

(2) Packet loss: Links are characterised by low packet loss probability. Considering wired technologies the reliability comes from well protected data paths, while in the case of wireless transmissions reliability is increased by protocol mechanisms, e.g. re-transmissions. The packet loss is minimised, since this increases the end-to-end transmission throughput and link utilisation.

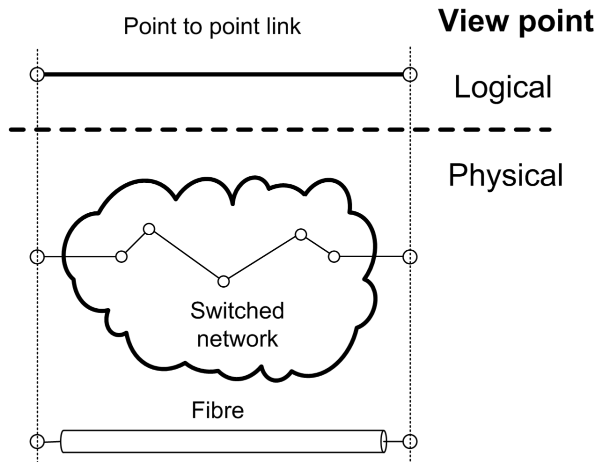


Figure 3.2: The link concept

(3) Link capacity: The link throughput is limited in bits per second. Thus, the shorter packets are the more of them can be transported over a link within a given time unit.

The first two properties greatly simplify the design of a link cache, since it does not need to employ mechanisms for packet recovery or for handling reordered packets. The third link property ensures that payload caching is meaningful, since a link can transport many more short packets than large packets within a time unit. If the link throughput was limited in packets per second, no gain could be obtained by caching.

While it is simple to associate a link with a fibre connection, the correct identification of a link in a broadcast medium may not be obvious. Figure 3.3 depicts three machines connected with Ethernet coaxial cable which acts as a broadcast medium. In this scenario, we identify the three following links: A-B, A-C, and B-C. Since the cable connects more than two machines, we use machine Ethernet addresses to identify link ends. It should be noted that our link abstraction does not capture broadcast transmission. It only supports the point-to-point communication. By this, CacheCast adheres to the Internet architecture which operates only on point-to-point links[1].

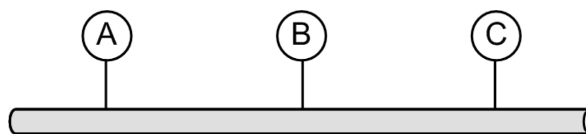


Figure 3.3: Ethernet network segment

[1]The broadcast communication in the Internet infrastructure is rarely available and only on the network edges. It causes waste of bandwidth, since messages are often sent to hosts which are not interested in their content.

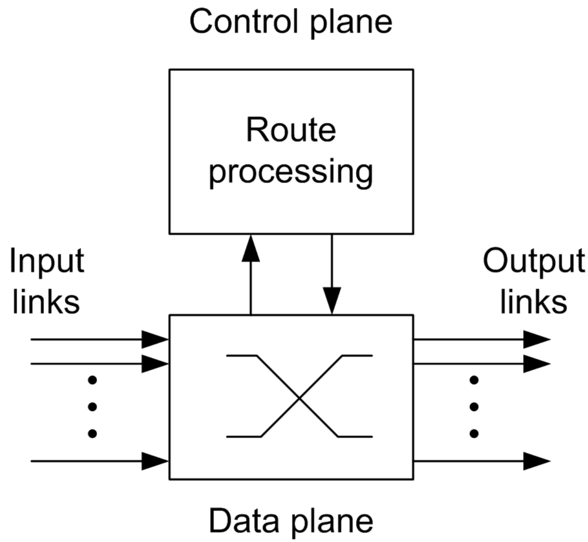


Figure 3.4: Conceptual view of a router

3.1.2 Router

A router is a network node that connects at least two links and switches packets between them. As depicted in Figure 3.4, a router has two processing planes: a control plane and a data plane. The control plane is responsible for computation and maintenance of network routes as defined by routing protocols like Open Shortest Path First (OSPF) or Border Gateway Protocol (BGP). Since these tasks are not time-critical, they are performed by a standard CPU. The router data plane is responsible for forwarding incoming packets based on their destination IP addresses and the precomputed routes. Packet forwarding is the most time-critical operation in a router. In order to match the arrival rate of packets from multiple links of high capacity, the forwarding operation requires hardware support and often a special type of hardware architecture.

CacheCast operates on a per packet basis, hence it functionally belongs to the data plane. The main task of the data plane is to forward packets and this is performed in the four stages (as depicted in Figure 3.5): (1) upon arrival, a packet is verified and if it has errors, it is discarded, (2) based on the packet destination IP address the output link is determined, next (3) the packet time-to-live (TTL) and checksum fields are updated, and finally (4) the packet is switched to the output link. Each output link has a queue that stores packets during bursty periods when packets from multiple input links are switched at the same time to the same output link exceeding its capacity.

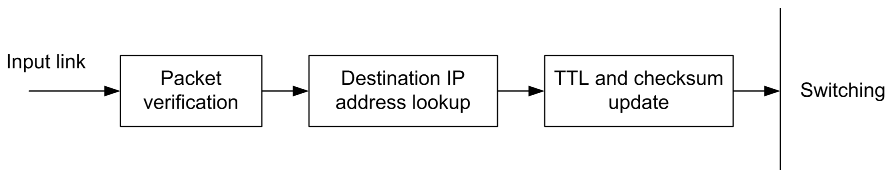


Figure 3.5: Input packet processing per link

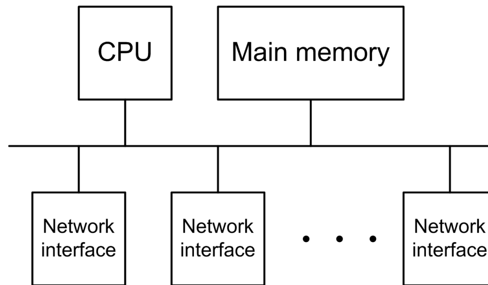


Figure 3.6: The first generation of router architecture

The first generation of routers perform all tasks of the data plane in a standard computer architecture (Figure 3.6). A packet arriving at one of the network interfaces is transferred to the main memory and then processed by the general purpose central processing unit (CPU). When the output link is determined the packet is queued and later transferred to the network interface associated with the output link. In this architecture the CPU performs all tasks related to the packet forwarding and the main memory acts as the packet switch, thus, the router forwarding rate is limited by per packet CPU processing and the memory bus speed. The second and third generation of routers are based on a distributed processing architecture. The data plane tasks related to the packet processing are performed directly on network interfaces and the switching is done via a shared memory, or a shared bus in routers of the second generation, and via a switch fabric in routers of the third generation. While the first generation routers are sufficient for small to medium size autonomous systems, the second and the third generation routers are necessary to handle traffic on Internet backbone links.

The router throughput is characterised by the number of packets that a router can forward within one second. Our fundamental assumption in this thesis is that throughput of a router does not depend on the packet size. This assumption holds for the first generation of routers, since the packet route evaluation and IP header update does not depend on packet size and the packet switching is performed inside the router main memory. However, in the second and third generations of routers this assumption may not hold. While these routers perform the same per packet operations, the packet switching involves packet transmission over the internal bus or switch fabric. Therefore, throughput of these routers depends on the packet size.

In this thesis we focus only on the first generation of routers that are used in the small to medium size autonomous systems. The second and third generations of routers require additional considerations which is a subject for future work. It is worth noting that for small to medium size networks the link cost is mainly associated with a physical infrastructure of copper wires or fibres and it greatly outweighs the cost of routers which are cost-efficient boxes. Therefore, the bandwidth savings that can be gained by caching in this type of network are important.

3.2 Requirements

CacheCast requirements arise from three aspects: (1) CacheCast as an intermediate system that supports single source multiple destinations transfer, must in principle preserve the end-to-end

relationship between communicating hosts. (2) CacheCast as a system performing per packet processing must be reliable and fast. (3) CacheCast as a system for reducing transmission costs must itself have minimum costs in terms of resource consumption and initial investment.

CacheCast as an intermediate system

The host communication in the Internet is based on the end-to-end principle, i.e. two hosts which communicate have all information about the connection stored locally and no information is stored in the network. Transport protocols like TCP or DCCP (which provide reliability or congestion control) are based on this principle. Additionally, the end-to-end relationship between hosts simplifies security control, since no third party is involved. Therefore, CacheCast as an intermediate system must preserve the end-to-end relationship.

IP Multicast is an example of a system which breaks the end-to-end principle. It requires a new set of protocols designed to work with an intermediate system. However, in this context it is difficult to provide reliability, congestion control, and security control which is a subject of ongoing research (cf. Section 2.1.2).

CacheCast as a packet processing system

CacheCast is a link layer system that performs per packet processing. Since the caching mechanism operates on individual packets, a single cache must be fast enough to process all packets arriving at the maximum speed. If this is not the case, the caching mechanism itself becomes a bottleneck.

CacheCast must be reliable. A single packet traversing a network hop-by-hop can be subjected multiple times to the caching mechanism. Since the packet is successfully delivered only if all of the caches on the packet path perform correctly, a single cache operation must be completely reliable. A failure of a single cache will result in failure of end-to-end packet transmission.

CacheCast as a system for cost reduction

The CacheCast purpose is to reduce costs of single source multiple destination transfers measured as bandwidth consumption. However, if CacheCast requires a considerable amount of resources to be deployed and utilised then the purpose is lost. Therefore, both investment and operational costs must be minimised.

In order to minimise costs of initial investments CacheCast must be incrementally deployable. An incrementally deployable system yields benefits already with the deployment of the first elements, thus creating a positive incentive for further investments. Systems requiring full deployment in order to work, create a high barrier of initial investment which may prove too difficult to overcome.

The operational costs of caches depend mainly on the amount of cache storage space and computational costs. Thus, in order to minimise the operational costs, CacheCast must use minimum storage space on each hop and employ a simple caching algorithm.

3.3 Fundamentals

The core problem that we address in this thesis is the redundancy in single source multiple destination datagram delivery in the Internet. In the absence of widely deployed multicast solutions this type of transmission is solved by a superposition of unicast transmissions. Obviously, this leads to multiple transmissions of identical payload over the same link.

The overhead introduced by multiple transmissions of the same element over the same communication channel is a well-known problem in computer systems and is addressed by caching. For example, multiple transfers of the same data/instruction from main memory to CPU are suppressed by multi-level CPU caches. Multiple transfers of the same pages from hard disk to main memory are suppressed by page caches. Multiple transfers of the same elements from web pages to the same client are suppressed by the client web browser caches. We argue that multiple transfers of the same payload over the same link should be also suppressed by caching. The link layer caching is transparent to packet transmission and therefore preserves the end-to-end relationship between communicating hosts. Thus, CacheCast as a link layer caching technique intrinsically satisfies the requirement imposed on intermediate systems as discussed in Section 3.2.

In the following subsections, we discuss two fundamental design issues for payload caching. The first issue is where to place cache elements. The second issue is whether a source should be aware or unaware of caching.

3.3.1 Placement of cache elements

According to the assumptions we have made, the link capacity is limited by the bit transmission rate and the router forwarding rate does not depend on the packet size. Therefore, the caching mechanism should be located on the edges of the link entity (see Figure 3.7). In this configuration, the caching mechanism is divided into two parts. The cache management unit located at the link entry removes redundant data from packets before they enter a link, thus increasing the link throughput. The cache store unit located at the link exit reconstructs the packets before they enter a router which does not impact router forwarding rate. Caches constructed on the link edge entity are simple to implement, since packets behave in a deterministic way on a link. Additionally, the caching mechanism is transparent to a router, since a router processes packets in a standard way (Figure 3.8).

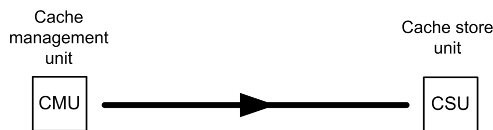


Figure 3.7: Extended directed link

A single link cache does not require any cooperation with other caches. It is a self-standing unit. The only information the cache requires is carried by a cacheable packet. Link caches are transparent to routers. As we have discussed in the previous section we focus on the first generation of routers. All cacheable packets that were transmitted without payload over a link are reconstructed inside the shared memory and then they are processed by a router. Before a cacheable packet is moved to the

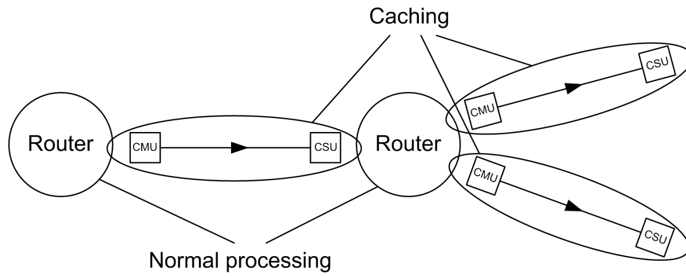


Figure 3.8: Caching links and routers

output interface, it is again subjected to the caching mechanism and its size may be reduced (see Figure 3.8). Since we move cached packets between network interfaces and shared memory and we aim at a very simple caching mechanism, the CacheCast impact on a router's switching capacity is negligible.

3.3.2 Caching aware source

Our second design decision is that a source should be aware of caching. A caching aware source cooperates with the network. It can ensure that the packets that carry the same payload are transmitted within the minimum interval time. Additionally, it can provide three key information elements to the network that simplifies caching. Firstly, it marks packets that carry the same payload as cacheable packets. Packets which do not carry duplicate payloads are not marked and are not considered for caching in the network. This in turn increases cache efficiency, since unique payloads do not unnecessarily consume cache space. Secondly, the payload part of a packet is of variable size. The source can easily provide information about the payload size. Thus, link caches know immediately which part of a packet to cache. Thirdly, a source can create an identifier for each payload, which combined with the source address, is unique in the Internet. Thus, caches perform the matching based on payload IDs instead of a whole payload match. A caching aware source can significantly relieve link caches both in terms of processing and memory requirements. The simplified cache processing results in reliable and fast link caches. Therefore, a caching aware source is the fundamental element of the CacheCast design by which we address the requirements of the fast and reliable packet processing and minimum resource consumption described in Section 3.2.

A source unaware of caching does not support the network. Caching is done transparently to the source and the link caches do not have the aforementioned advantages. These caches would require larger storage space to achieve a comparable efficiency (due to the wasted storage room for non-redundant payloads). They need to determine the payload part and need to compare the whole payload at the instance a cache hit occurs. However, the main advantage of the transparent caching is that it does not require any change at the source. Moreover, the transparent caching does not match payloads according to the IDs created at the source, rather it compares whole payloads. Thus, the same payloads that originate from different sources can be matched, which may provide the ability to cache content from unrelated sources. Nevertheless, we are interested in the single source multiple destination datagram delivery and inter-source matches are beyond the

scope of this work. Furthermore, our aim is to minimize the cache size which in turn minimizes the probability for inter-source matches. The advantages of the transparency are achieved by employing large caches and complex cache matching algorithms which will otherwise be too expensive for the Internet in the near future.

3.4 Link cache

CacheCast has been developed on the basis of our fundamental design decisions. It consists of two parts: a server support and a link cache. In this section we describe the functionality of the link cache and discuss issues related to the size, configuration, resource utilisation, and reliability of a link cache.

3.4.1 CacheCast header

The caching aware source (as we discussed in Section 3.3) marks packets as cacheable packets and provides information on payload ID and payload size of the packet. However, the question is: Where should this information be placed? We decided to create an extension header called the CacheCast header. Each packet which carries a CacheCast header is a cacheable packet. CacheCast works only on cacheable packets. Non-cacheable packets pass it untouched. The header is created at the source and contains three fields: the cache index (*INDEX*), the payload ID (*P_ID*), and the payload size (*P_SIZE*). The cache index is an administrative field for the caching mechanism and it points to the location of the payload in the cache store unit. The two remaining fields are filled by the source before packet transmission. The payload size describes the packet tail size which is cacheable. We assume that payload is always at the tail of a packet. The payload ID identifies packet payload uniquely at the source and when it is combined with the source address it identifies a packet uniquely in the Internet.

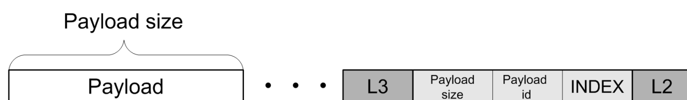


Figure 3.9: IPv4 Packet with the CacheCast header

Depending on the IP version available in a network, we consider two possible locations for the CacheCast header. If CacheCast works in an IPv6 network we use the IPv6 extension header to implement the CacheCast header. However, IPv6 is rarely deployed in the Internet at present and it is more likely that the caching mechanism is employed in IPv4 based networks. Since the IPv4 header does not provide any functionality enabling us to implement the CacheCast header, we follow the approach of the MPLS protocol. We locate the CacheCast header between link layer header (L2) and IP header (L3) (shown in Figure 3.9). We make the CSU unit responsible for removing the CacheCast header when a cacheable packet enters a router and the CMU unit responsible for inserting the CacheCast header before a packet re-enters a caching link. Since we have to maintain the payload ID and the payload size information when a cacheable packet is processed by a router, we move them to the packet meta-data describing the packet state on a router. We follow this approach in our Click router implementation (cf. Chapter 7).

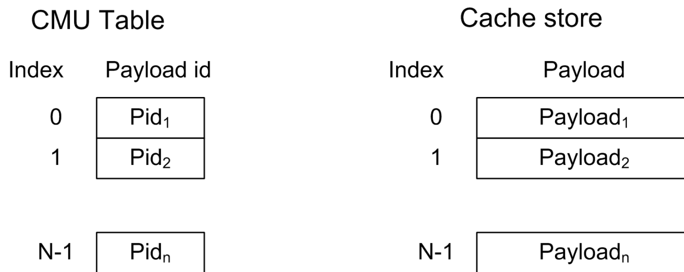


Figure 3.10: Relationship between the CMU table and the CSU

3.4.2 Link cache mechanism

Caching is done per link and is separated into a management unit and a cache store unit. The cache management unit (CMU) is placed on the link entry and the cache store unit (CSU) is placed on the link exit (Figure 3.7). The CMU is in full control of the CSU. It has a table where it keeps information on payloads stored in the CSU. The number of the CMU table entries is the same as the number of the CSU slots (Figure 3.10). We define the link cache behaviour for *cache hit* and *cache miss* in the following way:

Cache hit: If a payload identifier of a packet entering a link is found in the CMU table, we call it *cache hit*. This means that the packet payload is in the cache store unit on the exit side of the link and there is no need to transmit it again. Thus, CMU first puts the index of this payload identifier into the *INDEX* field in the CacheCast header. Next, it removes the payload and transmits only the header part of the packet. When the header arrives at the link exit the payload with the *INDEX* in the cache store unit is attached to it. Then, the whole packet is moved for further processing to the router.

Cache miss: If a payload identifier of a packet entering a link is not found in the CMU table, we call it *cache miss*. This means that the packet payload is not present in the cache store unit on the exit side. The CMU handles the cache miss in the following way. Firstly, it removes one entry from the table according to the selected cache *replacement policy*, inserting instead the payload identifier of the packet that caused the cache miss. Secondly, it inserts the index where the payload identifier was inserted in the CMU table into the *INDEX* field of the packet CacheCast header. Finally, the packet is transmitted over the link. Upon arrival at the link exit the payload is stored at the location pointed to by the *INDEX*. The previously stored payload can be safely overwritten since it has been evicted from the cache by the CMU. The payload identifier in the table and the corresponding payload in the cache store unit have the same indexes.

3.4.3 Link cache size

It is difficult to provide an approximate number of cache sizes in the Internet. In general, larger caches are more efficient, but they also require more processing capacity and more storage space. We decided to scale all caches according to the associated link capacity C. The scaling factor T is

Table 3.1: Number of headers in packet train transmitted within a given cache hold time

Source uplink speed	Cache hold time $2ms$	$10ms$	$50ms$
$512Kbps$	2	8	40
$1Mbps$	4	16	79
$10Mbps$	32	157	781
$100Mbps$	313	1561	7802

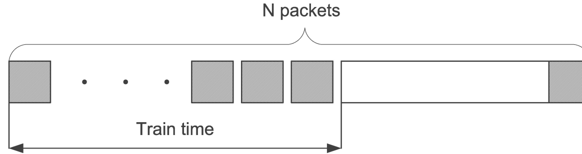


Figure 3.11: The packet train duration time

defined as the cache hold time and it is the minimum time a cache should be able to hold random payload. Therefore, the cache size S is given by $S = CT$.

We use the following observation to estimate the scaling factor T of the desired cache size. Let us consider a source that sends the same data chunk to multiple destinations in the CacheCast enabled network. Before packets carrying the data chunk enter the first hop link they are subjected to the CMU. Thus, only the first packet carries payload over the link while the remaining packets are without payload. We call the resulting packet chain – a packet train. An example of a packet train is shown in Figure 3.11.

Considering the packet train structure, we notice that it is sufficient to create caches that could hold a payload carried in the first packet until the last header of a packet train arrives. Therefore, the desired cache hold time depends on the packet train duration time. This, in turn, depends on the source uplink speed and the number of destinations.

We have calculated the example sizes of packet trains that can be inserted into a network within a given time window. Table 3.1 presents the calculation results performed for four different source uplink speeds and for three different time windows. The numbers in the table represent the amount of packet headers that can be transmitted by a given source within a given time window. Thus, the numbers correspond to the packet train size. The packet header size on a link varies depending on the link technology. In our calculations we assume the minimum Ethernet framing for a packet header which is 84 bytes including inter-frame gap for all uplink speeds.

In the calculations, we measure the packet train duration time as the difference between the time when the first packet and the last packet were transmitted by the source. Since only the head of a packet train carries the payload, the duration time of the packet train is equal to $(N-1)t_s$ where N is the number of receivers and t_s is the time to serialize one packet header.

Considering the numbers given in Table 3.1, i.e. the number of packet headers that can be inserted into a network for the different uplink speeds, we argue that the cache hold time T of 10ms is satisfactory. We support this claim with the following four key observations:

1. Clearly, the faster the uplink speed is the more packets a source can insert into a network

within a given time. We expect that sources with slow uplink speed send data to a few destinations while sources with fast uplink speed send data to many destinations.

2. Sources with a slow uplink speed often employ header compression techniques to shrink the header size to only a few bytes. Thus, slow sources may insert many more packet headers into a network within a given time.

3. The numbers provided in Table 3.1 do not determine the maximum number of receivers. For example, a source with $1Mbps$ uplink speed can still send data to more than 16 receivers, but in such a case the packet train duration time is larger than $10ms$. Therefore, it may cause additional payload transmissions decreasing the overall efficiency (cf. Chapter 4).

4. Sources with a fast uplink speed generate much more cacheable traffic in a network than sources with a slow uplink speed. Therefore, the major part of bandwidth savings is due to caching of traffic originating from the fast sources. The decrease in efficiency due to the slow sources that send to many destinations is low.

The small cache size is important both for the CSU and the CMU. The 10ms storage space required for the CSU is modest when compared to the storage space of a link queue. A link queue is designed to accommodate approximately 250ms of the total traffic flowing through an associated link [38] and, similar to a link cache, it requires fast access to memory. However, the link queue memory is large and cannot be implemented using a fast static memory. Therefore, hybrid approaches combining static and dynamic memory are investigated [39]. Nevertheless, this is not a case of a link cache storage which can be implemented using a static memory, since it is small.

The CMU has the table containing identifiers of payloads located in the cache store unit. It performs payload identifier lookup for each cacheable packet. Thus, when the table is large the lookup operation may become the bottleneck of the caching mechanism. Since it is sufficient to keep cacheable packets only for several milliseconds in a cache, it results in a small size of the CMU table. Let us consider a 1Gbps link as an example. If we assume that the average payload size is 1KB, and we cache link traffic for 10ms, we get the total amount of 1310 CMU entries. Additionally, if we take into account that only a part of the total link traffic is the cacheable traffic we can further reduce the number of entries. This size of a lookup table can be efficiently built based on hashing techniques [40].

3.4.4 Memory utilisation of a link cache

According to the description given in the link cache mechanism, the CSU memory is divided into slots and the size of each slot corresponds to the size of the link MTU. However, packets carry payloads of different sizes, therefore, the CSU slots are not fully utilised.

In order to increase the utilisation of the CSU memory the slot size should be smaller than the maximum payload size and payload should be stored in a variable number of slots. However, we do not consider this issue as a design specific issue, but rather as an implementation specific issue. We address the memory fragmentation in Chapter 7.

3.4.5 Configuration of a link cache

Before a link cache starts to operate, the CMU and the CSU must agree on the number of payload slots and the maximum slot size. This can be solved either by manual or by automatic configuration. While manual configuration of a link cache is straightforward, it creates a new source of possible errors and it must be avoided.

Automatic configuration based on exchange of messages between the CMU and the CSU during link cache initialisation cannot be realised, since some links are unidirectional (e.g. satellite links) and cannot support bi-directional communication. Therefore, either the CMU must configure the CSU or both elements are configured according to certain external parameters. We decided to configure the link cache elements according to the associated link capacity. The total cache store space is equivalent to the product of the link capacity and the time interval of 10ms. The cache store space is divided into slots of the same size in order to determine the number of slots and the CMU and the CSU are configured with this value.

3.4.6 Payload ID considerations

A source creates a payload identifier for each CacheCast packet it transmits. The payload ID combined with the source IP address uniquely identify this payload in the Internet. While in theory each payload can be annotated with a different payload ID, in practice a source can use only a limited amount of payload IDs. The maximum number of payload IDs is related to the size of the payload ID field in the CacheCast header. Therefore, when a source has used all unique payload IDs, it has to re-use them in order to identify new payloads. This creates a problem, since the same payload ID identifies two different payloads; thus, the question is: When can a source re-use a payload ID?

To better understand this question, let us consider the following scenario. A server S transmits data to a number of clients. The server annotates a data chunk A with the ID_A and transmits it. The data chunk traversing a network is stored in all link caches along the paths to the clients. Thus, the ID_A cannot be re-used, until it is not evicted from all CMUs which are related to these link caches. As we have established in Section 3.4.3, link caches should be able to hold payloads, thus also payload IDs, for at least 10ms. However, this does not imply that a payload ID is evicted after this time. In fact, it is rarely the case. Considering a random link cache which holds the data chunk A, the data chunk remains in this link cache until it is replaced by other data. Hence, this time period depends on the volume of the CacheCast traffic flowing through that link cache. If it is small, the data chunk A may remain in the cache for a few seconds or even minutes and the ID_A can not be re-used during this time.

In general, it cannot be guaranteed that when a source re-uses a payload ID, this ID is already evicted from all link caches. To solve this issue, we define a time period when a payload ID can remain in a CMU. When this time period expires the payload ID is evicted from the CMU. However, how long should the time period be? If the time to evict a payload ID from a CMU is long, a source cannot re-use the same payload ID for a long time and the amount of available IDs decreases. However, a source can use the same payload ID during packet re-transmission and benefit from CacheCast. On the other hand, if the time to evict a payload ID from the CMU is short, a source can re-use the same payload ID very fast and the amount of available IDs increases. However, link caches evict payloads very fast before packet re-transmission occurs.

The time period to evict payload ID can vary according to conditional parameters. However, we decided to set the time to evict a payload ID from a CMU to one second based on three arguments: (1) A longer time period would burden a source. (2) There is no reason to keep a payload ID longer than one second, since most re-transmissions occur within this time. (3) Assuming only 1% of the CacheCast traffic flowing though a link cache, a payload ID is evicted from this link cache after one second. Hence, we expect that most payload IDs will be evicted before the time period of one second expires.

3.4.7 Errors on a link

So far we have assumed that there is no loss and no packet re-ordering on a link. However, even though these events occur very rarely on current links (discussed in Section 3.1.1) the caching mechanism requires additional protection against them.

It should be noted that the link cache is built at the edge of the link entity. Thus, packet loss or packet re-ordering in a router does not affect the link caching. The only packet loss and packet re-ordering that is of concern to us is packet loss and packet re-ordering on that particular single link.

Packet loss

Packet loss on a link may affect the caching mechanism and cause cache inconsistencies, but only when the lost packet carried a payload. The following example illustrates this case. A cacheable packet with a new payload enters a link and is processed by the CMU. The packet causes a *cache miss*. The CMU inserts the packet payload ID into the table and puts the payload ID index into the packet header. Next, the packet is sent on the link but is lost. Thus, it does not reach the cache store unit and the payload is not stored on the link exit. Now, the payload ID in the CMU table does not reflect the correct payload in the cache store unit. Hence, the link cache is inconsistent. Each packet that enters the link and has the same payload ID as the lost packet causes a *cache hit*. Its payload is removed and only the header part is sent over the link. When the header arrives at the link exit a wrong payload is attached to it and the wrong payload is delivered to the packet destination.

To protect against errors caused by cache inconsistency we store the payload ID together with the payload in the cache store unit on the exit side of the link. Since the packet header always carries its payload ID, it can be compared with the payload ID in the cache store unit before the payload is attached. If the payload ID carried by the header does not match the payload ID in the cache store unit, the packet must be dropped. We do not consider any re-transmission schema of lost packets on a per link basis to keep the caching mechanism simple and fast. Therefore, the loss on a link of a cacheable packet with payload causes the dropping of the whole packet train. However, the loss of a packet without payload does not affect any subsequent packets.

Packet re-ordering

Packets that exit a link out of order may cause a temporal cache inconsistency. A simple illustration of this problem is the following. Let us consider two packets. The first packet entering the link causes a *cache hit*. Thus, its payload is removed and the CMU puts the corresponding index value

into the packet header. The second packet entering the link causes a *cache miss* and the CMU chooses an index according to a replacement policy, which appears to be the same as the index carried by the first packet. If the two packets are re-ordered on the link, the payload of the second packet will overwrite the payload being referred to by the first packet. Thus, the first packet receives a wrong payload, which is a severe error.

The solution that protects against errors due to packet loss also protects against errors caused by packet re-ordering. Applying it to the above example would cause the drop of the first packet (which arrives second at the link exit). Depending on the probability of the packet re-ordering on a link more sophisticated solutions may be applied to that link. However, in our basic design the packet dropping policy is considered sufficient.

3.5 Server support

According to our fundamental decisions described in Section 3.3.2, CacheCast requires server support. The support consists of two elements: (1) a server batches requests for the same data to transmit it within the minimum interval time, and (2) the creation of the CacheCast header for each cacheable packet. The first task of the server support is performed both by an application which transmits data to multiple destinations, since it must schedule transfer of the same data at the same time, and by an operating system (OS), since it must handle these transmission requests properly. The second task of the server support is related to low level packet modification and is fully performed by an OS. To better understand requirements for the OS support, we first describe steps that an application takes to transmit the same data to multiple destinations.

3.5.1 Application related tasks

In order to benefit from CacheCast, an application that transmits data to multiple destinations must batch requests for the same data. For example, a simple application could delay early requests to match the late coming requests and then send the data to all destinations. However, more advanced schemes like carousel content or fountain codes [41] can be used to synchronise in time transmissions to all destinations. These approaches are useful for static content. The live streaming type of content, served for example by IP TV or IP radio services, does not require this type of support, since the transmission is synchronised in time per se. In Chapter 6, we demonstrate live audio streaming in a CacheCast enabled network and we evaluate the benefits of this approach.

When a data chunk is scheduled for transfer, an application uses a system call to send the data chunk to a destination. The application that transmits the same data to multiple destinations must invoke the system call multiple times to send the data chunk to each destination. This, however, may result in loosely synchronised packet series at the network level. Moreover, the packets may be interleaved by other packets unrelated to the application data, since other applications can transmit at the same time. Therefore, the application requires proper support from an operating system in the form of a more suitable system call.

