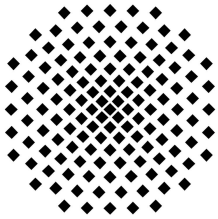


Universität Stuttgart

Fakultät Informatik, Elektrotechnik und Informationstechnik

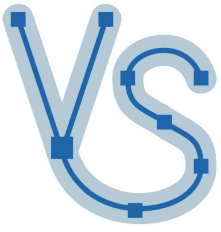
Diplomarbeit Nr. 2749

Protocol for Epoch Switching in a Distributed Time Virtualized Emulation Environment

Alexander Egorenkov

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. Kurt Rothermel
Betreuer:	Andreas Grau
begonnen am:	3. März 2008
beendet am:	2. September 2008
CR-Nummer:	C.2.1, C.2.2, C.2.5

Institut für Parallele
und Verteilte Systeme
Abteilung Verteilte Systeme
Universitätsstraße 38
D-70569 Stuttgart



Abstract

In this diploma thesis an efficient protocol with very small latency for group communication in Distributed Time Virtualized Emulation Environment (DTVEE) is designed and developed. DTVEE is a PC cluster and provides a distributed network emulation environment for large-scale distributed applications and network protocols. It allows to emulate network scenarios with thousands of nodes running unmodified software implementations. DTVEE uses node and time virtualization in order to support very large network topologies, to maximize hardware utilization and to minimize the time needed for network experiments. DTVEE can run an experiment with a factor (called time dilation factor, TDF) slower or faster than real-time and, therefore, emulate more CPU and network resources. It is better to adapt TDF to the current load in order to achieve best resource utilization and to shorten the runtime of an experiment. Therefore, continuous adaptation of TDF is required because the demand on the CPU and network resources changes during an experiment. The period of time between two TDF changes is called epoch. In this work, a protocol, that switches all nodes belonging to an experiment to a new epoch, shall be developed and evaluated. Since running nodes with a different TDF in the same experiment adulterates emulation results, the protocol has to change the TDF simultaneously.

Zusammenfassung

Diese Diplomarbeit hat den Entwurf und die Entwicklung eines effizienten Protokolls mit sehr kleiner Latenzzeit zur Gruppenkommunikation in Distributed Time Virtualized Environment (DTVEE) zum Ziel. DTVEE ist ein PC-Cluster and stellt eine verteilte Netzwerkemulationsumgebung für umfangreiche verteilte Anwendungen und Netzwerkprotokolle zur Verfügung. Es erlaubt uns Netzwerkszenarien mit tausenden von Knoten, die unmodifizierte Softwareimplementierungen ausführen, zu evaluieren. DTVEE verwendet Knoten- und Zeitvirtualisierung, um sehr große Netzwerktopologien zu unterstützen, Ausnutzung von Harware zu maximieren und die Zeit für Experimente zu minimieren. DTVEE kann ein Experiment um eine Konstante (TDF, time dilation factor) schneller oder langsamer laufen lassen und so mehr CPU- und Netzwerk-Ressourcen zu emulieren. Es ist besser TDF an die aktuelle Last anzupassen, um die beste Ausnutzung von Ressourcen zu erreichen und die Laufzeit eines Experimentes zu verkürzen. Deswegen is eine ständige Adaptation von TDF is notwendig, weil die Nachfrage nach CPU- und Netzwerk-Ressourcen sich während eines Experiments verändert. Die Zeitperiode zwischen zwei Änderungen von TDF wird Epoche genannt. In dieser Arbeit soll ein Protokoll entwickelt und evaluiert werden, das alle Klusterknoten, die zu einem Experiment gehören, in eine neue Epoche umschlatet. Weil zu einem Experiment gehörende Klusterknoten mit unterschiedlichen TDF die Ergebnisse des Experiments verfälschen können, soll das Protokoll TDF auf den Klusterknoten gleichzeitig umschalten.

Acknowledgments

I would like to sincerely thank my advisor Andreas Grau for his help, support and guidance during my diploma thesis. He put me on the road to doing good research and his easy accessibility to discuss various issues was invaluable during my research.

Contents

List of Tables vii

List of Figures ix

1. Introduction 1
- 1.1. Motivation . . . 1
- 1.2. Purpose of Study . . . 3
- 1.3. Outline . . . 3

2. Distributed Time Virtualized Emulation Environment (DTVEE) 5
- 2.1. System Model . . . 5
  - 2.1.1. System Architecture . . . 5
    - 2.1.1.1. PC Cluster . . . 5
    - 2.1.1.2. Time Virtualized Emulation Environment (TVEE) . . . 6
    - 2.1.1.3. Network Emulation . . . 7
  - 2.1.2. Epoch-based Virtual Time Concepts . . . 7
  - 2.1.3. System Properties . . . 8
- 2.2. Protocol Requirements . . . 9

3. Related Work 11
- 3.1. Real-time Introduction . . . 11
- 3.2. Real-time Scheduling . . . 11
  - 3.2.1. Real-time Linux . . . 13
- 3.3. Real-time Communication . . . 14
  - 3.3.1. Token Bus and Token Ring . . . 14
  - 3.3.2. Transport Protocols . . . 16
    - 3.3.2.1. Real-time Transport Protocol . . . 16
    - 3.3.2.2. RTCast . . . 16
  - 3.3.3. Ethernet-Based Approaches . . . 18
    - 3.3.3.1. Switched Ethernet . . . 19
    - 3.3.3.2. Token-Based Approaches . . . 20
  - 3.3.4. Wireless-Based Approaches . . . 22
    - 3.3.4.1. Wireless Rether . . . 22
    - 3.3.4.2. WRTP . . . 23
  - 3.3.5. Real-time Network Stacks . . . 24
    - 3.3.5.1. RTnet . . . 24

4. Design Issues **27**
4.1. Fundamentals of the Linux Kernel 2.6.18 Network Stack 27
4.1.1. The sk_buff structure 27
4.1.2. The net_device structure 29
4.1.3. Packet Reception 31
4.1.3.1. Link Layer Multicast 32
4.1.3.2. Layer 3 Protocol Handlers 33
4.1.3.3. Layer 4 Protocol Handlers 35
4.1.4. Packet Transmission 35
4.1.4.1. Frame Transmission 36
4.1.4.2. Transmission of IPv4 Packets 37
4.1.5. Intermediate Functional Block (IFB) Device 38
4.2. Possible Approaches to Protocol Design 38
4.2.1. User-space vs. Kernel-space Implementation 38
4.2.2. Simultaneous Packet Reception 39
4.2.3. Network Layer 40
4.2.4. Packet Latency Minimization 41
4.2.5. Simultaneous Independent Experiments 43

5. Protocol Design **44**
5.1. Architecture 44
5.2. Generic Part 45
5.2.1. Generic Protocol Module 45
5.2.1.1. Protocol Demultiplexing 46
5.2.1.2. Packet Priority and Latency 46
5.2.1.3. External Interface 47
5.2.1.4. /proc Interface 48
5.2.2. Experiment Module 48
5.2.2.1. External Interface 49
5.2.2.2. /proc Interface 49
5.3. PLACE 49
5.3.1. TDF Sender Module 50
5.3.1.1. External Interface 50
5.3.1.2. /proc Interface 51
5.3.2. TDF Receiver Module 51
5.3.2.1. External Interface 52
5.3.2.2. /proc Interface 52
5.3.3. Sequence Diagrams 53
5.3.3.1. Send TDF Change Request 53
5.3.3.2. Receive TDF Change Request 53
5.3.3.3. Join Experiment 53
5.3.3.4. Leave Experiment 54

6. Protocol Implementation **56**
6.1. Generic Part . . . 56
6.1.1. Generic Protocol Module . . . 56
6.1.1.1. Protocol Demultiplexing . . . 57
6.1.1.2. Packet Priority and Latency . . . 59
6.1.1.3. Module Parameters . . . 60
6.1.1.4. /proc Interface . . . 61
6.1.2. Experiment Module . . . 61
6.1.2.1. Module Parameters . . . 62
6.1.2.2. /proc Interface . . . 62
6.2. PLACE . . . 62
6.2.1. TDF Sender Module . . . 63
6.2.1.1. Module Parameters . . . 63
6.2.1.2. /proc Interface . . . 64
6.2.2. TDF Receiver Module . . . 64
6.2.2.1. Module Parameters . . . 65
6.2.2.2. /proc Interface . . . 65

7. Evaluation **67**
7.1. Evaluation Goals . . . 67
7.2. Evaluation Tools . . . 67
7.2.1. Network Load Generating . . . 67
7.2.2. CPU Load Generating . . . 68
7.2.3. Measurement of Packet Delay . . . 69
7.2.4. Measurement of CPU Load . . . 69
7.2.5. Protocol for Evaluation . . . 70
7.3. Scenario Description . . . 70
7.3.1. Scenario: Performance . . . 70
7.3.2. Scenario: Packet Delay and Packet Delay Variation . . . 71
7.3.3. Scenario: Packet Delay in Ingress Queue of Switch . . . 73
7.4. Evaluation Results . . . 74
7.4.1. Scenario: Performance . . . 74
7.4.2. Scenario: Packet Delay and Packet Delay Variation . . . 75
7.4.3. Scenario: Packet Delay in Ingress Queue of Switch . . . 82
7.5. Discussion of Results . . . 85

8. Conclusion **87**
8.1. Summary . . . 87
8.2. Limitations and Future Work . . . 88

Bibliography **93**

A. Appendix **94**
A.1. PLACE Use Cases 94
A.1.1. TDF Sender Module Use Cases 94
A.1.2. TDF Receiver Module Use Cases 95
A.2. Evaluation Results 96
A.3. Statement 102

List of Tables

7.1. Subscenarios for scenario "Packet delay and packet delay variation" . . . 72
7.2. Configuration of receiver nodes for scenario "Packet delay and packet delay variation" (- – no load, x – load) 73

List of Figures

2.1. TVEE Architecture 6
2.2. Epoch-based virtual time concepts 8

3.1. Wireless Rether Architecture 23
3.2. RTnet Architecture 24

4.1. Packet data storage 28
4.2. ptype_base and ptype_all data structures 34
4.3. Layer 4 protocol table 35

5.1. PLACE Architecture 44
5.2. TDF Receiver Module State Machine 51
5.3. Send TDF Change Request Sequence Diagram 53
5.4. Receive TDF Change Request Sequence Diagram 54
5.5. Join Experiment Sequence Diagram 54
5.6. Leave Experiment Sequence Diagram 55

6.1. Generic protocol header 56
6.2. gtype_base and gtype_all data structures 58
6.3. Packet priorities and scheduling 59
6.4. PLACE protocol header 63
6.5. expseq_base data structure 64

7.1. Topology for scenario "Performance" 71
7.2. Topology for scenario "Packet delay and packet delay variation" 71
7.3. Topology for scenario "Packet delay in ingress queue of switch" 73
7.4. Sender performance 74
7.5. Receiver performance 75
7.6. Packet delay variation in Xen 3 with credit scheduler and 30 ms time slices 75
7.7. Packet delay variation in Xen 3 with credit scheduler and 1 ms time slices 76
7.8. Subscenario 0 results 77
7.9. Subscenario 3 results 78
7.10. Subscenario 5 results 79
7.11. Subscenario 8 results 80
7.12. Subscenario 10 results 81
7.13. Packet delay distribution without TBF egress qdisc 82
7.14. Packet delay distribution with TBF egress qdisc 84

A.1. TDF Sender Module Use Cases 94
A.2. TDF Receiver Module Use Cases 95
A.3. Subscenario 1 results 96
A.4. Subscenario 2 results 97
A.5. Subscenario 4 results 98
A.6. Subscenario 6 results 99
A.7. Subscenario 7 results 100
A.8. Subscenario 9 results 101

Chapter 1.

Introduction

This chapter is composed of three sections. In the first section of this chapter, a short motivation for the problem of this diploma thesis is introduced. The second section of this chapter describes detail information of the problem of this diploma thesis. And the third section is an outline of the remainder of this document.

1.1. Motivation

Today, the ability to test, verify and evaluate a new network protocol or a brand new peer-to-peer application before distribution has become a very important task that takes a significant amount of development time. In dynamic large-scale distributed applications such as Chord peer-to-peer system [SMK+01] that generate large amounts of network traffic, the network plays an important part in overall application performance. These large-scale distributed applications run on thousands of cooperating nodes spread across the Internet. Therefore, deploying, administering, testing and evaluating of such systems "in the wild" becomes very difficult, expensive and in most cases impossible assignment. Further, results obtained from such deployments on the Internet are not reproducible or predictive of future behavior because it is impossible for researchers to control and change wide-area network conditions. Besides, evaluation approaches in realistic environment are restricted to existing technologies. However, there are another two known techniques to test and evaluate new network protocols or applications: network simulation and network emulation [GRL05, Fal99]. These techniques are not competing techniques, they can be used both for testing and evaluating and they complement each other in many ways. Therefore, network simulations and emulations have been used very often to explore the behaviour and the characteristics of network protocols and large-scale distributed applications. Network simulation and emulation enable larger experiment scenarios than obtainable using real elements alone.

Network simulation offers a low-cost, flexible, controllable and repeatable environment for testing and evaluating of network protocols and applications. The provided network simulation environment can be easily configured and has some level of abstraction [GRL05]. The notion of time in network simulation environments is virtual and is independent of real-time. Virtual time makes experiments controllable and repeatable. However, abstractions can compromise the results of a network simulation and make them useless. Further, network simulations do not support direct execution of software prototypes, they must be reimplemented in the network simulation environment

[GRL05].

Network emulation is a hybrid approach for testing and evaluating network protocols and large-scale distributed applications. It combines aspects of evaluation approaches in realistic environment and network simulation. Network emulation consists of real elements – such as implementation of software prototypes and network protocols – and simulated elements – such as network links and nodes. One important difference between network simulation and emulation is that network emulation supports direct execution of software prototypes and network protocols. Another important difference is that network emulations run in real-time. It is impossible to repeat an order of events in a network emulation due to the nondeterministic nature of its events and, often, a physically distributed environment infrastructure [GRL05].

Current advances in computing and networking technologies allow network emulators to test and evaluate simple topologies on a single node [VYW+02]. Virtualization techniques can be used to support the emulation of complete network stacks and operating systems. This technique is called node virtualization [AH06]. However, the computing and networking capacity of a single node is not sufficient to emulate topologies with thousands of participating nodes or large-scale peer-to-peer systems with thousands of instances. There are some possibilities to further increase the capacity of network emulation.

One of the possibilities to further increase the capacity of network emulation is a distributed network emulator environment, a cluster with nodes that are interconnected by a very fast local area network [AH06]. Hundreds of virtual nodes or test objects are distributed to each physical cluster node and multiplexed thanks to virtualization on these physical nodes. This approach allows us to emulate large topologies by segmenting this topology and distributing each segment to a single cluster node. However, each cluster node has a limited capacity of processing power and network and, therefore, the size of the supported scenarios has boundaries.

Another known virtualization technique is called time virtualization [GYM+06]. This virtualization technique allows us to scale computing power and network capacity. Time virtualization means that the time on a time virtualized node runs slower or faster than the real-time by a factor. This factor is known as time dilation factor (TDF) [GYM+06]. By slowing down the real-time by a factor, CPU and network appear to be faster to operating systems and applications. Time virtualization makes possible the emulation of physical resources that are not currently available.

The next step in increasing the capacity of network emulation is to combine both node and time virtualization. This virtualization technique is called hybrid virtualization. In hybrid virtualization, node virtualization is used for multiplexing isolated instances of test objects or virtual nodes on a physical cluster node and time virtualization is used for increasing the number of virtual nodes per single node of cluster. Slowing down the real-time allows us to further increase the number of test objects in an experiment. However, if physical cluster nodes do not use their physical resources at maximum, we can shorten the time of the network experiment by accelerating the time by a factor during the experiment and so maximizing the utilization of physical resources. During an experiment, physical cluster nodes can be overloaded and consequently experiment

results will be adulterated. Therefore, during an experiment, load of all cluster nodes must be watched and TDF must be adjusted on each single cluster node if necessary. To avoid adulteration of experiment results, TDF of each cluster node must be adjusted simultaneously. A network protocol is needed to simultaneously change TDF on all cluster nodes of an experiment in a network emulation environment.

In the next section, the purpose of the thesis is defined in detail.

1.2. Purpose of Study

The NET (Network Emulation Testbed) project of the Institute of Parallel and Distributed Systems (IPVS) at the University of Stuttgart [NET08] has established a network emulation system for computer networks at the Distributed Systems department. The emulation system consists of a PC cluster with flexibly configurable hardware and software tools. The system makes possible the emulation of specified network properties and the comparative performance analysis of network protocols and distributed applications.

Each cluster node in the network emulation environment runs a Time Virtualized Emulation Environment (TVEE) that is based on Xen Virtual Machine Monitor (VMM) or Hypervisor [DFH+03] and Linux OpenVZ [Ope08]. TVEE uses the hybrid virtualization technique and combines both node and time virtualization. In previous work, Xen was extended with a possibility to change TDF of a cluster node that runs this time virtualized emulation environment. Xen VMM provides time virtualization and Linux OpenVZ node virtualization. Currently it is possible to run experiments in this network emulation environment with thousands of virtual nodes and to test large-scale distributed applications such as BitTorrent but it is not yet possible to simultaneously change TDF on all cluster nodes during an experiment.

In this thesis, an efficient and low-latency network protocol providing one-to-many communication, that changes TDF of all nodes belonging to an experiment, shall be developed and evaluated. The network protocol has to change TDF of all cluster nodes, which belong to the same experiment, simultaneously because different TDFs on the cluster nodes in the same experiment mean adulteration of experiment results and, therefore, the experiment results become useless.

This diploma thesis presents the design, implementation and performance evaluation of PLACE, a **P**rotocol for **L**atency **A**ware **C**hanging of **E**pochs.

1.3. Outline

The reminder of this thesis is structured as follows.

Chapter 2 describes the architecture and the properties of the network emulation environment and TVEE. After that, Chapter 2 documents the requirements of the network protocol.

Chapter 3 presents related work and then shows the differences to this diploma thesis. Further, Chapter 3 points out the contribution of this diploma thesis.

Chapter 2 provides the base for design approaches studied in Chapter 4.

Chapter 5 describes and explains the architecture and design of the network protocol and presents the components of the network protocol and their behaviour.

Chapter 5 is the base for the implementation of the network protocol which is discussed in Chapter 6

In Chapter 7 procedures and results of the evaluation are described.

Finally, Chapter 8 gives a summary of the diploma thesis and shows possible extensions and enhancements of the network protocol developed in this diploma thesis.

Chapter 2.

Distributed Time Virtualized Emulation Environment (DTVEE)

This chapter describes the architecture and the properties of the Distributed Time Virtualized Emulation Environment (DTVEE). After that, the requirements of PLACE are documented.

2.1. System Model

DTVEE provides a distributed network emulation environment for large-scale distributed applications and network protocols. It allows to emulate network scenarios with thousands of nodes and to evaluate unmodified software implementations. DTVEE uses node and time virtualization in order to support very large network topologies, to maximize hardware utilization and to minimize the time needed for network experiments.

In the following sections, the overall architecture and the properties of DTVEE are described in detail.

2.1.1. System Architecture

The overall architecture of DTVEE is described in this section.

2.1.1.1. PC Cluster

DTVEE consists of 64 PC cluster nodes. Each cluster node of DTVEE is a Pentium4 2.4GHz machine with 512MB RAM and has two Ethernet network interface cards (NIC): Intel PRO/100 100MB/s and RealTek RTL8169 1GB/s. DTVEE provides two separate local area networks which interconnect all cluster nodes of DTVEE. The first Ethernet LAN is used only to control network experiment and the second Ethernet LAN is used only for network experiments. DTVEE uses one Cisco Catalyst 3550 switch and three Cisco Catalyst 2950 switches to build the control network and one Foundry Networks Fastiron II Plus switch with 64 ports to build the experiment network. DTVEE uses two separate networks in order to isolate control communications from data traffic generated during network experiments.