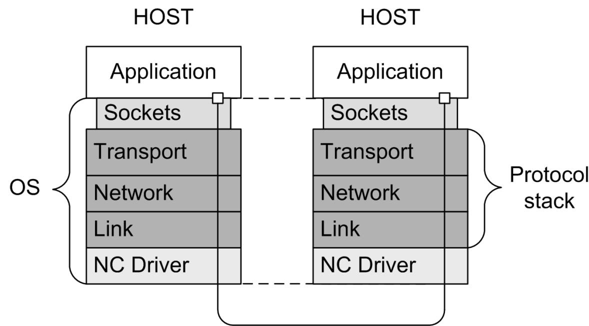


Figure 3.12: Network communication in OS

3.5.2 Operating system related tasks

While the application task is to schedule data transmission to all destinations at the same time, the OS tasks are to handle this transmission request properly and to create the CacheCast header for each packet. Before we describe in detail the OS support for CacheCast, we first give a brief overview of OS support for network communication in general. Within this framework we place later elements of the CacheCast support.

The central part of OS support for network communication is a protocol stack. It enables an application to communicate with remote processes in the Internet. As depicted in Figure 3.12 the protocol stack consists of three layers of protocols: transport protocols (e.g. TCP, UDP), network protocols (IP for the Internet), and link protocols (e.g. Ethernet for local area networks, Point-to-Point Protocol (PPP) for dial-up connections). An application accesses the protocol stack via sockets. In order to transmit data to a remote process an application sends the data to a socket, which passes it to the protocol stack. The resulting packet, which carries the application data, is transfered to a network card with the help of a network card driver (NC driver).

In the following paragraphs, we cover in detail the three elements of the OS support for network communication.

Socket

A socket is a standard way that most of operating systems use to handle an end-to-end connection. It is a communication end-point operated by the OS. An application accesses a socket by a socket interface which enables it to choose a communication protocol to establish and tear down a connection, and also to send and receive messages.

A socket is controlled by a set of dedicated system calls. In order to establish communication with a remote process, an application must first allocate a socket which is performed by the `socket()` system call. In return, the application receives a handler to a socket object which is used for subsequent system calls. The system call has three parameters that specify connection domain, type, and protocol. In the Internet domain the socket type specifies the service level provided by a socket, like e.g.: reliable stream oriented socket, unreliable datagram socket, raw socket, etc. The service is realised by a protocol. In most cases a socket of a given type is supported only by a single protocol. For example, unreliable datagram socket is only provided by UDP.

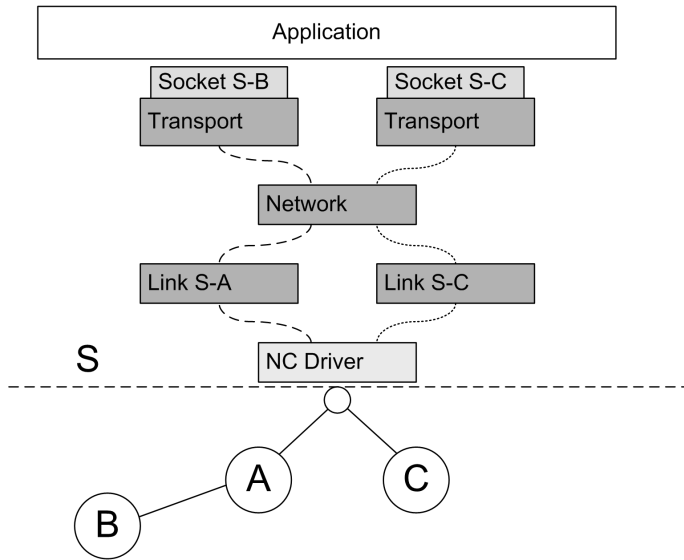


Figure 3.13: An application on server S connected to hosts B and C using two sockets

A socket abstracts communication details, such as network type, protocol type, protocol mechanism, or encapsulation process from an application. Thus, an application can use simple `send()` and `receive()` system calls to communicate with a remote process. Once an application sends a data chunk to a socket it is passed to a protocol stack for further processing.

Protocol stack

An operating system has a protocol suite which contains protocols of three different layers: transport, network, and link. The protocols in each layer have well defined interfaces which facilitate the composition of a protocol stack in a flexible way. The protocol stack consists of three protocols from transport, network, and link layer, which are stacked together in the listed order. Data sent by an application is successively processed by each protocol from the protocol stack and a packet is created. Each protocol adds its control information in front of the application data in a process called encapsulation. The resulting packet is handed to a device driver for transmission.

The composition of protocols in the working protocol stack is uniquely defined by a socket. The transport and network protocols are defined when an application allocates a socket. The domain parameter defines the network protocol whilst the type and protocol parameters define the transport protocol. The link protocol is defined when an application tells the OS to connect a socket to a remote socket. In the process of finding a route to the destination the first hop is determined and the protocol associated with the link to the first hop is set in the protocol stack.

The protocol stack can be seen as a composition of protocols, as we have described and as it is depicted in Figure 3.12. However, from an OS point-of-view each protocol requires an instantiation which holds the state of the protocol. Thus, the link protocol requires an instantiation for each link connected to a host. The network protocol requires an instantiation per network node running on a host. However, since a host usually appears in the Internet as a single node, there is only a single

Table 3.2: Comparison of transport protocols

Protocol	Data handling	Congestion control	Reliable
TCP	Byte stream	Yes	Yes
UDP	Message	No	No
SCTP	Message stream	Yes	Yes
DCCP	Message	Yes	No

instantiation in the OS. The transport protocols require an instantiation for each connection. We illustrate this idea in Figure 3.13. It depicts a server running an application which uses two sockets connected to hosts B and C. Each socket during allocation creates its own instance of a transport protocol. The existing network and link protocol instances are linked into the protocol stack of a socket. Therefore, when the application sends data to a socket, it is consecutively processed by protocol instances in the order defined by the socket. As a last step the packet is transmitted to a network card.

Data transmission

An application can send data only when a socket is in a connected state which implies that all protocols in the protocol stack associated with this socket are defined and linked. Depending on the transmission protocol, the `send()` system call may result in packet transmission or the data transmission may be delayed according to the protocol mechanism. Thus, even though an application sends the same data to multiple destinations at the same time, transmission of the resulting packets may be spread in time.

In considering transmission protocols that work in the Internet, there are only two possible sources of the delayed transmission: either a connection is congested and a protocol prohibits data transmission, or a protocol assembles small data chunks to send them in a large packet. The first behaviour is related to a congestion control mechanism often built into transport protocols which prevents congestion collapse in a network. The congestion control mechanism estimates how much data can be sent within a given time window and if this limit is reached it prohibits further transmission. In Table 3.2 we provide information on the presence of the congestion control mechanism in the Internet transport protocols.

The second source of the delayed transmission is related to the small-packet problem described in [42]. If each application send request resulted in a packet, then applications like telnet would incur high overhead in terms of network traffic, since each character created by a key stroke would be carried by a single packet. In order to save bandwidth, the transmission protocol should assemble these small data chunks into larger chunks. An example of this type of algorithm is Nagle's algorithm which is part of TCP. Other transport protocols listed in Table 3.2 do not assemble small data chunks, since they regard each data chunk provided with the `send()` system call as a single message to be sent rather than a byte stream.

A packet produced by a network stack contains application data encapsulated by three protocols from the protocol stack and it is ready for transmission. However, before the transmission occurs, the packet is subjected to a traffic control. Modern OSs provide a mechanism for packet classifi-

cation and queueing that enables the OS to control traffic when accessing congested medium.[2] As a result of the traffic control subsystem the order of packets created by the network stack may be disturbed.

3.5.3 Operating system support for CacheCast

According to our design an application should batch requests for the same data and send it at the same time. Since each end-to-end connection is handled by a different socket, the send procedure requires a sequence of the `send()` system calls to transmit the data to all clients. However, as we have discussed this does not result in a tight packet train on the network level. The sequence of system calls may be interrupted by other applications transmitting at the same time; packet transmission to some destinations may be delayed; or packet sequence may be unordered. To prevent this undesired behaviour, the OS must control the send procedure starting from the application send requests to the packet transmission to a network card.

System call

We decided to design a new system call that executes multiple send requests from an application in a controlled manner. The system call is a single entry point to the OS kernel. It takes as arguments a pointer to a data chunk to be sent and a set of socket handlers. When an application invokes the system call the control is passed to the OS which can supervise the process of data transmission. The system call tasks are:

1. It sends the application data to each socket from the set of sockets provided by an application.
2. It captures packets created by the protocol stack, appends the CacheCast header to each packet, and queues the packets for later transmission.
3. If a socket prohibits data transmission due to congestion control or data assembling process, the system call undoes the send request for this socket.
4. When the application data has been sent to all sockets, the system call sends previously queued packets to a device driver. However, only the first packet per link carries the application data. The remaining packets are truncated.

As a result of the system call, a tight packet train per link is created as it is depicted in Figure 3.14. The system call performs the CMU related tasks for the server and it is the only support a server requires to benefit from CacheCast.

The system call returns a status of the send requests. If some sockets from the set provided by an application could not be written successfully, the application is notified. Therefore, the application can decide itself how to handle these cases. For example, in Chapter 6 we present a modified paraslash server that streams an audio file to multiple clients in the CacheCast manner. The server counts how many times it could not write an audio sample to a given socket and if the failure number exceeds a certain threshold a connection related to the socket is regarded as a slow connection and is torn down.

[2] In Linux this mechanism is implemented using *qdisc* (an abbreviation of a queueing discipline).

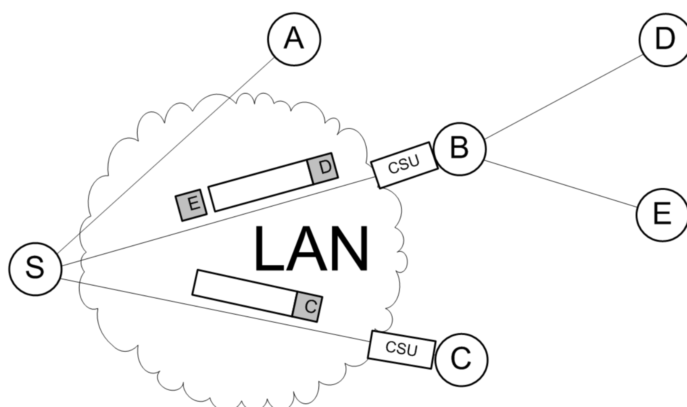


Figure 3.14: Server S transmitting the same data to machines C, D, and E

In the following paragraphs we discuss the exact place in the encapsulation process where a packet should be captured and queued, how to create a per link packet train, and how to build the CacheCast header. In Chapter 6 we present the `msend` system call – an instantiation of the system call for the Linux operating system. The `msend` system call currently supports only two transport protocols: UDP and DCCP.

Packet capturing and queuing

The second task of the system call is to capture packets created by the protocol stack, append the CacheCast header and queue them for later transmission. We decided to create a module which performs these tasks. The module is inserted in the protocol stack and is controlled by the system call. The following example explains how and where exactly the module operates.

Figure 3.15 depicts an application running on a server S which communicates with processes on hosts C, D, and E. Each connection is handled by a socket object. Since a path to the host D and a path to the host E share the same first hop link, the sockets handling connections to these hosts share the same link protocol instance. Thus, when the application sends the same data to all sockets, packets created by the sockets S-D and S-E are processed by the S-B link protocol instance.

In order to capture, process, and queue packets on per link basis, the CacheCast module must be installed either at the entry to a link protocol instance or at the exit. We decided to install the module at the entry as it is depicted in Figure 3.15. At this stage packets have transport and network headers, thus, they can be easily extended with the CacheCast header which should be stored between network and link headers in IPv4 networks. If the CacheCast module is installed at the exit of a link protocol instance, captured packets would be encapsulated in the link layer header which would require shift of the header in order to insert the CacheCast header.

Per link packet train

During the system call send operation the CacheCast modules capture all resulting packets on a per link basis and append the CacheCast headers. After the system call has sent application data to all sockets, it triggers transmission of the previously queued packets sequentially from all CacheCast

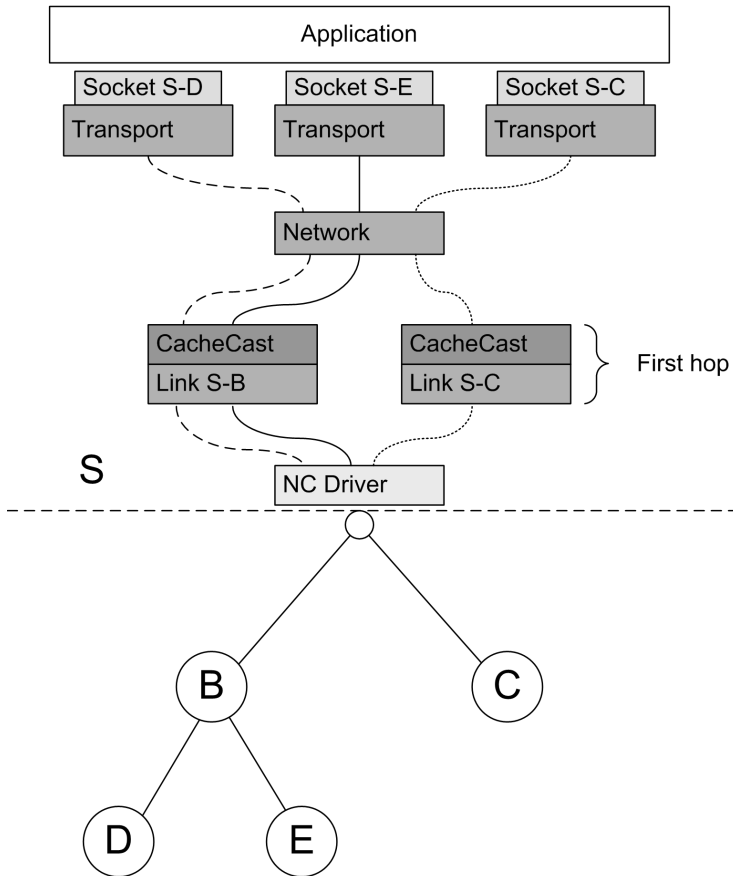


Figure 3.15: An application on a host S communicating with processes on hosts: C, D, and E

modules[3]. The trigger procedure is protected with a global lock which prohibits other applications from releasing previously queued packets at the same time. Thus, packet trains created by different applications are serialised on a link.

A single CacheCast module transmits only one packet with application data. The remaining packets which are queued in this module are truncated, thus, a single CacheCast module creates a tight packet train. Since packet transmission from the CacheCast modules is sequential, the resulting packet trains are sent to a device driver sequentially which ensures use of the minimum packet train time.

CacheCast header

The CacheCast header, as described in Section 3.4.1, carries vital information for simplifying packet processing on link caches. The information consists of the three key elements: payload size, payload ID, and index to payload in the CSU. We decided that a source is responsible for creating the CacheCast header for each cacheable packet.

[3] The number of CacheCast modules corresponds to the number of the first hop links.

The CacheCast header is created by the CacheCast module associated with the first hop link of a packet. Since the CacheCast module serialises all packet trains on a link, it is sufficient that the CSU located on the link exit has only one slot for one payload. This, in turn, implies that the index value stored in the CacheCast header can be set to zero for all packets pointing to that single slot. The payload size field is set to the size of application data carried by the packet.

The payload ID must be chosen cautiously, based on due regard to special considerations. Firstly, it must be set to the same value for all packets which carry the same application data. Secondly, the payload ID value cannot be generated sequentially. It should be obtained from a source specific random generator in view of security reasons which are discussed broader in Section 3.6. Thirdly, the payload ID value cannot be re-used until payload, which was previously identified by this ID, is evicted from all link caches. As we discussed in Section 3.4.6, the time period for which payload remains in a link cache depends only on the volume of the CacheCast traffic flowing through this link cache. To resolve this issue, we limit the time period a payload ID can remain in a CMU to one second. After this time period the payload ID is evicted. Therefore, a source can re-use a payload ID after one second.

3.6 Resilience and operational considerations

In principle, a new system running in the Internet should not create new security threads for the existing infrastructure and should itself be secure. CacheCast consists of two parts: a server support and a link cache, thus, it requires consideration of both host security and network security. The host security depends only on the implementation of the system call in the OS. Therefore, special care must be taken during the implementation process to check thoroughly the system call arguments and to prevent any type of memory leaks related to the CacheCast module operation. When implemented with precautions, the system call should not pose any new security threats.

The CacheCast security and the CacheCast impact on the network security are primarily related to the distributed architecture of the system. Operation of a single link cache does not depend on the operation of other link caches. A link cache is a self-standing unit which removes redundancy from packets and the only information it requires to process a packet is contained in the CacheCast header. Therefore, a failure of one link cache does not result is consecutive failures of other link caches and it is impossible to comprise the overall caching infrastructure. The only type of attack on the link cache infrastructure can be performed by injecting packets with forged CacheCast headers. Considering this type of thread, we have identified the following two scenarios:

- An attacker injects forged packets into a network to disturb link cache consistency.
- An attacker injects forged packets into a network to reduce link cache efficiency.

While discussing these scenarios we describe a possible behaviour of an attacker and the extent to which these actions impact the network. We also provide counter-measures.

3.6.1 Attack on the link cache consistency

Let us consider the following scenario. In the topology depicted in Figure 3.16 the server and the attacker access the Internet via the same link. The shared link is the second link on the path. All

links in the network have link caches. The server streams live media to many destinations located in the Internet; thus, it continuously transmits a new packet train carrying a new media chunk to all receivers. The attacker attempts to corrupt the media transmission by malicious use of the link cache mechanism.

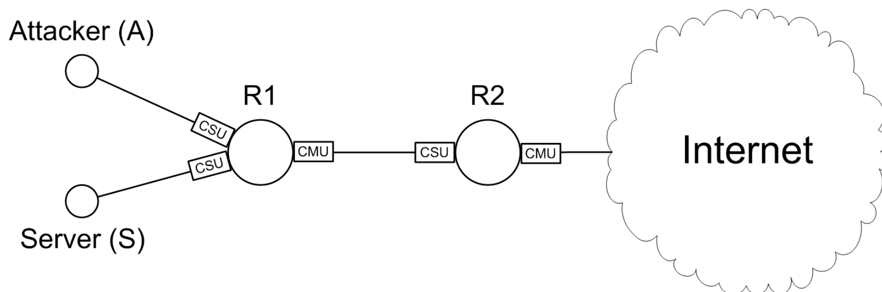


Figure 3.16: Attack scenario - network topology

In order to perform the attack, it is sufficient that the attacker manipulates the CMU located at the entry to the R1-R2 link. Let us assume that the attacker knows the ID of the payload carried in the next packet to be sent by the server. The attacker forges a CacheCast packet with the following properties: the payload ID field in the CacheCast header is set to the next payload ID which will be generated by the source (we call it ID_A), the source IP address is set to the server IP address, and the payload contains garbage. The attacker sends this packet to the network. As a result the CMU (link R1-R2) creates a mapping between the ID_A and the garbage. Therefore, when the server sends the next packet train carrying a media chunk identified by the ID_A, the CMU falsely recognises that this chunk is already on the next hop and truncates all packets in the packet train. Next, the CSU (link R1-R2) attaches the payload containing garbage to all packets. In consequence all live media receivers receive garbage.

To perform this type of attack, an attacker must know the payload ID which a victim uses to identify payload in the next CacheCast packet. A source can reduce the attackers' chances of finding this ID by generating payload IDs in a non-trivial way. In principle, a source must avoid assigning new IDs as subsequent numbers. The most efficient way to generate a new ID for payload is to use a payload checksum combined with a timestamp. The payload checksum is computed by most protocols; hence, a source can re-use it to build the ID. In the cases where an attacker is able to infer the checksum[4], the timestamp part of the ID will inhibit the attack.

3.6.2 Attack on the link cache efficiency

In this type of attack an attacker attempts to decrease link cache efficiency to the point where the cache mechanism is practically ineffective. We discuss this attack in the same context as the attack on the link cache consistency and based on the same topology as depicted in Figure 3.16. The server transmits many live streams to multiple destinations located in the Internet. The link caches located at the links S-R1 and R1-R2 remove redundant payload transmissions; thus, the server

[4]An attacker, who has access to the content streamed by a source, can compute the next chunk checksum and find the checksum ID.

Table 3.3: Number of CMU entries as a function of link capacity assuming 1500B slots

Link capacity	Number of CMU entries
$1Mbps$	1
$10Mbps$	8
$100Mbps$	85
$1Gbps$	873

can handle many more clients than without caching. In order to affect the server operation, the attacker attempts to make the link cache R1-R2 ineffective. When the link cache does not remove redundancy, the R1-R2 link becomes a bottleneck and the server must reduce the number of clients. The attacker injects forged packets into the network to force the CMU located at the entry to the link R1-R2 to evict the server content constantly. The question is whether the attacker can affect the link cache mechanism to the point where the server will not benefit from the link cache.

To evaluate this attack, we make the observation that it is sufficient to consider only a single packet train transmitted by the server. We do not have to consider multiple packet trains composing different live streams, since all packet trains generated by the source are serialised. Therefore, when a packet train has passed a link cache its payload can be evicted at no consequences to link cache efficiency. To perform a successful attack, the attacker must constantly evict the server's content from the R1-R2 link cache while the packet train passes through this link. This requires the attacker to fill all the CMU table entries before the second packet from the packet train arrives. In Table 3.3 we provide the number of CMU entries for different link capacities assuming 1500B slots in the associated CSU. Let us consider a case where the attacker and the server have the same uplink speed. Furthermore, the link R1-R2 has 1Mbps capacity; thus, there is only one entry in the CMU. In this case the attacker can make the link cache completely inefficient for the server. For each packet from a packet train transmitted by the server, the attacker forges a minimum size CacheCast packet that evicts the packet train payload from the link cache and forces payload re-transmission. When considering larger link capacities the attack is less effective. For example, considering 10Mbps link re-transmission occurs only every eighth packet from a packet train. The server and the attacker transmit packets at the same pace, since both are of the minimum size. Thus, the server content is evicted after the attacker transmitted the eighth packet.

In general, this attack is based on inefficient use of a link cache. The attacker forges CacheCast packets which do not contain payload or the payload is very small. In consequence, storage space in the CSU is wasted. To reduce the impact of this type of attack, CSU slots should be smaller (payloads larger than one slot are stored in contiguous slots). This increases storage space utilisation during attack. We follow this solution in the implementation part (see Chapter 7). Additionally, the link caches can be protected against this attack by ignoring CacheCast packets with sizes below a certain threshold.

3.7 Summary

In this chapter we have presented the idea and design of the CacheCast system that removes redundancy from single source multiple destination transfers. The CacheCast idea is based on a

caching technique applied on a per link basis. The design builds on precise analysis of network elements and requirements for this type of systems. The analysis of network elements indicates that a cache should operate on edges of a link entity and remove redundant payload transfers from the link, thereby increasing the link utilisation. Redundant payloads are restored when a packet enters a router. This, however, does not impact router performance, since the router throughput is independent of the packet size.

The requirements analysis provided in Section 3.2 guided further specification of the design. We recall them briefly here and compare how the final design addresses each of the listed requirements. Our main requirement is that CacheCast must preserve the end-to-end relationship between communicating end-points. This is fulfilled intrinsically, since CacheCast is a caching technique which acts on the packet level. It does not change the end-to-end semantics of the Internet communication. The second requirement refers to reliability of the caching infrastructure and speed of per packet processing. The reliability is achieved by the architecture of the CacheCast system which is based on infrastructure of independent link caches. Since link caches act independently, erroneous behaviour of one link cache does not impact other link caches. Avoiding dependencies between link caches simplifies the system and accounts for reliability. The speed of per packet processing is increased by the CacheCast header which carries a unique payload ID and information about the payload size. Thus, a link cache can immediately compare payloads based on payload IDs instead of payload content, and also knows which part of a packet is redundant. The third requirement is related to economy: CacheCast must have minimum cost of resource consumption and initial investment. We reduce the resource requirements in two ways. Firstly, CacheCast uses server support to transmit the same data within a short period of time; thus, the required cache storage space is reduced. Secondly, packets that carry redundant data are marked with the CacheCast header; thus, other packets do not consume storage space unnecessarily. Finally, since CacheCast is incrementally deployable, the costs of initial investment are significantly reduced.

CacheCast consists of two parts: distributed link cache infrastructure and server support. In the following chapters we evaluate them both. Firstly, we perform analytical evaluation of the link cache infrastructure. We compare CacheCast and multicast bandwidth savings; and we analyse link cache impact on end-to-end congestion control mechanisms. Secondly, we implement both elements and we perform micro-analysis of the implementation. Finally, we present a simple real world setup where we send an audio stream to a number of receivers in a CacheCast enabled network.

Chapter 4

CacheCast efficiency

This chapter is the first of four chapters in which we evaluate different aspects of the CacheCast system. In this chapter we assess the fundamental functionality of link caches, i.e. redundancy removal; and we estimate the total reduction in network traffic generated by single source multiple destination transfers. Following this, the next chapter evaluates fairness between the CacheCast traffic and the normal traffic. In the third and fourth chapter, we assess the computational complexity of the CacheCast elements. The evaluation of each aspect of CacheCast requires a different approach and a different method. To estimate the amount of removed redundancy and the total reduction in network traffic, we perform analytical analysis and simulations. We focus our analysis on four factors which impact the efficiency of the CacheCast system:

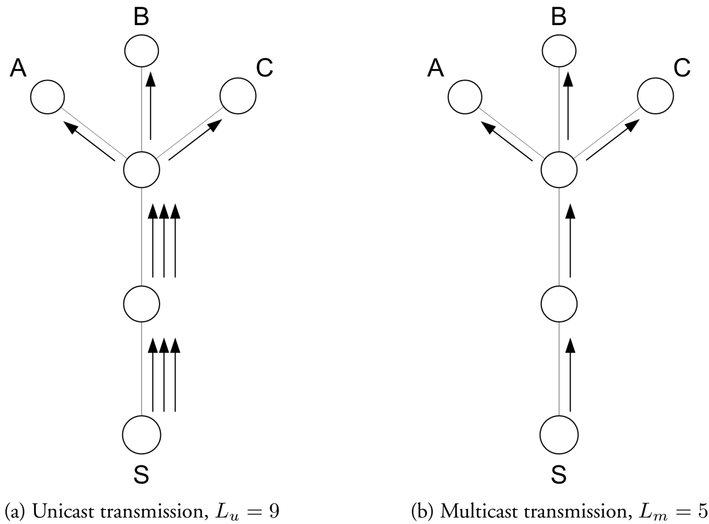
(1) Header size to payload size ratio: Only redundant transfers of the payload part of a packet are suppressed. Therefore, in order to use link caches optimally, a source should pack the maximum amount of data in each transmitted packet. The ratio of packet header size and payload size determines the maximum reduction in network traffic obtainable by link caches.

(2) Number of receivers: Similar to multicast, CacheCast efficiency depends on the number of receivers. The more destinations a source transmits to, the more redundancy in a network, and therefore the greater gains from link caches.

(3) Link cache hold time: Link caches store payload data for a very short period of time. When a source with the slow uplink speed transmits a packet train, this time period may be insufficient to remove all redundant payload transfers from links; which in turn reduces the link cache efficiency.

(4) Deployment range: The CacheCast system is incrementally deployable. Hence, a source can benefit from CacheCast from the deployment of the first link caches in a network. Nonetheless, the efficiency of the partially deployed CacheCast system is only a fraction of the maximum efficiency.

In the following sections we study in turn the outlined factors, but first we describe a metric which we use to assess the efficiency. While analysing a single factor we avoid influence of the remaining factors.



(a) Unicast transmission, $L_u = 9$

(b) Multicast transmission, $L_m = 5$

Figure 4.1: Transmission of the same data to three destinations: A, B, and C using (a) unicast and (b) multicast

4.1 Efficiency metric

To give an insight into the efficiency of the proposed link layer caching, we compare it with a "perfect" multicast scheme. The perfect multicast slightly differs from the IP Multicast. It does not require any additional signalling to establish a multicast tree and to deliver a datagram to receivers. Thus, it is strictly theoretical, but yet it gives us a good reference point.

In order to compare the efficiencies, we use the metric proposed by Chalmers and Almeroth [43]. It is expressed by the ratio of the total number of multicast links over the total number of unicast links that are traversed by datagrams during the delivery of the same data to all receivers in a group.

$$\delta = 1 - \frac{L_m}{L_u} \tag{4.1}$$

When the number of multicast links is similar to the number of unicast links, the efficiency δ is approximately zero, which means that there is no benefit from using multicast. On the contrary, as the efficiency δ approaches one the greater benefits are obtained from multicast. The metric expresses the reduction in the total traffic when using multicast instead of unicast.

For a simple example of the metric usage, let us consider a tree topology as shown in Figure 4.1. When the server S sends the same data to the hosts A, B, and C using unicast it must transmit three datagrams each traversing three links; thus, in total there are nine transmissions in the network and $L_u = 9$. When the server S uses multicast to send the same data to all hosts, only one datagram traverses the first two hops and is replicated at the branching point; thus, there are only five transmissions in the network and $L_m = 5$. The resulting efficiency is $\delta = 1 - \frac{5}{9} \approx 0.44$ which means that multicast reduces the total network traffic by 44%.

4.2 Header transmission costs

The first reduction in the efficiency of CacheCast, when compared to the perfect multicast, lies in the fact that it does not have a common header for all the destinations. Thus, the header part of a packet s_h needs to be *unicast* while the payload part s_p is *multicast*. This is reflected in a modified formula (4.1) for CacheCast efficiency as shown in 4.2:

$$\delta_c = 1 - \frac{s_h L_u + s_p L_m}{(s_h + s_p) L_u} \tag{4.2}$$

If we denote the ratio of the header size to the payload size by r ($r = \frac{s_h}{s_p}$), we can express the reduced efficiency using the perfect multicast efficiency δ (4.1) and the ratio r in the following way:

$$\delta_c = \frac{1}{1+r}\delta \tag{4.3}$$

The factor $\frac{1}{1+r}$ in Equation (4.3) limits the maximum efficiency obtainable by CacheCast. When the header to payload ratio r decreases, the efficiency of the link layer caching approaches the perfect multicast efficiency. However, when the ratio r increases, the efficiency degrades.

The CacheCast efficiency δ_c depends on packets composing cacheable traffic. The maximum efficiency is obtained when the packets are of the maximum size while the header part of the packets is of the minimum size. In the Internet the maximum packet size is limited by the standard maximum transfer unit (MTU) which is, at time of writing, 1500B. The minimum header consists of the link layer header, the CacheCast header, the IP header, and the transport header which is approximately the same size as the minimum packet size in the Ethernet network, i.e. 64B. Therefore, the CacheCast efficiency δ_c can achieve a maximum of approximately 96% of the multicast efficiency δ in the present Internet.

4.3 Finite cache size

The second reduction of the efficiency is related to the finite cache size resulting in additional transmissions of the same payload over the same link. To illustrate this, let us consider a packet train traversing a link. In the perfect case, only the first packet from the packet train carries the payload over the link and the trailing packets are truncated to the header part. However, in the presence of other CacheCast traffic on this link the payload may be evicted from the link cache before the whole packet train has passed the link. This results in additional transmission of the payload reducing the total efficiency.

In [44], it is shown that the efficiency of multicast increases with the growing number of receivers. However, the more receivers the longer is the packet train, and longer packet trains require larger caches, which contradicts our principle of keeping caches small. In order to get an insight into the relationship between the caching efficiency and the number of receivers, we conducted a series of simulations. The simulations are based on a scenario where a source transmits the same data to multiple destinations located in many autonomous systems. All links in the network have caches that are scaled according to the 10ms rule. We assume that payload is removed from a link cache after the 10ms time period. This implies that capacity of all links in the network is completely

Table 4.1: The size of 10ms packet train and its efficiency as a function of the source uplink speed

Source uplink speed	10ms packet train size	Efficiency δ
$512Kbps$	8	0.2325 ± 0.0495
$1Mbps$	16	0.3336 ± 0.0478
$10Mbps$	157	0.7275 ± 0.0070
$100Mbps$	1561	0.8873 ± 0.0006

utilised and the background traffic traversing through these links consists only of CacheCast packets with unique payloads. While this is the worst case scenario and it may not occur, it provides the bottom line for the CacheCast performance. In practice, the capacity of links is rarely fully utilised. Moreover, we expect that the CacheCast traffic will only be a fraction of the total traffic on a link.

4.3.1 Simulation setup

The network topology used in the simulations is based on the multicast tree topology collected in the *mwalk* project[1]. It was created using the *mwalk* tool by traversing paths from a source to a randomly chosen set of receivers. The tree topology has 1950 leaves and is claimed to retain the general characteristics of inter-domain multicast trees. We assume that the multicast spanning tree is to a great extent similar to the tree created by a superposition of unicast routes. The assumption is based on the similarity in the average path length and the underlying network infrastructure strongly constraining the shape of a tree [45].

The simulation is conducted in rounds. During each round a single source inserts only one packet to the tree which corresponds to a time t_s of serializing data on the source output interface. All link caches are using the FIFO replacement policy and have enough room to hold payload for $10ms$; after that time the payload is considered to be evicted. We measure the caching efficiency using (4.1) for the receiver group size varying from 2 to 1000, by choosing 10 times a random set of receivers from the 1950 leaves and then taking the arithmetic mean. The reduction in the efficiency related to header cost transmission, which we discussed in the previous section is not taken into account in the following results.

4.3.2 Impact of the finite cache size

The results of the simulation are presented in Figure 4.2. The y-axis denotes the CacheCast efficiency δ, i.e. the reduction in the total network traffic when compared to unicast. The results confirm that the CacheCast efficiency increases with the growing number of receivers. Furthermore, sources with high uplink speed achieve higher efficiencies when transmitting data to the large number of destinations. Sources with low uplink speed cannot transmit all packets within the cache hold time of 10ms; in consequence, additional payload transmissions decrease the total efficiency. The number of packet headers a source can send within the 10ms time period was previously given in Table 3.1. For ease of reference, we provide this relationship in Table 4.1 and additionally we complement it with the CacheCast efficiency δ corresponding to the given packet

[1] http://imj.ucsb.edu/mwalk/

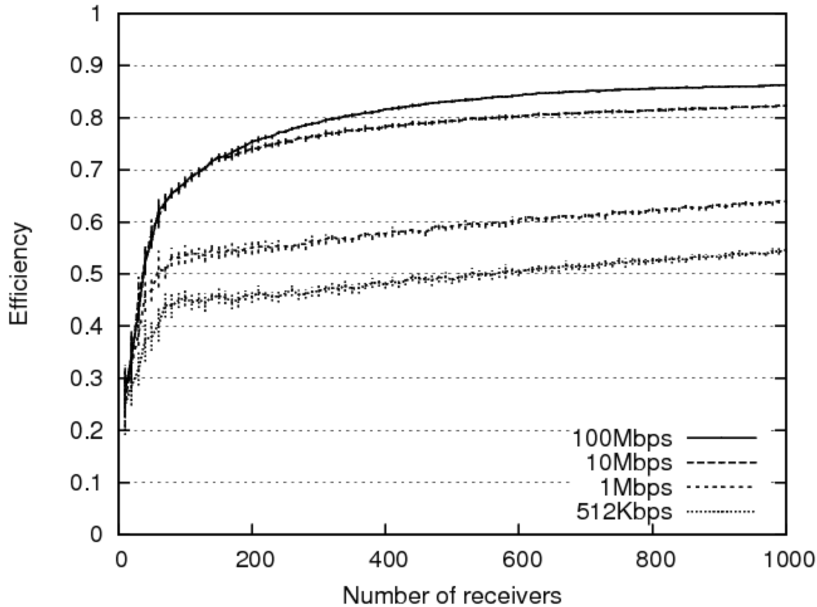


Figure 4.2: The efficiency of the link layer caching

train sizes.

According to Table 4.1, the $100Mbps$ source can transmit to 1561 destinations within the cache hold time. Therefore, for the group size below 1000 its efficiency is equal to the perfect multicast efficiency. Considering the sources with the lower uplink speed, we see that the growth of efficiency with the group size breaks at the point where the packet train time is equal to the cache hold time. At this point caches start to drop the oldest copies of payload. The exact efficiency at this point for all sources is given in Table 4.1. However, even though slow sources transmit only a few packet headers within the cache hold time, they can achieve relatively high efficiency when transmitting to a large group of receivers. The 10ms time period to evict payload from a link cache does not limit the growth in the efficiency with the group size, though it does slow it down.

4.4 Incremental deployment

CacheCast is incrementally deployable, it helps to save bandwidth in a network from the very beginning of deployment. This property is ensured by the link cache architecture. A cacheable packet that exits a link is fully reconstructed. The packet payload is attached to the packet header and the whole packet is further processed in a standard way at a router. If the next link on the packet path does not support caching the cacheable packet passes it unmodified (similar to a regular packet). Thus, it is not necessary to change all links in the Internet to start to gain from CacheCast. Installing a cache on the first hop link from a media server already yields benefits.

The maximum bandwidth savings are obtained with an Internet wide cache deployment. However, considering incremental deployment the question is: What percentage of these savings is

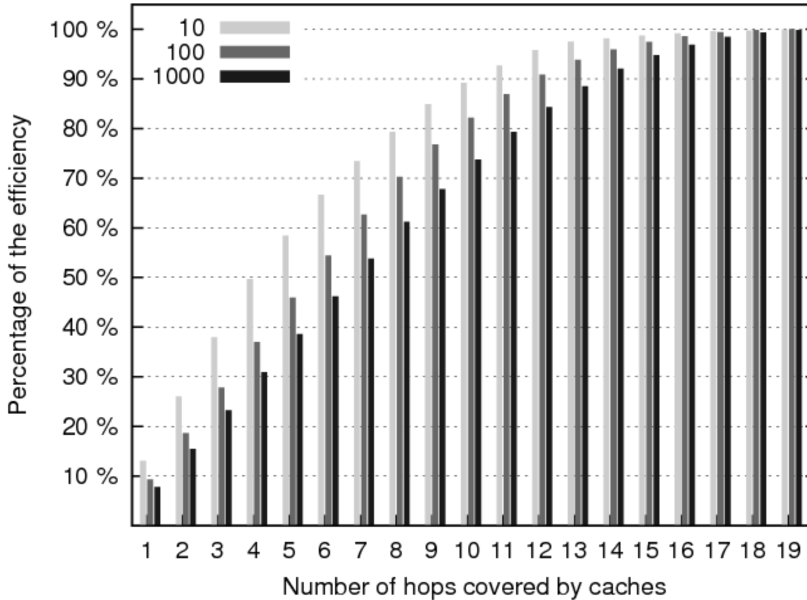


Figure 4.3: Incremental deployment

obtained with gradual cache deployment? We study this question in the simulation where we use the multicast tree topology from the *mwalk* project (cf. Section 4.3.1). In this experiment we assume that link caches are sufficiently large as to remove all redundant payload transmissions and the reduction in the CacheCast efficiency described in the previous section does not occur. We conduct the simulations for three different group sizes consisting of 10, 100, and 1000 receivers. For each group size we gradually deploy caches in the tree starting from the root and finishing at the leaves. In the first simulation only the first hop link from the source caches payloads, while remaining links do not. In the second simulation the first two hop links from the source cache payloads. We repeat the simulation until we cover the whole tree with caches. For each deployment range, we perform 10 measurements of the CacheCast efficiency and we compute the arithmetic mean. In each measurement, a single source transmits the same data to a set of receivers chosen randomly from among the 1950 leaves. We use the efficiency metric δ introduced in Section 4.1 (Equation (4.1)) to assess the amount of removed redundancy.

4.4.1 Efficiency gains per hop

Figure 4.3 shows the percentage of the maximum efficiency (the universal cache deployment) that can be achieved when deploying the link caches over a certain number of hops. The percentage varies depending on the receiver group size. Considering the small group sizes (represented by 10 receivers) the cache deployment over the first six hops yields already approximately 70% of what can be achieved. However, in order to achieve this percentage of the maximum efficiency for large group sizes (100 and 1000 receivers) it is necessary to deploy the link caches over the first nine hops.

The results presented depict the efficiency as a fraction of the maximum efficiency, i.e. the

efficiency that is obtained with Internet wide cache deployment. Additionally, only inter-domain multicast trees are taken into account. However, Internet service providers are mainly interested in their own gains. Thus, the relevant question is: What are the direct benefits of deploying caches inside a single ISP? These can be summarized in the following points:

- The traffic that is confined to a single ISP will achieve near multicast bandwidth utilization.
- The traffic that originates from the ISP and traverses other ISPs will be cached on the way between a streaming source and a gateway inside this ISP. Thus, there will be increased spare bandwidth on the ISP's links.

4.4.2 Cacheable and non-cacheable link

One of the core issues of the incremental deployment is the behaviour of the cacheable traffic on a boundary between a network with the link layer caching and a network without it. Let us consider a packet train with ten packets that originates from the network with link layer caching and which has destinations in another network without link layer caching. In the origin network its size on a link is the size of one payload and ten headers. However, in the destination network (which is without caching) its size on a link is the size of ten payloads and ten headers. Thus, it requires much more link capacity in the second network. Therefore, congestion may occur on the gateway between the networks.

We find this problem orthogonal to the link layer caching. The link layer caching increases link capacity. Thus, the problem resembles a situation where a high capacity link is connected with a low capacity link, or where the sum of input link capacities exceeds the capacity of the output link. This problem is addressed by congestion control algorithms, which are usually part of transport protocols. Similarly, the congestion control is responsible for handling the congestion on the boundary between a network with the link layer caching and a network without it. We show in Chapter 5 how the congestion control ensures "fair" capacity sharing in the presence of CacheCast and how it reduces the packet rate when caching efficiency decreases.

4.5 Summary

In this chapter we have analysed the efficiency of the CacheCast system. The metric, which we use to assess the efficiency, is the measure of the total redundancy removed from single source multiple destination transfers. To find the difference between the maximum efficiency and the CacheCast efficiency, we compare CacheCast with the perfect multicast - a conceptual transport mechanism that removes all redundancy and does not require any signalling.

The reduction in the efficiency of the CacheCast system when compared to the perfect multicast is related to the transmission of unique packet headers to all destinations, finite link cache size, and partial deployment. Considering these factors, we found the following: Firstly, in the present Internet, CacheCast can reach at maximum 96% of the perfect multicast efficiency. It is obtained when a source uses MTU size packets with a standard set of headers. Secondly, the 10ms link cache size is sufficient to achieve the maximum efficiency for sources with the $100Mbps$ uplink speed. Considering sources with slow uplink, even though they can transmit only a few packets within the 10ms time period, they can still achieve relatively high efficiency when transmitting to a large

group of receivers. Thirdly, deploying link caches over the first four hops removes already 30-50% of the total redundancy. Further deployment of link caches yields an additional 6-9% redundancy elimination per hop. The total efficiency reduction is a product of the reductions related to the three factors, i.e.: transmission of unique packet headers, finite link cache size, and partial deployment.

Chapter 5

TCP friendliness

In the previous chapter we have assessed the efficiency of the CacheCast system with respect to the amount of redundancy removed from network links. We found that CacheCast can achieve near perfect multicast efficiency. However, this is achieved on condition that we use large payloads for data transfers, and also on condition that all packets carrying the same content are transmitted within a short period of time of each other.

In this chapter, we investigate the CacheCast impact on fairness in the Internet. Internet fairness is based on the concept of equal resource utilisation of a bottleneck link by different packet flows. It is achieved by the Internet transport protocols that control the packet transmission rates of individual packet flows. However, the CacheCast mechanism modifies packet sizes on links, which may disturb operation of the protocols. This, in turn, will cause "unfair" resource utilisation. Furthermore, CacheCast requires synchronised packet transmission to all data receivers. However, it is not known exactly how this requirement impacts the data transmission rate to individual receivers. We evaluate these issues using the network simulator *ns-2*.

The rest of this chapter is organised in the following manner. In Section 5.1 we elaborate the problem of TCP friendliness in the context of CacheCast operation. Section 5.2 describes the implementation of the CacheCast elements in the network simulator *ns-2*. In Section 5.3, using the network simulator *ns-2* we measure the link cache impact on TCP friendliness in a bottleneck link topology. Subsequently, in Section 5.4, we evaluate the impact of synchronised transmission. Finally, we summarise this chapter in Section 5.5.

5.1 Requirement of TCP friendliness

The Internet traffic consists of individual flows that carry data between hosts. At present, the dominant part of the flows is controlled by the Transmission Control Protocol (TCP). One of the main tasks of the protocol is to ensure fair utilisation of network resources. TCP fairness is based on the idea that two flows which compete for the capacity of the same link should achieve the same share. This understanding of fairness can be extended to a number of flows which share a bottleneck link. In this case, each flow should obtain the same share. Since the dominant part of the Internet traffic is carried by TCP, other transport protocols must be designed to achieve fairness on the TCP basis, i.e. they must be TCP friendly. Furthermore, any new mechanism operating in the Internet should not disturb TCP fairness.

Unlike IP Multicast, CacheCast is independent of the transport protocol. It does not require a

common header to all destinations but only a common payload. Hence, a source can use existing protocols on top of the CacheCast mechanism, although this does require additional considerations. The protocols estimate the packet flow throughput based on factors such as packet size, packet arrival rate, and end-to-end delay. However, these factors are affected in the presence of link cache. Consequently, the protocols operating on top of the CacheCast mechanism may not be TCP friendly.

In order to understand the protocol behaviour in the presence of a link cache we perform simulations in a typical bottleneck link topology, where the bottleneck link is a caching link. We anticipate that the most promising application for the CacheCast system is multiple destination live streaming in the Internet. Therefore, for our simulation scenario, we have chosen the case of a single media server streaming to multiple receivers. We evaluate the CacheCast impact on the media server traffic in the network simulator *ns-2*. We implement CacheCast in *ns-2* for the evaluation, since *ns-2* does not support the CacheCast mechanisms.

5.2 *ns-2* implementation

The network simulator *ns-2* is widely used in networking research for protocol prototyping and evaluation. It provides a library of elements which can be roughly divided into four groups: applications, agents, nodes, and links. The applications and agents are used to create traffic inside a simulated network. The nodes and links form the network topology. A user defines a scenario for the *ns-2* simulation in the form of a script. While the elements are implemented using C++, the script is described with OTcl - an object oriented scripting language; thus *ns-2* combines performance and flexibility.

The *ns-2* CacheCast implementation consists of two parts: link cache and server support. Since the *ns-2* link does not correspond to our concept of a link defined in Section 3.1.1, we integrate the CMU and CSU components directly with the *ns-2* link. We distinguish between these two link models. The *ns-2* link cache implementation is explained in the next section. The server support is implemented as a new component operating on a network node. As a background for the *ns-2* CacheCast implementation we provide information on the *ns-2* packet structure and how an *ns-2* packet is processed by elements.

ns-2 packet structure

An *ns-2* packet is represented by an object depicted in Figure 5.2. It does not belong to any specific network layer and it does not carry any data. Instead, it is a generic object that can represent any type of a packet. The object has two basic pointers: pointer to a block of packet headers (`bits_`) and pointer to packet data (`data_`). The block of packet headers contains information related to different protocols and methods. However, these headers are unrelated to standard packet headers. The block contains at least a common header which describes the fundamental properties of a packet, such as packet type (`ptype_`) and packet size (`size_`) as indicated in Figure 5.2. The packet data pointer points to an object containing the real payload of this packet. However, in our simulation, packets do not have real payloads and the pointer is cleared.

The *ns-2* packet size in a simulated network is not related to the size of a packet object. It is simulated using the `size_` parameter stored in the common header. For example, a delay component

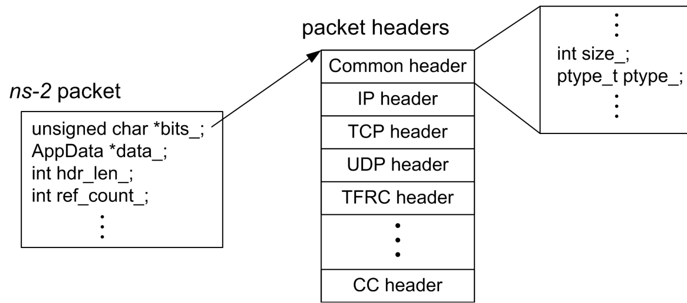


Figure 5.1: *ns-2* packet

of a link element uses this parameter to find the time to serialise a packet. Since the packet object size is not related to the simulated packet size, the packet header block contains all packet headers defined in *ns-2* by default. If memory utilisation during a simulation is an issue, a user can specify exactly which headers should be removed from the packet header block to reduce the memory footprint. However, we did not encounter this problem in our simulations.

Connectors and classifiers

Packet processing components are derived from two classes: connector class and classifier class. The connector class is used to implement components that process a packet in a pipeline such as agents or link components. The classifier class is used to implement components demultiplexing packets according to a predefined classification.

The CMU, CSU, and server support components are implemented using the connector class. This class defines an object with pointers to two downstream objects. The first pointer points to the next object in a processing pipeline and is called `target_`. The second pointer called `drop_` points to an object discarding packet (primarily used by a queue component). In order to pass a packet to a downstream object, a component invokes a standard receive function of the next object (`target_->recv(Packet *p, Handler *h)`). The *ns-2* objects are unidirectional and have no upstream connections. For instance, to model a bidirectional link, *ns-2* combines two simple links that transfer packets in the opposite directions.

The components form basic elements used in the simulation. For example, a link element consists of three components: queue, delay, and a component decreasing time-to-live. An element is defined in the form of an OTcl script which specifies connections between the element components. Before a simulation starts, *ns-2* binds together components using the `target_` and `drop_` pointers according to a simulation script.

5.2.1 *ns-2* CacheCast header

We define the *ns-2* CacheCast header as described in Listing 5.1. The header carries information on the payload ID (`payload_id_`), the size of a cacheable part (`cacheable_part_`), and the original size of a packet (`real_size_`), i.e. the packet size before CacheCast encapsulation. These information elements are set by our *ns-2* server support before a packet is sent to a link element. The (`offset_`) variable is a standard variable in any packet header and it provides information on the offset of this

header in the packet header block.

Listing 5.1: *ns-2* CacheCast header

```
struct hdr_cc {
  int payload_id_;          // the unique id for this address
  int cacheable_part_;      // the size of the cacheable part
  int real_size_;           // the original size of a packet
  int& payload_id() { return payload_id_; }
  int& cacheable_part() { return cacheable_part_; }
  int& real_size() { return real_size_; }

  static int offset_;
  inline static int& offset() { return offset_; }
  inline static hdr_cc* access(Packet* p) {
    return (hdr_cc*) p->access(offset_); }
};
```

We modify the *ns-2* files to attach the CacheCast header to the packet header block for all packets by default. Please note, that this does not change the size of any packet, since the packet header block size is not related to the size of an *ns-2* packet. The simulated CacheCast header is 12B which corresponds to the CacheCast header size in our Click implementation. The CMU component extends the simulated packet size by 12B before transmission on a link and the CSU component restores the original packet size. In order to distinguish between CacheCast packets and standard packets we use the field `cacheable_part` which is set to zero for all standard packets.

5.2.2 *ns-2* link cache

Design

The *ns-2* link is modelled by three chained components: queue, delay, and a component decreasing packet time-to-live (TTL) (see Figure 5.2). In order to find transmission delay of a packet on a link, the simulator uses the packet size parameter stored in the common header of the packet. The delay component calculates the time to serialise the packet on a link according to the packet size and the link capacity. This time is also used to schedule transmission of the next packet, because only when the current packet has been already serialised the next packet can be transmitted. Thus, the modification of the packet size variable (`size_`) is sufficient to simulate the payload removal and restore process.

Figure 5.2: *ns-2* link

We have implemented CMU as a table where we store the payload IDs of the CacheCast packets. When a packet is received by the CMU, its payload ID is compared against those stored in the table, and if a cache hit occurs the packet size is decreased by the payload size. If the payload ID is not found in the table, then the ID is inserted into the table according to the replacement policy. We place the CMU component after the queue, since the queue is conceptually a part of a router; and before the delay component. Therefore, the time to serialise the CacheCast packet and the

time to serve the next packet are computed according to the reduced packet size while the queue component evaluates packet drop based on the original packet size.

Since the payload is not removed from a packet, but only the size of the packet is changed, the implementation of the CSU component is straightforward. It simply restores the packet size to its original size. It should be noted that *ns-2* does not model any computational complexity and costs of copy operations within a node. The CSU component is placed between the delay and TTL components. Therefore, the caching link element consists of the following five components which are chained together: queue, CMU, delay, CSU, and TTL (depicted in Figure 5.3).

Figure 5.3: *ns-2* caching link

Implementation

The CMU and CSU components are derived from the connector class. We implement the components functionality in the receive function (`recv(Packet *p, Handler *h)`) which is called upon packet arrival. When a packet is processed we invoke the receive function of a downstream object pointed to by the variable `target_`. The core of the CMU implementation is presented in Listing 5.2. The core of the CSU implementation is presented in Listing 5.4.

The CMU component is integrated with the *ns-2* link element therefore all *ns-2* links cache packets by default. In order to simulate standard links we have to disable the CMU functionality. This is controlled by the `enable_` variable. If the variable is cleared, packets are immediately sent to a downstream object. To distinguish CacheCast and standard packets we check whether a packet has a cacheable part. If the `cacheable_part` variable is set to zero we pass this packet immediately further downstream. Otherwise, we compute a payload ID based on the packet payload ID and the packet source address.

In our simulations we measure impact of the link cache efficiency on the CacheCast traffic. The efficiency is controlled using the `max_eff_` variable. It determines a fraction of redundant data which is removed by the link cache. When the `max_eff_` variable value is 1.0 or higher, all packets that have payload in the link cache are truncated to the header size. If the variable value is below 1.0, only the fraction of packets that have payload in the link cache is truncated. In the last step the packet size is increased by the CacheCast header size, simulating CacheCast encapsulation and the packet is passed to a downstream component.

Listing 5.2: Cache management unit

```
void CacheManagementUnit::recv(Packet *p, Handler *h)
{
  struct hdr_cmn *cmn_h = HDR_CMN(p);  // pointer to the common header
  struct hdr_ip *ip_h = HDR_IP(p);     // pointer to the IP header
  struct hdr_cc *cc_h = HDR_CC(p);     // pointer to the CacheCast header
  int payload_id;

  if ((!enable_) || (cc_h->cacheable_part() == 0)) {
    /* Caching on the link is disabled or the packet does not carry
     * a cacheable content. We send the packet to the next component. */
```

```
    recv->target_(p, h);
    return;
  }

  payload_id = (cc_h->payload_id() & 0x0000FFFF) |
               (ip_h->src().addr_ << 16);

  if (cache_->update(payload_id))
    cacheHit_++;
  else
    cacheMiss_++;

  /* Simulate reduced efficiency */
  if ((cacheHit_ % 100) < (max_eff_ * 100))
    cmn_h->size() = cc_h->real_size() - cc_h->cacheable_part();

  cmn_h->size() += CC_HDR_LEN;

  target_->recv(p, h);
}
```

To describe the cache replacement policy we define a class named *Policies* and derive from it three subclasses for FIFO, Clock, and LRU replacement policies. The CMU is configured to cache packets according to one of these policies and it holds a pointer to the policy object in the `cache_` variable. To update the CMU cache the `cache_->update(payload_id)` function is invoked. If the payload ID is already in the cache the function returns the **true** value and the **false** value otherwise. We show the code of the FIFO replacement policy in Listing 5.3 as an example for the `update()` function. The replacement policy is implemented using a circular buffer. Since we simulate only small link caches, linear lookup for a payload ID in a cache is sufficient.

Listing 5.3: FIFO replacement policy

```
bool PolicyFIFO::update(int payload_id)
{
  int i;

  /* We perform linear lookup in the cache to find the payload ID. */
  for(i = 0; i < size; i++)
    if(cache[i] == payload_id)
      return true;

  /* The payload ID is not in the cache. We insert it at the current
   * beginning of the circular cache.
   */
  cache[cachePtr] = payload_id;
  cachePtr = (cachePtr + 1) % size;

  return false;
}
```

The only task of the CSU component is to restore the original size of the CacheCast packets. The CSU component does not implement a payload store, since packets do not carry content and

it does not implement error detection, since the simulated links do not drop and do not reorder packets.

Listing 5.4: Cache store unit

```
void CacheStoreUnit::recv(Packet *p, Handler *h)
{
  struct hdr_cmn *cmn_h = HDR_CMN(p);    // pointer to the common header
  struct hdr_cc *cc_h = HDR_CC(p);       // pointer to the CacheCast header

  if(cc_h->cacheable_part() > 0) {
    cmn_h->size() = cc_h->real_size();
  }

  target_->recv(p, h);
}
```

5.2.3 *ns-2* server support

Design

Network traffic in *ns-2* is generated by agents acting as communication end-points. An agent implements a transport protocol. It is attached to a network node and it has a counterpart – a sink-agent – attached to another node. Thus, this source sink agent pair defines a single packet flow in a simulated network. A source agent is often controlled by an application, which can, for example, be the file transport protocol (FTP), telnet, or a traffic generator such as constant bit rate (CBR) generator. For instance, in our simulation we generate a data stream between two nodes in a simulated network using a CBR generator on top of a TFRC agent attached to a source node and its counterpart TFRC-sink agent attached to a sink node. Nonetheless, not all agents require an application to trigger packet transmission. The aforementioned TFRC agent generates packets at the maximum rate permitted by the TFRC congestion control when there is no application to control its sending rate.

The *ns-2* server support is a single element operating at a network node. It processes all packets created by agents which are attached to this node as depicted in Figure 5.4. The *ns-2* server support is not functionally equal to the server support described in the design. It only creates the CacheCast header for each transmitted packet but it does not enforce the tight packet train structure. Since packets which are created by the agents do not carry data, the task of the *ns-2* server support is to determine which packets generated by different agents have the same payload. This is achieved using an epoch concept. All packets created by the agents within one epoch are considered to have the same payload and the payload ID of these packets is equal to the epoch number. A new epoch starts, when one of the agents creates the second packet within the same epoch. Thus, the epochs are generated based on the rate of the fastest flow from among all flows created by the node agents. We describe the implementation of this mechanism in the following paragraph.

In order to simulate the server support as it is designed, we use the CBR generators on top of the transport protocols. Since a single application can only drive a single agent, we install the CBR generator for each transport protocol instance. The generators are synchronised in time and send the same size packets at the same rate. If a congestion control mechanism of a transport protocol

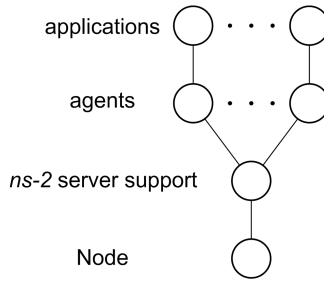


Figure 5.4: *ns-2* server support

prohibits packet transmission, the CBR send request is discarded. Thus, the *ns-2* server support generates a sequence of tightly packed packets. However, before we evaluate the CacheCast system as a union of the link caches and the server support mechanisms, we examine the impact of the link caches on the network traffic consisting of loosely synchronised data streams.

Implementation

The *ns-2* server support has two tasks: to find the cacheable part of a packet and to assign a payload ID. Unlike a network packet, an *ns-2* packet is not a result of the encapsulation process, where different protocol layers are clearly defined and application data is always at the tail. An *ns-2* packet is created by an agent which annotates it with a type. The type can be defined either by an agent or by an application and there is no rule for it. For example, the CBR application driving the UDP agent produces packets of the CBR type; however, the CBR application driving the TFRC agent produces packets of the TFRC type. Furthermore, some packet types provide the application data size while other types do not provide it. This difficulty is reflected in our implementation shown in Listing 5.5. Our implementation finds the payload size only for three different packet types. While the TFRC packet provides the payload size in its header, the TCP and CBR packets do not have this element of information and we have to compute it based on the standard header lengths.

The payload ID gets the value of the current epoch ID which is a natural number. We generate epoch IDs in the following way. For each attached agent we hold the information about the epoch ID at the point this agent transmitted the last packet. This information is stored in the `agent_epoch_` table, and agents are distinguished by their port number (`src().port_`). A new epoch begins when a certain agent transmits the second packet in the same epoch. In this case we increase the epoch ID by one. As a result new epochs are generated at the pace of the packet transmission rate of the fastest agent.

Listing 5.5: Server support

```
void ServerSupport::recv(Packet *p, Handler *h)
{
  struct hdr_cmn *cmn_h = HDR_CMN(p);        // pointer to the common header
  struct hdr_cc *cc_h = HDR_CC(p);           // pointer to the CacheCast header
  struct hdr_tfrc *tfrc_h = HDR_TFRC(p);     // pointer to the TFRC header
  struct hdr_ip *ip_h = HDR_IP(p);           // pointer to the IP header

  int cacheable_part = 0;   // size of the cacheable part of a packet
```

```
  if (cmn_h->ptype() == PT_TFRC)
    cacheable_part = tfrc_h->psize;
  else if (cmn_h->ptype() == PT_TCP)
    cacheable_part = cmn_h->size() - IP_HDR_LEN - TCP_HDR_LEN;
  else if (cmn_h->ptype() == PT_CBR)
    cacheable_part = cmn_h->size() - IP_HDR_LEN - UDP_HDR_LEN;

  if (cacheable_part > 0) {
    /* If any of the agents have sent already in this epoch advance the epoch */
    if (agent_epoch_[ip_h->src().port_] == current_epoch_)
      current_epoch_++;

    cc_h->real_size() = cmn_h->size();
    cc_h->payload_id() = current_epoch_;
    cc_h->cacheable_part() = cacheable_part;

    agent_epoch_[ip_h->src().port_] = current_epoch_;
  }

  recv->target_(p, h);
}
```

5.3 Loosely synchronised streams

We study the link cache impact on congestion controlled transport protocols in the single bottleneck link topology as depicted in Figure 5.5. All links in the topology are the caching links as described in Section 5.2.2. We install the *ns-2* server support at the streaming source. The bottleneck link queue is managed by the random early detection RED queuing discipline [46] in byte mode. We let *ns-2* automatically configure the queue parameters. Two types of traffic compete for the bottleneck link capacity.

1. **Cacheable traffic:** It is generated by the streaming source and consists of a number of streams. Each stream has a different receiver located behind the bottleneck link. We use TFRC [47] to perform congestion control over a single stream. Our decision is based on the fact that TFRC is designed for streaming media, it is well accepted in the research community, and it does not produce bursts of packets like TCP does. The round trip time (RTT) between the source and each receiver is the same, therefore the streaming rate of the individual flows is similar. The streams are not driven with the same rate by the source, rather we let each stream compete for the bottleneck link capacity individually. Thus, a single TFRC sender transmits a packet as soon as the congestion control algorithm permits it. We simulate a single TFRC stream using a stand-alone TFRC agent connected to the source node and a TFRC-sink agent connected to one of the TFRC sink nodes. The *ns-2* server support marks payloads with the same ID within one epoch which is determined by the rate of the fastest stream of all streams. The payload size is 1000B.

2. **Non-cacheable traffic:** It is represented by 100 TCP flows each originating from a distinct source and with a destination located behind the bottleneck link. The TCP flows are gen-

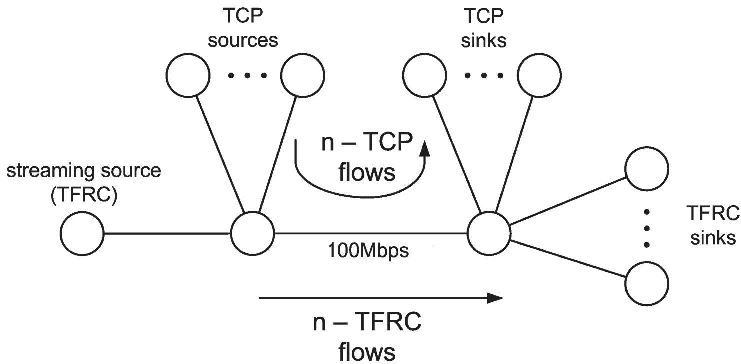


Figure 5.5: Single bottleneck scenario

erated by FTP applications which transfer unlimited amounts of data in 1000B payloads. We simulate a single TCP flow with an FTP application attached to a TCP agent. The TCP agent is connected to one of the TCP source nodes and transfers data to its counterpart TCP-sink agent connected to one of the TCP sink nodes. All TCP connections have the same RTT and their throughput is limited only by the bottleneck link capacity.