2.1.1.2. Time Virtualized Emulation Environment (TVEE)

Each cluster node of DTVEE runs TVEE. TVEE is a hybrid virtualization system for scaling network emulation to large topology sizes. Hybrid virtualization combines node virtualization as well as time virtualization.

Node virtualization allows one physical cluster node to emulate several virtual nodes in a network experiment and, therefore, to increase the size of possible network experiments beyond the number of the cluster nodes in DTVEE.

In TVEE, node virtualization is achieved through OpenVZ [Ope08]. OpenVZ is a lightweight virtualization system that provides independent, secure and isolated containers (virtual nodes) on a single physical machine. Each container appears like a separate single host and has its own users, root access, files, memory, IP addresses, applications and can be rebooted independently from other containers. OpenVZ is based on a modified Linux 2.6 kernel. Currently, TVEE uses OpenVZ that is based on a modified Linux 2.6.18 kernel. Each container in OpenVZ has its own protocol stack which consists of network, transport and application layer. The protocol stack of each virtual node is stacked on top of the virtual Ethernet device. TVEE uses software bridges to connect virtual nodes on same cluster node. In order to provide communication between virtual nodes of different cluster nodes, the uplink of the software bridge is connected to the Ethernet NIC of the cluster node.

Time virtualization allows to further increase the number of virtual nodes per single physical node by slowing down the real-time of the physical node. In that case, a network experiment runs slower but with this approach it is possible to emulate very large network topologies. On the other hand, if the resources of physical nodes aren't utilized at maximum, it is possible to shorten the time of a network experiment by accelerating the real-time of the cluster nodes which participate in the experiment.

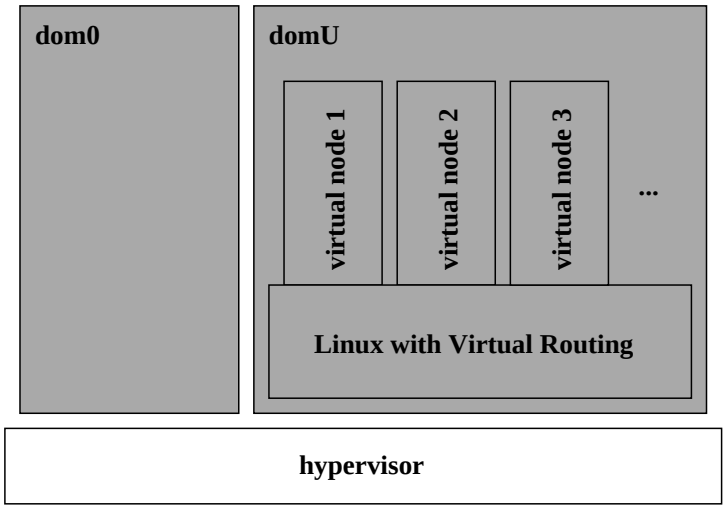


Figure 2.1.: TVEE Architecture

In TVEE, time virtualization is achieved through Xen. Xen is a virtual machine mon-

itor (VMM) or a hypervisor [DFH+03]. Xen uses the paravirtualization technique and runs directly on hardware of a single physical cluster node. The Xen hypervisor does not emulate the hardware for guest systems (also called domains) but allows guest systems to directly access hardware with small overhead. Therefore, the paravirtualization approach is very efficient, contrary to full hardware emulation approach. TVEE uses Xen 3.1.0. The dom0 domain is the first guest system started by the Xen hypervisor on boot. This domain has special privileges, it can start and stop new guest systems, which are called domU domains, and access the hardware directly. In TVEE, the Xen hypervisor of each cluster node runs two domains: the dom0 domain and one domU domain. The domU domain runs the before mentioned OpenVZ system. The Figure 2.1 shows the architecture of TVEE.

The original Xen hypervisor does not support time virtualization. Thus, the interface of the Xen hypervisor was previously extended with a new hypercall for time virtualization. Domains communicate with the Xen hypervisor using hypercalls. The new hypercall of the Xen hypervisor allows us to slow down or to accelerate the real-time of the domain domU by a factor which is called time dilation factor (TDF).

2.1.1.3. Network Emulation

In order to emulate various network properties, the network emulation tool is integrated into the device driver of the virtual Ethernet device which is used by each virtual node. The network emulation tool is placed inside the device driver of the virtual Ethernet device because it allows back pressure in case of saturation of the emulated network. With the network emulation tool it is possible to emulate frame delays, bandwidth limitation and frame loss. All these parameters can be configured for each pair of sender and receiver individually.

2.1.2. Epoch-based Virtual Time Concepts

During a long lasting network experiment, the resource utilization of physical nodes varies over time. By using virtual time based on discrete events, DTVEE could maximize resource utilization of physical nodes. By using a constant TDF, DTVEE does not need any synchronization during experiment but it results in a low average resource utilization of physical nodes.

Therefore, DTVEE uses epoch-based virtual time in order to maximize resource utilization during network experiments and to avoid high synchronization overhead. A network experiment is divided in epochs. During an epoch, the TDF on all physical nodes which participate in the experiment remains unchanged. At an epoch transition, the TDF of these cluster nodes is changed to a new TDF. Epoch-based virtual time allows to maximize resource utilization and to minimize time needed for a network experiment by selecting optimal TDF value and epoch duration for a given load.

During a network experiment, all physical nodes which participate in the experiment periodically send load reports to the central coordinator. The central coordinator can detect when physical nodes are overloaded or underloaded. When physical nodes are

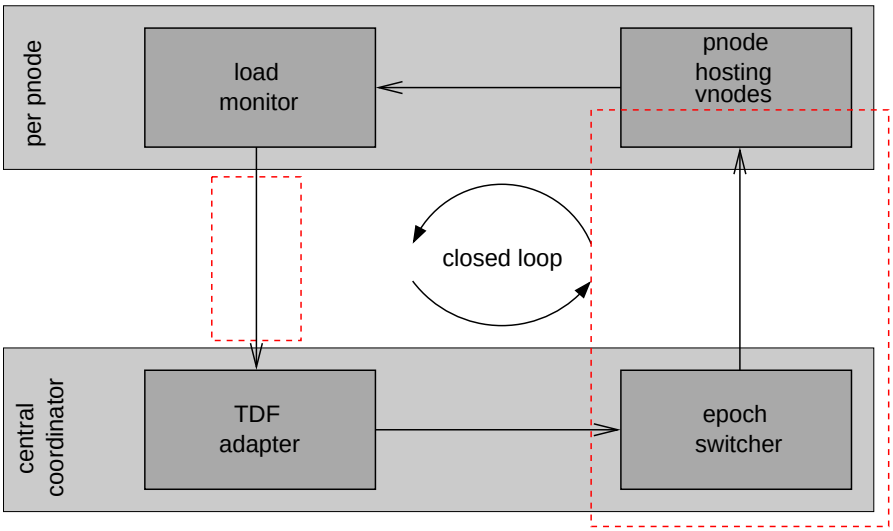


Figure 2.2.: Epoch-based virtual time concepts

overloaded or underloaded, the central coordinator computes a new optimal TDF for all physical nodes participating in the experiment and initiates a new epoch switch.

The Figure 2.2 shows the interactions between the central coordinator and physical cluster nodes which participate in an experiment. Every cluster node, which participate in an experiment, runs a load monitor that monitors the resource utilization of the cluster node and periodically sends reports to the central coordinator. The TDF adapter, which is running on the central coordinator, receives these reports and makes a decision based on these reports to initiate an epoch switch. The TDF adapter uses the PLACE protocol to distribute a new TDF to the cluster nodes of the experiment. The dashed line in the Figure 2.2 shows which tasks are undertaken by PLACE. The PLACE protocol provides the communication infrastructure for load reports, which are send by the load monitor on the cluster nodes, and TDF change requests, which are sent by the epoch switcher on the central coordinator.

2.1.3. System Properties

In this section, various important system properties are described which must be considered during the design of PLACE.

Under heavy network load conditions, a cluster node of DTVEE can occasionally drop a received frame because not enough memory for the received frame is available or the ingress queue of the cluster node is overflowed.

Frame delays are non-deterministic because an Ethernet switch through which a sent frame passes can affect the delay of the frame. Non-deterministic transition time through the standard Linux network stack of a cluster node also affects the delay of the frame. The length of the ingress and outgress queues of the standard Linux network stack can vary depending on actual network load and, thus, can also affect the frame delay.

DTVEE uses switched Ethernet and, therefore, frame delays aren't possible because of frame collisions.

A protocol implementation in user space has also to deal with process scheduling in Linux because the standard Linux scheduler is non-deterministic. In that case, the frame delay can be further affected by the standard non-deterministic Linux scheduler.

2.2. Protocol Requirements

Cluster nodes participating in an experiment can be overloaded or underloaded during the experiment. The main goal of PLACE is simultaneous TDF switching on all cluster nodes which participate in same experiment.

The overloaded cluster nodes can adulterate the results of the experiment. In order to avoid overloading of cluster nodes, the real-time of the cluster nodes participating in the experiment is slowed down by a factor. Cluster nodes with the slowed down real-time appear to have increased CPU and network capacity. The disadvantage of slowing down the real-time of the cluster nodes is the increased experiment time.

The underloaded cluster nodes do not cause the adulteration of experiment results but by speeding up the real-time of the cluster nodes participating in an experiment it is possible to maximize the hardware utilization of the nodes and, therefore, to finish the experiment more quickly.

It is very important that all cluster nodes which participate in same experiment have the same TDF because different TDFs will adulterate the experiment results. Therefore, it is very important to switch TDF of all cluster nodes participating in same experiment simultaneously.

Another important aspect of the required protocol is low latency. Low latency guarantees that the protocol quickly responds to overloading of cluster nodes and, therefore, will avoid the adulteration of experiment results.

The PLACE protocol will be used only in the control network of DTVEE.

The main requirements for PLACE are listed in the following:

1. Simultaneous TDF changing on all cluster nodes participating in same experiment
2. Low latency between initiating a TDF change request and TDF switching on all cluster nodes participating in same experiment, latency has to be smaller than 1 ms
3. No packet loss
4. Sending rate at least 1000 packets per second (1 packet every millisecond)
5. Concurrent TDF change requests have to be serialized
6. Implementation has to be generic and has to support other problems which require low-latency 1-to-n communication

7. Implementation has to provide an interface for user-space programs as well as kernel-space tasks, the interface has to provide a possibility to send a TDF change request, to obtain various statistical and debugging information, such as current TDF, time of last TDF change, number of TDF changes etc.
8. Target architecture is Xen/x86_32 but the protocol implementation must be easy portable to other architectures and, therefore, architecture-dependent code isn't allowed
9. Support for 65536 simultaneous independent experiments
10. Support of 8 priorities for the protocol packets
11. A cluster node can participate in not more than one experiment
12. Target kernel is Linux 2.6.18
13. Source code has to be well-commented

Chapter 3.

Related Work

In this chapter, related work of this diploma thesis is introduced. The focus of the diploma thesis is developing and evaluating an efficient and low-latency network protocol for one-to-many communication to change TDF of cluster nodes simultaneously. Therefore, network protocols and projects on related topics are discussed in the following sections. First, a short introduction to real-time systems will be presented. Second, approaches to real-time process scheduling will be introduced. Third, hardware- and software-based approaches to real-time communication will be described.

3.1. Real-time Introduction

Real-time means that a specified deadline which is triggered by an event has to be carried out unconditionally and guaranteedly or else a critical state will be entered or a catastrophe will occur. Further, the event handling routine must complete within the specified time in order to be able to response in time to new events. Real-time capability does not define a specific time but it promises that there is a defined time within which the system is able to answer to an event. The correctness of a real-time system does not only depend on correct results but also that the results to be produced within a specified deadline period [SGG04].

We can distinguish between hard and soft real-time.

If a deadline is missed in a hard real-time system, then a critical state is entered or a catastrophe occurs. Hard real-time is a vital part of many computing and control systems today. An example of a hard real-time system is an aircraft flight control system. An aircraft flight control system is a hard-real time system because a single flight error is fatal.

If a deadline is missed in a soft real-time system, then only the quality of service is reduced. Soft real-time normally can be seen in operating systems or in applications. An example of a soft real-time system is a video streaming system, e.g. Internet Protocol Television (IPTV).

3.2. Real-time Scheduling

A real-time operating system is an operating system which guarantees not only that the computed results are correct but also that the results are produced within a specified

deadline period. Results which are produced after the deadline are basically useless. Real-time operating systems are of two types: hard and soft real-time operating systems. A hard real-time requires that the critical real-time tasks are completed within their deadlines. A soft real-time system is less restrictive and guarantees only that the real-time tasks will receive higher priority than non-real-time tasks. Most modern general-purpose operating systems, such as Linux and Windows, are soft real-time operating systems and, therefore, can't be directly used for problems with hard real-time constraints [SGG04].

An operating system has to implement the following important features to be considered a real-time operating system: preemptive priority-based scheduling, preemptive kernel and minimized latency.

A priority-based scheduling algorithm is one of the most important characteristics of a real-time operating system. The priority-based scheduling algorithms assign each task a priority that is based on the importance of the task. The real-time tasks are assigned higher priorities than non-real-time tasks. A preemptive priority-based scheduling algorithm can withdraw the CPU from a lower-priority task if a higher-priority task becomes runnable. An operating system which provides a preemptive priority-based scheduling can only guarantees a soft real-time functionality. For example, Linux, Solaris and Windows provide a preemptive priority-based scheduling. These operating systems assign the highest priorities to the real-time tasks.

Scheduling for hard real-time operating systems can be classified into two types: static and dynamic scheduling. Static schedulers make decisions at compile time. A run-time schedule is generated before the real-time system runs and based on task parameters, such as maximum execution times and deadlines. The advantage of the static scheduling is the small run-time overhead. One example of a static real-time scheduling algorithm is Rate-Monotonic Scheduling. Dynamic scheduling makes decisions at run-time and, therefore, is very flexible and adaptive. But dynamic schedulers may cause significant overhead because of run-time processing. One example of a dynamic real-time scheduling algorithm is Earliest Deadline First (EDF). Preemptive or nonpreemptive scheduling of tasks is possible with both static and dynamic scheduling. In preemptive scheduling, the currently executing task will be preempted upon arrival of a higher priority task. In nonpreemptive scheduling, the currently executing task will not be preempted until completion.

Tasks running in kernel mode can't be preempted in nonnpreemptive kernels. Nonpreemptive kernels aren't well suitable for real-time applications because tasks in kernel mode may spend several milliseconds during system call, exception or interrupt handling. Preemptive kernels are very difficult to design but they are mandatory for hard real-time operating systems. There are many approaches for making a kernel preemptable. One approach is to insert preemption points into a kernel. The kernel checks at the preemption points if a higher priority task is ready to run. In that case, the kernel interrupts the execution of the current process and schedules the higher priority task [BC05].

3.2.1. Real-time Linux

Linux is a free Unix-like time-sharing operating system that runs on a variety of platforms, including PCs. Many Linux distributions such as Slackware, Gentoo, Debian and Ubuntu package the Linux OS with software and made Linux a very popular operating system.

The Linux scheduler, like the scheduler of Windows or MacOS, is designed to provide a best average response time. Thus, Linux feels interactive and fast even if running many tasks. However, Linux wasn't designed for real-time. In Linux, a task may be suspended for an arbitrary long time, for example, when a Linux network device driver services a frame reception [BC05].

There are many other operating systems which, in contrary to Linux, were designed from the beginning as real-time operating systems. These real-time operating systems, such as VxWorks, QNX or LynxOS, offer scheduling guarantees. These real-time operating systems aren't used for general purpose computing but, for example, in spacecrafts of NASA.

Although the Linux operating system wasn't designed from the beginning as a real-time operating system, there exist many successful free and commercial projects that have adapted Linux for real-time. These adaptations are called Real-time Linux.

There are two different approaches to support real-time performance with Linux.

First approach tries to improve the preemption of the Linux kernel. Linux kernel preemption project [Lin08b] is a project that uses this approach.

Second approach adds a new software layer beneath Linux kernel that has full control of interrupts and processor key features. RTLinux [RTL08], RTAI [RTA08] and Xenomai [Xen08b] are projects that use the second approach. The software layer beneath Linux kernel that has full control of interrupts and processor key features is a minimal real-time operating system core that runs Linux as a low priority background task. Shared memory, mailboxes, message queues and FIFO pipes may be used to share data between the Linux operating system and the real-time core. The low priority task that runs Linux is only allowed to run if there are no real-time tasks to run and there are resources to spare. Interrupts with hard real-time constraints are processed by the real-time core, while other interrupts are forwarded to the task that runs Linux. The real-time core are simply patches to the basic Linux kernel source code. Hard real-time guarantees are only provided in the real-time core. All real-time tasks are implemented as kernel modules and are restricted just as usual kernel modules in what they can do and have to be carefully designed and implemented. Especially, they can't use arbitrary functions from the shared libraries.

Real-time Linux adaptations alone do not guarantee a deterministic processing of the received packets on each cluster node of DTVEE because the Linux standard network stack is used for packet processing. Therefore, it is necessary to use Real-time Linux adaptations with a deterministic network stack which guarantees that a packet is processed by all cluster nodes within a bounded time, for example, the RTnet real-time network stack [KaZB05]. In that case, it is possible to guarantee a deterministic processing time for the received packets on all cluster nodes. Thus, the combination of a

Real-time Linux adaptation with a real-time network stack is a possible approach to guarantee a bounded delay for PLACE packets but it's too much effort to use a Real-time Linux adaptation with a real-time network stack only for purposes of PLACE. And furthermore, DTVEE does not need a Real-time Linux adaptation with a real-time network stack.

3.3. Real-time Communication

The fundamental requirement of network communications in real-time distributed systems is that there be a limited and known packet delivery latency despite overload. Timing constraints in real-time distributed systems are one of the most important characteristics. In a real-time distributed system it is a requirement that a message generated by an application must be received by the receiver within a defined time interval. A real-time packet, that isn't transmitted within the specific deadline, is simply useless for both sender and receiver.

Another important requirement of real-time distributed systems is a bounded delay jitter. Delay jitter can be removed by buffering at the receiver. However, the size of buffer that the receiver requires can be reduced if the communication network can give some guarantees about delay jitter. For a high bandwidth communication the reduction of delay jitter can be significant.

Further important concept of real-time distributed systems is simultaneous message delivery. Simultaneous message delivery requires that all receivers receive the same message at the same time. Therefore, simultaneous message delivery protocols has to meet a strict deadline and to ensure that each receiver will receive a message at the same time, regardless of the network conditions and probably differences between the local clocks in the receivers.

In the following sections various hardware- and software-based protocols are studied which are used in real-time distributed systems on the Internet and LANs. These approaches try to solve the before mentioned challenges of real-time distributed systems.

3.3.1. Token Bus and Token Ring

Token Bus [RV92] and Token Ring [PD03] are distributed shared medium access protocols which are based on the token passing mechanism. The token passing mechanism is a widely used technique in communication networks to provide a collision-free access to a shared communication medium. The token passing mechanism assumes that all stations connected to one shared network segment build a ring.

Token Bus supports an arbitrary linear or tree topology and Token Ring supports an arbitrary ring topology. The stations in a Token Bus network build a logical ring. Contrary to Token Bus, the stations in a Token Ring network are organized in a physical ring. The ring-based topology of Token Ring is viewed as a single shared medium, it does not behave as a collection of independent point-to-point links which are configured in a loop.

The token passing mechanism is a distributed protocol without master and assumes that a token circulates around a physical or logical ring and each station in the ring receives a token from its predecessor and then forwards the token to its successor. A token is a special sequence of bits and allows a station that has a token to transmit a frame over a shared communication medium. The token passing protocol is decentralized and has a high efficiency but it also has problems. The failure of a node in a ring can crash the entire ring or if the token is lost, then some recovery procedure have to be invoked to get the token back. The token passing mechanism also has to handle nodes that join and leave a ring dynamically. Furthermore, each node on a ring has to hold the token during a frame transmission. The token holding time (THT) has to be limited in order to be able to guarantee a bounded frame transmission delay. Another important quantity is the token rotation time (TRT), which is the amount of time it takes a token to traverse a ring as viewed by a given node. The token rotation time increases when the number of nodes on a ring increases and, therefore, we get worse deadlines. Thus, a ring can't contain a large number of nodes if small deadlines have to be provided. TRT is given by:

$$TRT = NumberOfNodes \cdot THT + RingLatency \tag{3.1}$$

Token Ring supports different level of priority and guarantees a deterministic behaviour for the packets with the highest priority level. The strict priority scheme of Token Ring may cause lower-priority packets to be locked out of a ring for extended periods of time if there are sufficient high-priority packets ready to be sent.

Token Bus also allows to support different level of priority but the priority scheme of Token Bus differs from the priority scheme of Token Ring. Token Bus protocol requires each station in a logical ring to implement a Synchronous (highest priority) message class. Three lower priority classes Urgent Asynchronous, Normal Asynchronous and Time Available do not have to be implemented by a station on the ring. Token Bus requires each station on the ring to implement the Synchronous priority classes and defines a variable called the Highest Priority Token Hold Time (HPTHT). This variable determines how long a station may service its Synchronous traffic on each token visitation [GW88].

Token Bus and Token Ring use a shared communication medium and, therefore, they both support broadcast and multicast communication.

Token Bus and Token Ring are obsolete technologies and were replaced by inexpensive high-speed Ethernet. The price of 16 Mbps Token Ring switches is still higher than 100 Mbps Ethernet switches.

It isn't possible to use Token Bus or Token Ring in DTVEE because they are obsolete, more expensive than Ethernet and do not provide enough bandwidth. Furthermore, Token Bus and Token Ring alone do not provide a bounded delay for packets if the standard non-deterministic Linux network stack and scheduler are used on cluster nodes of DTVEE.

3.3.2. Transport Protocols

Various transport protocols for the standard TCP/IP network stack are discussed in this section.

3.3.2.1. Real-time Transport Protocol

Real-time Transport Protocol [PD03, RTP96], called RTP, is a real-time end-to-end transport protocol for multimedia applications in the Internet. RTP is a very flexible protocol that supports many multimedia applications and can use various underlying protocols, such as TCP [PD03, TCP81], UDP [PD03, UDP80] or AAL5/ATM (ATM Adaption Layer 5/Asynchronous Transfer Mode) [PD03]. In most cases, RTP uses the connectionless UDP as a transport protocol because it is better than TCP for multimedia traffic and because UDP supports multicast communication.

RTP does not guarantee timely delivery of packets and does not keep the packets in order, RTP gives the responsibility for recovering lost segments and reordering of the packets to the application. RTP protocol provides following services for real-time multimedia applications: payload type identification, source identification, sequence numbering and timestamping.

RTP is accompanied by another transport protocol, called RTP Control Protocol (RTCP) [PD03, RTP96], which provides feedback of the quality of the data delivery and information about session participants.

RTP alone, like UDP, only provides a best-effort service. Real-time applications, which use RTP, may suffer from jitter, delay, and packet loss. Various approaches exist to solve these problems. The adaptive playout delay, forward error correction, and interleaving are some of these approaches, however, these approaches are not suitable for hard real-time systems because they will work only to some degree of loss, delay or jitter. Another different approach is to fix the unreliable, best-effort nature of the network layer in the Internet by means of Intserv and Diffserv. Although this approach offers a quality of service as reliable as TCP, it is very difficult to deploy this solution to all existing routers in the network core of the Internet [KR05].

This transport protocol is unsuitable for purposes of PLACE because it can't guarantee a bounded packet processing delay if the standard Linux scheduler and network stack are used.

3.3.2.2. RTCast

RTCast [ASJS96] is a lightweight fault-tolerant multicast communication and group membership protocol for exchanging periodic and aperiodic messages within a real-time process group. The RTCast protocol supports message transport within a deadline, atomicity (message is delivered to either all processes or to none at all) and order for multicast messages within a process group and tolerates process crashes and failures of communication media. Furthermore, the protocol guarantees the atomicity of membership changes and ensures that all processes within a process group agree on membership. The protocol is called lightweight because it does not use acknowledgments for

every message. The RTCast protocol is a pure software-based solution and is designed to run on standard non-real-time operating systems and hardware. Currently, there are implementations of the protocol for Linux, Solaris and Windows NT.

The RTCast protocol is implemented on top of the IP protocol and sends messages using broadcast or IP multicast if available. It provides group membership service and timed atomic multicast communication. The RTCast protocol assumes that the underlying communication network provides unreliable unicast communication.

Nodes of a single multicast group are organized as a logical ring. Each node on the ring has a unique identifier and there is a FIFO channel between any pair of nodes on the ring. And that these FIFO channels have a bounded transmission delay. Furthermore, the RTCast protocol requires that node clocks must be synchronized.

The RTCast protocol is capable to detect node failures and tolerates receive omissions. However, send omissions are treated like node failures and a node is halted if it does not receive its own message. The protocol does not consider permanent link failures because hardware redundancy may be used to handle these failures.

RTCast applies a token passing mechanism to regulate access to the network. Each process within a process group knows its predecessor and its successor. Each process multicasts a heartbeat after sending a message. The heartbeat mechanism is used to detect process crashes. Each sent message has a sequence number for detecting missed messages. If a process detects a missed message, it halts and does not send heartbeats. Therefore, other processes in the group will exclude this process from the group membership when they do not receive a heartbeat from the halted process.

First, when a process receives the token, it multicasts a membership change message, if any membership changes were detected during the last round. After that, the process may send data messages. The last data message is marked by setting a corresponding bit. Finally, the process multicasts a heartbeat which indicates that the process is still alive. A heartbeat received by a logical successor of a process is assumed to be the logical token.

Each process in a process group has a maximum token holding time (THT). A process that holds the token must release the token by multicasting the heartbeat when it has sent all data messages or when the maximum THT has expired. This guarantees a bounded token rotation time (TRT) and allows to detect the loss of a token by setting a timeout.

The RTCast protocol supports joining and leaving of processes. A member of a process group may leave the group by multicasting a membership change message. A new process can join a process group by sending a join request message to some process of the group which sends a membership change message to notify all other processes in the group. On a multiple access LAN such as Ethernet, a new joining process can cause problems because it may access the communication medium at the time assigned to some process in the group. To address this problem the RTCast protocol reserves a join slot which is large enough for sending a join request.

Th RTCast protocol is unsuitable for purposes of PLACE because it can't guarantee a bounded packet processing delay if the standard Linux scheduler and network stack are used. Furthermore, the RTCast protocol requires that the clocks of all cluster nodes

are synchronized. In addition, the RTCast protocol is a token-based approach and we get worse deadlines when the number of nodes on a logical ring increases.

3.3.3. Ethernet-Based Approaches

Originally, Ethernet was designed to interconnect office computers and printers. However, its wide availability, high bandwidth and low cost, made it appealing enough to be considered for use in other application domains, such as multimedia applications, industrial and embedded systems, which have real-time constraints [PD03, HHH+02].

However, Ethernet was not originally designed for hard real-time applications and does not directly support this sort of applications because Ethernet is not deterministic. Ethernet is a shared medium communication system. Collisions, random delay and transmission failures are all possible on Ethernet – especially on heavily-loaded networks. In such a communication system, it is impossible to promise hard real-time guarantees. In addition, Ethernet frames do not have priorities and that makes it unsuitable for real-time systems in which frames with higher priority should not be held up waiting for unimportant frames.

In order to make Ethernet suitable for real-time applications, a mechanism is needed to completely avoid all frame collisions on Ethernet. A collision domain is a network segment where simultaneous transmissions may produce a collision. The more there are transmitting stations in a segment the more increases the collision probability. Without collisions, it becomes possible to give hard real-time guarantees, because then the frame transmission needs a constant time. Completely avoiding collisions, therefore, offers a possibility to develop and use real-time applications over Ethernet.

There are several accepted and reliable methods to make Ethernet real-time capable [Ind08]:

- Limited Load
- Token Passing

The Limited Load is a method that is specific only for Ethernet. There is a special and famous situation when an Ethernet network completely breaks down because there are so many stations trying to initiate a frame transmission or a frame retransmission, that the Ethernet network is completely unable to handle the produced load, which is called the critical load. The Ethernet bus can guarantee that a frame will be delivered within defined time if it is guaranteed that the load for the Ethernet bus is far below the critical load. Ethernet switches provides a possibility to reduce the load of an Ethernet network. They provide a private collision domains for each one of their ports.

The Token Passing method is a widely used technique in communication networks and can also be used in an Ethernet network without any hardware modifications. Using a special software in each Ethernet station, passing the token from one station to another and only allowing the station that has the token to access the Ethernet bus, it is possible to provide real-time capability on the Ethernet bus.

This section presents an overview of the efforts towards hardware- and software-based real-time communication systems over Ethernet, which use the before mentioned Limited Load and Token Passing methods.

3.3.3.1. Switched Ethernet

An Ethernet switch [Spu00, PD03, Cas04], also called a switching hub, basically connects Ethernet devices with each other. An Ethernet switch has several ports to which an Ethernet device or another switch can be plugged in. An Ethernet switch receives frames on its ports that were transmitted by one Ethernet device and passes these frames to appropriate switch ports which connect to other Ethernet devices. As it passes these frames it also learns on which ports an Ethernet device may be reached and uses this gathered information in deciding to which ports received frames should be forwarded. This technique is known as the Backward Learning algorithm. This allows to reduce the load on an Ethernet network because frames are only sent to the appropriate switch ports where they need to go.

The main advantage of a switch is its ability to receive multiple frames simultaneously. An Ethernet switch, as also an Ethernet hub, basically buffers the frames which are received as a result of simultaneous transmission (collision). However, an Ethernet switch supports frame transmission in parallel if simultaneously received frames have to be forwarded to different ports and Ethernet devices on these ports are currently not transmitting. In contrast, an Ethernet hub passes all frames to all ports, excluding the port on which this frame has arrived, and, therefore, wasting a lot of bandwidth.

An Ethernet switch learns where Ethernet devices are located during the frame forwarding. It maintains a database of MAC addresses that contains dynamically learned entries. An Ethernet switch looks up the address table for the destination address of each received frame. If it does not find an appropriate entry for a received frame, then this frame is forwarded to all ports of the switch.

There are more than one switching method that an Ethernet switch can apply to forward an incoming frame. The latency of an Ethernet switch will vary depending on the switch load and the switch architecture.

With the store-and-forward switching method, an Ethernet switch copies the entire frame to its internal buffer and computes the CRC of the frame. If an error is detected, then the frame is discarded. If the frame does not have errors, then the destination address of the frame is looked up and the outgoing port is determined. The advantage of the store-and-forward switching method is that frames which contain errors aren't forwarded. The disadvantage of the store-and-forward switching method is a higher frame latency which depends on the frame length (up to several milliseconds).

With the cut-through switching method, an Ethernet switch copies only the destination address of a frame to its internal buffer. After that the destination address is looked up and the outgoing port is determined. The advantage of the cut-through switching method is a reduced frame latency. An incoming frame is forwarded as soon as its destination address is read. First disadvantage of the cut-through switching method is that frames with errors are forwarded and waisting the bandwidth. Second disadvantage of

this switching method is a higher probability of collisions.

Many Ethernet switches can combine the two switching methods. As long as the amount of collisions isn't large the cut-through switching method is used. If the amount of collisions increases, then an Ethernet switch applies the store-and-forward switching method.

With the fragment-free switching method, an Ethernet switch copies only the first 64 bytes of a frame. If this frame part is error-free, then the frame destination address is looked up and the outgoing port is determined. The most errors and collisions occur during the first 64 bytes of a frame. The fragment-free switching method is faster than the store-and-forward but slower than the cut-through switching method.

There are two types of Ethernet switches: managed and unmanaged switches. A managed switch is basically a switch that supports Simple Network Management Protocol (SNMP) [SNM90, PD03]. Most managed switches provide more features than SNMP. A managed switch allows to control a network. An unmanaged switch simply allows Ethernet devices to communicate. Advanced modern switches provide more sophisticated features, such as Quality of Service (QoS), Virtual Local Area Network (VLAN), Port Mirroring, IGMP Snooping and many more.

An advanced Ethernet switch with QoS ability can apply a higher priority to certain received frames. It can use a port on which the frame has arrived or a tag within the frame header to determine the priority of the frame (IEEE 802.1p and 802.1Q). This features help to improve the determinism of Ethernet networks.

DTVEE uses switched Ethernet but switched Ethernet alone does not guarantee a bounded packet delay if the standard non-deterministic Linux network stack and scheduler are used on cluster nodes.

3.3.3.2. Token-Based Approaches

This section describes software-based Ethernet protocols that supports real-time communications over Ethernet and do not require any modifications of hardware at all. They avoids frame collisions on the Ethernet bus by using the token passing mechanism mentioned before. The protocols require modifications of the standard network stack because they are built into the Ethernet device driver and operate over the data link layer.

Rether [Ven96, Tzi99] is an efficient delay/bandwidth guarantee mechanism over Ethernet for real-time multimedia applications. Rether was successfully implemented within the Ethernet device driver under Linux, FreeBSD and DOS.

The Rether protocol is transparent to the higher network protocols such as IP and transport protocols such as UDP and TCP, therefore, all existing network applications can run without any modifications at all. Rether provides also a new API for real-time applications which have to use this API for real-time. The Rether protocol supports only simplex uni-directional connections for real-time applications.

The Rether protocol has two modes of operation. Rether operates in the CSMA/CD [CSM] mode if there is no need for real-time but it switches to the token passing mode for real-time. As soon as the last application with real-time requirements is finished, Rether

switches back to the CSMA/CD mode. In the token passing mode, both real-time and non-real-time traffic is regulated by a token. In the token passing mode, only the node, which has the token, may send data over Ethernet.

During the token passing mode, a token circulates from station to station in cycles. The Rether protocol allows to configure the period of the token cycle. In each cycle, Rether first services all real-time applications of each station in the network. Only after that the Ethernet bus access is granted to non-real-time nodes in a round-robin fashion. The Rether protocol must ensure that non-real-time applications do not starve and, therefore, reserves some bandwidth for non-real-time traffic.

The Rether does not use a globally synchronized clock, therefore, the token itself contains a special counter, called the residual cycle time. At the beginning of each new token cycle, this counter is set to a full token cycle that is configurable and may be set during the system initialization. When a station receives the token, the station subtracts its token holding time from the residual cycle time. When the residual cycle time counter becomes zero, a new token cycle is initiated.

The Rether protocol tolerates token losses in case of node failures or random bit errors. The Rether protocol requires that each node in the network monitors the state of its successor node. Each node in the network must acknowledge the reception of a token. When the sender of the token does not receive an acknowledgment within a defined time, the monitoring station, which has sent a token, creates a new token.

The Rether protocol supports also switched Ethernet. Between the sender and the receiver, which are on different segments of the network, a logical connection is established. This logical connection consists of several per-segment reservations. Each network segment has a circulating token that is independent from other network segments.

The Rether protocol is unsuitable for purposes of PLACE because the Rether protocol is a token-based approach and, therefore, we get worse deadlines when the number of nodes on a logical ring increases.

Another software-based Ethernet protocol that supports real-time communication over Ethernet is Real-time Ethernet Protocol [MHG03, MH05], called RT-EP.

The RT-EP network is logically organized as a static ring in which the token rotates. Each node on the ring knows its successor and its predecessor. Each message which is sent by a node has fixed priority and each node has a priority queue where all packets to be transmitted are stored in priority order. Each node also has reception queues where received packets are stored in priority order. The number of reception queues is equal to the number of real-time applications. This protocol works only if the total number of communicating real-time applications is known in advance at configuration time.

The RT-EP protocol has two phases: a priority arbitration phase and a transmission phase. In the priority arbitration phase, the message with the highest priority is determined. In the transmission phase, the message with the highest priority is transmitted to the receiver. The priority arbitration phase may be initiated by an arbitrary node which is called token master. During the priority arbitration phase the token visits all nodes on the logical ring and each node checks information in the token to determine if one of its own messages has a priority higher than the priority contained in the token. In that case the station with the highest priority is updated in the token, otherwise the

token will not be changed. After that, the token is sent to the successor node. At the end of a priority arbitration phase, the token master has the token and the token master sends the token to the station with the highest message priority. This station becomes the new token master and may begin the transmission phase.

The RT-EP protocol tolerates the loss of a packet and guarantees the real-time behaviour in that case. Faults such as failure of a station and busy station are not supported by this protocol and considered to be a bad system design. RT-EP uses positive acknowledgment and retransmission mechanism to cope with the loss of a packet. If the acknowledgment isn't received after a defined number of retransmissions, then the station is considered as a failing station and is excluded from the logical ring.

The RT-EP protocol is unsuitable for purposes of PLACE because it requires a real-time operating system and does not support non-real-time traffic. Furthermore, the RT-EP protocol is a token-based approach and, therefore, we get worse deadlines when the number of nodes on a logical ring increases.

3.3.4. Wireless-Based Approaches

Real-time token-based protocols originally designed to support real-time traffic for shared Ethernet LANs appear also to be suitable for wireless LANs. But there are technological differences between 802.11 and Ethernet networks which make these protocols unsuitable for wireless LANs and they have to be redesigned in order to support real-time traffic in 802.11 networks.

The token passing mechanism, which ensures that collisions does not occur in an Ethernet segment, works very well for the Ethernet network because all stations can hear each other. In a wireless LAN, mobile nodes can move out of each other's transmission range and, therefore, direct token passing between wireless nodes is infeasible [PD03].

The protocols for wireless networks which are discussed in this section are unsuitable for DTVEE because DTVEE does not yet use wireless networks but it may be possible in the future.

3.3.4.1. Wireless Rether

One possibility to overcome this problem is to use the infrastructure mode in wireless networks. All mobile nodes communicate with the wired network and between each other through an access point and, therefore, token passing mechanism can be used in 802.11 networks.

Wireless Rether [SGZ+02], like Rether, is also a software-based solution to support real-time applications in 802.11 networks. The protocol is also placed in the network stack over the data link layer and is implemented in the device driver of the wireless network interface.

The token passing mechanism is implemented in a central server, called Wireless Rether Server (WRS). This server is responsible for passing the token to wireless nodes and is placed between the access point and the wired network. The wireless nodes are called Wireless Rether Clients (WRC). The WRS grants the token to wireless nodes in

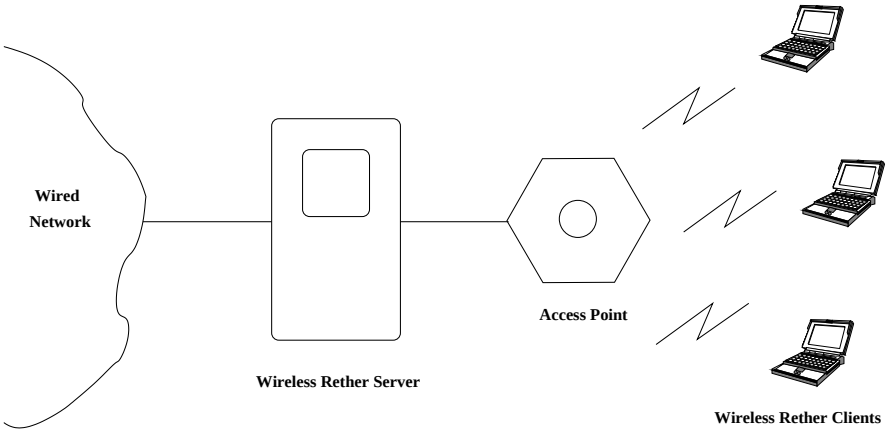


Figure 3.1.: Wireless Rether Architecture

a weighted round robin fashion. The weight associated with each WRC corresponds to the duration of the token holding time. The sum of all the weights must be smaller than the token cycle time and a portion of the token cycle time is reserved for non-real-time applications. The Figure 3.1 shows the architecture of Wireless Rether.