We perform two experiments in the aforementioned simulation setup. In both experiments we measure the share of the bottleneck link bandwidth that is consumed by TCP and TFRC flows. We also measure the average receiver throughput to gain the end host perspective. The throughput is measured on the network layer. To factor out the influence of the RTT on the behaviour of TFRC and TCP flows, the experiments are performed in three different RTT configurations:

1. **Low RTT** The TFRC and TCP flows have the same RTT of 40ms.

2. **High RTT** The TFRC and TCP flows have the same RTT of 100ms.

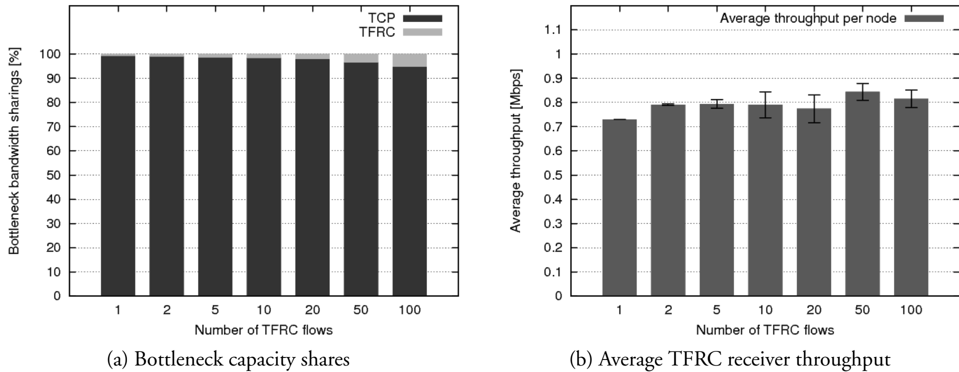
3. **Different RTT** The TFRC flows have 40ms RTT while the TCP flows have 100ms RTT.

The results are obtained in simulations which are run for 180 seconds. We remove the first 60 seconds from the measurements to avoid any influence of the transient behaviour of TCP and TFRC.

5.3.1 Effect of increasing the number of receivers

In the first experiment, we exponentially increase the number of receivers in the streaming session. The number of TCP flows is fixed to 100. We expect the TFRC flows to obtain incrementally more of the bottleneck link bandwidth share with the increasing number of flows; however, this is not the case.

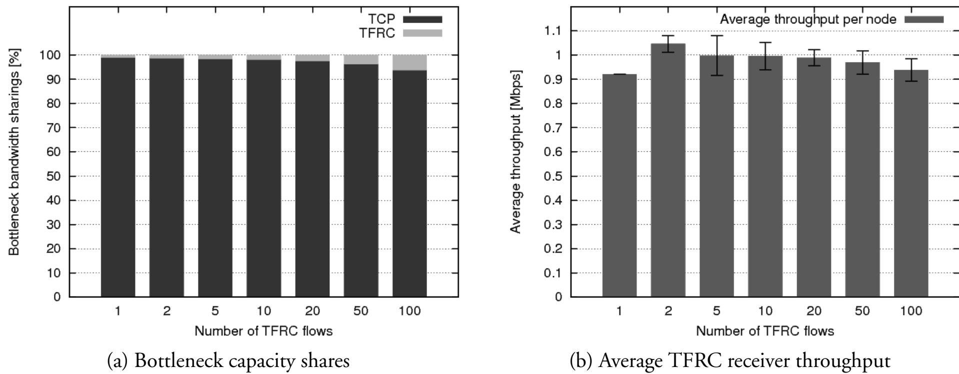
Figures 5.6, 5.7, and 5.8 present the TFRC and TCP shares of the bottleneck link capacity as a fraction of the total capacity and the average TFRC receiver throughput. The results show a surprising behaviour of TFRC. The increase in the number of TFRC streams has very little impact on the TFRC shares. In the **Low RTT** and **High RTT** configurations 100 TFRC flows obtain only 5% and the **Different RTT** obtains 15% of the bottleneck link capacity while we would expect



(a) Bottleneck capacity shares

(b) Average TFRC receiver throughput

Figure 5.6: Increasing the amount of TFRC flows on a bottleneck link, the **Low RTT** configuration



(a) Bottleneck capacity shares

(b) Average TFRC receiver throughput

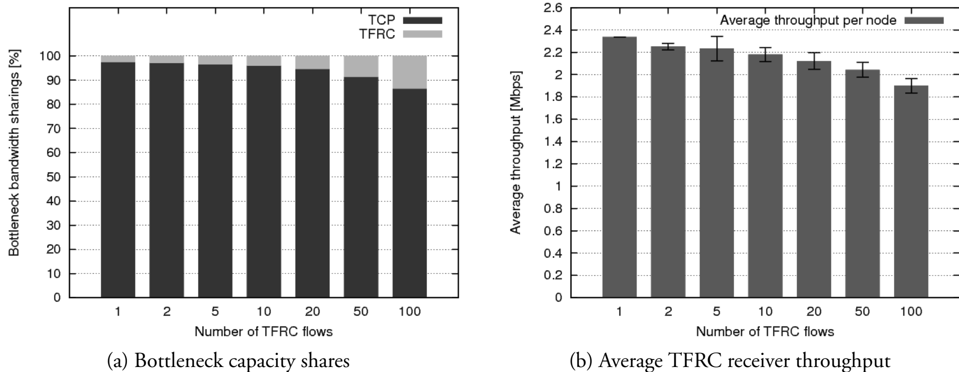
Figure 5.7: Increasing the amount of TFRC flows on a bottleneck link, the **High RTT** configuration

it to take 50%. However, when analysing the average receiver throughput, we notice that it is approximately constant regardless of the number of receivers.

When analysing the average receiver throughput, we find that in the **High RTT** and **Low RTT** configurations TFRC and TCP flows achieve similar end-to-end throughput. In the **High RTT** configuration these throughputs are almost equal. The average TFRC achieves approximately 0.95Mbps and the same throughput achieves the average TCP flow[1]. In the **Low RTT** configuration TFRC achieves approximately 0.8Mbps while TCP achieves 0.95Mbps which is still relatively similar. Considering the **Different RTT** configuration, the average TFRC receiver throughput achieves twice more than the average TCP receiver throughput. However, this difference is related to unfair competition of the TFRC flows with the TCP flows which have much larger RTTs. The next experiment confirms this issue.

We speculate that this behaviour originates from the TFRC congestion control, which competes with TCP on a rate basis. TFRC adjusts its sending rate to be "fair" with the TCP sending rate.

[1]The average TCP throughput is $t_h = 100Mbps * 95\% * \frac{1}{100} = 0.95Mbps$.



(a) Bottleneck capacity shares

(b) Average TFRC receiver throughput

Figure 5.8: Increasing the amount of TFRC flows on a bottleneck link, the **Different RTT** configuration

However, when caching is enabled TFRC packets carry only the header part of a packet thus they consume much less of the bottleneck link capacity which in turn can be utilised by TCP. In all figures, together with the average receiver throughput we also mark the standard deviation from the average. The low variation among receivers confirms our assumption on the cacheable traffic, i.e. that all TFRC streams achieve a similar rate.

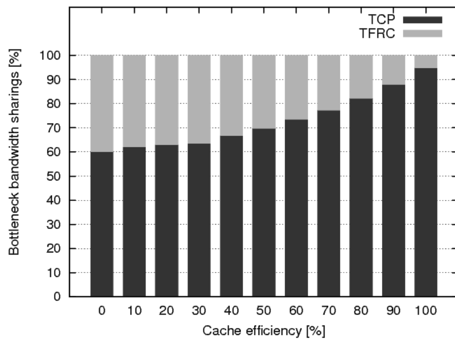
5.3.2 Effect of decreased caching efficiency

In the second experiment we analyse the impact of the caching efficiency on the bottleneck link. The caching efficiency metric denotes the ratio of the number of packets without payload to the total number of packets in a packet train. The first packet in a packet train always carries a payload; thus we do not count it. We fix the number of receivers to 100 and gradually increase the caching efficiency. We start with no caching, where all packets carry a payload, and finish with a perfect caching where only one payload is transmitted per packet train.

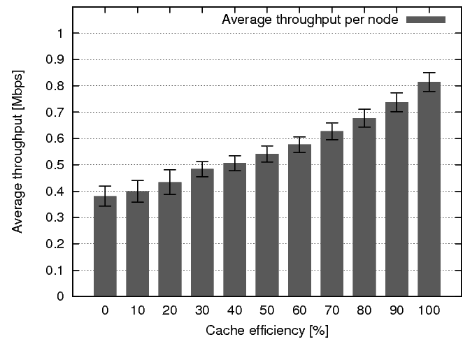
The results of the simulations for the three configurations are presented in Figures 5.9 , 5.10, and 5.11. When there is no caching the TFRC and TCP flows obtain "fair" shares of the bottleneck link capacity in the current understanding. We find that these shares are only equal in the **High RTT** configuration. In the remaining two configurations either TCP or TFRC obtains more of the bottleneck link capacity.

With the increasing caching efficiency the TFRC share of the bottleneck link capacity decreases and the TCP share increases respectively. However, when comparing these two types of traffic based on the end-to-end throughput, we observe a similar increase in the average receiver throughput in the presence of caching for both the TCP flows and TFRC flows. Specifically, in the configurations **Low RTT** and **High RTT** the average TFRC receiver throughput doubles, and the average TCP flow throughput increases by 50% in the **Low RTT** configuration and doubles in the **High RTT** configuration respectively. Similar, in the **Different RTT** configuration the TFRC throughput triples and the TCP throughput increases by 150%.

The link caches do not disrupt the current network understanding of fairness. Both TCP and TFRC take advantage of it. In the presence of caching, TFRC flows make space for TCP flows on

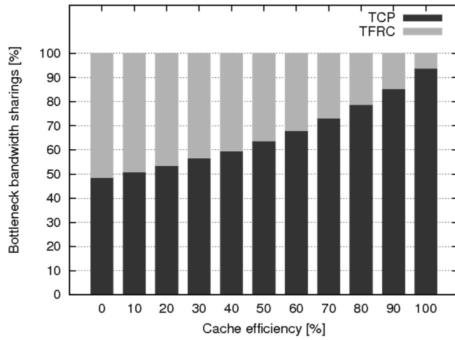


(a) Bottleneck capacity shares

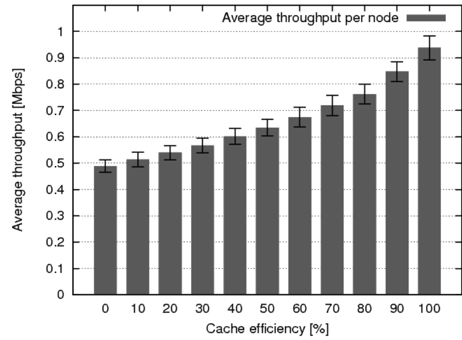


(b) Average TFRC receiver throughput

Figure 5.9: Increasing the caching efficiency on a bottleneck link, the **Low RTT** configuration

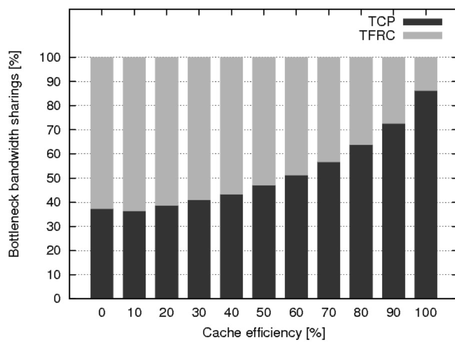


(a) Bottleneck capacity shares

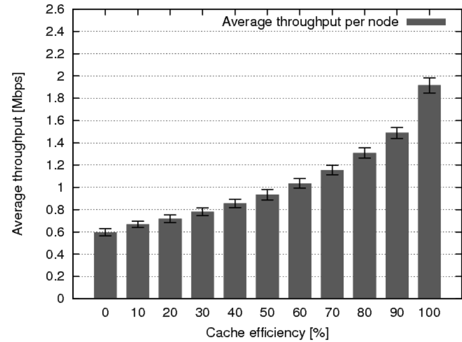


(b) Average TFRC receiver throughput

Figure 5.10: Increasing the caching efficiency on a bottleneck link, the **High RTT** configuration



(a) Bottleneck capacity shares



(b) Average TFRC receiver throughput

Figure 5.11: Increasing the caching efficiency on a bottleneck link, the **Different RTT** configuration

the bottleneck link. While not all the space is utilised by TCP, TFRC still benefits from it. In all configurations the average receiver throughput at least doubles in the presence of CacheCast.

5.4 Tightly synchronised streams

In the previous experiments we evaluated the link cache impact on the loosely synchronised TFRC streams. In this setting, a single TFRC stream controls its rate individually to achieve the maximum throughput. This, however, does not completely comply to the CacheCast design, where a source must transmit the same data chunk in the short time interval to all destinations. Thus, a single TFRC sender cannot transmit a packet immediately when the congestion control algorithm permits, but it must wait for an external source to trigger the transmission. To illustrate this problem, let us consider a simple scenario of an IP TV server streaming a single channel to a number of destinations using TFRC. The server creates media chunks according to the media stream bit rate and sends them to all TFRC senders at the same time. In this scenario, the task of the TFRC senders is not to achieve the maximum bit rate, but rather to follow the media stream rate and in the case of congestion to reduce transmission rate.

To evaluate the impact of the stream synchronisation, we use the same topology as described in Section 5.3 with the bottleneck link capacity of 100Mbps. We generate 100 TCP streams on the bottleneck link. The streaming source has 100 receivers located behind the bottleneck link. The source uses TFRC to control the streaming rate to each receiver. Both TCP and TFRC flows have RTT of 100ms which corresponds to the **High RTT** configuration from the previous experiment.

In this experiment all TFRC senders are driven by the CBR traffic generators which trigger packet transmission at the same time and according to the predefined rate. If the TFRC sender prohibits the packet transmission due to congestion, the transmission request issued by the CBR generator is ignored, which means the packet is dropped. The TFRC senders do not queue packets. As a result, the streaming source produces a tightly synchronised packet train. We conduct a series of simulations where the streaming source transmits data with increasingly higher rates. We probe the streaming rate in the range of 50Kbps to 1.5Mbps with the interval of 50Kbps. Please note that since a TFRC sender does not have an input queue, the TFRC stream will not achieve its maximum throughput on the bottleneck link when competing with other flows.

Figure 5.12 shows the average receiver throughput as a function of the source streaming rate along with the standard deviation. Considering the source streaming rate in the range of 50Kbps to 600Kbps we see that all receivers get the full stream. However, increasing the streaming rate further does not yield similar increases in the throughput on the receiver side. The TCP flows competing with the TFRC streams prohibit further consumption of the bottleneck link capacity. A part of the stream is dropped either at the streaming source or in the network at the bottleneck link queue. We have preformed measurements of the packet drop at the bottleneck link queue and we found that it is in the range of 1.5-1.8% regardless of the source streaming rate. Hence, the packet drop is caused by the TFRC sender which prohibits the CBR generator to send data.

We observe that the maximum average receiver throughput is 0.83Mbps when the source streams at the rate of 1.15Mbps. This is approximately 10% less than the loosely synchronised streams achieved when competing for the bottleneck link capacity in the previous experiment (see Figure 5.7). This confirms our hypothesis that the synchronised TFRC senders will not achieve the same throughput as the loosely synchronised streams. A simple solution for this problem would be

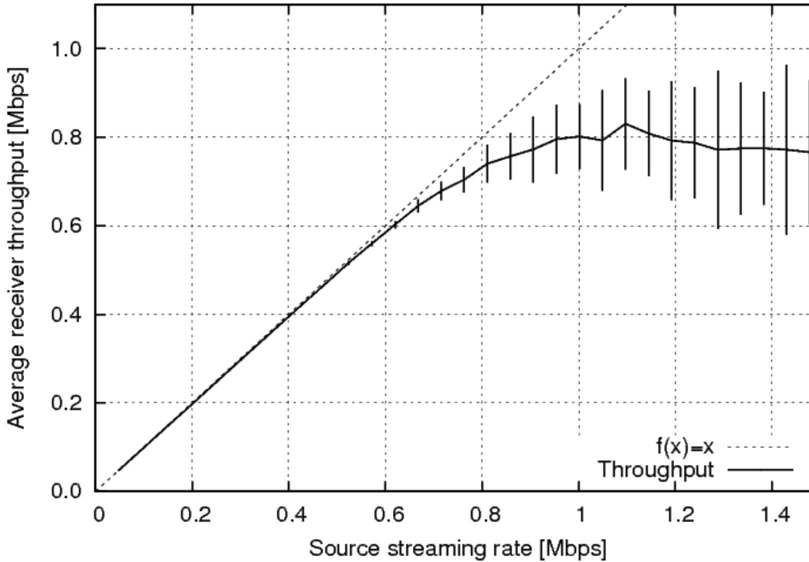


Figure 5.12: Increasing streaming rate in the bottleneck link topology

to take this difference into account in the TFRC formula. However, this requires further analysis outside the scope of this thesis.

5.5 Summary

In this chapter we have investigated the CacheCast impact on TCP friendly rate control. We have analysed the impact of the link cache and the server support separately. In our simulations we used TCP to transport non-cacheable data and UDP with TFRC to transport cacheable data. The simulations indicate that when caching is enabled in a network, TCP flows obtain much more of the bottleneck link capacity when competing with TFRC flows. This could suggest that link caches disturb fairness in the network. However, we argue that the opposite is true. Measuring the receiver throughput we find that even though TCP obtains more network resources it delivers approximately the same amount of data to a single receiver as TRFC does within the same time unit. Thus, from the end-to-end point of view CacheCast achieves fairness.

The server support synchronises data transmission over multiple connections. It sends a data chunk to all connections at the same time. Individual connections do not queue the data chunk when the congestion control algorithm prohibits sending it, but instead drop the data chunk. Queuing would lead to unsynchronised transmission. Since connections do not have input queues, they are not able to achieve the fair share of resources when competing with the standard flows. We have found that the synchronisation mechanism built in the server support reduces the average TFRC throughput by approximately 10%. Since the TFRC rate estimation is based on a formula, it could be modified to compensate for the error. However, this is an issue for future work.

Chapter 6

Computational complexity - server support

This is the third chapter of the evaluation part. In the first chapter we have performed analyses and simulations to estimate the amount of removed redundancy from the Internet links when using CacheCast. We have used the perfect multicast efficiency as the reference efficiency for the CacheCast system and evaluated how the CacheCast architecture limits this efficiency. We have studied the impact of the following three architecture elements: distinct packet headers, finite cache size, and limited deployment. In this and the following chapter, we discuss the last element which can reduce CacheCast efficiency when operating in the Internet, namely the computational complexity of the system. If the server support or the link cache elements require a considerable amount of computation to operate, it will render the system inefficient. In order to answer this question, we design and implement these two components of the CacheCast system, and then we perform a detailed analysis. This chapter covers the server support part of the system and the following chapter describes the link cache part.

The server side implementation consists of two elements: a system call and a kernel module. The system call provides the means to transmit the same data to a group of hosts, which is very similar to work done in [48] and [49]. However, unlike these related works, the CacheCast server support additionally eliminates redundant payload transmission. The kernel module provides an interface to control the CacheCast server support and provides a set of auxiliary functions for the system call. Just how the server support implementation works is demonstrated in a small testbed. The testbed consists of a server streaming an audio file, the CacheCast router, and two machines hosting clients. The results show that, for example, over a 25 Mbps link CacheCast can – compared to a traditional unicast solution – scale up the number of served clients by a factor of 10 and more.

The rest of this chapter is organised as follows: In Section 6.1 we briefly describe the CacheCast system elements and identify potential computational bottlenecks. Before we discuss in depth the server support implementation, in Section 6.2 we introduce a common structure of the CacheCast header for the server support and the link cache elements. In Section 6.3 we describe the Linux implementation of the server support. The evaluation of this element is given in Sections 6.4 and 6.5. Finally, we draw conclusions in Section 6.6.

6.1 Computational bottlenecks

CacheCast is a link layer caching technique that operates on point-to-point links. As it is depicted in Figure 6.1, we install the cache management unit (CMU) on the link entry and the cache store

unit (CSU) on the link exit. The CMU controls the CSU and it knows exactly which payloads are in the CSU. Whenever the CMU recognises that payload of the currently transmitted packet is in the CSU on the link exit, it replaces the payload with an index to the payload in the CSU and transmits it. Thus, when considering a series of packets that share the same payload, only the distinct packet headers and one copy of payload traverses a link. We refer to this structure as a *packet train*. On the link exit the payload index is used to find a relevant payload in the CSU. Next, the payload is attached back to the packet header and the packet is processed in a normal way on a router.

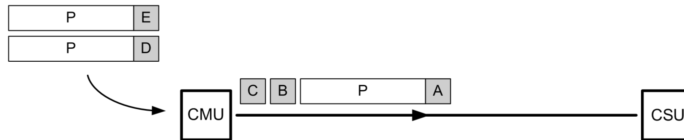


Figure 6.1: Extended directed link

Our core design assertion is that servers support CacheCast. For each packet to be subjected to CacheCast a server must first create the CacheCast header. Packets without the CacheCast header are ignored and not cached. The CacheCast header is placed between the link layer header and the IP header. A server stores the information on redundant payload size and payload ID in the CacheCast header which, combined with the server IP address, uniquely identifies the payload in the Internet. These two elements of information greatly simplify the caching process since the CMU does not need to perform any sophisticated redundancy detection on an incoming packet, but simply compares the payload ID with the IDs of the payloads that are in the CSU. The INDEX field is an administrative field used to pass an index value between the CMU and the CSU in the described manner.

Besides the cache processing complexity the CacheCast costs are also related to the size of the CSU. Therefore, a server aware of caching is responsible for batch requests for the same data and transmitting it within the minimum amount of time. Thus, the required CSU size is minimised. In Chapter 3, we estimated the required size based on different packet train lengths and different source uplink speeds. We found that the CSU size corresponding to 10ms of data traffic flowing through the associated link is sufficient.

Considering the CacheCast architecture, we find that the computational complexity is mainly related to the CMU element which must perform payload ID lookup and modify a packet immediately before link transmission. Additionally, the CSU element may become a bottleneck when storing or restoring large packet payloads. Avoiding the payload copy operation with, for example, a virtualisation mechanism could reduce this burden. Considering the server support, we do not find potential computational bottlenecks. Nonetheless, other types of problems may arise due to the specific characteristic of the CacheCast traffic.

6.2 CacheCast header in the Ethernet networks

The previous evaluations did not require specification of the CacheCast header structure and per field byte distribution in the header. In this chapter and the following chapter we describe the

CacheCast implementation which initially requires the introduction of a common format for the CacheCast header. Please notice that it is sufficient that only the CMU and CSU elements composing a single link cache (and respectively the server support and the first hop CSU element) must use the same structure for the CacheCast header. Hence, the CacheCast header structure can vary between link caches. Nonetheless, it is far more convenient to use a standard header across the same type of link technology.

Our CacheCast implementation operates only in the Ethernet networks; therefore, the CacheCast header is designed only for this type of networks. In Listing 6.1 we show the CacheCast header structure in the C language format. The header contains three information elements (INDEX, payload ID, and payload size) that are related to CacheCast and are broadly discussed in Section 3.4.1. Additionally, the CacheCast header contains the `packet_type` field. This field has the same semantics as the type field of the Ethernet header and it is used to store temporarily the Ethernet type of the encapsulated packet during transmission between the CMU and CSU elements (see Figure 6.2). The type field of the Ethernet header is used to indicate that the packet is a CacheCast packet. We use the Ethernet type of `0xCACA` to identify CacheCast packets. When CSU receives a packet with this Ethernet type, it assumes that it is a CacheCast packet and the original Ethernet type of the encapsulated packet is stored in the `packet_type` field. The CacheCast Ethernet type is not a standard type and it is chosen from among unallocated Ethernet types to resemble the CacheCast name.

Listing 6.1: CacheCast header for the server support and link cache implementation

```c
struct cachecast_header {
  uint32_t INDEX;
  uint32_t payload_id;
  uint16_t payload_size;
  uint16_t packet_type;
}
```

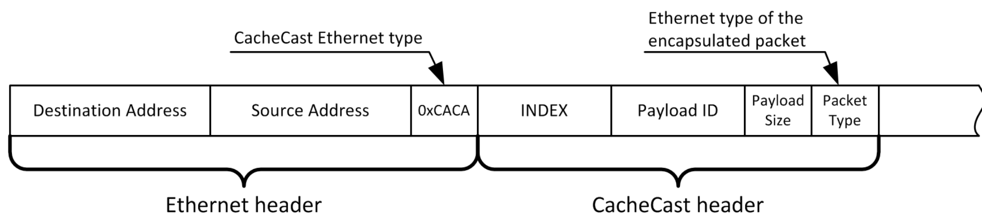


Figure 6.2: Use of the packet type field in the CacheCast header

The CacheCast header is twelve bytes long with the following byte distribution per field: four bytes for the INDEX value, four bytes for the payload ID value, two bytes for the payload size value, and two bytes for the packet type value. While the four-byte INDEX can address a huge table, in our prototype implementation we decided to allocate more bytes than necessary. To illustrate a number of unique IDs that can be encoded on the four-byte `payload_ID` field, we use an example of a source with 1Gbps uplink speed. Assuming that this source transmits only the minimum size Ethernet packets with unique payload IDs, the source uses all possible IDs after almost an hour. Since link caches discard payload IDs after one second (cf. Section 3.4.6), the four-byte field for the

payload ID value is sufficient. The packet payload cannot be larger than the size of an IP packet. Since the IP packet size is stored in a two-byte field in the IP header, the two-byte `payload_size` field is sufficient. The size of the `packet_type` field follows from the Ethernet specification. This structure of the CacheCast header is used in both the server support implementation described in this chapter and in the link cache elements implementation described in the following chapter.

6.3 Server support in Linux

6.3.1 Design

CacheCast imposes on the source the following three tasks: (1) identify and annotate packets that carry the same payloads with a unique ID and the payload size, (2) create the CacheCast header, and (3) minimise the transmission time of packets that carry the same payload. Our initial investigation shows that these tasks cannot be accomplished by current OS. Thus, we have designed an extension to the Linux OS that fulfils them. We chose the Linux OS due to its open architecture and its wide use as a server platform. The Linux extension consists of two elements: a system call and a shell command. The new system call, which we have named `msend` (the abbreviation of "multiple send"), allows an application to send the same data to many destinations. The reason to substitute multiple `send` system calls with one `msend` call is to provide a single entry point to a kernel for the application. Therefore, all send requests are batched in time and the kernel can handle them efficiently.

The design of the `msend` system call can be best understood by analysing the graph depicted in Figure 6.3. Similar to the normal `send` system call, `msend` operates only on connected sockets, thus, an application must first establish connections (1). To transmit data via a set of sockets, an application uses `msend` providing this set and the data as the arguments (2). Then, the kernel sends the data to the sockets using the normal kernel send call (3). However, the resulting packets are intercepted before link transmission, the CacheCast headers are created, and the packets are queued on a per-neighbour basis. When the data is sent to all sockets, the queued packets are released one by one, but only the first packet destined to a given neighbour carries the payload. The remaining packets are truncated and only packet headers are sent (4). As a result a tight packet train per neighbour is created. We use a lock to serialise the packet train transmissions. Thus, it is enough for the neighbour CSU to be able to hold only one payload.

At present, the system design does not include a mechanism for auto-discovery of CacheCast capable neighbours. Therefore, an administrator must enter this information manually. This is achieved using a shell command tool named `cachecast` which provides an interface to the caching mechanisms. The administrator can install the CMU on any link which is connected to the host machine. The caching link is identified by a host device name and a neighbour IP address.

6.3.2 Linux networking subsystem

We have implemented the server support for CacheCast in Linux OS based on kernel 2.6.24.7. It is implemented as a system call and a kernel module responsible for per neighbour CacheCast support. The implementation uses many kernel structures and is integrated into the networking subsystem in many places; therefore, before we describe the server support implementation, in the

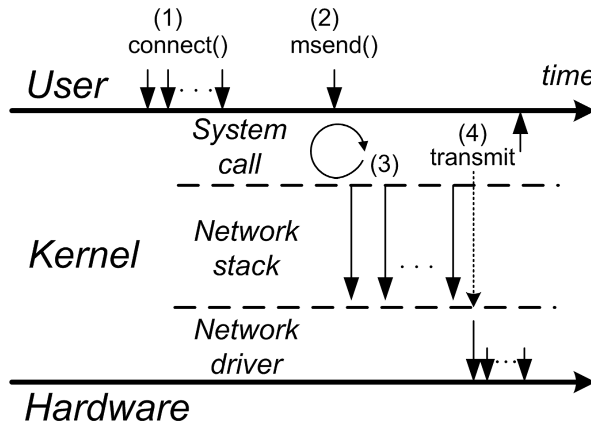


Figure 6.3: The msend system call design

following paragraphs we give a brief overview of the operation of the Linux networking subsystem and the fundamental data structures which are relevant for our implementation.

Sockets

A socket represents a communication end-point operated by the OS. It is used for interprocess communication, mainly for processes located on different machines in a network. Similar to Unix, Linux applies the file abstraction to all devices and system resources; and hence also to sockets. Thus, a socket is identified by a file descriptor, and a process can use the standard `read` and `write` file operations in order to receive and send data across a network. Nonetheless, the file abstraction is not completely realised for sockets. The main part of the socket functionality is provided with the Berkeley sockets application programming interface (API). The API defines a set of operations on sockets which enable an application to create a socket, to manage a socket connection, and to transfer data over a socket. We limit the socket description to the elements which are related to the transmission path.

The sockets constitute a very thin layer in the Linux networking subsystem. The main tasks of the socket layer are: (1) to perform a security check, (2) to provide an interface to the protocol specific functions, and (3) to translate application data to the common message format. A socket is created using the `socket(int domain, int type, int protocol)` system call which allocates a socket object in the Linux kernel and returns a file descriptor associated with this socket. The socket object is described with a generic `socket` structure shown in Listing 6.2. It is primarily used to store general information about the socket (like the socket state, or type) and to enable file operations on a socket. The key entries in this structure are: a pointer to the `sock` structure which contains the network layer representation of a socket and a pointer to the protocol specific set of socket operations stored in the `proto_ops` structure.

Listing 6.2: Socket structure

```
struct socket {
  socket_state          state;
  unsigned long         flags;
```

```
  const struct proto_ops *ops;
  struct fasync_struct   *fasync_list;
  struct file            *file;
  struct sock            *sk;
  wait_queue_head_t       wait;
  short                   type;
};
```

While the socket structure is brief and contains only generic information relevant to all socket types, the sock structure contains a large number of elements which in part are relevant to all sockets and in part are relevant only to a TCP socket due to legacy issues. Since the sock structure is large we show in Listing 6.3 only two elements which are related to our server support implementation.

Listing 6.3: Sock structure

```
struct sock {
  volatile unsigned char  sk_state;
  ...
  struct sk_buff_head     sk_write_queue;
  ...
};
```

The sk_state describes a protocol level socket state which is complementary to the state described in the socket structure. For example, a DCCP socket in a connected state is described with sock->state=SS_CONNECTED and sock->sk->sk_state=DCCP_OPEN[1]. Since the msend system call operates only on connected sockets we inspect this variable to determine the socket state.

The next element sk_write_queue is a queue head where socket buffers are stored temporarily. The queue is mainly used in two cases: to assemble large packets from small data chunks stored in multiple subsequent socket buffers, and to delay the transmission time (e.g. when a congestion control mechanism prohibits the transmission). The msend system call investigates the sk_write_queue after transmitting a packet. If the socket buffer containing the packet is on the queue, this means that the transmission is delayed due to congestion in a network. Since the server support requirement is to transmit the tight packet train, the socket buffer must be removed from the queue in this case.