The central architecture of Wireless Rether has the advantage that the loss of a token isn't fatal because the WRS can monitor the mobile nodes and regenerate the token if it's lost. However, the central architecture of Wireless Rether has also the disadvantage of a single point of failure.

3.3.4.2. WRTP

The Wireless Token Ring Protocol (WTRP) [ELA+02] is another token-based distributed medium access control protocol for wireless networks. Contrary to the Wireless Rether protocol, WRTP is a medium access control protocol for wireless ad-hoc networks. Therefore, the WRTP protocol has no single point of failure and supports topologies in which not all nodes on a logical ring have to be connected to a single master as in the Wireless Rether protocol. However, one of the biggest challenges that the WRTP protocol have to overcome is partial connectivity. The Wireless Rether protocol does not have to deal with this problem due to its centralized architecture.

The WRTP protocol allows nodes to join and leave a logical ring dynamically. A node is allowed to join a logical ring only if the token rotation time wouldn't grow unacceptably with the addition of the new node. Each node on a logical ring has a connectivity table. The connectivity table of a node contains an ordered list of nodes in its own ring. A node builds its connectivity table by monitoring transmissions from its own logical ring. When a node joins a ring, the node looks up the prospective predecessor and the successor in its connectivity table. When a node leaves a ring, the predecessor of the leaving node finds the next available node to close the ring in its connectivity table.

3.3.5. Real-time Network Stacks

Various real-time network stacks are discussed in this section.

3.3.5.1. RTnet

RTnet [KaZB05] is a modularized framework for hard real-time communication systems and adds the real-time capability to the standard UDP/IP protocols such as the IP, ICMP and UDP protocols. TCP is not supported by RTnet because it is impossible to make TCP real-time capable. RTnet is a pure software-based approach to support real-time traffic over standard IP networks and currently supports the Ethernet network and the FireWire bus. For the use of RTnet is a hard real-time compliant system platform needed. Currently, there are implementations of RTnet for RTAI Linux [RTA08] and Xenomai [Xen08b]. In the Figure 3.2 the overall architecture of the RTnet framework is shown.

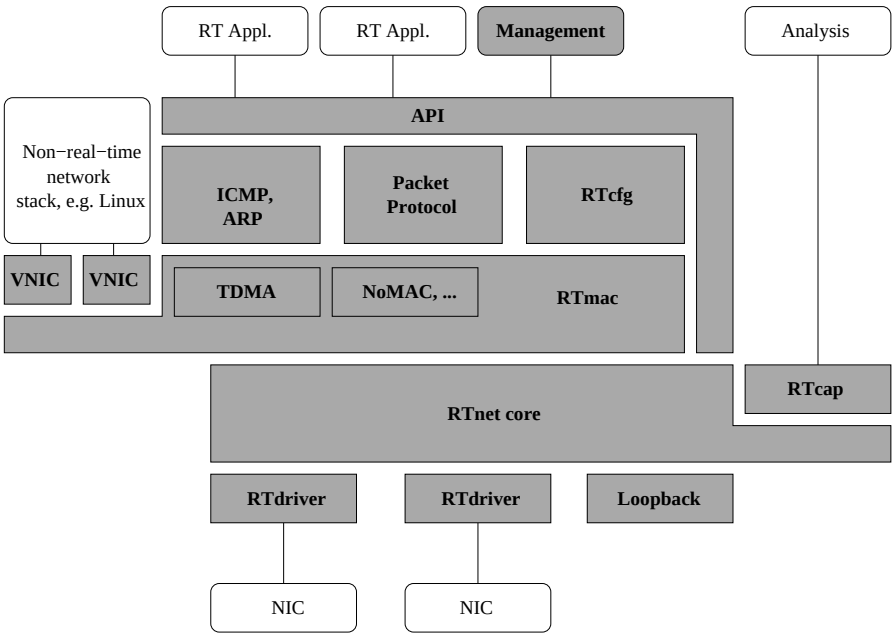


Figure 3.2.: RTnet Architecture

One of the important parts of a network stack is the packet management. The RTnet framework uses a data structure called rtskb for the management of packets. This data structure was derived from the Linux sk_buff data structure. The RTnet stack has to preallocate all packet buffers during setup because of the real-time requirements.

In the RTnet framework, network interface cards (NIC) are attached to the RTnet stack via a Linux-like driver interface. Therefore, it is very easy to port Linux drivers to RTnet and several widely-used NICs are already ported to RTnet. Some of the ported NICs are Gigabit Ethernet network cards. A RTnet NIC driver has to provide a very accurate timestamping for incoming and outgoing packets. Therefore, the packet

timestamp for incoming packets has to be taken in the beginning of the interrupt routine. In addition, a NIC driver has to provide the capability to store a timestamp in an outgoing packet.

For real-time communication, a real-time capable network stack is as important as a deterministic communication media. In the RTnet framework, the RTmac layer is an optional extension to the RTnet stack and is required only if the underlying communication media, such as standard Ethernet, isn't deterministic. The RTnet framework already provides a timeslot-based MAC discipline, called Time Division Multiple Access (TDMA). This MAC discipline is mainly for the use with standard Ethernet. TDMA is an access method for shared medium networks such as the Ethernet bus. This technique divides the shared medium in discrete time slots and only one station may transmit data within each time slot, therefore, no collisions are possible in this network. TDMA technique requires a global clock so all nodes in the network can stay synchronized. The master periodically issues synchronization messages and synchronizes the clock of RTnet nodes in within a network segment. On the participant nodes, all packets are sorted according to their priority. TCP/IP data has the lowest priority and is only transmitted when it do not hinder time critical communication.

The RTnet framework has a deterministic UDP/IP network stack. Several modifications to a standard UDP/IP network stack were performed due to real-time requirements.

The dynamic Address Resolution Protocol (ARP) has to be converted into a static address resolution mechanism. All destination MAC addresses have to be known at setup time. If a destination MAC address can't be resolved later, then no address resolution is performed and an error is returned to the caller.

The routing process was simplified and the routing tables were optimized for the limited amount of entries which are used with RTnet.

In order to optimize the IP fragmentation mechanism, some modifications were performed to the IP layer. The IP layer of the RTnet network stack tries to avoid packet fragmentation. Furthermore, IP packet fragments are accepted in a strictly ascending order. If packet fragments arrive out of order, then the hole fragment chain is dropped. The total number of fragment chains is limited in order to guarantee an upper bound for the lookup latency.

The RTnet framework offers a generic configuration and monitoring service, called the Real-time Configuration Service (RTcfg). This service is independent of a communication media which has to support broadcast transmissions. RTcfg distributes configuration data in order to allow real-time nodes to join real-time networks on-the-fly. RTcfg monitors the state of active nodes and exchanges of their hardware addresses, for example to setup and maintain the static ARP tables. Furthermore, it allows to synchronize the real-time network startup procedure.

The RTnet framework allows to tunnel time uncritical communication through the real-time network. The full access to participants in the RTnet network is provided by a gateway to other networks via stream-oriented protocols such as TCP/IP. Diagnosis and maintenance tasks can be performed.

The RTnet framework offers a powerful capturing extension called RTcap. This plug-in allows to capture both incoming and outgoing packets over NICs. Therefore, network

analysis tools such as Ethereal can be used with RTnet.

The RTnet framework has a POSIX-confirming socket and I/O interface which allow applications to attach to the RTnet stack. UDP and packet sockets allow to exchange user data deterministically. User space applications which use Linux networking are almost source code compatible with the socket interface of RTnet.

The RTnet real-time network stack needs a real-time operating system and, therefore, is unsuitable for DTVEE that does not use a real-time operating system. It is too much effort to use a real-time operating system with the RTnet real-time network stack only for purposes of PLACE.

Chapter 4.

Design Issues

This chapter discusses possible design approaches for the PLACE protocol. At first, basic concepts of the Linux kernel 2.6 network stack will be introduced. It is important to understand these concepts of the Linux kernel 2.6 network stack before considering approaches to protocol design because these fundamentals of the Linux kernel 2.6 network stack serve as a basis for approaches to protocol design. Furthermore, these fundamentals are also important for protocol design and implementation which are discussed in Chapter 5 and Chapter 6. After that, various approaches to fulfill the main requirements of the PLACE protocol, their advantages and disadvantages are presented.

4.1. Fundamentals of the Linux Kernel 2.6.18 Network Stack

The Linux network stack [CKHR05, Ben05, WPR+04] is originated from the BSD network stack and since then is considerably improved and extended. The Linux network stack provides a free, rich, efficient and very flexible network functionality which can be individually configured and adapted to special requirements.

The architecture of the Linux network stack is based on the five-layer TCP/IP reference model for network protocols and has static structure. It is completely implemented in the Linux kernel. The network layers of the Linux network stack interact closely and, therefore, the Linux networking code is very efficient. However, this architecture has also the disadvantage, that the network layers has not always clearly defined interfaces.

The implementation of the Linux network stack is designed to be independent of a specific protocol. That applies to the transport and network layer protocols (TCP/IP, IPX/SPX, etc.) as well as to Layer 2 protocols (Ethernet, token ring, etc.). Other protocols can be added to any layer of the Linux network stack without a need for major changes.

In the following sections, most important data structures, packet reception and transmission in the lower network layers of the Linux network stack are discussed in detail.

4.1.1. The sk_buff structure

The socket buffer structure sk_buff [CKHR05, Ben05, WPR+04] is the most important data structure in the Linux networking code. It represents data and headers of a packet during its passing through the Linux network stack. All network layers of the Linux network stack use this data structure which describes a packet.

A socket buffer consists of two parts: payload and management data. Payload is the storage location which contains data that was received over a network or has to be sent over a network. Management data is additional data (pointers, timers, etc.) which is required by the network protocols that process a packet represented by the socket buffer.

Packet headers can be efficiently prepended or stripped while a socket buffer passes through the network layers of the Linux network stack. The Linux networking code avoids coping the payload of a socket buffer by reserving sufficient space for data and headers. Only cheap pointer operations are used in order to prepend or strip a packet header. Actually, the payload of a packet is copied only twice in the Linux network stack. First, a payload is copied from or to the user- space when an application calls the socket system call to send or receive data. Second, a payload is copied when it is passed to or received from a network adapter. Since kernel version 2.4 Linux also supports a zero-copy approach [Bro05] which eliminates all the data duplication done by kernel when a user-space program has to send data over a network adapter but the network adapter has to support scatter/gather I/O. Scatter/gather I/O simply means that the data waiting for transmission does not need to be in consecutive memory. It can be scattered through various memory locations. Zero-copy approach not only reduces several context switches but it also eliminates data copying done by the CPU. The Linux network stack supports zero copying of files through specific API's such as sendfile. The sendfile system call offers significant performance benefits to applications such as web servers and ftp servers which has to efficiently send files. The semantics of sendfile is to transmit data of the specified length, or completely, from one file descriptor to another, for example, socket descriptor, without copying of it to the user address space. Therefore, it is only usable in situations where the user application is only interested in data transmission and does not need to process the data. Since the transmitted data never crosses the user/kernel boundary, the sendfile system call greatly reduces costs of data transmission.

This architecture of the socket buffers is one of the main reasons for flexibility and efficiency of the Linux networking code.

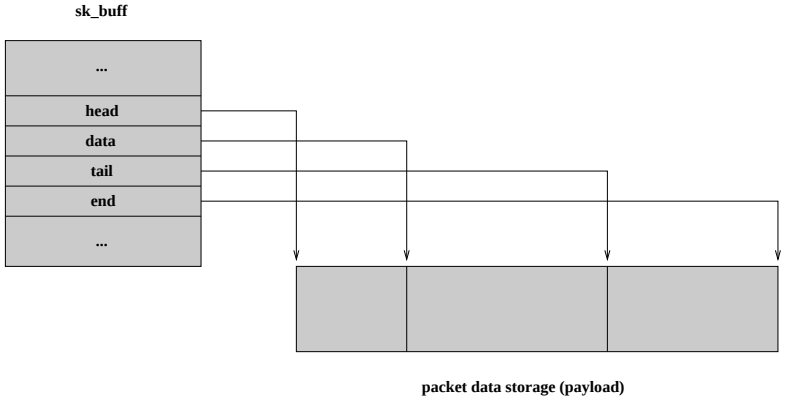


Figure 4.1.: Packet data storage

The sk_buff structure contains pointer variables to address the data in a packet. The head pointer points to the beginning of the allocated space of the packet payload. The

data pointer points to the beginning of the valid bytes of the packet payload and is usually slightly greater than head pointer. The tail pointer points to the end of the valid bytes of the packet payload and the end pointer points to the maximum address which the tail is allowed to reach.

Another important variables of the sk_buff structure are union variables to address the headers of various network layers: h, nh and mac. Each pointer variable in the union variables points to a different type of data structure. The h union variable contains pointers to transport layer headers, the nh union variable contains pointers to network layer headers and the mac variable includes link layer headers.

The dev variable in the sk_buff structure is a pointer to the net_device structure which is discussed in the next section. It depends on whether the packet stored in the socket buffer is about to be sent or has just been received. The dev pointer points to the receiving network device when the packet was received. And it points to the sending network device through which it will be sent out when the packet is to be transmitted.

Normally, when packets are currently not processed by any protocol instance of the Linux network stack, they are organized in queues. In order to manage packets in queues, the Linux networking code uses the sk_buff_head data structure. The socket buffer queue is implemented as a circular doubly-linked list that allows quick navigation in both directions.

The Linux networking code offers a lot of functions, usually very short and simple, to manipulate socket buffers or socket buffer queues. These functions allows to create, initialize, destroy, copy socket buffers and manipulate parameters and pointers of socket buffers or socket buffer queues. Most of these functions are defined as inline and have only little functionality, however, they are very important and are very often used. The inline procedures are no real procedures. The body of an inline procedure is built into the body of the calling procedure, similarly to macros. Inlining reduces overhead of a procedure call and, therefore, makes code execute faster which is very important for often used procedures but it also makes the Linux kernel slightly larger.

Most socket buffer or socket buffer queue operations are executed in critical sections or they can be interrupted by higher-priority operations, such as interrupt handlers, softirqs or tasklets. Therefore, the data of the sk_buff and sk_buff_head structures should be handled in an atomic way. In order to achieve that, spinlocks and semaphores have to be used which introduces some additional cost. But it is the only way to prevent inconsistent states.

4.1.2. The net_device structure

In the Linux network stack, a network device is represented and managed by the interface or the data structure net_device [CKHR05, Ben05, WPR+04]. The net_device structure serves as a basis for each network device in the Linux kernel. It provides not only information about the network adapter hardware (interrupt, I/O ports, driver functions, etc.), but also the configuration data of the the higher network protocols (IP address, subnet mask, etc.).

The network device structure can represent and manage not only a physical network

adapter but also a logical network device, such as the loopback network device. The network device structures for all physical and virtual network devices are maintained in a global list.

The network device structure is the interface between the higher network layers and a network adapter. This interface is implemented by the network device driver of a network adapter. It abstracts from the technical properties of a network adapter and provides a uniform interface to the higher network layers of the Linux network stack. The properties of different network devices are hidden at the net_device interface. The net_device structure provides a uniform view of network devices to the higher network protocols.

For efficient implementation of this interface, the concept of function pointers is used. Higher network protocols can only use these function pointers to indirectly call hardware-specific methods of a network device driver. The device driver of a network adapter has to map driver functions to a uniform interface so that higher protocols can access the functions of the network device driver.

Each network device has two identifiers: name and ifindex. Both identifiers uniquely identify a network device in the Linux kernel. name is the name of the network device. ifindex is a second identifier of a network device and is assigned by the Linux kernel when the network device is created. ifindex allows us to quickly find a network device with the specified ifindex in the global list of all network devices. The search by using ifindex is more efficient than the search by using the name attribute. The Linux networking code offers the method dev_get_by_name to find a network device by its name and the method dev_get_by_index to find a network device by its ifindex.

A network device has first to be registered to the Linux kernel before it can be used. A registered network device is put into the global list of all network devices, regardless of whether it is activated. Network devices can be registered at compile time or run time of the Linux kernel. The Linux kernel offers methods to register or unregister a network device: register_netdevice and unregister_netdevice.

Network devices and the Linux kernel can use several approaches to exchange data: polling, interrupts and the combination of the two techniques.

With polling, the Linux kernel constantly checks whether a network device has anything to say. The Linux kernel is continually reading a memory register of the network device or the kernel checks it after a timer expires. This technique can easily waste a lot of system resources and, therefore, it is rarely used if the Linux kernel can use other techniques such as interrupts.

Most network device drivers use interrupt handlers to exchange data with a network adapter. A network device interrupts the processor to signal one of three possible events: a new packet has arrived, transmission of an outgoing packet is complete or an error situation has occurred. The interrupt handler of a network device driver can tell the difference between the arrival of a new packet, a transmitting notification and error situations by checking the status register of the physical network adapter. This technique is quite common and represents the best option under low traffic loads. However, it does not perform very well under high traffic loads. Under high traffic loads, the CPU wastes all of its time to handle interrupts. This problem is commonly referred to as the receive

livelock. If packets are being received very fast, the Linux kernel never gets to process them because interrupts are being generated too fast and the CPU spends 100% of its time in interrupt handling. Interrupts have the advantage of very low latency between the reception of a packet and its processing. Packet reception is handled more in detail in the following section.

The third technique combines polling with interrupts and performs very well under very high traffic loads. Polling and interrupts have some advantages and disadvantages. It is possible to combine them and obtain something even better. This technique is also discussed more in detail in the following section.

4.1.3. Packet Reception

The path of each received packet which was not generated locally begins in a network adapter. Most network device drivers use interrupts to notify the Linux kernel about the arrival of a packet [CKHR05, Ben05, WPR+04].

The interrupt handler of a network device can use programmed I/O (PIO) to copy a received packet from the memory of the network adapter to a socket buffer but this technique wastes CPU cycles.

All modern PCI network adapters support direct memory access (DMA) and bus-mastering I/O. In that case, the device driver of a network adapter preallocates socket buffers for received packets and the network adapter triggers the interrupt when a received packet is already copied to one of the preallocated socket buffers by the network device. This technique does not waste CPU cycles, unlike the PIO technique.

Interrupt handlers are nonpreemtible and non-reentrant. During the execution of an interrupt handler, interrupts are disabled for the CPU that is serving the interrupt. Therefore, the CPU cannot receive other interrupts, whether of the same type or of different type. This has serious effects on performance and responsiveness of the Linux kernel and, therefore, interrupt handlers have to be very short.

In the Linux networking code, processing of received packets consists of two parts: top half and bottom half. The top half is the interrupt handler of a network adapter and it is very short. It only puts received packets into a backlog queue for further processing. Each CPU in the Linux kernel has its own backlog queue for incoming packets. In order to put a new packet into a backlog queue, an interrupt handler passes the received packet to the netif_rx procedure which puts the received packet into the backlog queue of the current CPU and schedules the network bottom half. The bottom half runs all non time-critical operations which could not be handled in the interrupt handler. In Linux, the bottom half for further packet processing is implemented by the software interrupt NET_RX_SOFTIRQ. The software interrupt NET_RX_SOFTIRQ is implemented by the net_rx_action procedure which dequeues packets from a backlog queue and calls the procedure netif_receive_skb for further packet handling.