Send system calls

When a socket is created, an application receives a file descriptor which identifies this socket. The file descriptor is used in the subsequent communication with the socket to send and receive data via the socket, and to control the connection state. In order to transmit data via the socket, the application can use one of the four standard system calls depicted in Figure 6.4 (the text in parenthesis describes the data type which the system call handles). The send, sendto, and write system calls handle a byte string transmission; and the sendmsg system call handles a message transmission. Regardless of the system call, the socket layer translates the application data to a message format and invokes sock_sendmsg function. This function, in turn, invokes the protocol specific sendmsg function like tcp_sendmsg(), udp_sendmsg(), or dccp_sendmsg(); by this passing the message to the transport layer. At the transport layer the application data is moved from the message to a socket buffer for packet assembly. We describe these two formats in the next paragraph.

[1] The Linux convention is to refer to the socket structure using the sock variable.

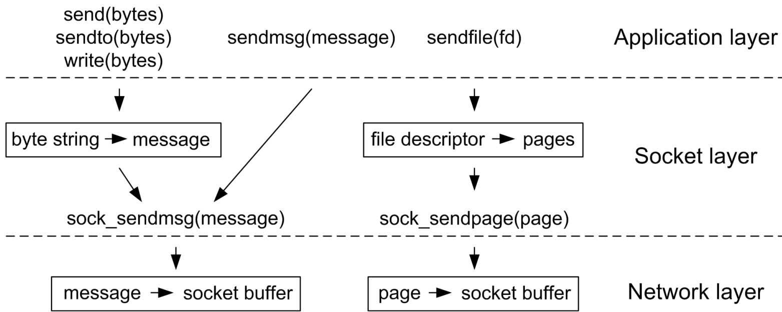


Figure 6.4: Application data representation on different layers

Additional to the aforementioned standard system calls, Linux provides the sendfile() system call for efficient transmission of files (see Figure 6.4). The system call uses a memory page interface to pass data from a file to a socket without expensive copy operations. The file and the socket are provided in the form of file descriptors; the first file descriptor points to a socket in write mode and the second file descriptor points to a memory mapped file in read mode. The memory mapped file is a file which completely resides in the OS virtual memory in structures called pages. The data transfer between the file and the socket is achieved by passing a pointer to a memory page, where the data is stored. At the socket layer, the page transfer is handled by the sock_sendpage function. This function, in turn, invokes a protocol specific sendpage function at the transport layer, like udp_sendpage or tcp_sendpage. Finally, a protocol specific sendpage function creates a socket buffer structure for packet assembly and stores the page pointer in it. During protocol encapsulation, the protocol headers are stored in a buffer which is always attached in front of the data stored in the memory pages.

The msend system call uses both sendmsg and sendpage functions to create a CacheCast packet. The message interface is used to create an empty socket buffer and to pass CacheCast relevant information to the transport layer (specifically a synchronisation queue number described in Section 6.3.3). The memory page interface is used to pass application data, which is copied from user space to preallocated memory pages.

Message structure

The socket layer uses a message as a standard format to pass application data to the transport layer. A message format enables the socket layer to pass the application data along with control information. In Listing 6.4, we provide the message structure as it is described in the Linux kernel. The application data is pointed to by a vector (msg_iov) which describes fragments of data scattered in the memory. If data occupies a continuous block of memory the vector contains only a single pointer to the data and the data length. The control information is attached to a message using the msg_control pointer. It is described by an additional structure cmsghdr which specifies a protocol that created this control message, the type of the control message, and the content. In our server support implementation we use the control message to pass CacheCast specific information to the transport protocol layer.

Additional to the control message option, the message handling can be controlled with the

msg_flags variable. It enables an application to transmit, for example, a non-blocking message (MSG_DONTWAIT), or to order a socket to wait for more messages before transmission (MSG_WAIT). We use these two options to assemble a CacheCast packet in the transport layer.

Listing 6.4: Message structure

```
struct msghdr {
  void            *msg_name;      // Socket name
  int              msg_namelen;   // Length of name
  struct iovec    *msg_iov;       // Data blocks
  __kernel_size_t  msg_iovlen;    // Number of blocks
  void            *msg_control;
            // Per protocol magic (e.g. BSD file descriptor passing)
  __kernel_size_t  msg_controllen; // Length of control message list
  unsigned         msg_flags;
};

struct cmsghdr {
  __kernel_size_t  cmsg_len;    /* data byte count, including hdr */
  int              cmsg_level;  /* originating protocol */
  int              cmsg_type;   /* protocol-specific type */
};
```

Socket buffers

The sk_buff structure is a basic structure for packet assembly in the Linux networking subsystem. It provides a buffer for packet data and auxiliary information necessary for packet processing. In Listing 6.5 we highlight the most important information elements of the sk_buff structure. The first two pointers in the socket buffer structure enable networking modules to queue the socket buffer on different queues in the system (e.g. socket buffers are queued on the aforementioned sk_write_queue when a congestion control mechanism prohibits transmission). All socket buffers which are created by one of the send system calls belong to a socket connection which is referenced by the sk pointer.

Listing 6.5: Socket buffer structure

```
struct sk_buff {
  struct sk_buff     *next;
  struct sk_buff     *prev;

  struct sock        *sk;
  struct dst_entry   *dst;

  unsigned int        len;
  unsigned int        data_len;

  __be16              protocol;

  unsigned char      *head:
  unsigned char      *data;
  unsigned char      *tail;
  unsigned char      *end;
```

```
    unsigned int            cc_queue;
    ...
};
```

When the first hop of a packet carried in a socket buffer is determined, the dst_entry structure describing this first hop is referenced by the dst pointer. The dst_entry structure has a pointer to the neighbour structure which handles link layer encapsulation and provides a method for link layer packet transmission. We provide more details about the neighbour structure in the next paragraph. Considering the IP stack, the dst entry in the sk_buff structure is set by the IP routing subsystem before the IP encapsulation. In the last step of the IP layer processing, the neighbour specific transmission function is invoked (pointed to by the dst->neighbour->output function pointer) and the socket buffer is passed to the link layer. The server support modifies this execution order. Before the neighbour specific transmission function is invoked, we check whether the neighbour is a CacheCast capable neighbour and a socket buffer carries a CacheCast packet. If these two conditions are met, we invoke the cc_queue_skb() function of the server support which enqueues this socket buffer.

The protocol variable stored in the sk_buff structure describes the packet type carried in the socket buffer. It is primarily used in the Ethernet encapsulation process. For example, before the IP layer passes a socket buffer to the link layer, it sets the protocol value to the IP Ethernet type 0x0800. The server support modifies the protocol variable for all CacheCast packets to hold the value of the CacheCast Ethernet type which is 0xCACA.

The socket buffer provides two types of storage space for a packet: a linear buffer and an array of pointers to memory pages. The total size of a packet stored in these two types of storage space is given in the len variable. The size of data stored only in the memory pages is given in the data_len variable. Since the msend system call uses the memory page interface (i.e. the sendpage() function) to pass application data to the transport layer, the data_len variable gives the application data size. Protocol headers created in the encapsulation process are stored in a linear buffer part. We use the data_len variable to compute the payload size value stored in the CacheCast header.

Figure 6.5 depicts data layout in a socket buffer. The sk_buff structure describes the linear buffer with four pointers: the head pointer points to the beginning of the buffer, the data pointer points to the beginning of valid data in the buffer, the tail pointer points to the end of the valid data in the buffer, and the end pointer points to the end of buffer space. The space between addresses pointed to by the head and data pointers is called headroom and is reserved for headers which are added in the process of encapsulation. Since application data in CacheCast packets is provided with pointers to memory pages, the linear buffer carries only protocol headers. At the end of the linear buffer the skb_shared_info structure is located. It contains an array of pointers to memory pages along with offsets and sizes of data stored in the pages.

When the execution thread of the socket send operation reaches the network layer, application data is moved to a socket buffer. As it is depicted in Figure 6.4, if the application data is carried in a message, the message data is copied to the linear buffer and if the application data is provided with a pointer to a memory page, the pointer is stored in the skb_shared_info structure. The final packet is the sum of the data stored in the buffer space and in the memory pages.

In order to identify a socket buffer which carries a CacheCast packet, we have added the cc_queue element in the sk_buff structure. The cc_queue value has a double meaning. The

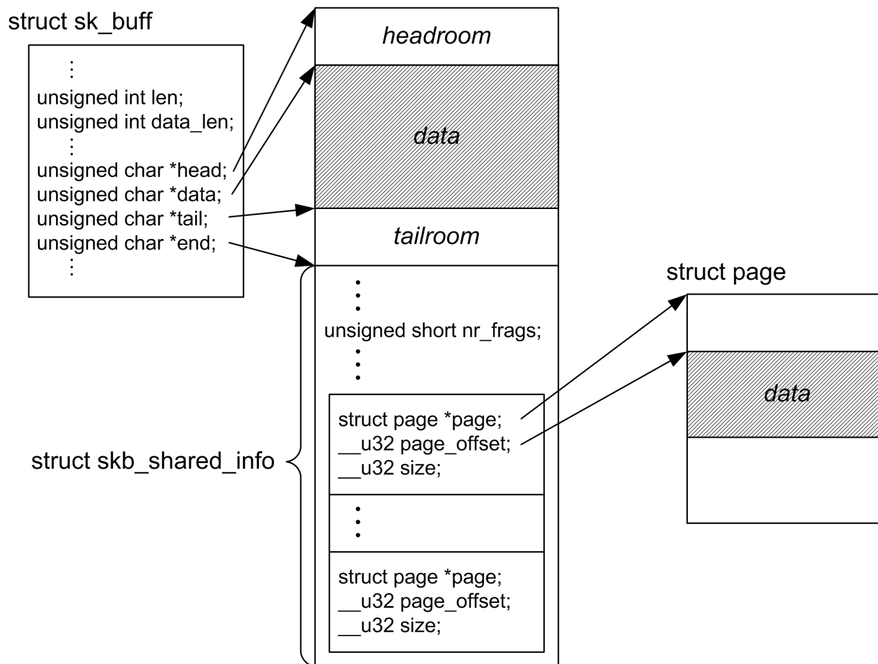


Figure 6.5: Socket buffer - data layout

value set to zero implies that the socket buffer carries a standard packet. However, a value greater than zero implies that the socket buffer carries a CacheCast packet and the cc_queue value is interpreted as a synchronisation queue number (which we describe in the next section). When a socket buffer is allocated the sk_buff structure is cleared including the cc_queue element. Thus, all socket buffers carry by default standard packets.

Neighbouring subsystem

Neighbours of a host are network nodes which are within one hop range from the host (measuring the hop count on the network layer). The neighbouring nodes are direct recipients of packets created by a host, and thus are essential for communication. If a host does not have any neighbours it is considered to be disconnected.

Linux neighbour management is performed by the neighbouring subsystem which has two tasks: neighbour discovery and the network layer to link layer address translation for neighbouring nodes. The neighbour discovery is performed by one of the discovery protocols (such as Address Resolution Protocol (ARP) [50] in IPv4 networks or Neighbour Discovery Protocol (NDP) [51] in IPv6 networks) which operate within the neighbouring subsystem framework. A key structure in the neighbouring subsystem is the neighbour structure which provides information about a state of a neighbour, network and link layer addresses of the neighbour, a method to transmit a packet to the neighbour, and more. In Listing 6.6, we show only the key elements of the structure which are relevant for the server support implementation.

Listing 6.6: Neighbour structure

```
struct neighbour {
  struct net_device    *dev;
   // Pointer to a network device through which the neighbour is reachable
  __u8                 nud_state;
   // state of the NUD state machine
  unsigned char        ha[ALIGN(MAX_ADDR_LEN, sizeof(unsigned long))];
   // Hardware address of the neighbour
  int                  (*output)(struct sk_buff *skb);
   // Transmit method to the neighbour
  struct sk_buff_head  arp_queue;
   // Queue to store temporarily socket buffer during solicitation request
  struct cc_neigh_info *cachecast_ptr;
   // Pointer to CacheCast specific neighbour information
  ...
};
```

A neighbour is always associated with a single network device through which it is reachable. If a neighbour is reachable via multiple devices which are present at a host, each device - neighbour pair is described with a distinct `neighbour` structure. A pointer to the network device is stored in the `dev` pointer.

Linux implements a neighbour unreachability detection (NUD) state machine which keeps neighbour information up to date. The NUD state machine determines when a neighbour is considered to be connected, when to send solicit requests to a neighbour, or when a neighbour is considered to be unreachable. The machine state is stored in the `nud_state` variable in the `neighbour` structure. The server support queries the NUD state using an auxiliary function `neigh_is_valid` to find whether the network to link layer address mapping is valid for the neighbour. If the mapping is invalid during packet transmission, a socket buffer carrying the packet is put on the `arp_queue` queue and a solicit request (e.g. ARP who-is packet) is sent to obtain the correct address mapping for this neighbour. CacheCast packets cannot be stored temporarily, while waiting for the solicit response, since this would disturb CacheCast packet transmission. Therefore, if the network to link layer address mapping is invalid for a neighbour, socket buffers carrying CacheCast packets towards this neighbour are sent in a standard way, i.e. the packets are not CacheCast encapsulated and do not form a packet train. Please notice, that the network to link layer address mapping for a neighbour is only invalidated after a considerable time period when there is no communication with the neighbour. Thus, when a host transmits a media stream this mapping is valid.

We install in the `neighbour` structure a pointer to the CacheCast neighbour specific structure (the `cachecast_ptr` pointer). If the pointer is set, the neighbour associated with this `neighbour` structure is regarded as capable of receiving CacheCast packets. If the pointer is cleared, the neighbour does not recognise CacheCast packets and the server support will send standard packets to this neighbour. We describe the function of the `cc_neigh_info` structure in the following section.

6.3.3 Implementation

The Linux server support implementation fully complies with the design introduced in Section 6.3.1. It is capable of creating a per-neighbour packet train. Packet trains are serialized on a per-neighbour basis. For each packet in a packet train we create the CacheCast header with the same

payload size and the same ID. The payload size corresponds to the data size provided by the application. Since we serialise packet trains on a per-neighbour basis, it is only required that the neighbouring CSU is able to hold one payload. Thus, we set the INDEX field to zero for all packets. The ID is a subsequent number of a packet train transmitted by this host. The sequential payload ID makes the server content vulnerable to malicious attacks; however, the server support is a prototype implementation for evaluation purposes and does not incorporate all security countermeasures, which are discussed later in Chapter 3.

We assume that not all neighbours in a network are CacheCast capable. Therefore, a user must define the neighbours that support CacheCast. This knowledge is maintained internally by the OS, and packet trains are created only to those neighbours that support CacheCast. Those neighbours that do not support CacheCast receive regular packets. Until the first neighbour is defined, the msend system call will only send normal packets. This is very similar to the functionality of the system calls provided in [48, 49].

The server support is implemented as a system call and a Linux module which supports the system call. Additionally, we make small changes in the Linux networking subsystem to link it with the server support. In the following paragraphs, we describe the complete server support implementation beginning with a description of the msend system call which gives an overview of the implementation.

msend **system call**

We separate the CacheCast concerns from applications with the msend system call which transmits the same data chunk. The system call API is based on the select system call API and is as follows:

```
int msend(fd_set *fds_write,
          fd_set *fds_written,
          char *buf, int len)
```

The sockets are provided in the form of file descriptors and we use the standard fd_set structure to pass a set of file descriptors between the user space and the kernel space.

With the fds_write variable an application provides a set of sockets to be written. Currently the implementation can handle connections based on the UDP and DCCP protocols. The sockets that were successfully written can be found by inspecting the fds_written variable. If a socket provided in the fds_write set was written to, a bit corresponding to the file descriptor of this socket is set in the fds_written set. Otherwise the bit is cleared. The buf and len variables describe the data chunk to be sent. The system call returns the total number of successfully written connections.

Figure 6.6 depicts the execution flow of the msend system call. The underlined functions cc_alloc_queues(), cc_release_queues(), and cc_queue_skb() are part of the server support and are described in detail in the following paragraphs. In the first step, the system call allocates memory pages for application data and copies the data into the allocated pages. Considering a standard Linux configuration for a desktop machine, a single page has 4KB which is enough to accommodate a packet of a standard MTU size (1.5KB). The system call does not fragment data larger than the MTU size. Moreover, we disable the IP fragmentation for CacheCast packets, since this would disturb the packet train structure. Therefore, an application is responsible for fragmenting data into chunks which fit in the network MTU together with the transport, network, and CacheCast headers.

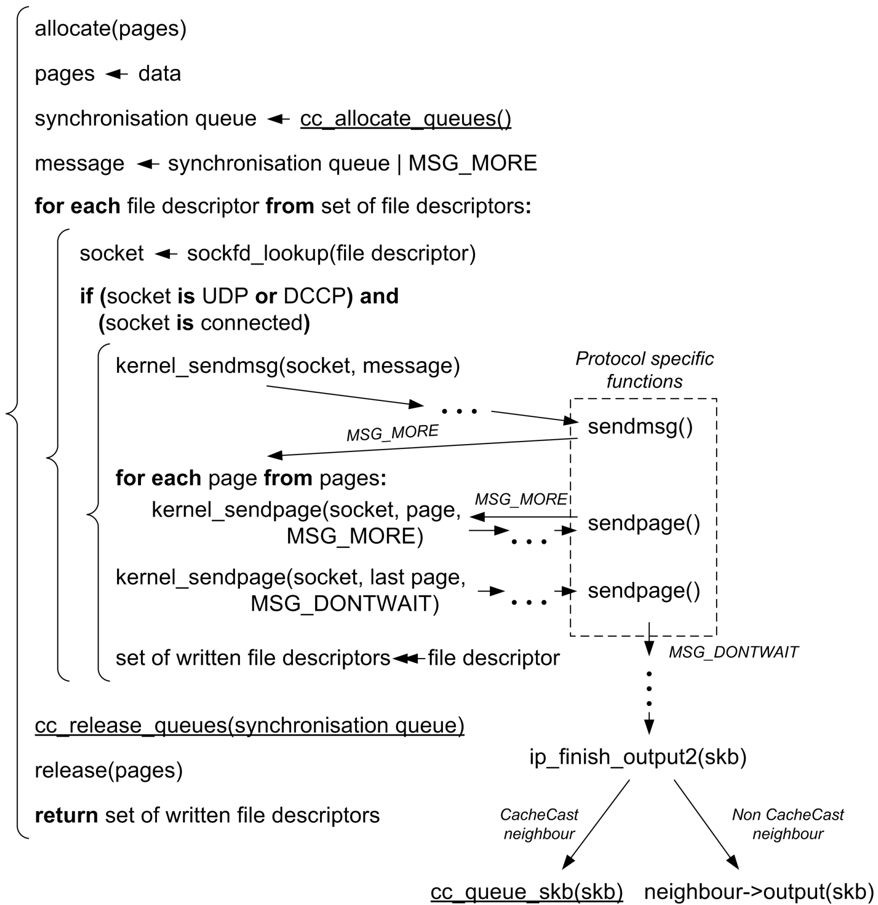


Figure 6.6: The msend system call execution flow

In the second step, the system call using the cc_alloc_queues() function allocates synchronisation queues for CacheCast capable neighbours. Each CacheCast capable neighbour has its own synchronisation queue which is used to enqueue socket buffers carrying CacheCast packets before the link layer encapsulation. This is performed in the ip_finish_output2() function with the call to the cc_queue_skb() function. The synchronisation queue has a double purpose. Firstly, socket buffers queued on a synchronisation queue form a packet train. Thus, when the system call has sent application data to all sockets, the synchronisation queue contains all socket buffers destined to a given neighbour. The socket buffers are de-queued using the cc_release_queues() function which builds packet trains. Secondly, a synchronisation queue synchronises packet transmission before the link layer. At this point in the network stack, socket buffers carry almost complete packets. Thus, when the socket buffers are de-queued, packets are only Ethernet encapsulated and transmitted. This low processing overhead between consecutive transmissions of packets from a packet train reduces transmission time of the complete packet train.

The msend system call can be executed by multiple processes at the same time. This requires a distinct synchronisation queue for each process per CacheCast capable neighbour. We solve the issue by preallocating a set of synchronisation queues per neighbour. The msend system call allocates a single synchronisation queue per neighbour from the set using the cc_alloc_queues() function which returns the queue number. In order to enqueue a socket buffer created by this msend system call on the allocated queue, the socket buffer must be annotated with the synchronisation queue number. The information is stored in the cc_queue entry in the sk_buff structure (see Listing 6.5). Please note that the msend system call operates at the socket layer and has no access to the socket buffer structure (cf. Figure 6.4); therefore, to set the cc_queue value in the sk_buff structure we use a message interface. The system call creates an empty message with control information carrying the synchronisation queue number.

In the third step, the system call prepares all packets which are waiting in the allocated synchronisation queues for transmission. For each file descriptor provided by the application, the system call translates it into the corresponding socket using the sockfd_lookup() function. If this is a UDP socket or a DCCP socket and it is in a connected state, the system call sends the control message to build an empty socket buffer annotated with the synchronisation queue number[2]. Next, it sends application data stored in the pages using the sendpage() interface. Since the DCCP implementation in the Linux kernel 2.6.24.7 does not support the sendpage() interface, we have implemented it with the minimum functionality necessary to support the msend system call. The sendmsg() and sendpage() functions build a socket buffer which has a small linear buffer space for packet headers and a set of pointers to memory pages where the application data is stored. The last memory page is sent with the MSG_DONTWAIT flag set which triggers the send operation. The socket buffer is intercepted at the output of the network layer. If the destination neighbour is CacheCast capable and its hardware address is valid (cf. discussion in the neighbouring subsystem paragraph), the socket buffer is enqueued with the cc_queue_skb() function on the synchronisation queue identified with the synchronisation queue number.

In the last step, the system call transmits all packets enqueued on the synchronisation queues in the form of a packet train. This is achieved with an auxiliary function cc_release_queues() which takes as an argument the synchronisation queue number. Additionally, the function de-allocates the synchronisation queues.

Kernel module

The CacheCast kernel module is responsible for handling the state related to operation of the server support. Specifically, the module maintains the knowledge of CacheCast capable neighbours and manages synchronisation queues. The key information elements are stored in the cachecast_module and cc_neigh_info structures described in Listing 6.7. The cachecast_module structure is allocated and initialised during module initialisation stage. We describe this structure in the context of functionalities provided by the module.

Listing 6.7: CacheCast kernel module structures

```
// structure describing CacheCast module state
struct cachecast_module {
```

[2]We have modified the UDP and DCCP implementation of the sendmsg() function to initialise the cc_queue element in the sk_buff structure according to the CacheCast control message.

```
  struct list_head   cc_neigh_list;
    // List head of CacheCast capable neighbours
  uint32_t           queue_map[SYNCH_QUEUE_MAP_SIZE];
  spinlock_t         queue_map_lock;
    // Allocation map of synchronisation queues
  uint32_t           queue_payload_id[SYNCH_QUEUE_MAX];
  atomic_t           cur_payload_id;
    // Per synchronisation queue payload ID
  struct mutex       queue_xmit_mutex;
    // Global lock for packet train transmission
};

// Neighbour specific information
struct cc_neigh_info {
  struct list_head     cc_neigh_list;
    // Linked list hook
  struct sk_buff_head  queue_skb_head[SYNCH_QUEUE_MAX];
    // A set of synchronisation queues
  unsigned int         tx_cc_packets;
  unsigned int         tx_cc_trains;
    // Statistics
  struct neighbour     *neighbour;
    // Back reference to the neighbour owning this element
};
```

The first element in the structure is a list head of CacheCast capable neighbours. As we have described in the neighbouring subsystem paragraph, a neighbour is regarded as capable of receiving CacheCast packets, if the cc_neigh_info pointer in the neighbour structure is set, i.e., it points to the cc_neigh_info element. The list of CacheCast capable neighbours consists of the cc_neigh_info elements which reference back the neighbours; this is depicted in Figure 6.7. When a user defines a neighbour as capable of receiving CacheCast packets the module allocates a new cc_neigh_info element, sets the cc_neigh_info pointer in the neighbour structure to the new element, and adds the element to the list. A reverse operation is performed when a user defines a neighbour as incapable of receiving CacheCast packets. This functionality is presented to a user with the following system command:

```
cachecast [add|rm] dev_name neigh_ip_addr
```

The keywords "add" and "rm" indicate whether the cc_neigh_info structure should be installed or removed from the neighbour structure. The network device name and the neighbour IP address uniquely identify a single neighbour in the Linux neighbouring subsystem.

The cc_neigh_info element is not only used to mark CacheCast capable neighbours but also to provide a set of synchronisation queues (the queue_skb_head array) where socket buffers are stored temporarily during the msend system call invocation. The queue_map element in the cachecast_module structure keeps information about allocation of these synchronisation queues. The msend system call allocates a single synchronisation queue per neighbour using the cc_alloc_queues() function. The function performs a lookup for the first clear bit in the map, indicating an unallocated queue, and sets the bit. The bit position corresponds to the synchronisation queue number. The function also creates a payload ID for all packets stored in the allocated queues. The payload

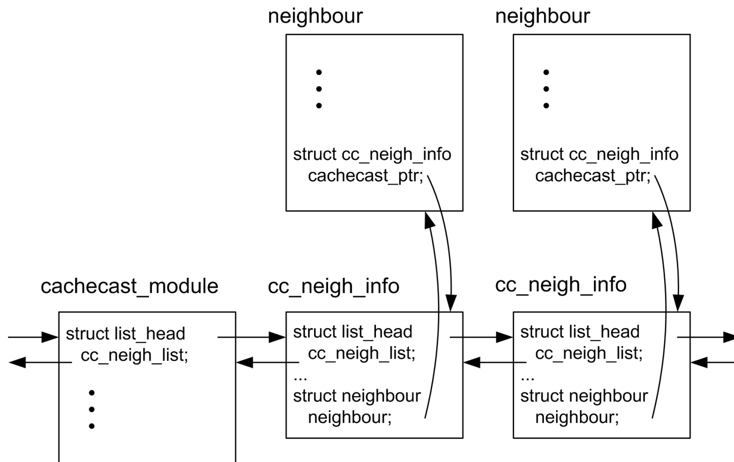


Figure 6.7: The CacheCast kernel module and the neighbouring subsystem

ID is generated using the cur_payload_id counter and is stored in the queue_payload_id array at the synchronisation queue number position. Since the system call operates in the multi-thread environment, the allocation process must be protected with the queue_map_lock lock. The payload ID counter is of the atomic type and is incremented using atomic operations.

Additional to the cc_alloc_queues() function, the kernel module exports the cc_queue_skb() and cc_release_queues() functions which were already mentioned in the context of msend system call. The cc_queue_skb() function is called by the ip_finish_output2() function to store a socket buffer on a synchronisation queue. However, before the socket buffer is enqueued, the function builds the CacheCast header. At this point, the socket buffer carries the IP encapsulated packet; thus, the CacheCast header is simply appended in front of the IP header. The CacheCast header fields are filled in the following way: The INDEX value in the CacheCast header is set to zero, and the payload ID value is set to the value found in the queue_payload_id array at the synchronisation queue number position. The payload size value requires special consideration. If the header size is at least of the minimum packet size, the payload size is set to the application data size. However, if the header size is less than the minimum packet size, the payload size must be reduced to ensure the minimum packet size for packets without payload. Otherwise, the network driver extends the truncated packets to the minimum packet size with random data which will cause corruption of CacheCast packets during payload restore at the CSU. After the CacheCast encapsulation, the socket buffer is enqueued on a synchronisation queue in the queue_skb_head array in the cc_neigh_info element. The queue number and the neighbour are provided in the socket buffer structure.

The cc_release_queues() creates a packet train per neighbour from the socket buffer previously enqueued in the synchronisation queues. The function takes as an argument the synchronisation queue number and for each cc_neigh_info element from the CacheCast capable neighbour list (cc_neigh_list) performs the following actions. It de-queues the first socket buffer enqueued in the queue_skb_head array at the synchronisation queue number and transmits it using the neighbour->output() function. For the remaining socket buffers, the function first de-queues the

socket buffer, next truncates it according to the payload size, clears the payload size field in the CacheCast header, and finally transmits it. The complete operation of the packet train transmissions is protected with the `queue_xmit_mutex` lock. Therefore, concurrent processes will not disturb packet train structure during transmission. When all socket buffers have been sent to the link layer, the `cc_release_queues()` function clears a bit in the `queue_map` allocation map at the synchronisation queue number position to indicate that the queues are free.

6.4 Micro evaluation

In this section we evaluate the described `msend` system call implementation. We assess the performance of the system call execution for a wide range of input parameters and relate it to the standard `send` system call. We also establish bottlenecks of packet transmission operation. The measurements are conducted on a 2.4GHz Intel machine with 512MB 266MHz SDRAM and a 2.5GHz AMD Athlon 64bit duo-core machine with 2GB 667MHz SDRAM. On both machines we run the Ubuntu Server 9.04 with our modified Linux kernel 2.6.24.7. The machines have an Intel 82541PI Gigabit Ethernet network card which is capable of transmitting Jumbo frames of up to 16110B.

The server support for CacheCast consists of two elements: the `msend` system call and a process that creates the packet trains. Since packet trains are formed only for those destinations that are defined with the `cachecast` shell command, it is possible to measure the performance of the raw system call and also the burden incurred due to packet train creation. To understand the results of the measurements we compare them with the performance of the `send` system call under the equivalent workload. Thus, we perform measurements in the following three configurations:

send_loop: Using the `send` system call in a loop we transmit the data block to a group of hosts.

msend_no_CMU: Using the `msend` system call we transmit the data block to a group of hosts that do not support CacheCast.

msend_CMU: Using the `msend` system call we transmit the data block to a CacheCast enabled group of hosts located behind a caching link.

6.4.1 Evaluation methodology

The `msend` system call performance depends on the destination group size and the data block size. Thus, we perform a number of experiments varying these two factors. The costs of a system call are expressed as the per-packet processing time, i.e. the time to build and transmit a single packet to a single destination. To obtain this time we measure the time of the `msend` system call invocation or, in the case of **send_loop** configuration, the execution time of the system call loop. If the destination group size is low, we invoke the system call/execute the loop multiple times, so that the total amount of transmitted packets is at least 100 and we measure the total time. This is due to two reasons: (1) the measurement precision is insufficient to capture the accurate value of a single system call invocation; (2) the processor cache affects the measurement results severely when invoking a system call only once for a small group of destinations. To obtain the per-packet processing cost the measured time is divided by the total number of transmitted packets. We ensure

that the system call results in packet transmission to a given number of destinations by monitoring the resulting network traffic. For each of the three configurations we perform the measurements ten times and we report the average value.

6.4.2 Costs wrt. group size

The first experiment is performed only on the Intel machine and its purpose is to give a general idea of the transmission costs wrt. the group size. We transmit a data block to a growing number of destinations (1-1000) using UDP. We use data blocks of four different sizes: 500B, 1500B, 9000B, and 16000B. The smaller two block sizes correspond to the packet sizes that are currently used in the Internet to transport data. The resulting UDP packet for data block size of 1500B can be greater than the standard Ethernet MTU size; however, no fragmentation occurs since our network card MTU is set to 16110B. The larger two data blocks result in Ethernet Jumbo frames. These are used to show potential benefits of the system call in networks with large MTUs.