In the Linux kernel version 2.5, a new API for handling ingress frames was introduced, known as NAPI (New API) to handle the problem of the receive livelock under high traffic loads. Since then, a network device driver can notify the Linux kernel about a new packet: by means of the old procedure netif_rx and by means of the NAPI mechanism. Very few

network device drivers supports NAPI and some of them allows to choose between the two techniques during a kernel configuration.

Instead of using only interrupts to exchange data between a network adapter and the Linux kernel, NAPI uses a mix of interrupts and polling. When a new packet is received, the interrupt handler of a network adapter adds the network device to a poll list and lets the Linux kernel know that there is some work to be done on the device. Each CPU in the Linux kernel has its own poll list. After that, the interrupt handler of the network adapter disables further interrupts on the device caused by reception of new packets. Then, the interrupt handler schedules NET_RX_SOFTIRQ.

A network device driver implements the polling functionality by poll function pointer in struct net_device. The poll function is called by net_rx_action and processes received packets. The kernel sets a limit on the total number of packets that the poll function of each network adapter in the poll list can process. It ensures fairness amongst network devices. If the poll function of a network device could process all outstanding packets of the device then it reenables receive interrupts for this network device. Interrupt will not be enabled, if not all received packets could be processed by the poll function. In order to process received packets, the poll function passes these packets to the netif_receive_skb function. In the Linux kernel 2.6, backlog queues for network device that do not use NAPI are implemented as pseudo network devices which use NAPI.

NAPI reduces the ratio of interrupts under high traffic loads. NAPI reduces packet latency and increases throughput under high traffic loads. And under low traffic loads, it converges to the interrupt-driven scheme.

The NET_RX_SOFTIRQ software IRQ is invoked upon return from an interrupt handler and will process received packets. Thus, if packets arrives very fast then the NET_RX_SOFTIRQ software IRQ will keep processing received packets and, therefore, user programs will never get to the CPU and they will simply starve. In order to avoid this situation, the NET_RX_SOFTIRQ software IRQ processes at most netdev_max_backlog packets which is set to 300 by default. Furthermore, the net_rx_action function may run not more than one clock tick. If there are more received packets to be handled, then net_rx_action schedules itself again. When the net_rx_action function returns and notices that it has been scheduled again, it wakes up a low priority kernel thread, known as ksoftirqd, to process the remaining packets.

4.1.3.1. Link Layer Multicast

A multicast frame is meant to be received by more than one host but not by all hosts. Multicast groups are assigned special hardware addresses. In Ethernet [Spu00], for example, a multicast address has the least significant bit of the first address byte set. Transmission of a multicast frame is very simple in the Linux kernel because it looks exactly like any other frame. A network device transmits multicast frames without looking at their destination addresses.

In order to receive multicast frames, a network device driver has to keep track of all interesting multicast addresses and deliver to the Linux kernel only those multicast frames which belongs to one of these subscribed multicast groups. In the Linux kernel, a

network device driver accepts a list of multicast addresses which should be delivered to higher network protocols for further processing. How a network device driver implements this functionality is dependent on the hardware of a physical network card.

Typically, network adapters can belong to one of three classes, as far as multicast is concerned: adapters that cannot deal with multicast, interfaces that can distinguish between multicast frames and other frames, and interfaces that can perform hardware filtering of multicast frames.

Network adapters that cannot deal with multicast frames can receive frames which are directed directly to their hardware address or receive every frame. These network adapters can only receive multicast frames if they receive every frame. Therefore, a host can be overflooded by a lot of frames which are not directed to the host and, thus, wasting a lot of CPU cycles to process these unimportant frames.

Network adapters that can distinguish between multicast frames and other frames can be instructed to receive multicast frames which are analyzed by a network device driver if they are interesting for the host or not. In that case, the overhead is acceptable because the amount of multicast frames on a normal network is very low.

Network adapters that can perform hardware filtering of multicast addresses are the optimal case for the Linux kernel because they does not waste CPU time analyzing and dropping uninteresting multicast frames which were received by the network device.

Most modern PCI Ethernet network interfaces support hardware filtering of multicast addresses but this filtering is often not perfect. Modern network cards use hashing to implement hardware filtering of multicast addresses. These network cards has a built-in bit vector and they hash with Ethernet CRC algorithm multicast addresses to obtain the index into this bit vector. By setting a bit of the bit vector, a network device driver instructs the network device to deliver multicast frames with addresses which hash to the index of this bit in the bit vector of the network device. Typically, the size of the bit vector is 64. High end network adapters support also perfect filtering of multicast addresses.

The Linux networking code provides two methods to manage multicast group membership: dev_mc_add and dev_mc_delete. The dev_mc_add instructs a network device driver to deliver multicast frames with a specified multicast address. The dev_mc_delete procedure instructs a network device driver not to deliver any more multicast frames with a specified multicast address.

4.1.3.2. Layer 3 Protocol Handlers

This section describes how the Linux networking code manages Layer 3 protocols and how an arrived packet is processed from Layer 2 upward in the Linux network stack.

The Linux networking code distinguish between two types of Layer 3 protocols: a protocol which receives all arrived Layer 3 packets and a protocol which receives only packets with the correct Layer 3 protocol identifier [CKHR05, Ben05, WPR+04].

The Linux networking code uses the data structure packet_type to manage Layer 3 protocols. The list ptype_all stores all protocols that should receive all incoming packets. The hash table ptype_base stores all other Layer 3 protocols. There is a packet_type data

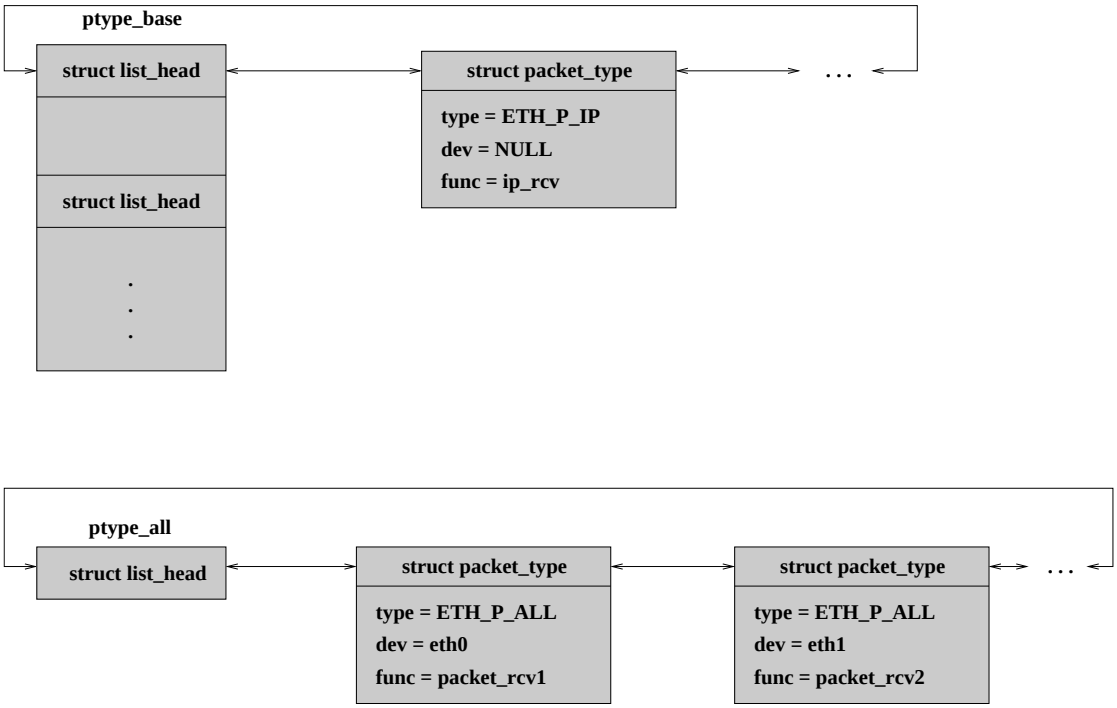


Figure 4.2.: ptype_base and ptype_all data structures

structure for each Layer 3 protocol in the Linux kernel. The packet_type data structure contains a function pointer func which is the handling routine of a Layer 3 protocol.

For every received Layer 3 packet, the Linux kernel calls netif_receive_skb procedure and passes to it the pointer to the socket buffer of the packet. At first, the netif_receive_skb procedure passes a copy of the packet to the handler routine of every Layer 3 protocol which wants to receive all Layer 3 packets. These Layer 3 protocols are maintained in the list ptype_all. After that, netif_receive_skb procedure passes a copy of the packet to the Layer 3 protocol with the correct protocol identifier if there is a Layer 3 protocol registered with the correct protocol identifier. The netif_receive_skb procedure tries to find that Layer 3 protocol in the hash table ptype_base.

There are two functions to manage the Layer 3 protocols: dev_add_pack and dev_remove_pack. The dev_add_pack function allows to register a new Layer 3 protocol with the Linux network architecture. The dev_add_remove function allows to remove an already registered Layer 3 protocol from the Linux network architecture.

To handle incoming IP packets, the Linux networking code statically registers the function ip_rcv as the Layer 3 handler for the IP protocol. The ip_rcv handler processes all incoming IP packets destined to the local host and forwards IP packets destined to other hosts if the forwarding functionality is enabled in the Linux kernel else the Linux kernel drops IP packets which are not destined to the local host.

4.1.3.3. Layer 4 Protocol Handlers

This section describes how the Linux networking code manages Layer 4 protocols and how an arrived packet is processed from Layer 3 upward in the Linux network stack.

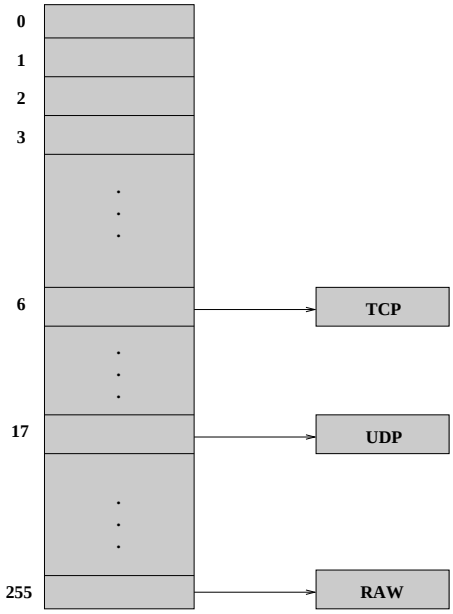


Figure 4.3.: Layer 4 protocol table

The Linux networking code stores all registered Layer 4 protocols in a table named inet_protos [CKHR05, Ben05, WPR+04]. The inet_protos table is a simple array which contains 256 items, for each of the possible Layer 4 protocols. Each layer 4 protocol is described by the data structure net_protocol. The net_protocol consists of three fields: handler, err_handler and no_policy. The function pointer handler points to the handler for incoming packets of a Layer 4 protocol. The function pointer err_handler points to the handler which is used by the ICMP protocol handler to inform a Layer 4 protocol about the reception of an ICMP_UNREACHABLE [ICM81] message.

The Linux networking code provides two functions to manage the Layer 4 protocols: inet_add_protocol and inet_del_protocol. The inet_add_protocol function allows us to register a new Layer 4 protocol with the Linux network stack and the function inet_del_protocol to unregister an already registered Layer 4 protocol.

The Layer 4 protocols ICMP, UDP and TCP are statically added to the inet_protos table and are always available. The IGMP protocol is only registered when the Linux kernel is compiled with support for IP multicast. Not all Layer 4 protocol handlers are handled by the Linux kernel like UDP or TCP protocols. For example, the OSPF protocol is handled by user-space applications.

4.1.4. Packet Transmission

This section discusses packet transmission at the Layer 2 and at the Layer 3 in Linux.

4.1.4.1. Frame Transmission

This section discusses packet transmission [CKHR05, Ben05, WPR+04] at the Layer 2 in Linux.

Every network device driver provides a method for sending a packet over a network. The function pointer hard_start_xmit in the net_device structure points to a driver-specific transmission function. This method is responsible for sending a packet in the form of a socket buffer. A socket buffer passed to the hard_start_xmit contains a physical packet as it should appear on the media, complete with the transmission-level headers. The network device does not need to modify the data being transmitted. The data pointer of the socket buffer points to the packet being transmitted and the len field of the socket buffer is its length in bytes.

In the Linux networking code, higher protocols do not use the hard_start_xmit function of a network device directly. They use the dev_queue_xmit method to send a packet in the form of a socket buffer over a network device. The network device is specified by the dev parameter of the socket buffer that is passed to the dev_queue_xmit function.

In the Linux kernel, a network device can have a queue for outgoing packets, known as the egress queue. Backlog queues for incoming packets are simple FIFO queues but egress queues are much more complex and can be hierarchical, represented by trees of queues. The Linux kernel uses algorithms known as *queueing disciplines* to provide traffic control and quality of service in a network. Queueing disciplines arrange outgoing packets in some specified order for further transmission. When a packet is to be sent to a network interface by the Linux kernel, it is enqueued to the queueing discipline configured for that network interface. The Linux kernel then tries to get as many packets as possible from the queueing discipline and hands them to the network adapter driver.

Some network devices, such as the loopback network device, do not have an egress queue. A packet transmitted over the loopback network device is immediately delivered.

The dev_queue_xmit function places a passed socket buffer in the egress queue of the specified network device by using the queueing discipline of the network device and triggers further handling of packets ready to be sent. The queueing discipline of the network device is responsible for delivering the next packet which is passed to the hard_start_xmit function of the network device for transmission over a network.

The hard_start_xmit function is protected by a spinlock in the net_device structure to serialize concurrent calls of this function. When the hard_start_xmit function returns, it can be called again. Most physical network adapters transmit packets asynchronously and have a limited amount of built-in memory available to store packets that have to be transmitted over a network. The hard_start_xmit function returns as soon as it is done instructing the network device about packet transmission. Therefore, when this memory is exhausted, the network device driver stops any other transmission attempts until the network device has free memory available for further outgoing packets. A network device driver calls the netif_stop_queue function to stop the egress queue of the network device. When the network device is ready to accept packets for transmission, the network device driver calls the netif_wake_queue to enable the egress queue of the network device.

4.1.4.2. Transmission of IPv4 Packets

This section discusses packet transmission [CKHR05, Ben05, WPR+04] at the Layer 3 (IP layer) in Linux.

Transmission of IPv4 packets can be initiated by Layer 4 transport protocols, such as TCP or UDP. The Linux kernel itself can also generate IP packets, e.g. ICMP [ICM81] or IGMP [IGM97] packets. Furthermore, if a computer is configured as a router and the forwarding of IP packets is enabled in the Linux kernel, then received IP packets that are addressed to other remote computers will be forwarded and transmitted by the Linux kernel.

The Linux networking code provides several functions that perform transmission of IP packets. Each of these functions is specially written and optimized for a specific case. The reason for this is that the Layer 4 protocols like TCP prepare and fragment data which they send. In that case, the IP layer does not need to do much work. But the Layer 4 protocol like UDP leave the preparation and fragmentation of data to the IP layer. Each network has a maximum frame size which is called Maximum Transfer Unit (MTU). Only frames of the size which does not exceed the MTU can be transported over the network. Therefore, the IP protocol has to be capable to adapt the size of IP packets to the network MTU. If the MTU of the network is smaller than the size of an IP packet, then the IP packet has to be split into multiple smaller IP packets. For example, the MTU of the Ethernet network is 1500 bytes.

The transport protocols like TCP or SCTP use the function ip_queue_xmit to pass data to the IP layer for transmission. The function ip_queue_xmit receives a pointer to the socket buffer which contains the data for transmission and a flag which indicates whether fragmentation is allowed. The socket buffer provides all the necessary information needed to process the packet by the ip_queue_xmit function.

The transport protocols like UDP or the network protocols like ICMP use the ip_append_data and ip_push_pending_frames functions to pass their data to the IP layer for transmission. The protocol which uses these two functions does not fragment or help to fragment their data. Therefore, the IP layer has to fragment the data if necessary. With these two functions, it is possible to store several transmission requests by calling the function ip_append_data multiple times without actually transmitting anything. The function ip_push_pending_frames flushes the output queue that was created by the function ip_append_data, performs fragmentation if necessary and passes the resulting packets to the next lower protocol layer for transmission. The function ip_append_data does not only buffer data for transmission but also generates data fragments of the size which is easier for fragmentation by the IP layer. Therefore, the IP layer does not need to copy data from one buffer to another while it handles fragmentation. In that case, the performance of the IP layer can be significantly increased.

The routing subsystem of the Linux networking code has to be consulted before a locally generated IP packet or an IP packet forwarded from other remote host can be transmitted over the network. The routing subsystem of the Linux kernel provides several functions to lookup the routing table and the routing cache of the routing subsystem. The result of the lookup operation is stored in the dst field of the structure sk_buff which

represents an IP packet for transmission. The field dst of the structure sk_buff is a pointer to the structure dst_entry and contains among other important fields the function pointer output. All transmissions of IP packets which were generated locally or forwarded from other hosts pass through the function dst_output on their way to a destination host. The function dst_output invokes the function pointer output of the socket buffer which was passed to the function dst_output. The function pointer output of the structure sk_buff will be initialized by the routing subsystem to the function ip_output if the destination address of the IP packet is unicast and will be initialized to the function ip_mc_output if the destination address is multicast. And at last, the function ip_finish_output is invoked to interface with the neighbouring subsystem of the Linux networking code. In an Ethernet network, the neighbouring subsystem is ARP.

4.1.5. Intermediate Functional Block (IFB) Device

The standard Linux network stack can only do traffic shaping on egress queues. IFB [Lin08a] allows us to setup a virtual network device between physical network devices and the Linux network stack. These virtual devices allow us attaching queueing disciplines to incoming packets instead of dropping. An IFB device can use every queueing discipline that can be used with egress queues. Packets are redirected to these devices using tc/action mirred redirect construct. IFB devices provides functionality similar to IMQ [IMQ08].

4.2. Possible Approaches to Protocol Design

This section presents several possible approaches to design the PLACE protocol and discusses their advantages and disadvantages. Which one of the approaches presented below will actually be used to design the PLACE protocol and why will be discussed in detail in Chapter 5.

4.2.1. User-space vs. Kernel-space Implementation

This section discusses the advantages and the disadvantages of the user-space and kernel-space implementation of the PLACE protocol.

There are two kinds of environment in Linux in which software can operate: user space and kernel space [BC05].

Kernel space is a privileged mode of operation in Linux and is used by code compiled into the Linux kernel or loaded as loadable kernel module (LKM) [CKHR05] after the initial boot process. For example, device drivers are executed in kernel space because they have to access and manage hardware. There are low-level functions in kernel space which are not available in user space.

User space is a least-privileged environment in Linux. User applications, for example, daemons, interactive or batch applications, operate in that environment.

The reason for the separation between kernel space and user space is that otherwise user data and kernel data could disturb each other which would result in less performance

and instability of the Linux system.

Both user-space and kernel-space implementations of the PLACE protocol can be designed as a Layer 3, Layer 4 or Layer 5 protocol of the standard Linux network stack. However, a user-space implementation have to be granted root privileges in order to be able to operate in Layer 3 or Layer 4 of the standard Linux network stack. A user-space approach has to use the standard BSD socket API. With a kernel-space approach, the standard BSD socket API is not available for the PLACE protocol and, therefore, a kernel-space implementation of the PLACE protocol has to deal with the standard Linux network stack which is more complex than the BSD socket API.