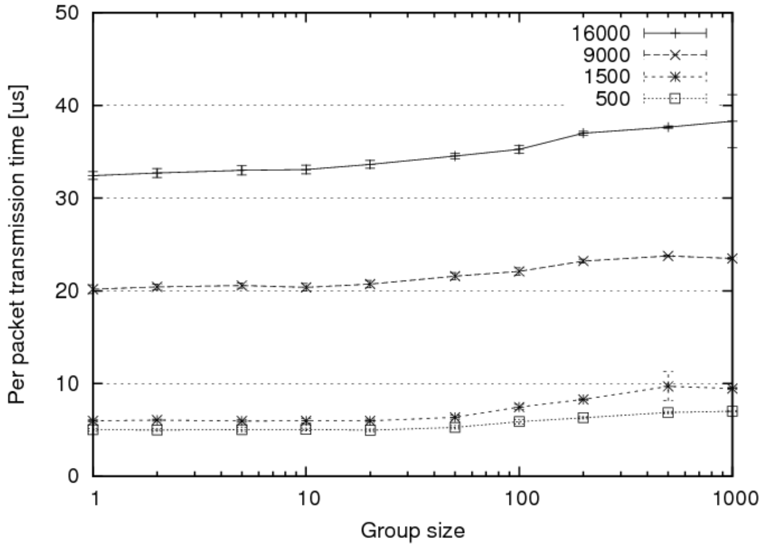
The experiment results depicted in Figure 6.8 show the time required to transmit a single packet. In the case of the **send_loop** configuration (Figure 6.8a), the per-packet cost does not change significantly with the group size which is obvious since we invoke the `send` system call to transmit each packet. Surprisingly, in the case of the **msend_no_CMU** configuration (Figure 6.8b) the per-packet cost decreases very quickly for all payload sizes with increases in group size and already with the destination group size at above 50 it does not depend on the payload size. The per-packet cost is almost inversely proportional to the group size, however, it does not decrease below $5\mu s$. It appears that the `msend` system call costs related to the page allocation and data copying from user space to kernel space are quickly amortised with the growing destination group size.

To find the cost of the CMU operation we compute the difference between the measurements obtained in the **msend_CMU** and **msend_no_CMU** configurations. Regardless of the packet size for all group sizes the difference stays approximately in the range of $-1\mu s$ to $1\mu s$ with the confidence interval of $2\mu s$. It appears that the CMU operation has insignificant impact on the `msend` system call performance. Thus, even though we have conducted further experiments with the **msend_CMU** configuration we do not present them in the analysis, since these are very similar to the **msend_no_CMU** configuration.

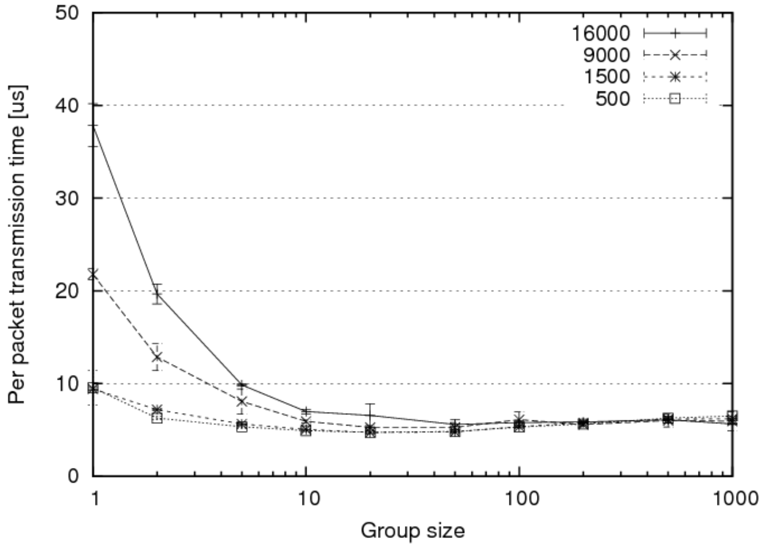
6.4.3 Costs wrt. payload size

In the second experiment, we focus on the relationship between the payload size and the per-packet transmission time. We perform a similar experiment to the first one, however we increase the data block size from 100B up to 16000B (with a step of 2000B) and transmit it to a group of destinations in the following four sizes: 1, 10, 100, and 1000. To obtain sufficient confidence in the results and in order to investigate the impact of the memory speed, we perform the experiment on both our Intel and the AMD machines.

In Figure 6.9 we show a sample graph representing the relationship of the system call cost and the payload size for the group size of 100, as measured on the Intel machine. In studying the `send` system call, we observe a distinct linear increase in the per-packet transmission time with the increase in packet size. However, in the case of the `msend` system call the per-packet transmission time is constant in the range of the probed packet sizes. We can send much more data when using



(a) The send_loop results



(b) The msend_no_CMU results

Figure 6.8: Per packet send time as a function of the group size for four different payload sizes

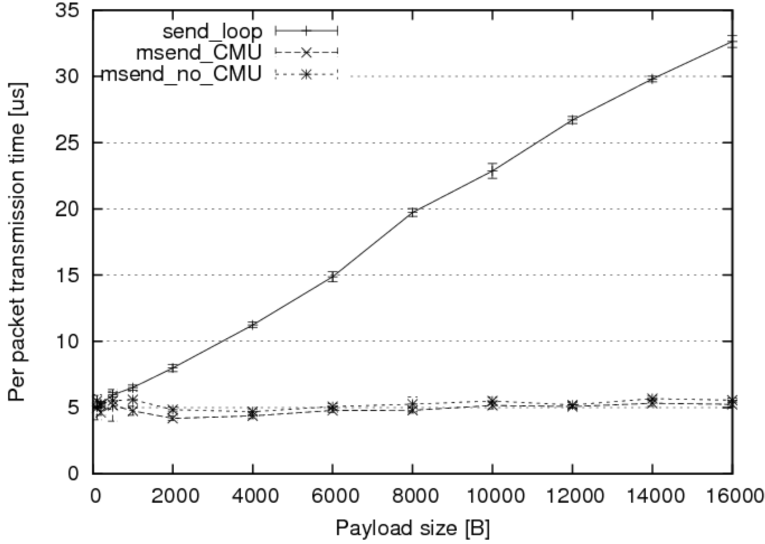


Figure 6.9: Per packet send time as a function of the payload size for group size of 100 (the Intel machine)

Table 6.1: Per packet system call cost - the linear function fit results for the Intel machine

Group size	**send_loop**	**msend_no_CMU**
1	$a = 1.68e{-3} \pm 1.9\%$ $b = 3.84 \pm 7.0\%$	$a = 1.74e{-3} \pm 2.7\%$ $b = 6.80 \pm 5.7\%$
10	$a = 1.67e{-3} \pm 1.9\%$ $b = 4.02 \pm 6.5\%$	$a = 1.21e{-4} \pm 15.3\%$ $b = 4.95 \pm 3.0\%$
100	$a = 1.78e{-3} \pm 1.4\%$ $b = 4.76 \pm 4.4\%$	$a = 1.37e{-5} \pm 120.5\%$ $b = 5.22 \pm 2.6\%$
1000	$a = 1.93e{-3} \pm 1.5\%$ $b = 5.33 \pm 4.6\%$	$a = -2.22e{-5} \pm 99.3\%$ $b = 5.79 \pm 3.1\%$

larger packet sizes at the same transmission cost. This prompts a question: *what is the cost of sending a single byte wrt. the transmitted packet size and the group size.* The cost can be split into two parts:

(a) Per-byte cost - this cost is related to copying data from user space to kernel space, payload checksum calculation, etc.

(b) Per-packet cost - this cost is related to user-kernel mode switch, message preparation, header building, socket state update, in kernel packet route evaluation, etc.

To obtain these values we fit a linear function to the data set of the second experiment. The factors a and b of the linear function $f(x) = ax + b$ represent respectively per-byte cost (a) and per-packet cost (b).

The results of the linear function fit for the Intel machine are shown in Table 6.1 and for the AMD machine in Table 6.2. The a and b factors are given together with the accuracy of the fit

Table 6.2: Per packet system call cost - the linear function fit results for the AMD machine

Group size	**send_loop**	**msend_no_CMU**
1	$a = 6.13e{-4} \pm 2.1\%$ $b = 3.78 \pm 2.8\%$	$a = 6.24e{-4} \pm 4.0\%$ $b = 5.85 \pm 3.6\%$
10	$a = 6.44e{-4} \pm 2.8\%$ $b = 3.70 \pm 4.0\%$	$a = 6.42e{-5} \pm 28.5\%$ $b = 4.36 \pm 3.4\%$
100	$a = 6.80e{-4} \pm 1.0\%$ $b = 3.92 \pm 1.5\%$	$a = 1.98e{-5} \pm 65.3\%$ $b = 4.27 \pm 2.5\%$
1000	$a = 6.46e{-4} \pm 1.6\%$ $b = 4.47 \pm 1.9\%$	$a = -1.15e{-5} \pm 117\%$ $b = 4.75 \pm 2.3\%$

operation. The accuracy of the a factor is especially low in the **msend_no_CMU** configuration for large groups. This is due to high sensitivity of the a factor when fitting to near constant linear function.

Considering transmission to one destination, the per-byte cost (a) of the `msend` system call and the `send` system call are similar on both the Intel and AMD machines. However, with the increase in group size the per-byte cost of the `msend` system call decreases very quickly, as we reported in the previous experiment. For the group size of 10 destinations, the cost is 14 times smaller on the Intel machine and 10 times smaller on the AMD machine. We even obtain negative values for the per-byte cost when the group size is 1000 which is an artifact of the fit operation. This indicates that the per-byte cost of the `msend` system call is negligible when the group size is large. It implies that when a server streams data to a large number of destinations using the `msend` system call its load is inversely proportional to the size of packets that compose the stream. For example, a radio server which serves hundreds of clients transmitting an audio block in a single packet could halve its load by transmitting two audio blocks per packet. We demonstrate this later in Section 6.5 where we show how a live streaming server can handle almost eight times more receivers when using eight times larger packets.

The per-packet cost (b) of the `msend` system call is higher than the regular `send` system call especially when sending only to one destination. It is 1.77 and 1.54 times higher on the Intel and AMD machines respectively. However, this cost ratio decreases with the growing destination group size and for the group size of 1000 it is insignificant (i.e. 1.08 on the Intel machine and 1.06 on the AMD machine). The reduction of the per-packet cost is due to the fact that the `msend` system call performs only one user-kernel mode switch and prepares a message only once for an arbitrary large group of destinations, while the `send` system call performs these tasks for each transmitted packet.

6.4.4 Per-byte and per-packet cost contribution to the total cost

Depending on the payload size the total packet transmission cost is dominated either by the per-byte cost (a) or by the per-packet cost (b). We compute the payload size s_{eq} for which these costs are in equilibrium. If the payload size is smaller, than the s_{eq} size then the total cost is dominated by the per-packet cost. If the payload size is greater than the s_{eq} size, then the total cost is dominated by the per-byte cost.

Considering the `send` system call, regardless of the group size, the costs equilibrium is achieved

with the payload size s_{eq} of approximately 2300B for the Intel machine, and 6100B for the AMD machine when using UDP. We expect that for more advanced protocols than UDP (like DCCP) the per-packet cost increases and, thus the costs equilibrium is achieved with the larger payload size s_{eq}. The msend system call for a single destination achieves the costs equilibrium with the payload size s_{eq} which is larger than in the case of the send system call, i.e. approximately 3900B for the Intel machine, and 9300B for the AMD machine. This is mainly due to higher per-packet cost. However, considering larger group sizes, where the per-packet costs decrease, the payload size s_{eq} increases further. For example with the group size of 10, the costs equilibrium is achieved with the payload size s_{eq} of approximately 41kB for the Intel machine and 68kB for the AMD machine. Obviously, the size s_{eq} is an approximation based on the assumption that the linear dependency between the packet size and the transmission cost holds above the measured payload sizes, however, it gives a good indication of how dominant the per-packet cost is.

In all cases, the payload size s_{eq} (where the costs are in equilibrium) is always greater than the standard 1500B MTU size in the Internet. Thus, the per-packet cost is the dominant cost of packet transmission in the Internet and the load on a modern server related to packet transmission is mainly dependent on the number of transmitted packets rather than the amount of transmitted data. This means that a substantial amount of server CPU cycles can be saved by using larger packets.

6.4.5 Memory speed impact

The per-byte cost (a) measured on the AMD machine is approximately 2.7 times smaller than on the Intel machine for the send system call and for the msend system call when sending to few destinations. This cost reduction can be directly related to the difference in the memory speed between the AMD and Intel machines, since the AMD machine memory bus runs at 2.5 times higher frequency.

The per-packet costs (b) are not reduced significantly by the increase in the memory speed. We observe that the cost decreases only by 1 to 18% on the AMD machine when compared to the Intel machine. Since the per-packet cost dominates the total cost of packet transmission in the Internet, increasing the memory speed by a given factor will not yield reductions in the packet transmission costs of the same order.

6.4.6 Cost of user-kernel mode switch

Similar to the msend system call, the sendgroup system call [49] performs only one user-kernel mode switch to transmit data to a group of destinations while the equivalent loop of the send system call requires per destination one mode switch. In [49] the authors recognise that this reduction in the number of user-kernel mode switches is “a good strategy for improving performance”.

Considering the msend system call we also observe the performance improvement. The per-packet cost (b), which captures the cost of a user-kernel mode switch, decreases by 14 to 27% when transmitting to more than one destination. For instance, in the case of the Intel machine the cost is initially $6.80\mu s$ and falls to $4.95\mu s$ with a group size of ten destinations. For larger destination group sizes it increases slightly. Interestingly, when compared to the per-packet cost of the send system call the cost of msend is greater even with a group size of 1000 destinations. The main advantage of using the msend system call comes from the reduction in the per-byte costs (see

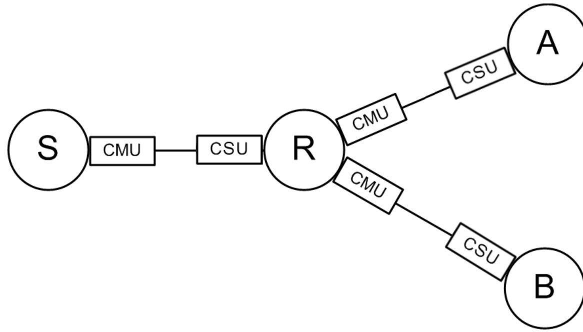


Figure 6.10: Real system performance testbed.

Figure 6.9) but not from a reduction in the number of user-kernel mode switches. This insight is different from results gained in [49].

6.5 Testbed Evaluation

To show the benefits a real application can obtain when using CacheCast we built a testbed. This is depicted in Figure 6.10. It consists of four machines: two Intel 2.4GHz and two AMD duo-core 2.5GHz. We use the Intel machines for the server (S) and the router (R). The AMD machines (A, B) host the clients. We install caches on links S-R, R-A, and R-B and we limit the S-R link capacity to 25Mbps. The capacity is reduced in order to obtain comparable results. Initially, we performed the same experiments using 1Gbps links. However, with CacheCast the bottleneck is the server CPU power while without CacheCast the bottleneck is the S-R link capacity which makes the experiments incomparable.

The server S uses the `paraslash`[3] software to stream an audio file. Our decision is based on the fact that the software implements streaming using the DCCP protocol [52]. The DCCP protocol controls congestion in the network, thus, we do not risk the server being overloaded or the network being congested. We have modified the `paraslash` software so that it uses the `msend` system call to stream the audio file. To obtain maximum cache efficiency, we demand all the client streams to be synchronised in time. We do not queue a sample that is not sent to a client (e.g. due to congestion) instead it is dropped. If a client exceeds a threshold of 10% of dropped samples in a certain time window, we interpret it as a slow connection and the client is disconnected.

The router (R) is based on the Click Modular Router software which we describe in the following chapter. On each input and output connection we install the CSU and the CMU elements. The router was configured as it is depicted in Figure 6.10 and acts as an IP router connecting three different sub-networks. The client machines (A and B) use the `paraslash` receiver software to request an audio stream from the server S. We install the Click CSU element on each machine.

We perform the following experiment: The server S streams an mp3 file at a rate of 320Kbps using 1024B data chunks. We gradually increase the amount of clients requesting the content equally from the machines A and B until we fill the bottleneck link. Then, we query the server S for the number of actually connected clients. We measure the server CPU utilisation for one minute

[3] http://paraslash.systemlinux.org/

Table 6.3: Server S streaming 320Kbps audio in 25Mbps network.

Server type	Chunk size	Users	CPU
Original	1024	74	2.71 ± 1.1
CacheCast	1024	1020	44.21 ± 1.6
CacheCast	2066	2066	59.63 ± 1.7
CacheCast	4184	4097	58.30 ± 1.0
CacheCast	8364	8001	87.17 ± 1.8
CacheCast	15674	12454	91.35 ± 3.049[a]

[a]Limited by clients resources at approx. 80% utilisation of the network capacity

and record the average value together with the standard deviation. The measurements are taken in two configurations: (1) using the original paraslash software, and (2) the modified CacheCast implementation of the software. In order to evaluate the impact of the packet size that is used to carry the audio stream on the CacheCast implementation, we also vary the data chunk size carried by packets.

Table 6.3 shows the number of users connected to the server and the server CPU load for different data chunk sizes. The original server implementation can only handle 74 clients due to the bandwidth limitation. Using the same data chunk size the CacheCast server can handle more clients, since the transmission of the redundant payloads does not consume scarce link capacity. This can be further improved by using larger chunks. The reason is that with the growing data chunk size a single client consumes less bandwidth. With larger chunks the amount of packets per unit time in the network decreases. Since almost all packets are truncated by CacheCast to the minimum size, the utilised bandwidth is proportional to the amount of packets transmitted per time unit.

The analysis of the msend system call described in Section 6.4 indicates that its performance does not depend on the transmitted data size when the destination group size is large. The testbed results confirm this general trend. Increasing the chunk size eight times we could handle eight times more clients, while the server load increased only twice. One could expect no increase in the CPU load; however, the server performance does not solely depend on the msend system call but also on other client related tasks. The testbed experiment shows the importance of the system call performance and how larger data chunks can reduce the server load substantially.

6.6 Summary

In this chapter we have conducted an initial evaluation of the computational complexity of the server support. The server support is integrated with the Linux OS and an application can access it with the msend system call. The evaluation indicates that the msend system call achieves similar performance to the standard send system call when transmitting to a single destination. However, when transmitting to a large group of hosts, the server support does not only remove the redundancy from transmitted packets, but also reduces the server load. We support this claim with an experiment in a small testbed.

The testbed evaluation shows that the CacheCast system can be introduced into the existing

network, bringing near multicast performance while still maintaining the end-to-end relationship between a client and a server. We experience no difficulties while modifying the paraslash server to use the `msend` system call in the DCCP mode. The modified server could perform suitable AAA services with the DCCP protocol controlling individual connections to each client. In the next chapter we evaluate the computational complexity of a link cache.

Chapter 7

Computational complexity - link cache

This is the second chapter in which we analyse the computational complexity of the CacheCast system. In the previous chapter we evaluated the server support part of the system. The results indicate that the server support does not create additional burden on a server, rather, it can improve the server performance. In this chapter we evaluate the second part of the system, i.e. a link cache. A link cache consists of two elements located at the edges of a link. The first element CMU is located at the link entry and removes payloads that are already present in a cache on the link exit. The second element CSU is located at the link exit and restores from a local cache the payloads that were removed by the CMU element. While the link cache algorithm is simple, it is not given how fast it can operate, how much resources it requires, and what are its bottlenecks. To study these issues, we implement the link cache elements in the Click modular router software [53] and evaluate them in the context of router operation.

Since the link cache elements performance is highly dependent on the host hardware architecture which varies greatly among routers, we chose not to assess the performance of the elements as stand-alone units. Instead, we decided to evaluate the elements in the context of complete router operations and to measure how the elements impact router performance. Using the Click modular router software we build an IP router on a standard desktop machine. To understand the impact of the link cache elements on the router performance, we measure the router performance before installing the CacheCast elements and after. The results indicate that it is always beneficial to install CacheCast elements even if they consume additional CPU cycles.

The rest of this chapter is organised as follows: In Section 7.1 we describe the link cache design and its implementation in Click modular router software. We also provide a brief overview of the Click software components. Section 7.2 presents the evaluation of the link cache elements in the context of router operations. We show how the performance of a standard router changes when installing the CacheCast elements. Finally, we conclude this chapter in Section 7.3.

7.1 Router Elements

Since off-the-shelf routers are not programmable, we decided to build a CacheCast capable router using a desktop machine and the Click modular router software [53]. Click provides elements for packet processing and a flexible framework for composing them. The composition of elements is called a configuration. A part of the Click software is a standard IP router compliant configuration consisting of sixteen elements. In order to run a CacheCast capable router, we also need to imple-

ment and insert the CSU and CMU elements as the first and last elements in the packet processing path.

The CacheCast elements are implemented according to the functional description presented in Section 3.4 with modification in the CMU table and CSU memory slot design. The slot size of the maximum payload size causes severely reduced utilization of the CSU memory, since even small payloads occupy the whole slot. To mitigate this problem, we use slots which are smaller than the maximum payload size and store large payloads in multiple contiguous slots. Decreasing the slot size increases utilisation of the CSU memory. However, it also increases the number of slots and, thus, the management burden on the CMU side. We have chosen the default slot size of 512B, which we consider to be a good trade-off between the CSU memory fragmentation and the CMU management burden. Nonetheless, our Click implementation can operate also with other slot sizes. In the following sections we describe the differences between the original design presented in Section 3.4 and the modifications. Next, we provide a brief overview of Click software architecture and finally we describe the link cache implementation in Click software.

7.1.1 CMU and CSU design

As in the original design, the CMU table contains the same amount of entries as the corresponding CSU slots (cf. Figure 7.1). An entry located under a given index in the CMU table describes the content of a memory slot located under the same index in the CSU. Since a packet payload can be stored in multiple contiguous slots, the CMU entries no longer describe whole payloads but rather payload chunks. We decided that only the CMU entry which refers to the first chunk of a payload should contain a payload ID. The CMU entries referring to the remaining chunks of the payload are set to zero. Therefore, the payload ID lookup operation always returns an index to the first chunk of a payload, if there is a match.

The linear payload ID lookup in the CMU table is unacceptable due to incurred delays; therefore we facilitate the lookup operation with a hash table. We use the payload ID as the key for the hashing function and we store the index of the payload ID in the resulting entry of the hash table. The hash table is consistent with the CMU table. If we overwrite a payload ID in the CMU table, we also remove the corresponding payload ID key from the hash table. Similarly, if we insert a payload ID into the CMU table, we also update the hash table.

The cache has the round robin replacement policy that is implemented with the current index pointer on the CMU side. The pointer refers to the next entry in the CMU table to be replaced. If a payload ID of a CacheCast packet is not found in the CMU table, it is inserted in the table under the current index. If the payload is stored in multiple CSU slots, the consecutive table entries are set to zero. The hash table is updated and the current index advances over the modified entries.

When traversing a link, a CacheCast packet carries the CacheCast related information in the CacheCast header (see Figure 7.2). However, when the packet enters the router, the CSU removes the CacheCast header and stores the payload ID and the payload size in the packet annotations. Thus, the IP router processes the packet in the standard way. The payload size is always set to the valid value in the annotations. The index is not relevant in the router context and is dropped. The CMU identifies the CacheCast packet by present CacheCast annotations and constructs the CacheCast header before the link transmission. The link cache elements use the same CacheCast header structure as the server support described in Chapter 6.

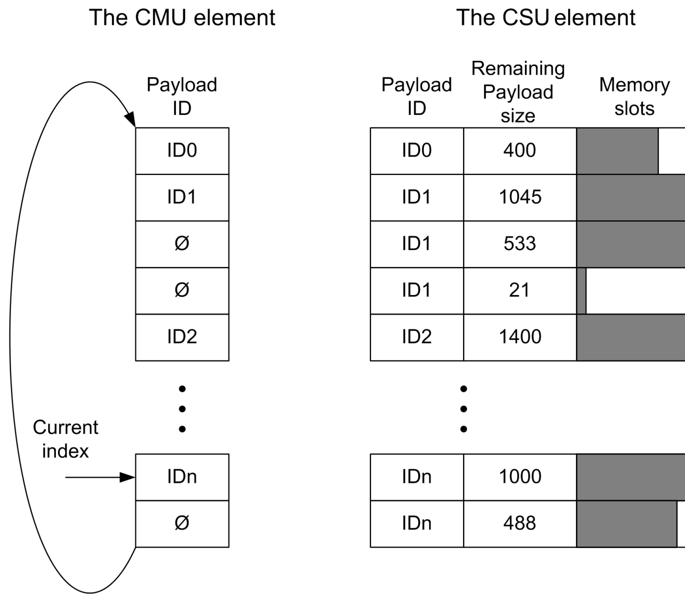


Figure 7.1: CMU and CSU relation

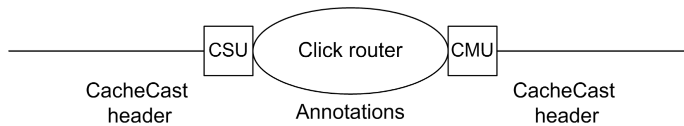


Figure 7.2: Handling the CacheCast header related information in the Click router

To indicate that a CacheCast packet is truncated, the CMU sets the payload size field in the CacheCast header to zero. However, since the packet payload may be stored in multiple slots in the CSU, the information on the total payload size is necessary to restore the payload. We decided to store the information in the meta-data of the first payload chunk. The meta-data of the subsequent payload contains the remaining payload size (see Figure 7.1). Thus, when the CSU receives a truncated packet it knows immediately its payload size.

Handling errors on a link

The CMU controls where packet payloads are stored in the CSU using the INDEX field in the CacheCast header. However, the CMU table and the CSU slots may become inconsistent due to packet drop. If a packet with payload is dropped on a link, the CMU table entry containing this packet payload ID is inconsistent with the CSU slot which contains old payload. Subsequent packets with the same payload are truncated on the link entry and carry the index of the old payload which is then attached to them on the link exit. To avoid this packet corruption, the CSU stores payloads along with their IDs. Thus, before the CSU restores payload, it compares the ID from the packet CacheCast header with the ID of the payload pointed to by the index. If these two match, the payload is restored, otherwise the packet is dropped. This mechanism protects against packet

corruption caused by packet drop and packet reordering on a link.

Cache consistency

To keep a cache consistent it is necessary that the CMU and the associated CSU have the same amount of table entries/slots and use the same slot size. This is achieved during the configuration of elements. The slot size has the default value of 512B, and the total amount of CSU memory is scaled with the associated link capacity. By default it is equivalent to the maximum amount of data that traverses the link within a 10ms time period.

7.1.2 Click modular router software

The Click modular router software provides a quick and easy way to build a router using a standard desktop machine running Linux or FreeBSD operating system. The key components of the Click architecture are: a library of packet processing elements, a router configuration, and a router driver. In order to start a router, a router driver is loaded with a configuration which defines connections between packet processing elements. The router driver parses the configuration, connects and initialises the elements, and starts up the router. In the following paragraphs we briefly describe the elements and configuration part of the Click architecture.

Click elements

A Click element is the smallest processing unit of a Click router which provides a simple functionality such as a packet queue, a packet scheduler, a packet classifier, or a network driver interface. Complex functions are composed of simple elements connected together. For example, to build a queue with the random early detection (RED) policy, we connect together the *RED* element implementing the RED policy and the *Queue* element implementing a packet storage. A Click element is characterised by four general properties:

1. **Element class:** Each element operating in a router belongs to a class that specifies a code to be executed when an element processes a packet. The class of an element is identified by the class name such as, for example, the previously mentioned *RED* or *Queue* classes.

2. **Ports:** An element has a number of input and output ports that can operate in one of two different modes: pull and push. If element operation does not constrain the type of ports, the ports can be defined as agnostic, i.e., they can act both as either pull or push ports.

3. **Configuration string:** To pass configuration arguments to an element, a user can use a configuration string which is parsed by an element during the router initialisation phase.

4. **Method interface:** Elements can communicate using method interfaces. An element can export one (or more) interface that provides data or methods. For instance, the *Queue* element exports an interface to query the queue length.

Click elements are implemented in C++ as objects which are derived from the `Element` class. The class is the base class for all Click elements and it provides a skeleton implementation of an element. In order to create a new element, it is necessary to complete the skeleton implementation with just a few declarations. As an example of element implementation, in Listing 7.1 we show

a null element implementation provided in the Click software documentation. The element does not perform any action, does not accept any configuration strings, and does not export any method interface. The element only forwards packets from its input port to its output port. The key declarations in the MyNullElement element implementation are the class_name(), port_count(), and processing() functions which describe element. The element operation is defined in the push() function which forwards packets to the first output port.

Listing 7.1: Null element

```
class MyNullElement : public Element { public:
  MyNullElement() { }
  ~MyNullElement() { }
  const char *class_name() const { return "MyNullElement"; }
  const char *port_count() const { return PORTS_1_1; }
  const char *processing() const { return PUSH; }
  void push(int port, Packet *p) {
    output(0).push(p);
  }
};
```

A part of Click elements acts as a simple packet filter with only a single input and a single output port. These elements can operate both in push and pull mode and are therefore defined as agnostic in terms of processing. To facilitate their implementation, the Element class provides a simple_action() function that, by default, is called both by the push() and pull() functions. Thus, regardless of the processing type of the element, the simple_action() function is invoked to process a packet. We use this skeleton function to implement the link cache elements.

Packet processing

Packet processing in a router can be initialised either by a source element creating packets or by a sink element consuming packets. When a source element creates a packet, it pushes the packet to its downstream elements in a series of push() function calls until the packet is consumed (e.g. it is stored on a queue). When a sink element is ready to consume a packet, it pulls the packet from its upstream elements in a series of pull() function calls.

The complete operation of the push and pull packet processing can be best illustrated in the simple configuration depicted in Figure 7.3. The *FromDevice* and *ToDevice* elements implement network driver interface receiving and transmitting parts respectively. When the *FromDevice* element receives a packet from the eth0 device, it pushes the packet to the *Queue* element which stores the packet temporarily. When the *ToDevice* receives a signal that the eth1 device is ready to transmit it pulls the *Queue* element, which either returns a previously enqueued packet or a null object if the queue is empty.

Figure 7.3: Push-pull example for a Click router configuration

Configuration

The Click configuration describes packet flows between router elements. It is a directed graph where elements are vertices and edges denote possible packet paths between the elements. Correct configuration requires that only elements with a compatible port type are connected. If an element has agnostic ports, the type of ports is resolved to either pull or push type based on adjacent elements. Additionally, input and output ports of an agnostic element must be of the same type. The port constraints are propagated until all port types are resolved. In the event that not all constraints can be successfully resolved, a router driver returns an error message pointing to the problem.

Click configurations are described with a simple language that enables a user to declare elements and then to specify connections between them. In Listing 7.2 we provide two alternative descriptions of the configuration presented in Figure 7.3. Firstly, in three declarations we create the *FromDevice*, *ToDevice* and *Queue* elements. The *FromDevice* and *ToDevice* elements require a network device name to read and write to. This information element is provided with a configuration string in parenthesis. Secondly, the declared elements are connected together using an arrow syntax. Alternatively, the complete configuration can be written as a continuous chain of elements without preceding declarations.

Listing 7.2: Push-pull example configuration

```
// Declarations:
 src  :: FromDevice(eth0);
 dst  :: ToDevice(eth1);
 q    :: Queue;
// Connections:
 src -> q;
 q -> dst;

// Declaration are not required, thus alternatively:
 FromDevice(eth0) -> Queue -> ToDevice(eth1);
```

7.1.3 CMU and CSU implementation in Click software

The CMU and CSU functionalities are implemented as two distinct Click elements with class name *CacheManagementUnit* and *CacheStoreUnit* respectively. The elements perform a simple processing which can be executed both in push and pull mode. Figure 7.4 shows how these elements should be installed in a simple IP router. The IP router internals are replaced with a cloud, since these are irrelevant for the CacheCast operation[1]. The router switches packets between two links connected with the `eth0` and `eth1` interfaces. We use the *FromDevice* and *ToDevice* elements to read and write packets to these interfaces. The *CacheStoreUnit* elements are installed immediately after the *FromDevice* elements to restore packet payloads and remove the CacheCast header before the IP router processing. The elements operate in push mode, which is enforced by the *FromDevice* elements. The *CacheManagemntUnit* elements are inserted between the transmission queue and the *ToDevice* elements. Therefore, packet drop in the transmission queue does not affect link cache consistency. The elements remove redundant payloads and create the CacheCast header before link

[1]For the complete IP router configuration please refer to [53].

transmission. In this configuration, the operation of the CMU element is crucial, since it introduces delays in packet transmission. The following explains it: When the *ToDevice* element is ready for transmission it pulls a packet from the transmission queue. If there is a packet in the queue, it is first processed by the CMU and then it is forwarded to the *ToDevice* element for transmission. Therefore, the delay introduced by the CMU increases the idle time of the link transmission channel.