A kernel-space implementation has several advantages over a user-space implementation. One advantage of a kernel-space implementation is that it is more efficient than a user-space implementation. A user-space approach requires context switches in order to transmit or to receive a packet of the PLACE protocol. In Linux, context switches between user and kernel mode and vice versa are very expensive. Another disadvantage of a user-space implementation is the non-deterministic behaviour of the standard Linux process scheduler. In Linux, a user-space application can be suspended for an arbitrary long time, especially on a heavy-loaded machine. In that case, it will be impossible to simultaneously change TDF of several cluster nodes and to guarantee very low latency for the PLACE protocol.

A user-space implementation has also several advantages over a kernel-space implementation. A user-space implementation is transparent to modifications of the underlying Linux kernel and relies for network communication only on the standard BSD socket API. Therefore, a user-space approach is more portable and is easier to deploy, especially on machines administered by other users or with a different Linux kernel version. Furthermore, it is easier to develop and test a user-space implementation due to ease of modification and deployment. Errors in the kernel space can result in system failure but errors in the user space only cause the termination of the user-space application. In Linux, a user-space program executes in a space isolated from other user-space processes and critical system data. This environment protects the user-space application from mistakes in other processes but the Linux kernel is assumed to be correct and responsible.

4.2.2. Simultaneous Packet Reception

This section discusses possibilities to provide simultaneous packet reception on the cluster nodes of DTVEE.

Since the cluster nodes of DTVEE are connected by an Ethernet LAN, there are two approaches to guarantee that the cluster nodes simultaneously receive a packet: broadcast and multicast communication.

Broadcast packets are received by every network device connected to the same Ethernet broadcast domain. In case of DTVEE, it means that every cluster node of DTVEE receives a broadcast packet which was sent over one of the two local area networks of DTVEE. Broadcast packets tie up system resources as well as consume network bandwidth. Every node in a given broadcast domain has to process each broadcast packet

it receives. When a network device of a node receives a broadcast frame, it generates an interrupt. In turn, each interrupt consumes some amount of processing time by the node. Furthermore, every received broadcast packet is processed by the Linux network stack. Excessive amounts of broadcast traffic not only waste bandwidth but also degrade the performance of every network device attached to the network. Thus, if the PLACE protocol would use broadcast communication to guarantee simultaneous packet reception on the cluster nodes which participate in the same experiment, a cluster node of DTVEE has to process the PLACE packets of an experiment regardless if the cluster node participates in the experiment or not.

A multicast packet is processed only by those nodes which are interested in the packet. A network device passes a multicast frame to the Linux network stack for further processing only if the network device was explicitly told to pass upwards the multicast frames with a given multicast address. A cluster node has to subscribe to a multicast group that is identified by a multicast address in order to receive the multicast packets addressed to that multicast group. Therefore, multicast communication saves system resources because multicast packets which belong to a multicast group that is not subscribed by the node are not processed by the Linux network stack of this node. With multicast communication, it is also possible to reduce the wasted bandwidth and the workload at the cluster nodes if the PLACE protocol would use IP protocol for multicast communication. Modern high-end Ethernet switches support IGMP [IGM97] Snooping with help of which it is possible to reduce the wasted bandwidth on an Ethernet LAN. With IGMP Snooping, an Ethernet switch analyzes all the IGMP packets. When a switch receives an IGMP Join packet from a node for a given multicast address, it adds the port of the node to the multicast list for that group. When the switch receives an IGMP Leave packet, it deletes the port of the node from the multicast list for that multicast group. With IGMP Snooping, Ethernet switches can make intelligent multicast forwarding decisions by examining the contents of the IP header of each received frame.

However, broadcast and multicast communication does not solve the problem in the situation when a node sending a TDF change request for a given experiment participates in the same experiment because the sending node can't predict in advance when other cluster nodes participating in the same experiment receive the TDF change request. One possible solution for this problem is to make sure that the node that can send TDF change requests for a given experiment does not participate in the same experiment. One cluster node of DTVEE could be reserved for this purpose. This cluster node wouldn't participate in any experiments.

4.2.3. Network Layer

This section discusses in which network layer of the Linux network stack the PLACE protocol could be placed.

There are three network layers in which the PLACE protocol could be placed: Layer 3, Layer 4 and Layer 5. All three approaches can be implemented in user space as well as in kernel space.

A user-space implementation of the PLACE protocol has to use the standard BSD

socket API for communication. By placing the PLACE protocol in Layer 4 or Layer 5, it is possible to use multicast or broadcast communication. These approaches has also the advantage that IGMP Snooping could be used. A Layer 5 implementation can use standard UDP sockets and a Layer 4 implementation has to use raw sockets or packet sockets. By placing the PLACE protocol in Layer 3, it is also possible to use multicast and broadcast communication but IGMP Snooping cannot be easily used because a Layer 3 implementation couldn't be able to use the IP protocol for communication. A Layer 3 implementation in user space has to use packet sockets for communication.

A kernel-space implementation of the PLACE protocol cannot use the standard BSD socket API for communication and has to communicate with the Linux network stack directly. By placing a kernel-space implementation in Layer 4 or Layer 5 of the network stack, it is also possible to use multicast or broadcast communication and IGMP Snooping. A Layer 5 implementation cannot use the standard BSD socket API but it can use UDP sockets of the Linux networking code and a Layer 4 implementation can use raw sockets of the Linux networking code. Furthermore, a Layer 4 implementation of the PLACE protocol can create and send IP packets directly. In that case, it has also to provide a Layer 4 packet handler. By placing the PLACE protocol in Layer 3, it is possible to use link layer multicast and broadcast communication but IGMP Snooping cannot be easily used in that case. A Layer 3 implementation has to provide a Layer 3 packet handler to the Linux networking code.

Another important advantage of the Layer 4 and Layer 5 implementations over the Layer 3 implementations is the possibility to transmit packets that are larger than the Ethernet maximum transmission unit (MTU) which is 1500 byte. By using the IP protocol for communication, the PLACE protocol could send packets with size a little smaller then 64 kB because the maximum size of an IP packet is 65535 bytes [IP81].

4.2.4. Packet Latency Minimization

This section discusses several approaches to minimize time of packet processing by the Linux network stack.

Under conditions of high network load, a time to process a packet by the Linux network stack is not deterministic and can vary widely. Therefore, the design of the PLACE protocol has to guarantee a very low latency for PLACE packets even under high network load. There are several approaches which can be used to provide a low latency for the PLACE protocol. These approaches can be partitioned into two groups: latency minimization for incoming and outgoing packets of the PLACE protocol.

In the Linux networking code, outgoing packets are handed to the queueing discipline of the network device which will transmit these packets over a network. The dev_queue_xmit function enqueues a packet in the queueing discipline of the network device which is stored in the dev field of the socket buffer that manages the packet. The queueing discipline of the network device is responsible for scheduling the enqueued packets and for passing them to the network device driver's hard_start_xmit function for sending over a network. The default queueing discipline is a simple FIFO queue called pfifo_fast. The pfifo_fast queueing discipline actually consists of three FIFO bands. A

very long time can pass before an enqueued packet is handed to the hard_start_xmit function for transmission over a network. One possible approach to minimize the delay of the packets of the PLACE protocol in the queueing discipline of a network device is to assign to a network device a new queueing discipline which will prefer the packets of the PLACE protocol over other packets and will pass them to the hard_start_xmit function of the network device driver before other packets. One possibility to achieve that is to use the prio queueing discipline. The prio queueing discipline is a priority queueing discipline and can have multiple priority queues. In the prio queueing discipline, packets are first classified using filters and then enqueued into different priority queues, which by default are three. Packets are scheduled from the head of a given queue only if all queues of higher priority are empty. Within each of the priority queues, packets are scheduled in FIFO order. By assigning a prio queueing discipline with at least two priority queues (the queue with the highest priority is only for the packets of the PLACE protocol) to each network device over which the packets of the PLACE protocol could be sent, we could guarantee that the packets of the PLACE protocol are sent first.

Latency minimization for incoming packets of the PLACE protocol is more complex than for outgoing packets because ingress queue are simple FIFO queues in the Linux kernel. It is not possible to assign a prio queueing discipline or any other queueing discipline to an ingress queue. Furthermore, only network device drivers which do not use NAPI put incoming packets into ingress queues, also known as backlog queues which is one per CPU. Network device drivers that use NAPI do not put incoming packets into backlog queue and directly call the netif_receive_skb for packet processing. In order to minimize latency for the incoming packets of the PLACE protocol, the Linux network stack has to process the packets of the PLACE protocol first. One way to achieve this is to use an IFB device. To guarantee that the incoming packets of the PLACE protocol are processed by a cluster node before other received packets, an IFB device has to be installed on each cluster node and all incoming traffic of a cluster node has to be forwarded to the IFB device before it travels the Linux network stack upwards. That can be achieved by configuring a traffic control filter on each physical network device of a cluster node. Furthermore, a prio queueing discipline which prefers the packets of the PLACE protocol over other packets has to be assigned to the egress queue of the IFB device. An IFB device will enqueue packets from its egress queue by using the prio queueing discipline and put them into backlog queues of the Linux network stack. Because the packets of the PLACE protocol are placed in the priority queue with the highest priority, they will be put into backlog queues first and, therefore, the packets of the PLACE protocol will be processed before other received packets.

Another important aspect of packet latency minimization is to ensure that the packets of the PLACE protocol are forwarded with minimum delay by the network switches. The Cisco Catalyst 2950 [Cis08a] and 3550 [Cis08b] switches which build the control network of DTVEE support QoS with egress queueing and scheduling. Without QoS, the Cisco switches of the control network offer only best-effort service to each packet, regardless of the packet contents or size and they transmit a packet without any assurance of delay bounds or reliability. By using the QoS feature of the Cisco switches, we can prioritize the PLACE packets and, therefore, ensure that these packets are forwarded with the

minimum possible delay by the Cisco switches. The Cisco switches can classify the received packets either by prioritization values in the VLAN tag of the Layer 2 frames or by prioritization values in the IP header (ToS field) of the Layer 3 packets. A Layer 4 and a Layer 5 implementations of the PLACE protocol could use either the VLAN tag or the ToS field of the IP header to ensure that the PLACE packets are handled with the highest priority by the switches of the control network of DTVEE. A Layer 3 implementation of the PLACE protocol can only use the VLAN tag in the Layer 2 frames to assign the highest priority to packets of the PLACE protocol.

4.2.5. Simultaneous Independent Experiments

This section discusses different approaches to support simultaneous independent experiments by the PLACE protocol.

Each cluster node which participates in an experiment has to know in which one it participates and has to accept only those packets that belong to this experiment. Therefore, the PLACE protocol has to provide a possibility to distinguish between packets which belong to different experiments.

With multicast communication, the PLACE protocol can use the multicast address of a packet to distinguish between packets of different experiments. Therefore, each experiment must have a multicast address that can identify it from other simultaneous experiments. The multicast address of the packets that belong to the same experiment is the identifier of this experiment. This approach has the advantage that a cluster node will receive only the packets of the PLACE protocol which belong to the experiment of the cluster node and will not waste CPU cycles to processing packets of other simultaneous experiments. The PLACE protocol can use an IP multicast address as the identifier of an experiment, if the protocol will be placed in Layer 4 or Layer 5 of the Linux network stack. The IPv4 local scope multicast address range 239.255.0.0/16 [Adm98] provides exactly 65536 multicast addresses which is the number of maximum experiments supported by the PLACE protocol. The PLACE protocol also can use a link layer multicast address as the identifier of an experiment, if the protocol will be placed in Layer 3 of the Linux network stack. In that case, the PLACE protocol can use the user-defined Ethernet multicast address range 03:00:00:01:00:00 – 03:00:40:00:00:00 [Eth08].

With broadcast communication, the PLACE protocol cannot use the IP address or the link layer address of a received packet to find out to which experiment the packet belongs. In that case, the payload of each packet of the PLACE protocol has to provide additional information that will reveal the experiment to which the packet belongs. This can be achieved by providing a field in each packet of the PLACE protocol that will store the identifier of the experiment to which the packet belongs. This approach has the disadvantage that each cluster node will receive and inspect even packets that belong to experiments in which the cluster node does not participate. In that case, the cluster node will waste CPU cycles.

Chapter 5.

Protocol Design

This chapter presents the design of the PLACE protocol and the design decisions which are based on the design issues discussed in the previous chapter. The protocol design provides a basis for the implementation of the PLACE protocol. First, the overall architecture of the protocol is presented and after that the most important components of this architecture are discussed more in detail.

5.1. Architecture

This section presents the overall architecture of the PLACE protocol.

The protocol architecture consists of two major parts: the generic part of the protocol and the PLACE protocol itself. The Figure 5.1 shows the overall protocol architecture. The PLACE protocol is only a minor part of the protocol architecture and relies heavily on the generic part of the protocol. The protocol architecture consists of these two parts because the generic part of the protocol can be used not only by the PLACE protocol but also by other protocols with similar requirements which are distribution of data to multiple receivers with minimum possible delay, e.g. the protocol for sending CPU load of a cluster node to the coordinator.

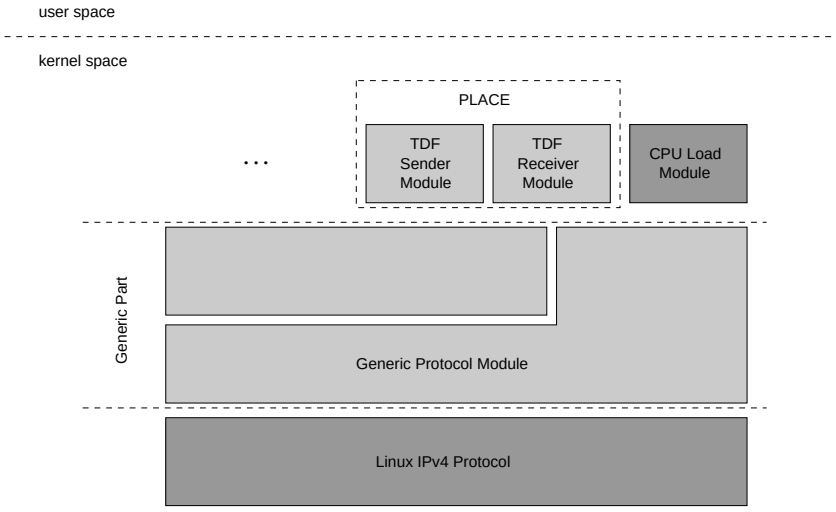


Figure 5.1.: PLACE Architecture

Both parts of the protocol are placed in the Linux kernel space of the domain dom0 in order to minimize the bad effects of the non-deterministic process scheduling of the standard Linux kernel on the packet latency.

The generic part of the PLACE protocol and the PLACE protocol itself are placed in Layer 4 of the Linux network stack and use the Linux IPv4 protocol for communication. The implementation in the Layer 3 of the Linux network stack has no real advantages over the implementation of the protocol in the Layer 4. And furthermore, the Layer 3 implementation can only handle packets of the size that does not exceed the Ethernet MTU which is 1500 bytes.

5.2. Generic Part

This section discusses the design of the generic part of the PLACE protocol.

The generic part of the protocol is the most important part of the protocol architecture and provides a low-latency multicast communication protocol to the PLACE protocol. It also can be used by other protocols which desire to distribute data to multiple receivers and that with minimum delay.

The generic part of the protocol consists of two modules: the generic protocol module and the experiment module. In the following sections, the design of these modules is discussed in detail.

5.2.1. Generic Protocol Module

This section discusses the design of the generic protocol module.

The generic protocol module is a loadable kernel module. The main goal of the generic protocol module is to provide the multicast communication and the packet priorities to the higher protocols like the PLACE protocol.

The generic protocol directly uses the Linux IPv4 network protocol to transmit its packets and to provide multiple packet priorities. Therefore, the destination of a generic protocol packet is identified by an IPv4 address which can be arbitrary unicast, multicast or broadcast IPv4 address. The generic protocol does not only provide the multicast communication to the higher protocols but also the unicast and the broadcast communication. But the multicast communication is the major goal of the generic protocol.

Because the generic protocol uses the IPv4 protocol to transmit its packets, the packets of the generic protocol cannot be larger than an IPv4 packet but the generic protocol packets can exceed the Ethernet MTU. In that case, the fragmentation of the generic protocol packets that are bigger than this MTU is handled by the Linux IPv4 protocol.

As mentioned before, the most important goal of the generic protocol is to provide the multicast communication and packet priorities to the higher protocols such as the PLACE protocol. But the IPv4 protocol already provides this functionality for the IPv4 packets. Therefore, it seems that the generic protocol does not provide any additional functionality which isn't provided by the IPv4 protocol. The IPv4 protocol supports only 256 different higher protocols and many of the IPv4 protocol values are already reserved and cannot be used. Therefore, the generic protocol uses only one IPv4 protocol value

and provides its own protocol field for demultiplexing of the higher protocols. With the generic protocol, it is possible to support more higher protocols which need the multicast communication and the packet priorities than the IPv4 protocol could support.

5.2.1.1. Protocol Demultiplexing

The generic protocol uses an unreserved IPv4 protocol value to identify its packets and registers itself to the Linux networking code as the receiver for these packets. The packet header of the generic protocol contains a protocol field which is used to demultiplex packets of the higher protocols that transmit their packets on behalf of the generic protocol. The protocol field must be large enough to support at least 256 different protocols.

The generic protocol allows to register callback functions in order to handle packet reception of the higher protocols. A packet arrived for a higher protocol will be passed to the callback function which was registered for the protocol to which this packet belongs. Each callback function will not only be associated with a protocol value but also an IPv4 address. This means, that an arrived packet is passed to a registered callback function only if the protocol value and the destination IPv4 address of the packet are identical to the protocol value and the destination IPv4 address of the callback function.

The destination IPv4 address associated with a callback function is not a necessary attribute and can be a wildcard IPv4 address. In that case, the generic protocol protocol passes all received packets to this callback function if the protocol value of the packet and the protocol value associated with the callback function are the same. In that case, the destination IPv4 address of the packet isn't considered by the generic protocol.

It is also possible to register multiple callback functions which have the same protocol value and the destination IPv4 address. In that case, a received packet is delivered to each callback function which has the same protocol value and the same destination IPv4 address as in the received packet.

Furthermore, the generic protocol also allows to register callback functions which should receive every packet destined to any higher protocols.

5.2.1.2. Packet Priority and Latency

The generic protocol supports 8 (0-7) different packet priorities. Priority level 7 is the highest priority level, and priority level 0 is the lowest. The priority of a generic protocol packet is stored in the ToS (Type of Service) [DSC98] field of the IPv4 header.

The priority of a generic protocol packet indicates the importance of this packet and has a large effect on the latency of the packet. The higher the priority of a packet is, the more important is this packet and the shorter is the latency of this packet.

In addition to the ToS field in the IPv4 header, the generic protocol uses egress queue scheduling in the Cisco Catalyst 2950 and 3550 switches of the control network of DTVEE and ingress and egress queue scheduling with the Ethernet network adapters of the cluster nodes of DTVEE in order to provide 8 different packet priorities and to minimize the latency of the generic protocol packets with a high priority.

The Cisco Catalyst 2950 [Cis08a] and 3550 [Cis08b] switches of the DTVEE control network do not support ingress queue scheduling. Therefore, the egress queue scheduling on the cluster nodes is very important to achieve a low latency for the packets of the generic protocol. It is especially important on the cluster nodes which can send generic protocol packets because a large ingress queue in a switch can drastically increase the latency of a packet. In order to avoid this situation, we must ensure that the ingress queues of the switches are never very large.

The generic protocol uses the priority queueing discipline in order to guarantee that the packets of the generic protocol are sent first if there are several packets ready to be sent in the outgoing queue of the network adapter which is connected to the control network. In addition to the priority queueing discipline, the generic protocol also uses the token bucket queueing discipline in order to ensure that the ingress queues of the Cisco switches are small.

The Cisco Catalyst 2950 and 3550 switches of the DTVEE control network support egress queue scheduling and allow to prioritize the packets of the generic protocol and, therefore, it allows us to minimize the delay of these packets in the output queues of the switches. The Cisco switches support the strict priority scheduling. The Cisco switches can read packet priorities stored in ToS field of the IPv4 header and place a packet into the output queue which is associated with the packet priority in the ToS field of the IPv4 header. Packets from the output queue with the highest priority are sent first and if this queue is empty then packets from the output queue with the second highest priority are sent and so on. Furthermore, the Cisco 2950 and 3550 switches also support weighted round-robin queue scheduling which avoids the starvation of the queues with lower priorities if the queue with the highest priority is never empty.

In order to further minimize the delay of the generic protocol packets, the generic protocol also uses the ingress queue scheduling on the cluster nodes of DTVEE. The ingress queue scheduling ensures that the packets of the generic protocol are handled first by the Linux kernel. Each cluster node in DTVEE has two network interfaces, the first network interface is connected to the control network and the second network interface is connected to the experiment network. Thus, high traffic load in the experiment network can increase the delay of a generic protocol packet received over the control network because the Linux kernel has to handle a large amount of the packets received over the experiment network.

The ingress queue scheduling can be realized with the IFB device. In that case, all incoming packets from the both network adapters of the cluster node are forwarded to the IFB device. A priority queueing discipline installed on this IFB device ensures that the packets of the generic protocol are handled first by the Linux kernel.

5.2.1.3. External Interface

This section describes the external interface of the generic part of the PLACE protocol in an abstract way and, later during the implementation of the PLACE protocol, this interface can be changed in order to improve the performance of the generic part. The number of parameters and the parameters themselves will remain unchanged and only

the parameter passing can be changed for efficiency and performance.

send_packet(IP address, protocol, priority, data, len) sends data of length len with priority priority as a packet of protocol protocol to address IP address.

add_protocol(IP address, protocol, func_ptr(data, len)) registers packet handler func_ptr(data, len) as packet handler for packets of protocol protocol and which are destined to address IP address.

del_protocol(IP address, protocol, func_ptr(data, len)) removes packet handler func_ptr(data, len) as packet handler for packets of protocol protocol.

5.2.1.4. /proc Interface

This section describes the /proc-Interface of the generic part of the PLACE protocol.

/proc/tvee/tdf/generic/stats contains generic protocol statistics.

5.2.2. Experiment Module

This section discusses the design of the experiment module.

The experiment module is also a loadable kernel module and introduces the notion of an experiment as a method for addressing a set of cluster nodes in DTVEE which participate in the same network experiment. An experiment is simply an integer value which identifies a set of cluster nodes. The experiment module provides an additional and simple form of addressing of a set of cluster nodes in DTVEE which belong to the same network experiment.

With the generic protocol, it is possible to send packets to a specified IPv4 address and to receive packets destined to a specified IPv4 address. But the experiment module allows us to send packets to a set of cluster nodes which is identified by an integer value or to receive packets destined to this set of cluster nodes.

With the generic protocol, the higher protocols can use the IPv4 multicast communication to efficiently distribute data to a set of cluster nodes in DTVEE. The cluster nodes in this set have only to join an IPv4 multicast address and wait for packets which will be sent to this IPv4 multicast address. Each cluster node that wants to send data to this set of cluster nodes has to know the IPv4 multicast address of this set of cluster nodes.

With the experiment module, a set of cluster nodes is identified by an integer value which is mapped to an IPv4 multicast address by the experiment module. The experiment module hides this mapping and provides an uniform interface to the higher protocols which can use an abstract integer identifier to address a set of cluster nodes in DTVEE and send data to these cluster nodes or to receive data destined to this set of cluster nodes.

The advantage that provides the experiment module is that the mapping of an experiment identifier to the corresponding IPv4 multicast address is hidden in the experiment

module and can be changed without affecting the higher protocols. Furthermore, the mapping function from an experiment identifier to the corresponding IPv4 multicast address has not to be defined in each of the higher protocols which use the notion of experiment to address a set of cluster nodes.

The experiment module is directly placed above the generic protocol module and uses only the interface provided by the generic protocol to send packets or to register a receiver for packets. The only functionality provided by the experiment module is the mapping function that maps a specified experiment identifier to the corresponding IPv4 multicast address.

5.2.2.1. External Interface

This section describes the external interface of the experiment module of the PLACE protocol in an abstract way and, later during the implementation of the PLACE protocol, this interface can be changed in order to improve the performance of the experiment module. The number of parameters and the parameters themselves will remain unchanged and only the parameter passing can be changed for efficiency and performance.

send_packet(experiment, protocol, priority, data, len) sends data of length len with priority priority as a packet of protocol protocol to experiment with identifier experiment.

add_protocol(experiment, protocol, func_ptr(data, len)) registers packet handler func_ptr(data, len) as packet handler for packets of protocol protocol and which are destined to experiment experiment.

del_protocol(experiment, protocol, func_ptr(data, len)) removes packet handler func_ptr(data, len) as packet handler for packets of protocol protocol.

5.2.2.2. /proc Interface

This section describes the /proc-Interface of the experiment module of the PLACE protocol.

/proc/tvee/tdf/experiment/stats contains statistics of the experiment module.

5.3. PLACE

This section presents the design of the **P**rotocol for **L**atency **A**ware **C**hanging of **E**pochs (PLACE). At first, the design of the sending and the receiving instance of the PLACE protocol will be presented. Finally, several sequence diagrams will be presented which show the interactions between the modules of the PLACE protocol in several most important situations.

The main goals of the PLACE protocol is to distribute TDF change requests to a specified set of cluster nodes in DTVEE participating in the same network experiment with

lowest possible delay, but the most important is to simultaneously deliver a TDF change request to the destination cluster nodes. In order to achieve these goals, the PLACE protocol heavily relies on the generic protocol module and the experiment module.

The PLACE protocol distinguishes between the sending and the receiving instances.

The sending instance of the PLACE protocol can only send TDF change requests triggered by an external source. A TDF change request is simply an IPv4 packet that contains, among other things, a TDF value destined to a specified set of cluster nodes of DTVEE participating in the same experiment.

The receiving instance of the PLACE protocol does not send any network packets and only listens for any incoming TDF change requests. Upon receiving a TDF change request, the receiver instance of a cluster node initiates the switching of the TDF value of the Xen hypervisor on the cluster nodes.

The PLACE protocol uses sequence numbers in its packets in order to serialize concurrent TDF change requests and to provide the possibility for receiving instances of the PLACE protocol to detect a packet loss.

In the following sections, the design of the sending and the receiving instances of the PLACE protocol are described more in detail.

5.3.1. TDF Sender Module

This section presents the design of the sending instance of the PLACE protocol.

The TDF sender module is a loadable kernel module and realizes the sending instance of the PLACE protocol. The TDF sender module provides the capability to send a TDF change request to a set of cluster nodes in DTVEE which participate in the same network experiment identified by an integer value.

The TDF sender module is able to send TDF change requests to multiple experiments simultaneously. Because each experiment has an independent sequence number for its TDF change requests, the TDF sender module has to support up to 65536 independent experiments simultaneously and has to manage the sequence numbers of these experiments. The TDF sender module maintains an independent sequence number for each of 65536 possible experiments. For each outgoing TDF change request destined to a specified experiment, the TDF sender module automatically increments the sequence number of this experiment.

5.3.1.1. External Interface

This section describes the external interface of the TDF sender module of the PLACE protocol in an abstract way and, later during the implementation of the PLACE protocol, this interface can be changed in order to improve the performance of the TDF sender module. The number of parameters and the parameters themselves will remain unchanged and only the parameter passing can be changed for efficiency and performance.

send_tdf(experiment, tdf) sends a TDF packet with TDF tdf to experiment experiment.

get_stats() returns statistics of the TDF sender module.

5.3.1.2. /proc Interface

This section describes the /proc-Interface of the TDF sender module of the PLACE protocol.

/proc/tvee/tdf/sender/send_tdf enables to send a TDF packet from user space.

/proc/tvee/tdf/sender/stats contains statistics of the TDF sender module.

5.3.2. TDF Receiver Module

This section presents the design of the receiving instance of the PLACE protocol.

The TDF receiver module is also a loadable kernel module and realizes the receiving instance of the PLACE protocol. The main goal of the TDF receiver module is to listen for incoming TDF change requests destined to an experiment, to read the TDF value stored in these TDF change requests and to adjust the TDF value of the Xen hypervisor on a cluster node. Each cluster node can participate at most in one experiment and, therefore, the TDF receiver module is able to receive TDF change requests only belonging to a single experiment.

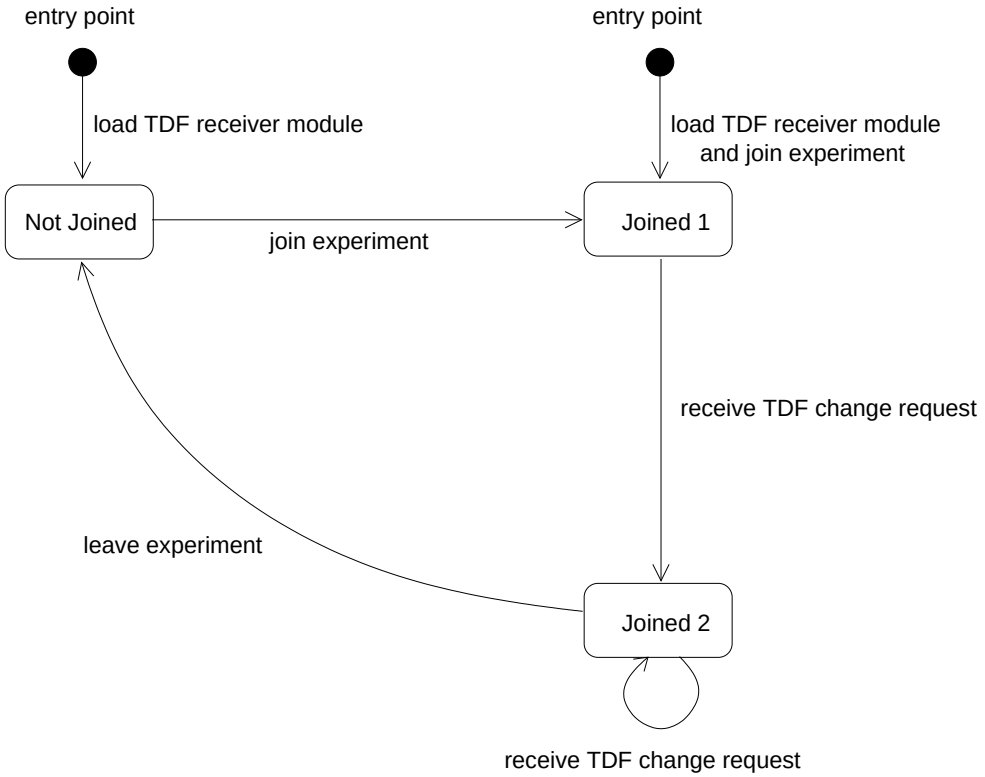


Figure 5.2.: TDF Receiver Module State Machine

The TDF receiver module realizes the finite state machine showed in the Figure 5.2. The finite state machine has three states: Not Joined, Joined 1 and Joined 2. The receiving instance of the PLACE protocol can start either in the state Not Joined or in the state Joined 1. The TDF receiver module is in the state Not Joined after it was loaded and didn't join an experiment. And the TDF receiver module is in the state Joined 1 after it has been loaded and joined a specified experiment. It is possible to pass an experiment identifier to the TDF receiver module at loading time. In that case, the TDF receiver module will join the specified experiment directly after it has been loaded and will start in the state Joined 1. The receiving instance of the PLACE protocol will stay in the state Joined 1 until it receives the first TDF change request of the newly joined experiment. After that the TDF receiver module goes to the state Joined 2 and it will stay in this state until it leaves the newly joined experiment or join another experiment. The goal of the state Joined 1 is to figure out the current packet sequence number used in the experiment that was newly joined by the TDF receiver module. The sequence number in a TDF change request makes it possible for the TDF receiver module to detect a packet loss and to report it. The TDF receiver module has to distinguish between the state Joined 1 and the state Joined 2 because it does not know the sequence number of the first TDF change request that will be received after the receiving instance has joined an experiment.

5.3.2.1. External Interface

This section describes the external interface of the TDF receiver module of the PLACE protocol in an abstract way and, later during the implementation of the PLACE protocol, this interface can be changed in order to improve the performance of the TDF receiver module. The number of parameters and the parameters themselves will remain unchanged and only the parameter passing can be changed for efficiency and performance.

set_experiment(experiment) joins or leaves experiment experiment.

get_experiment() returns currently joined experiment.

set_change_tdf(yesno) enables or disables local TDF changing.

get_change_tdf() shows if TDF changing is enabled or disabled.

get_stats() returns statistics of the TDF receiver module.

5.3.2.2. /proc Interface

This section describes the /proc-Interface of the TDF receiver module of the PLACE protocol.

/proc/tvee/tdf/receiver/experiment allows to join or to leave experiment from user space and contains currently joined experiment.

/proc/tvee/tdf/receiver/change_tdf allows to enable or to disable TDF changing and shows if TDF changing is currently enabled or disabled.

/proc/tvee/tdf/receiver/stats contains statistics of the TDF receiver module.

5.3.3. Sequence Diagrams

This sections presents several important interactions of the PLACE protocol modules in different situations for a better understanding of the overall architecture of the PLACE protocol.

5.3.3.1. Send TDF Change Request

The Figure 5.3 shows the normal course of interactions between the TDF sender module, the experiment module, the generic protocol module and the Linux IPv4 protocol when the sending instance of the PLACE protocol sends a TDF change request.

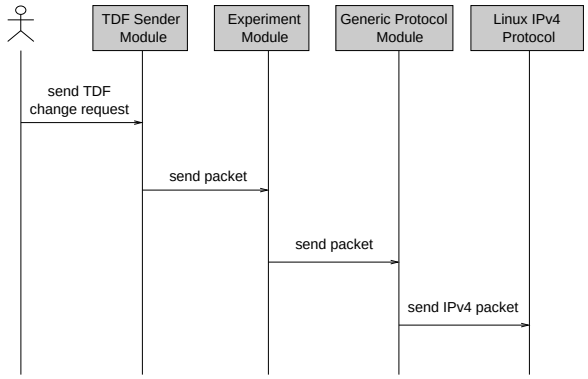


Figure 5.3.: Send TDF Change Request Sequence Diagram

5.3.3.2. Receive TDF Change Request

The Figure 5.4 shows the normal course of interactions between the Linux IPv4 protocol, the generic protocol module, the TDF receiver module and the Xen hypervisor when the receiving instance of the PLACE protocol receives a TDF change request.

5.3.3.3. Join Experiment

The Figure 5.5 shows the normal course of interactions between the TDF receiver module, the experiment module, the generic protocol module and the Linux IPv4 protocol when the receiving instance of the PLACE protocol joins an experiment.

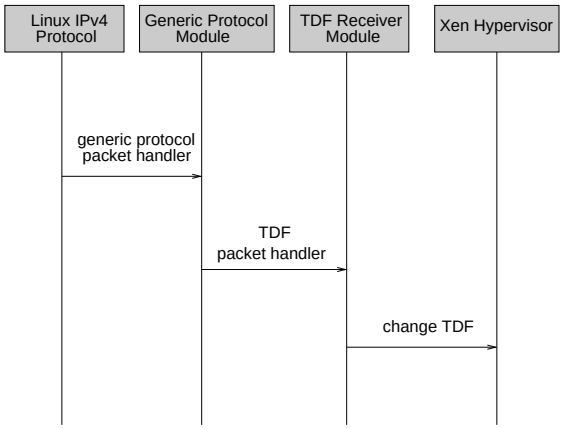


Figure 5.4.: Receive TDF Change Request Sequence Diagram

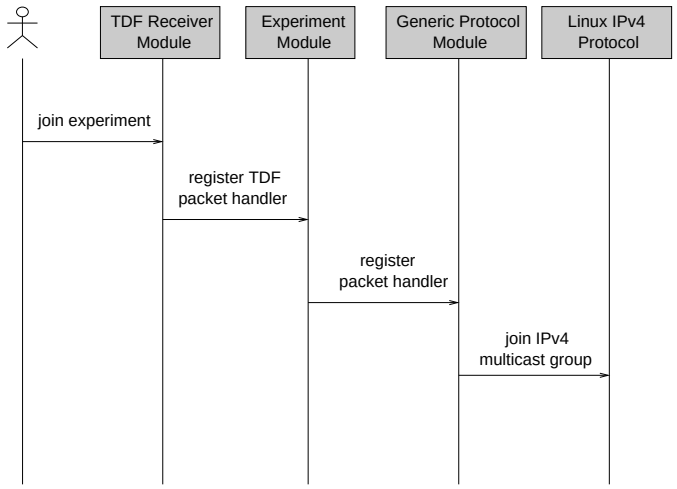


Figure 5.5.: Join Experiment Sequence Diagram

5.3.3.4. Leave Experiment

The Figure 5.6 shows the normal course of interactions between the TDF receiver module, the experiment module, the generic protocol module and the Linux IPv4 protocol when the receiving instance of the PLACE protocol leaves the previously joined experiment.

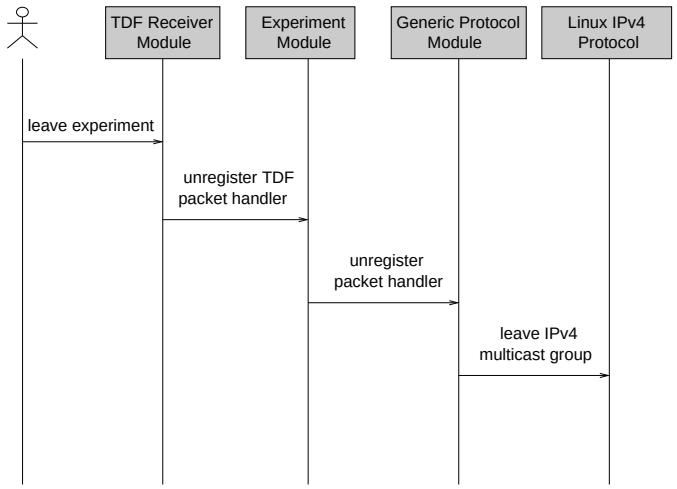


Figure 5.6.: Leave Experiment Sequence Diagram

Chapter 6.

Protocol Implementation

This chapter presents the implementation details of the protocol components described in the previous chapter. Each part of the protocol is implemented in kernel space as a single kernel module and, therefore, all components of the protocol were completely written in C.

6.1. Generic Part

This section describes the implementation details of the generic part of the PLACE protocol.

6.1.1. Generic Protocol Module

This section describes the implementation details of the generic protocol module.

The generic protocol uses the IPv4 protocol [pro08] value 254 to transport its packets over network. But this protocol value is not hardcoded and can be changed by a kernel module parameter at loading time of the generic protocol module.

At loading time, the generic protocol module must be provided with the name of a valid Ethernet network device. The generic protocol module uses only the specified network interface for sending and receiving of the generic protocol packets and it is not possible to change the specified network interface at run-time of the generic protocol module. Furthermore, is is also not possible to use more than one network interface with the generic protocol module. In order to use another network interface, the generic protocol module must be reloaded.