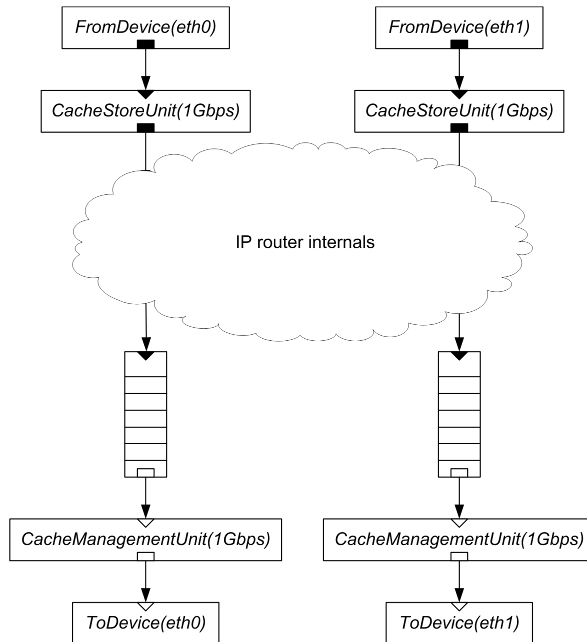


Figure 7.4: Click CacheCast IP router

The *CacheManagementUnit* and *CacheStoreUnit* elements accept only two configuration parameters: `RATE` defining the associated link rate and `CHUNK_SIZE` specifying the memory slot size. If the `CHUNK_SIZE` parameter is not given the elements are configured with the default value of 512B. Based on these two parameters and the assumption that CSU should accommodate $10ms$ of traffic flowing through it, the elements calculate the number of memory slots in CSU. On the CMU side this number describes the size of a table where payload IDs are stored and on the CSU side this number describes the number of memory slots. A pair of CMU and CSU elements that compose a single link cache must have the same configuration parameters in order to ensure correct operation of the link cache. In Listing 7.3 we show example declarations of the CacheCast elements in the Click configuration language.

Listing 7.3: Example declarations of CacheCast elements

```
cmu :: CacheManagementUnit(1Gbps);
  // Declares the CMU element connected to 1Gbps link
  //  with the default slot size of 512B
  // The RATE parameter is read implicitly
csu :: CacheStoreUnit(RATE 100Mbps, CHUNK_SIZE 256B);
```

```
// Declares the CSU element connected to 100Mbps link
//  with a non-standard slot size of 256B
```

In the following paragraphs we discuss specifics of the link cache elements' implementation as well as assumptions made.

CacheManagmentUnit element

CacheManagementUnit acts as a simple processing element that receives a packet on its input port, processes the packet, and forwards it further to its output port. All packets with cleared CacheCast annotations are immediately forwarded without any processing. If the element receives a packet with the CacheCast annotations set, it processes the packet under the assumptions that the packet is Ethernet encapsulated and without the CacheCast header. Packets processed by the element have the CacheCast header inserted between the Ethernet header and the IP header. This is with the exception of packets that do not carry CacheCast annotations and packets with payloads larger than the CSU storage space.

The *CacheManagmentUnit* functionality is completely implemented in the `simple_action()` function shown in Listing 7.4. The function uses three classes from the Click software that require a brief explanation. A Click packet is represented with the `Packet` class. The class defines two buffers: a buffer for packet data and a buffer for annotations. The packet data is stored in the former buffer and its boundaries are determined with the `data` and `end_data` pointers. To allocate additional space at the head or at the tail of the packet data, we use `push()` and `put()` functions; to truncate the head or the tail of the packet data, we use `pull()` and `take()` functions. The buffer for packet annotations is a fixed block of memory without structure. To store data in the annotation buffer, we call `set_anno_u*(offset, value)` function providing the offset in the memory block and the value to be stored as arguments. For example, to store a payload ID in the annotations we call `set_anno_u32(CACHECAST_PAYLOAD_ID, payload_id)`. A similar function `anno_u*(offset)` is used to read annotations. Click does not provide any allocation mechanism for the annotation buffer therefore care must be taken when chosing the offset values. We define two offsets in the annotation buffer to store the payload ID and the payload size. In order to create or to destroy a packet, we call `make()` and `kill()` functions.

The Packet class implements only immutable packets. This constraint enables efficient packet duplication, since a copy of a packet can share a data buffer with an original packet. Nonetheless, the link cache elements modify packet data and, thus, require mutable packets. The mutable packets are implemented with the `WritablePacket` class. To make a packet mutable, we use a `uniqueify()` function provided in the `Packet` class which returns a packet with a private buffer. Additionally, the aforementioned functions that change the boundaries of the packet data in the packet buffer return mutable packets.

The Click software provides the `HashTable` class template for efficient implementation of hash tables or hash sets. The implementation is based on a chained hash table, where elements with the same hash value are put on a linked list. The class template has a set of functions for lookup, insert, delete, and other types of operation on a hash table. However, we use only three of them in our implementation: `find_prefer()`, `find_insert()`, and `erase()`. The `find_prefer(key)` function returns an iterator for an element with key `key`. If the key is not in the table, the function returns an iterator of the end of the table. If the key is in the table, an element associated with this key is

moved to the front of the linked list. This speeds up the lookup operation for the same element if there are more elements with the same hash value. We use this function, since we anticipate frequent lookups for the same payload ID within a short period of time, which are related to the packet train structure. The find_insert(key) function inserts an element with key key to a hash table and returns an iterator for the element. If an element associated with the key is already in the table, the function returns an iterator for this element. The erase(key) function removes an element associated with key key.

We use the HashTable class to implement an associative array (the payload_IdToIndex table) mapping payload IDs to INDEX values. The lookup with key ID in the table returns an iterator for the element containing the INDEX value in case of a cache hit, or an iterator for the end of the table in case of a cache miss. Initially, along with the INDEX value, a table element contained also a timestamp describing a point in time when this element was created. We have used the timestamp to invalidate entries that are older than one second, as discussed in Section 3.4.6. However, a call to read the current system time consumed much more CPU cycles than the whole element processing. Consequently, we have removed this functionality, since we focus our measurements on the CMU performance.

Listing 7.4: Source code of the simple_action function for the *CacheManagementUnit* element

```
Packet *CacheManagementUnit::simple_action(Packet *p_in)
{
  uint32_t _chunks_count; // number of chunks required to store payload
  uint32_t _payload_id;
  uint32_t _packet_ipv4_addr;
  uint16_t _payload_size;
  struct cachecast_header *cc_h;  // pointer to the CacheCast header
  struct click_ether   *ether_h;  // pointer to the Ethernet header
  struct click_ip      *ip_h;     // pointer to the IP header
  WritablePacket *p;

  /* CacheCast packets are marked with the CacheCast annotations */
  if(!p_in->anno_u32(CACHECAST_PAYLOAD_ID))
    return p_in;

  /* Copy the packet annotations to the local variables */
  _payload_id = p_in->anno_u32(CACHECAST_PAYLOAD_ID);
  _payload_size = p_in->anno_u16(CACHECAST_PAYLOAD_SIZE);

  /* Does the packet fit the associated CSU on the remote end? */
  _chunks_count = _payload_size / _chunk_size + 1;
  if(_chunks_count > _table_size)
    return p_in;

  /* Extend the packet head by the size of the CacheCast header */
  p = p_in->push(sizeof(struct cachecast_header));
  if(!p)
    return NULL;

  cc_h = (struct cachecast_header *)(p->data() + ETH_LEN);
  ether_h = (struct click_ether *)p->data();
```

```
/* Move the Ethernet header to the head of the packet buffer */
memmove(p->data(), p->data() + CH_LEN, ETH_ALEN * 2);
/* Set the CacheCast Ethernet type */
ether_h->ether_type = htons(ETHERTYPE_CACHECAST);

/* get packet IP address */
ip_h = (struct click_ip *)(p_in->data() + ETH_LEN + CH_LEN);
memcpy(&_packet_ipv4_addr, &ip_h->ip_src, sizeof(_packet_ipv4_addr));

Table::iterator _iter;
uint32_t _INDEX;
uint64_t _id;
uint64_t _evict_id;

/* The unique payload ID is a combination of the packet IP address
 * and the payload ID */
_id = ((uint64_t)_packet_ipv4_addr << 32) | _payload_id;

/* Lookup the ID in the ID to index hash table */
_iter = _table_IdToIndex->find_prefer(_id);

/* Cache Hit */
if (_iter != _table_IdToIndex->end()) {
  p->take(_payload_size);    // remove the packet payload
  _payload_size = 0;         // clear payload size
  _INDEX = _iter->_INDEX;    // copy the index value for this ID
} else {
/* Cache Miss */
  _INDEX = _cur_INDEX;       // the index value is the current index
  /* Allocate sufficient amount of chunks to store payload by
   * removing IDs referring to payloads which are invalid after
   * packet transmission */
  for(int i = 0; i < _chunks_count; i++) {
    _evict_id = _table_IndexToId[_cur_INDEX];
    if(_evict_id != INVALID_ID)      {
      _table_IndexToId[_cur_INDEX] = INVALID_ID;
      _table_IdToIndex->erase(_evict_id); // synchronises the hash table
    }
    _cur_INDEX = (_cur_INDEX + 1) % _table_size;
  }

  _iter = _table_IdToIndex->find_insert(_id);
  _iter->_INDEX = _INDEX;
  _table_IndexToId[_INDEX] = _id;
}

/* Build the CacheCast header */
cc_h->payload_id = htonl(_payload_id);
cc_h->payload_size = htons(_payload_size);
cc_h->INDEX = htonl(_INDEX);
// packet type is set, since ch_h->packet_type == (old) ether_h->type

return p;
```

```
}
```

CacheStoreUnit element

Similar to *CacheManagementUnit*, *CacheStoreUnit* acts as a simple processing element with only a single input port and a single output port. The complete functionality of the *CacheStoreUnit* element is implemented in the `simple_action()` function shown in Listing 7.5. The element operates under the assumption that it receives Ethernet encapsulated packets. If the Ethernet type of a packet is different from the CacheCast Ethernet type, the packet is immediately forwarded further downstream. Based on the `payload_size` field in the CacheCast header, the element determines whether it is only a header part of a packet or a complete packet. If a header part of a packet arrives, the element copies payload chunks from the cache store slots to the packet buffer. If a complete packet arrives, the reverse operation is performed. To secure a packet against possible corruption, as discussed in Section 7.1.1, we store the unique payload ID together with the payload chunks. As the last step in packet processing, we copy CacheCast related information to the packet annotations and remove the CacheCast header.

The implementation of the CacheCast elements is not optimised. While the CSU element copies payloads to each compressed CacheCast packet that arrives, this could be avoided. If a router could store a packet in multiple buffers (like the **mbuf** structure in the BSD OS) the CSU element would simply link payload to a packet header. However, the Click router does not support this kind of structure and we are forced to use the copy operation.

Listing 7.5: Source code of the `simple_action` function for the *CacheStoreUnit* element

```
Packet *CacheStore::simple_action(Packet *p_in)
{
  uint32_t _chunks_count;
  uint32_t _rem_chunks_size;     // size of the remaining chunks
  struct cachecast_header *cc_h;  // pointer to the CacheCast header
  struct click_ether   *ether_h;  // pointer to the Ethernet header
  struct click_ip      *ip_h;     // pointer to the IP header
  WritablePacket *p;

  /* Forward non-CacheCast packets immediately */
  ether_h = (struct click_ether *)p_in->data();
  if(ntohs(ether_h->ether_type) != ETHERTYPE_CACHECAST)
    return p_in;

  /* Received packet is a CacheCast packet */
  cc_h = (struct cachecast_header *)(p_in->data() + ETH_LEN);

  uint32_t _INDEX = ntohl(cc_h->INDEX);
  uint32_t _payload_id = ntohl(cc_h->payload_id);
  uint16_t _payload_size = ntohs(cc_h->payload_size);

  /* Does payload fit the cache store? */
  _chunks_count = (_payload_size - 1) / _chunk_size + 1;
  if(_chunks_count > _store_size) {
    p_in->kill();
    return NULL;
```

```
}

/* Verify the INDEX value */
if (_INDEX > _store_size)
  return p_in;

/* Unique id is a sum of a payload ID and a source IP address */
uint32_t _packet_ipv4_addr;
uint64_t _id;

ip_h = (struct click_ip *)(p_in->data() + ETH_LEN + CH_LEN);
memcpy(&_packet_ipv4_addr, &ip_h->ip_src, sizeof(_packet_ipv4_addr));

_id = ((uint64_t) _packet_ipv4_addr << 32) | _payload_id;

/* Is it a complete packet or just a header? */
if(_payload_size == 0) {
  /* Header arrived */
  _payload_size = _chunk_store[_INDEX].chunk_size;

  /* Extend packet buffer by the payload size */
  p = p_in->put(_payload_size);
  if(!p)
    return NULL;

  for (int i = 0; i < _chunks_count; i++) {
    /* Does the payload id and the chunk id match? */
    if(_chunk_store[_INDEX].id != _id) {
      p->kill();
      return NULL;
    }
    /* Copy payload from the slots to the packet buffer */
    memcpy((p->end_data() - _payload_size) + _chunk_size * i, // to
      _chunk_store[_INDEX].chunk,                                 // from
      _chunk_store[_INDEX].chunk_size > _chunk_size ?
        _chunk_size : _chunk_store[_INDEX].chunk_size);           // size

    _INDEX = (_INDEX + 1) % _store_size;
  }
} else {
  /* Complete packet arrived */
  _rem_chunks_size = _payload_size;

  for(int i = 0; i < _chunks_count; i++) {
    _chunk_store[_INDEX].id = _id;
    _chunk_store[_INDEX].chunk_size = _rem_chunks_size;

    /* Copy the packet payload to the slots */
    memcpy(_chunk_store[_INDEX].chunk,                         // to
      (p_in->end_data() - _payload_size) + i * _chunk_size, // from
      _rem_chunks_size > _chunk_size ? _chunk_size : _rem_chunks_size);

    _INDEX = (_INDEX + 1) % _store_size;
```

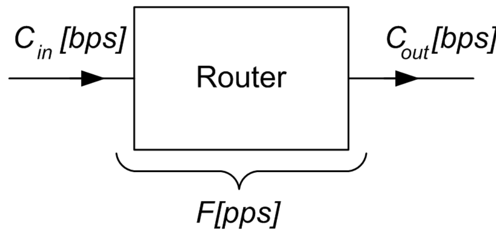


Figure 7.5: The router model

```
            _rem_chunks_size -= _chunk_size;
        }

        /* Make a mutable packet */
        p = p_in->uniqueify();
        if(!p)
                        return NULL;
    }

    /* set packet annotations */
    p->set_anno_u32(CACHECAST_PAYLOAD_ID, _payload_id);
    p->set_anno_u16(CACHECAST_PAYLOAD_SIZE, _payload_size);

    /* remove the CacheCast header */
    memmove(p->data() + CH_LEN, p->data(), ETH_ALEN * 2);
    p->pull(CH_LEN);

    return p;
}
```

7.2 Evaluation

7.2.1 Router model

Since the CSU and CMU elements operate in a router, we run performance analysis in the context of router operations. We use the following router model (see Figure 7.5). A router is a store and forward network node which performs the IP router related tasks as defined in [54]. It has a shared memory architecture and its input and output capacity is limited in terms of bits per second. We assume that input capacity C_{in} is equal to output capacity C_{out}. The forwarding rate of a router F is limited by its processing capabilities and is expressed in packets per second. A router can forward packets of minimum size at line rate. This implies that the forwarding rate F is at least C_{in}/S_{min} packets per seconds, where S_{min} denotes the minimum packet size in a network.

The router is built based on a 2.4GHz Intel machine using the standard-compliant Click router software [53]. To simplify measurements and interpretation of results, we read packets from a memory mapped trace file and send them directly to the Click router software. The packets are discarded after being processed by the router.

7.2.2 Trace files

The trace files we use in the following experiments are prepared using two machines. The first machine generates packets while the second machine records the resulting network traffic to trace files. Each trace file contains approximately 100MB of packets. The first trace file has non-CacheCast minimum size IP packets, which we use to establish the router F, C_{in} and C_{out} parameters. The remaining trace files contain CacheCast packets of the *packet train* structure, which we generate using the `msend` system call. Since the index value in the CacheCast header is set to zero for all generated packets, we send them additionally through a cascade of the CSU and CMU elements. This is create indexes in the CacheCast header of the packets. The traces are generated for three group sizes: 10, 100, and 1000, and with four different payload sizes: 500B, 1500B, 9000B, and 16000B. We use them to evaluate the CacheCast router performance.

7.2.3 Router parameters

In order to obtain the forwarding rate F and the maximum input/output capacity C_{in}/C_{out} of the Click router, we use the first trace file containing minimum size IP packets. We load packets from the trace file directly to the IP router software and measure the average time to process a single packet. Based on the average packet processing time we calculated that the router forwarding rate is $763 \pm 2.7\% Kpps$ and the corresponding maximum input/output capacity is $366.2 \pm 2.7\% Mbps$. We assume that the minimum packet size S_{min} is 60B.

7.2.4 CacheCast router performance

We install the CSU as the first element and the CMU as the last element on the Click router forwarding path as described in Section 7.1.3. The elements are configured with the following parameters: `RATE` $366.2 Mbps$ and `CHUNK_SIZE` $512B$. This results in the CSU of 468KB storage space divided into 937 slots and the equivalent number of table entries in the CMU. Since CacheCast elements impose an additional burden on the router CPU, this decreases the forwarding rate of the router F. Based on the *packet train* structure, the average packet size is $\overline{s} = s_p/n + s_h$, where s_p, s_h, and n denote payload size, header size, and destination group size respectively. Thus, the CacheCast average packet size is low or even has minimum size when the destination group is large. We expect that with decreased forwarding rate F' the router is not able to fully utilise the output capacity C_{out} and the router efficiency is affected. Without CacheCast this does not occur. However, the same packets carry payloads and thus fill the input capacity C_{in} much faster than CacheCast packets. Now, the bottleneck is the input capacity C_{in}. Therefore, the question is: *Is it beneficial to install the CSU and CMU elements in a router?* We demonstrate that, even with our simple CSU and CMU implementation, CacheCast yields substantial benefits.

Similar to the minimum packet size measurements, we load packets from the trace files containing CacheCast packets directly to the router software and measure the average time to process a single packet. Based on the measurements we calculate the CacheCast router forwarding rate F' and the corresponding output capacity utilisation.

The results depicted in Figure 7.6 show the utilisation of the output capacity of the router. Even though the router forwarding rate F is reduced, it can still utilise 100% of the output capacity when forwarding short *packet trains*. However, when the *packet train* size increases to 100 packets and

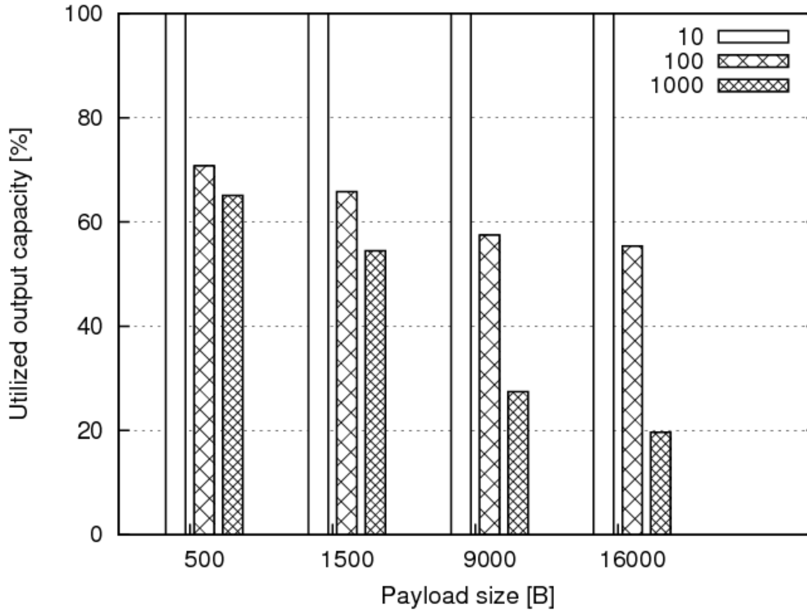


Figure 7.6: Utilised output capacity of the Click router for three different *packet train* sizes and three different payload sizes

above, the router becomes the bottleneck and the output capacity utilisation decreases. The reason for this is that the average packet size $\overline{s}$ in the large *packet train* decreases to the minimum size while the router is not able to forward the minimum size packets due to CacheCast operations. The utilisation is especially low when forwarding *packet trains* with large payloads, since the CSU, which mainly performs payload store and restore operations, consumes more router CPU cycles.

The original router forwards traffic with redundant payloads while the CacheCast router avoids the redundant payload transmission. To understand the benefits of the CacheCast router we compare the volume of traffic forwarded by both routers within a given time unit. However, we count the CacheCast traffic volume as if it carried the redundant payloads. We refer to it as the effective throughput. In Table 7.1 we show the ratio of the effective throughputs of the CacheCast router and the original router. Obviously, the effective throughput of the CacheCast router is higher than the original router when forwarding short *packet trains*, since it avoids redundant payload transmissions while still achieving 100% utilisation of the output capacity. Moreover, the effective throughput ratio is substantially higher for large *packet trains*, even though the output capacity is not fully utilised. For example, even with the 57% utilisation of the output capacity the CacheCast router can effectively forward 13 times more data of the following *packet train* structure: 1000 packets with the payload size of 1500B.

Table 7.1: The ratio of the effective throughputs of the CacheCast router and the original router

Group size	Payload size 500B	1500B	9000B	16000B
10	4.49	7.35	9.42	9.67
100	5.82	13.36	34.37	40.11
1000	5.80	13.52	35.38	40.10

7.3 Summary

While the server support can be implemented and evaluated according to conditions which are close to the real world, the link cache elements could not be implemented in a standard router. Instead, we have used the Click modular router software to implement the link cache elements and we have evaluated them on a single CPU machine. The evaluation was designed to give us insight that could be extended over the first generation routers. We considered a router with the fixed computational capacity designed to forward the minimum size packets with the line rate. Since operation of the link cache elements consumes a fraction of the router computational capacity, the router may not be able to utilise 100% of the output link capacity. However, the evaluation shows that installing the CacheCast functionality in the router is always beneficial. If the CacheCast traffic consists of short packet trains, the router is still able to fully utilise the output link capacity. The utilisation decreases only when the CacheCast traffic consists of long packet trains. It is most likely that this will only occur at the network routers close to servers.

Chapter 8

Related work

This chapter presents the related work for CacheCast. While we have already described closely related systems in Chapter 2, this chapter covers a broader range of packet caching systems. Furthermore, it provides a detailed comparison between CacheCast and the system of Anand et al. presented in [36]. We describe how the decision to split caching burden between a server and link caches simplifies the link cache implementation when compared to the system of Anand et al.

8.1 Network-wide redundancy elimination

Link layer caching has been recently studied in the context of network-wide redundancy suppression by Anand et al. in [36]. The authors envisage that all future routers will have the ability to remove redundant content from packets on the fly. Thus, the proposed system is more general than CacheCast, since CacheCast aims to remove only a specific type of redundancy.

While [36] briefly covers link layer caching and focuses mainly on the "redundancy-aware" routing protocol, CacheCast concentrates on the design of a link cache and its feasibility. CacheCast design decisions follow from the principle that both link caches and a server should make contribute removing redundancy from single source multiple destination transfer. A server is responsible for batch requests for the same data and for transmitting packets carrying the same data chunks within a minimum time window. We show how the server support greatly simplifies implementation of link caches by comparing it with the approach of Anand et al. [36]. Based on this comparison, we later explain differences with other related systems.

Both CacheCast and the system of Anand et al. are designed to operate in the wired Internet. This imposes very strong constraints on the cache storage space and on computational complexity of the caching algorithms. A large storage space improves redundancy detection in the link traffic. However, it also costs more, both in economical and computational terms. The computational complexity of the cache algorithm must be minimised to decrease per packet processing time. Ideally, the amount of packets per second processed by a cache should match link throughput. This, however, is difficult to achieve when considering modern fibre links transmitting up to 40Gbps.

8.1.1 Storage space

The approach from Anand et al. requires two types of storage space on both link ends, i.e., a packet store and a fingerprint store. The packet store contains packets that were recently transmitted over

a link and the fingerprint store contains fingerprints of packet chunks. Each fingerprint identifies a 64B chunk of a packet from the packet store. In a default configuration, a single packet in the packet store is represented by 32 fingerprints in the fingerprint store. To find redundancy in an incoming packet, a set of fingerprints is generated and then compared with those stored in the fingerprint store. If a match is found, it means that most probably the packet carries redundant content. To verify this[1] and to find the length of the matching string, the matching packet is fetched from the packet store and compared byte-by-byte with the incoming packet. Next, the matching string is replaced by a description carrying the information necessary to decode the string at the link egress. Therefore, to reconstruct the packet, the fingerprint store and the packet store must be coherent with those on the link entry.

Since the approach of Anand et al. is agnostic to network traffic, in order to achieve a meaningful reduction in a network footprint it requires to cache approximately 10 seconds of the associated link traffic[2]. Additionally, the necessary storage space for the cache must be increased by the fingerprint store size which is approximately half of the packet store size. This requires a considerable amount of memory on the ingress and egress side of a link. For example, to build a link cache on a 1Gbps link we need approximately 2GB of memory on each side of a link. Another downside of the large cache is the necessity to use cheaper DRAMs which do not provide necessary access speed to cache packets at high line rates. For example, in April 2011 the cost of one MB of DRAMs memory with 50ns access latency was approx. $0.01, while the cost of one MB of SRAM memory with 6ns access latency was approx. $15.

Conversely, CacheCast is based on the assumption that a source is aware of caching (as discussed in Section 3.3.2) and it makes an effort to transmit packets with the same payload within a minimum time interval. In our work, we estimate that 10ms caches are sufficient for this type of traffic, which is three orders of magnitude less than in the work of Anand et al. Additionally, CacheCast does not require storing packet payloads on the ingress side of a link. Unique payload IDs guarantee that the same payload IDs refer to the same content. Since CacheCast stores packet payloads in large slots, the resulting size of the CMU table is much smaller than the fingerprint store. Considering the three orders of magnitude smaller cache and the slot size of 512B, we obtain a four orders of magnitude smaller CMU table. Thus, to build a link cache on a 1Gbps link based on CacheCast design we need to allocate approximately 1.3MB for payload store on the egress side of a link and 30KB for the CMU table on the ingress side. Caches of this size can be efficiently implemented in SRAM memory that provide low access latency.

8.1.2 Computational complexity

The main bottleneck during packet encoding at the ingress side of a link cache is access latency to a storage space. Anand's algorithm requires 32 lookups in the fingerprint store, to find redundant chunks. Then, the matching packets are fetched from the packet store and compared byte-by-byte with the source packet. Finally, redundant strings are replaced with descriptions. While some of the operations can be executed in a pipeline, it requires dedicated hardware architecture. Furthermore, the sequence of 32 lookups in the fingerprint store poses a severe barrier to processing speed, since

[1]The fingerprint match does not guarantee that the 64B chunks are the same, since different 64B chunks might have the same fingerprints.

[2]Though, the cache size is not explicitly discussed in paper from Anand et al., the value of 10 seconds for the cache size is consistently used in the evaluation in [36, 37].

the fingerprint store is implemented with DRAMs. To address some of the limitations, Anand et al. propose reducing the number of lookups in the fingerprint store in favour of processing speed; however, consequently less redundancy is removed.

The CacheCast objective is to remove redundancy with the minimum amount of effort. A part of redundancy detection is already performed at the source and the information necessary to identify the redundant part of a packet is stored in the packet header. Therefore, the CMU element performs only one lookup to determine whether or not a packet carries redundant information. Furthermore, it is simple to remove and restore the redundant part of a packet, because it is always at the packet tail. These simple operations can be executed very efficiently and swiftly, because the storage needs for CMU and CSU are very small and faster SRAM storage is affordable for implementing these units.

8.1.3 Other considerations

Compared to the approach of Anand et al., CacheCast does not remove the redundancy originating from different sources sending the same data. However, it must be noted that the shorter the caching time is on a link the less is the likelihood to find this type of redundancy. Within a 10ms time window it is very unlikely to find redundancy other than the one which originates from the single source sending data to multiple destinations. Although, CacheCast poses an additional burden on the server side we show in Section 6.4 that it is negligible.

8.1.4 SmartRE

Based on the experience with the redundancy suppression system described in [36], Anand et al. proposed SmartRE [37] a new architecture of coordinated link caches. The authors acknowledge that with the naive approach presented in [36] the line rate encoding is not possible for high speed fibre links. To solve this problem, the packet encoding and decoding operations are distributed. Encoded packets are not decoded immediately at the downstream router, but can traverse a few hops before decoding. This reduces the number of operations per link cache. However, it also requires intelligent assignment of encoding and decoding operations between link caches.

While SmartRE reduces the encoding burden per link cache, it requires a distributed algorithm for coordination of encoding and decoding operations. This, in turn, reduces the reliability of the system. Moreover, SmartRE requires coherency between caches located within a distance of a few hops. However, this is difficult to achieve during congestion in a network, because under heavy load router queues drop packets causing inconsistency in distributed caches. It should be noticed that redundancy suppression is mainly useful in congested or near-congested networks. In the over-provisioned network there is no utility for redundancy suppression. Finally, the authors do not address the need for large storage space for packet and fingerprint storage.

8.2 Other related work

Packet level caching has been studied in the context of wired and wireless networks, slow access links, and point-to-point network paths. The system of Anand et al. is designed to operate in the

context of wired networks, thus, it is the closest related work to CacheCast. This section covers briefly other related systems that operate in the remaining contexts.

8.2.1 Redundancy elimination in multi-hop wireless networks

Wireless networks are characterised by low throughput and high packet loss which is mainly a result of interference between communicating stations. Therefore, hop-by-hop caching systems for wireless network operate under different constraints than systems for wired networks. Firstly, since the wireless networks have lower throughput, the time to process a packet is longer; thus, more complex algorithms can be employed. Secondly, under high packet loss, it is difficult to keep cache coherency between communicating hosts. Thirdly, the broadcast nature of the wireless medium opens new design opportunities for exploiting this property.

Afanasyev et al. [55] propose to cache recently overheard packets in order to eliminate redundancy in unicast transfers over a multi-hop path. Packets are annotated with IDs; therefore, before a station sends a packet, it transmits an RTS-id (request to send) message containing an ID of the packet. If the destination has overheard this packet, it replies with a CTS-ACK (clear to send) message acknowledging that the packet has been already received and the stations behave as if the transmission has already occurred. The technique removes redundant transmissions when a packet traverses a multi-hop path where hops are in close proximity. It also eliminates redundancy in the infrastructure based wireless networks where stations communicate via access-point. However, since IDs are generated based on the IP packet content, it does not remove redundancy from transfers of the same data to different destinations. This technique could be complemented with CacheCast. Information carried in the CacheCast header can be used to identify redundancy in packet payloads.

To enable application independent caching, Ditto [56] uses the data oriented transfer (DOT) technique introduced in [57]. DOT objects consist of small chunks identified with unique IDs. To download an object, a client first queries a server for a set of chunk IDs. Then, the client fetches the object chunks by IDs using DOT transport service. During data transfer, chunks are cached at on-path nodes and opportunistically (based on overhearing) on nearby nodes. Thus, if subsequently another client in the same wireless subnet requests the same chunks, it is served from the proxy nodes. While the system removes redundancy from single source multiple destination transfers, it requires large caches. Based on aggregated statistics from a real deployment of a Meraki wireless mesh network[3], the authors estimated that 3GB caches should suffice.

8.2.2 Point-to-point redundancy removal

The first systems for redundancy suppression were designed to improve web-caching (cf. Section 2.2.2). They consist of two coherent caches installed on both sides of the point-to-point path between a subnet gateway and a web-server where the ingress cache removes redundancy and encodes packets; and the egress cache decodes packets, thereby inserting back redundant information. These types of systems are usually deployed on slow access links or point-to-point network paths. Therefore, the systems can use large storage space and complex algorithms for redundancy detection.

[3] http://meraki.com/

Santos and Wetherall [58] describe the first system of this type. The redundancy detection is based on comparison of whole packet payloads. Therefore, any misalignments caused by TCP mechanisms, or different HTTP headers in packets, result in redundant transfers. CacheCast also detects redundancy by comparing whole packet payload. However, the server support part forces redundant data to be packet aligned. In the follow-up work, Wetherall et al. [30] propose detecting redundancy by comparing packet chunks. For this purpose they use a technique developed by Manber for finding similar files in a large file system [59]. While the technique can detect and eliminate redundancy in any part of a packet regardless of data misalignments, it requires more computational effort. Later the technique was applied to hop-by-hop redundancy elimination by Anand et al. [36].

8.2.3 Redundancy removal by compression

Removing redundancy from link layer data transfers is also addressed by different compression techniques of a packet header on a link (for example: RFC1144 [60], RFC2507 [61], RFC2508 [62], or RFC3095 [63]). These techniques are mainly deployed on dial-up and wireless links, where link capacities are low, and they are orthogonal to CacheCast. Thus, they can be used in parallel to provide even greater benefits.

8.3 Summary

The link layer packet caching system presented by Anand et al. in [36] is the closest related work to CacheCast. Therefore, in this chapter we have explained the key differences in the approaches of both systems and discussed implications of the design decisions. While the CacheCast approach is minimalistic in terms of necessary computational and storage resources, the approach of Anand et al. in contrast employs complex redundancy detection algorithms and requires substantial storage space, both of which are difficult to justify based on the gains envisioned.

To give a complete picture of the redundancy suppression in network transfers we have also briefly touched on other related systems operating within the context of wireless networks, access links, and point-to-point network paths (e.g. like paths between gateways and web-servers).

Chapter 9

Conclusions

This chapter provides thesis conclusions. In Chapter 1 we introduced the problem of single source multiple destination transfers in the Internet. We also gave a brief overview of the thesis. In Chapter 2 we analysed former solutions to the problem and drew general guidelines for the CacheCast system design. Chapter 3 provided fundamental information on the basic network elements that CacheCast builds on, i.e. link and router. Based on this information and the guidelines discussed in the previous chapter, we described the CacheCast design and presnented our rationale for the design decisions. The first insights into the system performance are presented in Chapter 4. We estimated the amount of redundancy that can be eliminated from single source multiple destination transfers when using CacheCast. We also discussed the case of incremental deployment and assessed the related bandwidth savings. Chapter 5 investigated the link cache impact on congestion control mechanisms built into transport protocols. We verified that CacheCast preserves the TCP fairness. In Chapters 6 and 7 we described the implementation of the server support and the link cache. The elements were evaluated with regard to the computational complexity, and the evaluation results confirmed the feasibility of the CacheCast approach. To show benefits of the CacheCast system in a real setup, in Chapter 6 we demonstrated how the `paraslash` media streaming software can serve up to thousands of clients in a small testbed network.

In this chapter, we provide a summary of the thesis contributions. We also revisit the initial claims posed in Chapter 1 and offer a critical discussion. While some of the claims have been confirmed throughout the thesis, others appeared to be conditional or applicable only to a limited extent. Finally, we discuss system elements that require further work and new research directions that this thesis opens for.

9.1 Summary of contributions

The contributions of this thesis include the following elements: a set of principles for design and implementation of a link layer caching system, a feasibility study of the proposed principles based on the CacheCast system, analysis of the environmental impact of the proposed architecture, and other related studies.

9.1.1 Principles

The fundamental contribution of this thesis is a set of principles for design and implementation of a link layer caching system. Based on these principles we have design the CacheCast system described in Chapter 3 CacheCast is the first system that addresses the problem of multicast transmission in the Internet using packet caches on links. Bearing in mind the difficulties of the previous multicast technologies, CacheCast is designed to use a minimum amount of resources and requires minimum changes to the network operation. CacheCast does not break the end-to-end relationship between a server and clients. This enables the server to perform authorisation, authentication, accounting, and also congestion control on a per-client basis which is necessary for content delivery and which the IP Multicast model does not provide. Furthermore, the server can communicate with clients located behind firewalls or NATs [64] because CacheCast preserves the end-to-end relationship between a server and clients.

CacheCast is a link layer caching technique that removes redundancy from packets using small caches on links. Unlike related systems, CacheCast distributes the caching burden between the source of data and the infrastructure of link caches. The source of redundant data must batch transmissions of the same data and annotate the data with information that simplifies redundancy elimination. This helps to considerably reduce both the storage space and the computational requirements of link caches. Thus, link caches can be implemented with small and fast SRAM memories which in turn allows to operate at line speed.

9.1.2 Feasibility study

Server support

The next key contribution of this thesis is the design of the server support described in Chapter 3. The server support provides a mechanism for an application to transmit the same data to multiple destinations using unicast connections. The individual send operations are batched in time and the server transmits a burst of packets, where only the first packet carries the data while the remaining packets are truncated to the header size. Packets created by the server support carry the CacheCast header that identifies redundant data. This greatly simplifies redundancy suppression on links.

We have implemented the server support as a system call named `msend` and an auxiliary tool that manages CacheCast connections. As input arguments, the `msend` system call takes a set of descriptors of open connections and a pointer to data. On invocation, the system call sends the data over the connections. However, packets created by the network stack are intercepted before transmission, batched, and transmitted in the form of a packet train. If a connection does not permit immediate packet transmission (e.g. due to congestion in a network), it is regarded as an error. The system call treats this and other errors as a failure in transmission and it returns to the calling application a list of connection descriptors that transmitted data successfully. The `msend` system call is implemented in Linux. Our evaluation presented in Chapter 6 indicates that it does not create additional burden on a server when compared to the standard `send` system call. On the contrary, when transmitting large blocks of data to multiple destinations, the `msend` system call outperforms the standard `send` system call. For example, when transmitting data using the Internet maximum transfer units, the `msend` system call can reduce the CPU load by 10-30%.

In order to benefit from CacheCast, existing applications must use the `msend` system call instead

of the `send` system call for data transmission. However, the required changes in the streaming server are often minimal. For example in Chapter 6 we show the `paraslash` media streaming application that transmits data using the `msend` system call. We made only minor changes in the `paraslash` software to enable CacheCast transmission.

Link cache

The next significant contribution is the link cache element presented in Chapter 3. The link cache is designed based on the principle that all complexity should be moved from the link cache to the source of redundant data. This results in a simple link cache architecture where the ingress side of a link has only a very small table of payload IDs and the egress side of the link keeps the related payloads in a small storage unit. The ingress side removes payloads based on payload ID matches and the egress side restores the payloads.

The storage and computational requirements of the link cache are minimal. Since link cache storage units are small, they can be implemented with fast but expensive SRAM memory that guarantees necessary access speed for packet processing on fast links. Moreover, the link cache remove and restore functions require only a few computations. Therefore, the link cache operation does not create a bottleneck in packet processing on a router (see Chapter 8).

We built a prototype implementation of the link cache elements in the Click modular router software which is described in Chapter 7. Even though this is a naive implementation, it indicates that link caches can greatly increase network throughput for single source multiple destination data transfers. In our testbed, this is further confirmed by a real application that can stream media up to thousands of clients (cf. Chapter 6).

9.1.3 Environmental impact

The next contribution of this thesis is the study of the link cache impact on fairness in the Internet. Link caches remove redundancy from single source multiple destination transfers and thereby provide more bandwidth for other data transfers. Congestion control algorithms that limit the transmission rate of a data flow estimate the flow throughput based on factors such as packet arrival time, packet size, loss ratio, or end-to-end delay. These parameters, however, are disturbed by the link cache operation. This, in turn, alters the operation of congestion control algorithms.

In Chapter 5 we performed a case study where we investigate the link cache impact on competition between TCP and TFRC flows for the bandwidth of a bottleneck link. The TCP flows carried non-redundant traffic and the TFRC flows carried redundant traffic originating from a single source. The simulations reveal that when a link cache removes redundancy from the TFRC flows, the TCP flows obtain much more of the bottleneck link capacity. However, when comparing the flows from the end-to-end perspective, we found that both TCP and TFRC increase proportionally their throughputs when link cache is operating. This indicates that the TCP fairness is preserved.

9.1.4 Other contributions

In the thesis we have also investigated the problem of synchronised transmission over multiple connections. When transmitting data, the server support requires that the data chunk is either

transmitted immediately or dropped by a connection. It cannot be queued when a congestion control algorithm temporally prohibits the transmission. Since the connection does not queue packets, it cannot transmit data immediately when congestion control algorithms permit it, but the connection remains idle waiting for a new data chunk to be sent by an application. Therefore, the connections controlled by the server support do not achieve the maximum throughput. We have investigated this problem in Chapter 5. We found that the throughput of connections controlled by the server support is reduced by approximately 10% when compared to standard connections.

Finally, the thesis presents the first live streaming application that uses the CacheCast system to deliver an audio stream to thousands of clients in a small network (described in Chapter 6). We have modified the paraslash audio streaming server to transmit audio files using the CacheCast server support. We experienced no difficulties when adapting the software to the CacheCast approach.

9.2 Critical review of claims

This section revisits the thesis claims introduced in Chapter 1. Based on the studies of the CacheCast system described throughout the thesis, we assess critically the claims and define the claims scope.

Claim 1: *A system of link caches can achieve near multicast bandwidth savings for a superposition of unicast connections.* We evaluated this claim in Chapter 4. CacheCast was compared to a "perfect" multicast which does not require any signalling to deliver data to receivers. Considering fully deployed CacheCast, the reduction in bandwidth saving when compared to the perfect multicast originates from (1) transmission of unique packet headers and (2) finite cache size. Depending on the payload to header ratio, the impact of unique packet headers varies from negligible to considerable. In order to reduce the negative impact of unique packet headers, transfered data should be encapsulated in the maximum transfer units. With this configuration, CacheCast bandwidth savings reach approximately 96% of the perfect multicast savings. Considering the impact of the finite cache size, it is especially severe for slow sources sending to the large number of destinations. However, it is a rare case. Usually, slow sources transmit to a few destinations, while well connected sources transmit to multiple destinations. For this configuration, the finite cache size does not reduce bandwidth savings.

A system of link caches can achieve near multicast bandwidth savings but under condition that data is transfered using the maximum transfer units. With the decreasing size of a transfer unit, the bandwidth savings diminish.

Claim 2: *A system of link caches requires server support in order to be feasible.* This claim follows from our initial investigation into the CacheCast system requirements presented in Chapter 3 and is supported throughout the thesis. The CacheCast system is designed to meet very strict constraints for packet processing elements in wired networks. To process packets at line rate, a link cache should employ simple algorithms and use fast memories. This further implies that the cache storage space should be minimised in order to be implemented with fast but expensive SRAM memory. The server support greatly reduces the required resources necessary to build a link cache. Therefore, it makes the system economically viable and technically doable. These arguments are broadly discussed in Chapter 8 and complemented

with a detailed comparison between the CacheCast system and the related system proposed by Anand et al. which do not employ any server support. In Chapter 6 we verified that the support can be implemented in OS and that it does not burden the server. Finally, the testbed evaluation indicates that the server support can be easily integrated into existing streaming applications, thus immediately reducing the amount of application traffic.

Claim 3: *A system of link caches is incrementally deployable.* Link caches that remove redundancy from packets on the link entry and reconstruct the packets on the link exit are transparent to network operation and thereby also incrementally deployable. However, the CacheCast design presented in Chapter 3 imposes an additional constraint on the link cache deployment. The server support element of the system annotates packets with information that simplify redundancy removal. In order to propagate this information in a network, CacheCast requires all routers on a packet path to be able to pass on this information. Therefore, link caches are incrementally deployable but only when deploying them hop-by-hop from sources towards receivers. In Chapter 4 we analysed the benefits of incremental deployment. We found that when deploying link caches a few hops from a source, we can remove a significant portion of redundancy. For example, when considering data transfer to a small number of destinations, link caches deployed over the first six hops can already remove 70% of what could be removed with the network wide cache deployment.

Claim 4: *A system of link caches maintains fairness with respect to bandwidth sharing on a bottleneck link.* Fairness in the Internet is based on the assumption that data flows traversing a bottleneck link should obtain equal shares of the bottleneck link capacity. This is achieved by congestion control algorithms that limit flow throughput and that are an integral part of many transport protocols. Based on the end-to-end delays and the packet loss rate, a transmission protocol adjusts the data transmission rate to the conditions on a bottleneck link. Introducing link caches into a network may disturb the protocol operation and thereby also fairness in the network. We evaluated the link cache impact in Chapter 5. The evaluation was performed in a bottleneck link topology where a number of TCP flows competed for the bottleneck link capacity with a number of TFRC flows. The TCP flows carried non-redundant traffic, while the TFRC flows carried CacheCast annotated redundant traffic. The results show that from the end-to-end perspective link cache does not disturb fairness in a network.

9.3 Future work

This thesis has explored the core issues related to the design and implementation of a caching system for single source multiple destinations transfer in the Internet. However, our reference system - CacheCast - requires further development. Furthermore, introducing CacheCast into the Internet opens new research challenges.

9.3.1 CacheCast elements

The CacheCast system presented in this thesis implements only the fundamental functionalities necessary for system operation. However, in order to be more than a research tool, CacheCast

requires further development of the server support and link cache elements.

Server support

The msend system call API provides very limited return information about the data transmission. After the system call execution, an application receives a bitmap that informs which connections were written successfully and which connections refused to transmit data. However, the application does not receive error codes related to the transmission failures. Therefore, the system call API should be modified to return all necessary information to identify the source of erroneous transmission.

The server support guarantees that the transmission of packet trains created by different processes is serialised on a link. We achieved this with a global lock in the Linux network device subsystem. When a single process holds the lock, the remaining processes cannot interrupt transmission of a packet train. However, this approach does not guarantee the serialised transmission of packet trains when the Linux traffic control (TC) subsystem is enabled. The TC subsystem processes packets after the network device subsystem. Therefore, it can re-order packets in packet trains and cause errors in transmission. This requires further work to ensure correct operation of the server support in any circumstances.

The msend system call supports two types of connection: DCCP and UDP. This list should be extended to embrace other transport protocols including, for example, the SCTP media streaming protocol. Furthermore, it is not given whether protocols like TCP can be supported by the system call. This requires further research.

Link cache

The configurable parameters of a link cache are the cache slot size and the cache capacity computed based on the associated link throughput. We have chosen the default values for these parameters; thus, the CMU and CSU elements that operate on the same link have consistent configurations. In order to build caches of different capacity, both cache elements must be configured manually. This is cumbersome and may cause inconsistencies when reconfiguring a large number of link caches. Therefore, a configuration protocol is required that can negotiate the link cache size and the slot size between the CMU and CSU elements automatically. The same protocol should also notify the server support about the presence of the CacheCast links. Thus, no additional tool for managing CacheCast connections would be required on a server.

The main bottleneck in the CSU operation is the copy operation from a payload store to packet payload. When a truncated packet arrives at the egress side of a link, the CSU element restores the packet payload from a local memory. CSU allocates necessary room for payload at the packet tail and it copies byte-by-byte the payload data from the local memory to the allocated space. However, the copy operation is slow and creates a bottleneck in packet processing on a router. To mitigate the problem, payloads should be attached to packet headers virtually. For example, in FreeBSD OS[1] a packet is constructed from a list of small buffers called *mbuf*. The *mbuf* packet structure can be used to attach payload to the packet header simply by linking together buffers that contain the packet header and the payload data.

[1] http://www.freebsd.org/

9.3.2 Future research directions

In the thesis, we have shown a simple usage scenario for CacheCast where a server streamed live content to multiple clients. The stream rate was constant and when a client could not receive the stream due to insufficient downlink speed it was disconnected. It would be more interesting to build a mechanism for handling heterogeneous clients. For example, a server could stream the same content in various qualities using different bit rates. Thus, clients with poor downlink speed would be switched to lower quality streams. A server could also adopt the streaming rate to an average client. These and many more policies can be explored in this context.

At present, CacheCast supports only UDP and DCCP transport protocols, which are unreliable. Therefore, in order to transfer data to multiple destinations using CacheCast, it is necessary to build a reliable transport mechanism on top of these protocols. This type of transport mechanism should either re-transmit lost data chunks or use erasure codes to enable a receiver to reconstruct original information. Furthermore, it should batch requests for the same data to benefit from CacheCast. In the past, researchers have proposed many schemes for reliable data transfer using IP Multicast. These could be adopted taking into account differences between CacheCast and IP Multicast.

CacheCast operates on point-to-point logical links. For example, three hosts A, B, and C connected by a broadcast medium (such as a coaxial cable) from the CacheCast perspective are connected with separate links A-B, A-C, and B-C. Therefore, when host A transmits the same data to hosts B and C, CacheCast does not suppress the second transfer even though host C has already received the data. Similarly, CacheCast does not suppress redundant transfers in link layer switched networks (such as switched Ethernet), since from the CacheCast perspective all network hosts are connected with separate links. Considering the growth of local area networks, it would be valuable to extend the CacheCast approach to remove redundancy from link layer transfers.

In the first generation of routers, packets are switched between input and output interfaces using shared memory. The router CPU reads a packet from an input interface directly to the shared memory where the CacheCast CSU element restores the redundant data that has been previously removed. When the packet output port is determined, the CacheCast CMU element removes from the packet the data that is present in the next hop cache and the packet is transfered to the output interface. Therefore, packets carrying redundant data are not transfered between the main memory and I/O interfaces. However, in the second and third generation of routes, packets are switched using a shared bus or a switch fabric. When a packet arrives at an input interface, the CSU element reconstructs the packet. Then the output port is determined and the packet carrying redundant data is transfered over a switch fabric to the output interface where the CMU element removes the data that is present in the next hop cache. Since, in the second and third generation of routers, packets carrying redundant data are transfered over a switch fabric, it may create congestion on the switch fabric. Therefore, these routers require additional mechanisms to mitigate this effect.

Bibliography

[1] L. Aguilar, “Datagram routing for internet multicasting,” *ACM SIGCOMM Computer Communication Review*, vol. 14, no. 2, pp. 58–63, 1984.

[2] D. Cheriton and S. Deering, “Host groups: A multicast extension for datagram internetworks,” in *Proceedings of the ninth symposium on Data communications*. ACM New York, NY, USA, 1985, pp. 172–179.

[3] R. Boivie, N. Feldman, Y. Imai, W. Livens, D. Ooms, and O. Paridaens, “Explicit multicast (xcast) basic specification,” IETF Internet draft, October 2000, rFC 5058.

[4] Y.-h. Chu, S. G. Rao, and H. Zhang, “A case for end system multicast (keynote address),” in *SIGMETRICS ’00: Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM, 2000, pp. 1–12.

[5] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica, “A data-oriented (and beyond) network architecture,” in *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, ser. SIGCOMM ’07. New York, NY, USA: ACM, 2007, pp. 181–192. [Online]. Available: http://doi.acm.org/10.1145/1282380.1282402

[6] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, “Networking named content,” in *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, ser. CoNEXT ’09. New York, NY, USA: ACM, 2009, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/1658939.1658941

[7] D. Ooms, W. Livens, and O. Paridaens, “Connectionless multicast,” *IETF Draft, draft-ooms-cl-multicast-00. txt, work in progress*, 1999.

[8] R. Boivie, N. Feldman, and C. Metz, “Small group multicast: A new solution for multicasting on the internet,” *IEEE Internet Computing*, vol. 4, no. 3, pp. 75–79, 2000.

[9] R. Boivie, N. Feldman, Y. Imai, W. Livens, and D. Ooms, “Explicit Multicast (Xcast) Concepts and Options,” RFC 5058 (Experimental), Internet Engineering Task Force, Nov. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc5058.txt

[10] B. Cain, S. Deering, I. Kouvelas, B. Fenner, and A. Thyagarajan, “Internet Group Management Protocol, Version 3,” RFC 3376 (Proposed Standard), Internet Engineering Task Force, Oct. 2002, updated by RFC 4604. [Online]. Available: http://www.ietf.org/rfc/rfc3376.txt

[11] C. Diot, B. Levine, B. Lyles, H. Kassem, and D. Balensiefen, “Deployment issues for the ip multicast service and architecture,” *Network, IEEE*, vol. 14, no. 1, pp. 78–88, Jan/Feb 2000.

[12] H. W. Holbrook and D. R. Cheriton, “Ip multicast channels: Express support for large-scale single-source applications,” *SIGCOMM Comput. Commun. Rev.*, vol. 29, pp. 65–78, August 1999. [Online]. Available: http://doi.acm.org/10.1145/316194.316207

[13] Y. Yang and S. Lam, “Internet multicast congestion control: A survey,” in *Proc. of ICT, Acapulco, Mexico*, 2000.

[14] S. Banerjee, B. Bhattacharjee, and C. Kommareddy, “Scalable application layer multicast,” in *SIGCOMM ’02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*. New York, NY, USA: ACM, 2002, pp. 205–217.

[15] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O’Toole, Jr., “Overcast: reliable multicasting with on overlay network,” in *OSDI’00: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*. Berkeley, CA, USA: USENIX Association, 2000, pp. 14–14.

[16] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh, “Splitstream: high-bandwidth multicast in cooperative environments,” in *SOSP ’03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 2003, pp. 298–313.

[17] “Pplive.” [Online]. Available: http://www.pptv.com/

[18] “Ppstream.” [Online]. Available: http://www.ppstream.com/

[19] “Sopcast.” [Online]. Available: http://www.sopcast.com/

[20] “Qqlive.” [Online]. Available: http://www.live.qq.com/

[21] J. H. Saltzer, D. P. Reed, and D. D. Clark, “End-to-end arguments in system design,” *ACM Trans. Comput. Syst.*, vol. 2, pp. 277–288, November 1984. [Online]. Available: http://doi.acm.org/10.1145/357401.357402

[22] J. T. Robinson and M. V. Devarakonda, “Data cache management using frequency-based replacement,” *SIGMETRICS Perform. Eval. Rev.*, vol. 18, pp. 134–142, April 1990. [Online]. Available: http://doi.acm.org/10.1145/98460.98523

[23] A. S. Tanenbaum and A. S. Woodhull, *Operating Systems Design and Implementation (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2005.

[24] S. Jiang, F. Chen, and X. Zhang, “Clock-pro: an effective improvement of the clock replacement,” in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ser. ATEC ’05. Berkeley, CA, USA: USENIX Association, 2005, pp. 35–35. [Online]. Available: http://portal.acm.org/citation.cfm?id=1247360.1247395

[25] A. Dingle and T. Pártl, “Web cache coherence,” *Comput. Netw. ISDN Syst.*, vol. 28, pp. 907–920, May 1996. [Online]. Available: http://dx.doi.org/10.1016/0169-7552(96)00020-7

[26] A. J. Smith, "Cache memories," *ACM Comput. Surv.*, vol. 14, pp. 473–530, September 1982. [Online]. Available: http://doi.acm.org/10.1145/356887.356892

[27] S. Przybylski, M. Horowitz, and J. Hennessy, "Characteristics of performance-optimal multi-level cache hierarchies," *SIGARCH Comput. Archit. News*, vol. 17, pp. 114–121, April 1989. [Online]. Available: http://doi.acm.org/10.1145/74926.74939

[28] A. J. Smith, "Disk cache—miss ratio analysis and design considerations," *ACM Trans. Comput. Syst.*, vol. 3, pp. 161–203, August 1985. [Online]. Available: http://doi.acm.org/10.1145/3959.3961

[29] A. Luotonen and K. Altis, "World-wide web proxies," *Comput. Netw. ISDN Syst.*, vol. 27, pp. 147–154, November 1994. [Online]. Available: http://portal.acm.org/citation.cfm?id=195676.195678

[30] N. T. Spring and D. Wetherall, "A protocol-independent technique for eliminating redundant network traffic," in *SIGCOMM '00: Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. New York, NY, USA: ACM, 2000, pp. 87–95.

[31] "Riverbed." [Online]. Available: http://www.riverbed.com/

[32] "Cisco." [Online]. Available: http://www.cisco.com/

[33] "Juniper." [Online]. Available: http://www.juniper.net/

[34] "Bluecoat." [Online]. Available: http://www.bluecoat.com/

[35] B. Aggarwal, A. Akella, A. Anand, A. Balachandran, P. Chitnis, C. Muthukrishnan, R. Ramjee, and G. Varghese, "Endre: an end-system redundancy elimination service for enterprises," in *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, ser. NSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 28–28. [Online]. Available: http://portal.acm.org/citation.cfm?id=1855711.1855739

[36] A. Anand, A. Gupta, A. Akella, S. Seshan, and S. Shenker, "Packet caches on routers: the implications of universal redundant traffic elimination," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 219–230, 2008.

[37] A. Anand, V. Sekar, and A. Akella, "Smartre: an architecture for coordinated network-wide redundancy elimination," *SIGCOMM Comput. Commun. Rev.*, vol. 39, pp. 87–98, August 2009. [Online]. Available: http://doi.acm.org/10.1145/1594977.1592580

[38] D. Wischik and N. McKeown, "Part i: buffer sizes for core routers," *SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 3, pp. 75–78, 2005.

[39] S. Iyer, R. R. Kompella, and N. McKeown, "Designing packet buffers for router linecards," *IEEE/ACM Trans. Netw.*, vol. 16, no. 3, pp. 705–717, 2008.

[40] A. Kirsch and M. Mitzenmacher, "The power of one move: Hashing schemes for hardware," in *IEEE INFOCOM 2008. The 27th Conference on Computer Communications*, 2008, pp. 106–110.

[41] J. Byers, M. Luby, M. Mitzenmacher, and A. Rege, "A digital fountain approach to reliable distribution of bulk data," in *Proceedings of the ACM SIGCOMM'98 conference on Applications, technologies, architectures, and protocols for computer communication*. ACM, 1998, pp. 56–67.

[42] J. Nagle, "Congestion Control in IP/TCP Internetworks," RFC 896, Internet Engineering Task Force, Jan. 1984. [Online]. Available: http://www.ietf.org/rfc/rfc896.txt

[43] R. Chalmers and K. Almeroth, "Developing a multicast metric," in *Global Telecommunications Conference, 2000. GLOBECOM '00. IEEE*, 2000, pp. 382–386.

[44] J. C. i. Chuang and M. A. Sirbu, "Pricing multicast communications: A cost-based approach," in *Telecommunication Systems*, 1998, pp. 281–297.

[45] R. C. Chalmers and K. C. Almeroth, "On the topology of multicast trees," *IEEE/ACM Trans. Netw.*, vol. 11, no. 1, pp. 153–165, 2003.

[46] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *Networking, IEEE/ACM Transactions on*, vol. 1, no. 4, pp. 397 –413, Aug. 1993.

[47] M. Handley, S. Floyd, J. Padhye, and J. Widmer, "TCP Friendly Rate Control (TFRC): Protocol Specification," RFC 3448 (Proposed Standard), Internet Engineering Task Force, Jan. 2003, obsoleted by RFC 5348. [Online]. Available: http://www.ietf.org/rfc/rfc3448.txt

[48] M. Karsten, J. Song, M. Kwok, and T. Brecht, "Efficient operating system support for group unicast," in *NOSSDAV '05: Proceedings of the international workshop on Network and operating systems support for digital audio and video*. New York, NY, USA: ACM, 2005, pp. 153–158.

[49] E. Lahav, M. Karsten, T. Brecht, W. Wang, and T. Zhao, "Group unicast for the real world," in *NOSSDAV '08: Proceedings of the 18th International Workshop on Network and Operating Systems Support for Digital Audio and Video*. New York, NY, USA: ACM, 2008, pp. 27–32.

[50] D. Plummer, "Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware," RFC 826 (Standard), Internet Engineering Task Force, Nov. 1982, updated by RFCs 5227, 5494. [Online]. Available: http://www.ietf.org/rfc/rfc826.txt

[51] T. Narten, E. Nordmark, and W. Simpson, "Neighbor Discovery for IP Version 6 (IPv6)," RFC 2461 (Draft Standard), Internet Engineering Task Force, Dec. 1998, obsoleted by RFC 4861, updated by RFC 4311. [Online]. Available: http://www.ietf.org/rfc/rfc2461.txt

[52] E. Kohler, M. Handley, S. Floyd, and J. Padhye, "Datagram congestion control protocol (DCCP)," *Work in progress*, 2003.

[53] R. Morris, E. Kohler, J. Jannotti, and M. Kaashoek, "The Click modular router," in *Proceedings of the seventeenth ACM symposium on Operating systems principles*. ACM New York, USA, 1999, pp. 217–231.

[54] F. Baker, "Requirements for IP Version 4 Routers," RFC 1812 (Proposed Standard), Internet Engineering Task Force, Jun. 1995, updated by RFC 2644. [Online]. Available: http://www.ietf.org/rfc/rfc1812.txt

[55] M. Afanasyev, D. G. Andersen, and A. C. Snoeren, "Efficiency through eavesdropping: link-layer packet caching," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 105–118. [Online]. Available: http://portal.acm.org/citation.cfm?id=1387589.1387597

[56] F. R. Dogar, A. Phanishayee, H. Pucha, O. Ruwase, and D. G. Andersen, "Ditto: a system for opportunistic caching in multi-hop wireless networks," in *Proceedings of the 14th ACM international conference on Mobile computing and networking*, ser. MobiCom '08. New York, NY, USA: ACM, 2008, pp. 279–290. [Online]. Available: http://doi.acm.org/10.1145/1409944.1409977

[57] N. Tolia, M. Kaminsky, D. G. Andersen, and S. Patil, "An architecture for internet data transfer," in *Proceedings of the 3rd conference on Networked Systems Design & Implementation - Volume 3*, ser. NSDI'06. Berkeley, CA, USA: USENIX Association, 2006, pp. 19–19. [Online]. Available: http://portal.acm.org/citation.cfm?id=1267680.1267699

[58] J. Santos and D. Wetherall, "Increasing effective link bandwidth by suppressing replicated data," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ser. ATEC '98. Berkeley, CA, USA: USENIX Association, 1998, pp. 18–18. [Online]. Available: http://portal.acm.org/citation.cfm?id=1268256.1268274

[59] U. Manber, "Finding similar files in a large file system," in *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*. Berkeley, CA, USA: USENIX Association, 1994, pp. 2–2. [Online]. Available: http://portal.acm.org/citation.cfm?id=1267074.1267076

[60] V. Jacobson, "Compressing TCP/IP Headers for Low-Speed Serial Links," RFC 1144 (Proposed Standard), Internet Engineering Task Force, Feb. 1990. [Online]. Available: http://www.ietf.org/rfc/rfc1144.txt

[61] M. Degermark, B. Nordgren, and S. Pink, "IP Header Compression," RFC 2507 (Proposed Standard), Internet Engineering Task Force, Feb. 1999. [Online]. Available: http://www.ietf.org/rfc/rfc2507.txt

[62] S. Casner and V. Jacobson, "Compressing IP/UDP/RTP Headers for Low-Speed Serial Links," RFC 2508 (Proposed Standard), Internet Engineering Task Force, Feb. 1999. [Online]. Available: http://www.ietf.org/rfc/rfc2508.txt

[63] C. Bormann, C. Burmeister, M. Degermark, H. Fukushima, H. Hannu, L.-E. Jonsson, R. Hakenberg, T. Koren, K. Le, Z. Liu, A. Martensson, A. Miyazaki, K. Svanbro, T. Wiebke, T. Yoshimura, and H. Zheng, "RObust Header Compression (ROHC): Framework and four profiles: RTP, UDP, ESP, and uncompressed," RFC 3095 (Proposed Standard), Internet Engineering Task Force, Jul. 2001, updated by RFCs 3759, 4815. [Online]. Available: http://www.ietf.org/rfc/rfc3095.txt

[64] K. Egevang and P. Francis, "The IP Network Address Translator (NAT)," RFC 1631 (Informational), Internet Engineering Task Force, May 1994, obsoleted by RFC 3022. [Online]. Available: http://www.ietf.org/rfc/rfc1631.txt