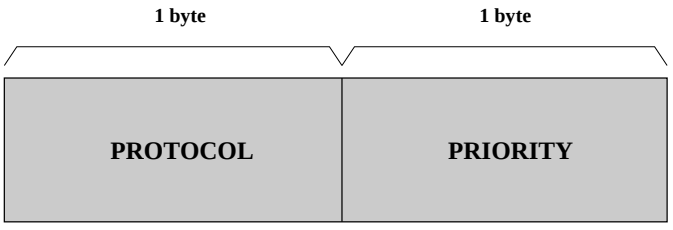


Figure 6.1.: Generic protocol header

Every packet of the generic protocol starts with the generic protocol header. The Figure 6.1 shows the header of the generic protocol. The header of the generic protocol

consists of two fields of size 1 byte. The purpose of the first field in the generic protocol header, which is called protocol, is demultiplexing of received packets to the higher protocols that use the generic protocol for communication. The second field of the generic protocol header, which is called priority, stores the priority of the generic protocol packet. The valid values for this field are only 0-7, 0 represents the lowest packet priority and 7 is the highest packet priority. This field is somewhat redundant because the generic protocol uses the DSCP field in the IPv4 header to provide packet priorities. And only the DSCP field is used for packet scheduling in cluster nodes and Ethernet switches of DTVEE. The generic protocol itself does not use the field priority. That field exist for the purpose of debugging of the generic protocol and additionally allows to pass a priority value of a packet to the generic protocol module efficiently.

At loading time, the generic protocol module registers a packet handler by use of the inet_add_protocol function of the Linux networking code in order to receive incoming packets of the generic protocol.

The generic protocol module provides two functions for packet sending:
generic_send_packet and generic_alloc_skb. The function generic_send_packet expects two parameters: a pointer to a sk_buff data structure representing a packet that should be sent and an IPv4 destination address. The sk_buff data structure representing a packet to be sent must have enough reserved space for the IPv4 header and the Ethernet header. Furthermore, the field data of the sk_buff data structure must point to the generic protocol header of the packet and the fields of the generic protocol header should be filled with valid values. The generic_send_packet function fills the IPv4 header and sends the passed packet to the destination which is identified by the IPv4 address provided to the function. Therefore, the user of the generic_send_packet function is responsible for allocation of the packet and the filling of the generic protocol header. In order to make it easier for users to create generic protocol packets, the generic protocol module also provides the generic_alloc_skb function which allocates a generic protocol packet, fills the generic protocol header of the packet and returns a pointer to the packet buffer where the payload of the packet is located and so makes it possible for users of the generic protocol module to fill the allocated packet with data. The generic protocol module uses pointers to the sk_buff data structure in order to avoid data copying which affects the efficiency of the generic protocol.

For the purpose of debugging, the generic protocol module provides the function generic_get_stats which returns a pointer to the static variable of type struct generic_stats. It provides statistical information to the users of the generic protocol module: number of sent or received packets etc.

6.1.1.1. Protocol Demultiplexing

The generic protocol supports up to 256 higher protocols. Furthermore, the generic protocol module provides the possibility to register a packet handler which will receive packets of all 256 higher protocols without registering of 256 packet handlers. In order to implement this functionality, the generic protocol module manages the packet handlers which want to receive packets of one single higher protocol in the hash table

called gtype_base and it manages the packet handlers which want to receive the packets belonging to any higher protocols in the doubly-linked list called gtype_all. The Figure 6.2 shows the gtype_base and the gtype_all data structures.

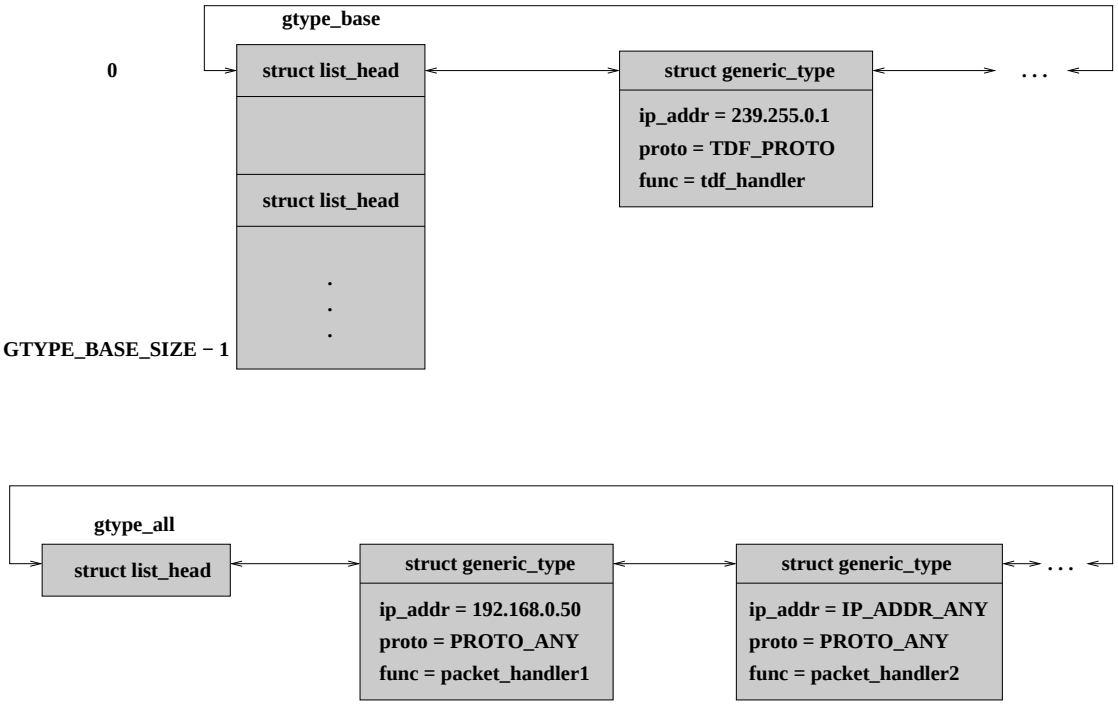


Figure 6.2.: gtype_base and gtype_all data structures

The packet handlers are managed by the structure struct generic_type. This structure has 4 fields: ip_addr, proto, func and list. The variable ip_addr holds an IPv4 address or can be the wildcard IP_ADDR_ANY. If ip_addr is not the wildcard then the packet handler can receive only packets which are destined to the IPv4 address stored in this variable. The variable proto holds the protocol value of a higher protocol which uses the generic protocol to transport its packets. It can be either a valid protocol value or the wildcard PROTO_ANY. If proto is the wildcard then the packet handler receives packets of any higher protocols. The variable func is the function pointer to a packet handler. The list variable of the generic_type structure is used to manage packet handlers in the hash table gtype_base and in the doubly-linked list gtype_all.

The generic protocol module provides two functions to add and to remove a packet handler for packet receiving: generic_add_protocol and generic_del_protocol. Both functions receives a pointer to a filled generic_type structure. The variables ip_addr, proto and func of that structure must be valid. A generic_type structure may not be freed until it was unregistered by calling the generic_del_protocol function. The function generic_del_protocol may only be passed generic_type structures which were already registered with the function generic_add_protocol because the generic protocol module uses the passed generic_type structures in order to implement the hash table gtype_base and the doubly-linked list

gtype_all.

6.1.1.2. Packet Priority and Latency

This section describes the configuration of cluster nodes and Cisco switches of the control network of DTVEE. This configuration is highly necessary in order to provide priorities and low latency for packets of the generic protocol. In every cluster node which intends to send or receive generic protocol packets and in every Cisco switch of the control network of DTVEE, packet scheduling has to be enabled and configured which will guarantee that packets of the generic protocol are processed by the cluster nodes and the switches with the lowest possible delay.

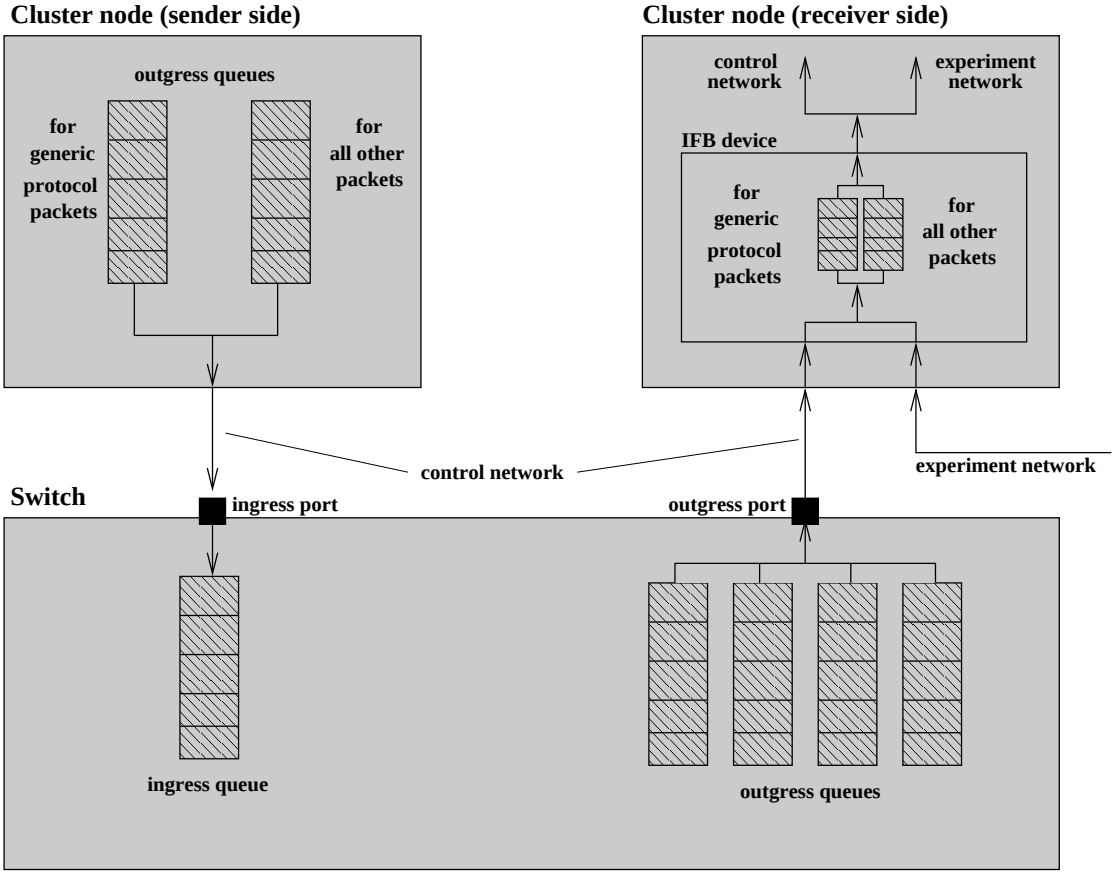


Figure 6.3.: Packet priorities and scheduling

The Figure 6.3 shows the configuration of ingress and outgress queues in cluster nodes and Cisco switches.

On all outgress ports of every Cisco switch in the control network of DTVEE, strict priority scheduling with four egress queues is enabled. On all ingress ports of the Cisco switches, trusting DSCP is enabled. Cisco switches do not trust DSCP by default. Cisco switches uses the dscp-to-cos map to convert a DSCP value to an integer in range from 0

to 7 which represents the priority of a packet. This priority value is mapped to one of the outgress queues of the outgress port to which the packet will be forwarded. Because Cisco switches 2950 and 3550 have only four egress queues and eight packet priorities, some packet priorities are mapped to the same outgress queue. By default, Cisco switches 2950 and 3550 map the packet priorities 0 and 1 to the outgress queue with the lowest priority, the packet priorities 2 and 3 to the outgress queue with the third-highest priority, the packet priorities 4 and 5 to the outgress queue with the second-highest priority, and the packet priorities 6 and 7 to the outgress queue with the highest priority. The generic protocol module uses also a map which makes the opposite mapping and converts a priority in range from 0 to 7 to a DSCP value. The generic protocol module implements this mapping by using the prio_to_dscp_map array. The generic protocol module uses the same mapping which is also used by Cisco switches. But it can be modified at module loading time.

On every cluster node which wants to send packets of the generic protocol, the outgress queueing discipline which guarantees that generic protocol packets are sent before other packets over network has to be configured. Furthermore, the outgress queueing discipline of the cluster node has to restrict the sending bandwidth of the cluster node in order to guarantee that the ingress queue of the switch port to which the cluster node is connected does not become too large. In order to achieve that, a priority queueing discipline with two bands is installed as the root queueing discipline on the control network interface. The first band of the root queueing discipline is a FIFO queue which has the highest priority and stores only outgoing packets of the generic protocol. The second band of the priority queue is a token-bucket queueing discipline. It restricts the outgoing bandwidth of the cluster node for all other packets. The first band of the priority queueing discipline, that stores packets of the generic protocol, does not differentiate between generic protocol packets of different priorities because this FIFO queue is expected to be almost always empty and, therefore, the delay of packets in this queue is insignificant. Furthermore, additional eight FIFO bands for every possible packet priority of the generic protocol would only cost unnecessary overhead to the cluster nodes.

On every cluster node which wants to receive packets of the generic protocol, an IFB device for ingress scheduling in the cluster node has to be configured. A cluster node is connected to the control network as well as to the experiment network. Thus, packets that are received from the experiment network can cause a delay in processing of incoming generic protocol packets. In order to avoid this situation, the network traffic from the control network and the experiment network is forwarded to an IFB device before it will be processed further by the Linux network stack. Furthermore, a priority queueing discipline with 2 bands is configured on the IFB device in order to guarantee that packets of the generic protocol are processed first.

6.1.1.3. Module Parameters

This section describes the module parameters of the generic protocol module.

ifname The parameter ifname is the name of a network interface which will be used

by the generic protocol to send and to receive its packets. This parameter is mandatory and is of type string.

proto The parameter proto is the IPv4 protocol value used by the generic protocol module to identify is packets. This parameter is not mandatory and is equal to 254 by default. The parameter is of type unsigned integer.

prio_to_dscp_map The parameter prio_to_dscp_map is a map used by the generic protocol module to convert a packet priority in range from 0 to 7 to a DSCP value. This parameter is not mandatory and is of type unsigned integer array of size 7. The parameter prio_to_dscp_map is equal to 0 10 18 26 34 56 48 56 by default.

6.1.1.4. /proc Interface

This section describes the /proc interface to the generic protocol module.

/proc/tvee/tdf/generic/stats The /proc/tvee/tdf/generic/stats proc-file is a read-only file which contains various statistical counters, such as number of packets sent and received.

6.1.2. Experiment Module

This section describes the implementation details of the experiment module.

The generic protocol module uses IPv4 addresses to send generic protocol packets and to register handlers for incoming generic protocol packets. In contrast to the generic protocol module, the experiment module does not use IPv4 addresses to send generic protocol packets and to register handlers for incoming generic protocol packets. The experiment module provides the same external interface to users as the generic protocol module but it uses experiment identifiers instead of IPv4 addresses for packet sending and receiving.

The experiment module heavily relies on the generic protocol module and provides only one new functionality: the mapping of an experiment identifier to an IPv4 multicast address and vice versa.

The experiment module uses the following equation to convert an experiment identifier of size 2 byte to an IPv4 multicast address under the condition that the experiment identifier is equal to or greater than 0 and equal to or less than 65535:

```
mc_ip_addr = 239.255.<(exp_id >> 8) & 0xff>.<exp_id & 0xff>
```

And it uses the following equation to convert an IPv4 multicast address to an experiment identifier under the condition that the IPv4 multicast address has the form 239.255.*.*:

```
exp_id = mc_ip_addr & 0xffff
```

The Intel PRO/1000 NIC [int08] used by the cluster nodes in the control network does not have implemented perfect multicast filtering in hardware. This network interface uses a hash table of size 64 bit to implement multicast filtering. It would be advantageous if the function, which converts an experiment identifier to an IPv4 multicast address, would create IPv4 multicast addresses so that the number of collisions is very small. The smaller the number of multicast address collisions, the fewer CPU cycles and memory of cluster nodes are wasted. The above presented function, which converts an experiment identifier to an IPv4 multicast address, guarantees that the first 64 experiment identifiers (0-63) are mapped to the IPv4 multicast addresses which do not cause any collisions in the hash table of the Intel PRO/100 NIC.

For the purpose of debugging, the experiment module provides the function experiment_get_stats which returns a pointer to the static variable of type struct experiment_stats. It provides statistical information to the users of the experiment module: number of sent packets etc.

6.1.2.1. Module Parameters

This section describes the module parameters of the experiment module.

The experiment module has no parameters.

6.1.2.2. /proc Interface

This section describes the /proc interface to the experiment module.

/proc/tvee/tdf/experiment/stats The /proc/tvee/tdf/experiment/stats proc-file is a read-only file which contains various statistical counters, such as number of packets sent etc.

6.2. PLACE

This section presents the implementation details of the **P**rotocol for **L**atency **A**ware **C**hanging of **E**pochs (PLACE).

The PLACE protocol uses the generic protocol to transport its packets. The generic protocol value for the packets of the PLACE protocol is 0 by default. And the packets of the PLACE protocol has the priority of 7, the highest priority which is provided by the generic protocol.

Every packet of the PLACE protocol begins with the PLACE protocol header which is shown in the Figure 6.4. The PLACE protocol header consists of three fields: experiment, sequence number and time dilation factor.

The field experiment is of size 2 bytes and identifies the experiment to which the PLACE packet is destined. The field experiment is not really necessary in the PLACE protocol header because the experiment identifier can be extracted from the IPv4 multicast address of the received packet. The field sequence number is of size 4 bytes and contains the current packet sequence number for the experiment specified in the field

experiment of the PLACE protocol header. The field tdf is of size 4 bytes and contains a TDF value.

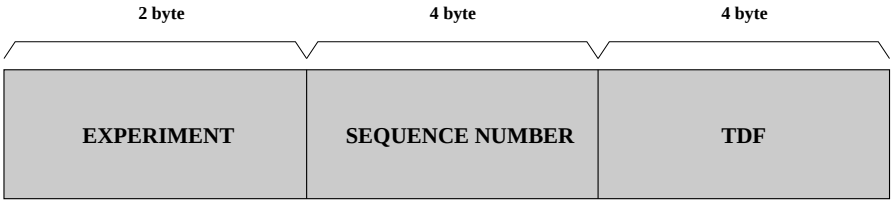


Figure 6.4.: PLACE protocol header

The packets of the PLACE protocol do not contain any payload. They only have a PLACE protocol header.

6.2.1. TDF Sender Module

This section describes the implementation details of the TDF sender module.

The TDF sender module implements the sending instance of the PLACE protocol and is capable to send PLACE packets to multiple experiments simultaneously. In order to achieve that goal, the TDF sender module has to manage packet sequence numbers for all possible experiments. The TDF sender module does not allocate memory for experiment sequence number variables immediately and, therefore, it manages packets sequence numbers in the hash table as shown in the Figure 6.5. When the TDF sender module is prompted to send the first PLACE packet for an experiment then the TDF sender module allocates a new experiment_seq data structure, initializes the sequence number for the experiment to 0 and places this newly allocated data structure in the hash table exp_seq. After that, all PLACE packets sent to the same experiment get the sequence number from the experiment_seq data structure in the hash table exp_seq. The sequence number is incremented every time a PLACE packet is sent to the experiment which is associated with the sequence number.

The main function provided by the TDF sender module is the function tdf_sender_send_tdf. This function receives two integer parameters: experiment and tdf. The experiment variable identifies the experiment to which a PLACE packet will be sent and the variable tdf stores the TDF value which will be stored in the PLACE packet.

For the purpose of debugging, the TDF sender module provides the function tdf_sender_get_stats which returns a pointer to the static variable of type struct tdf_sender_stats. It provides statistical information to the users of the TDF sender module: number of sent packets etc.

6.2.1.1. Module Parameters

This section describes the module parameters of the TDF sender module.

proto The parameter proto is the generic protocol value used by the TDF sender module for packets of the PLACE protocol. This parameter is not mandatory and is of

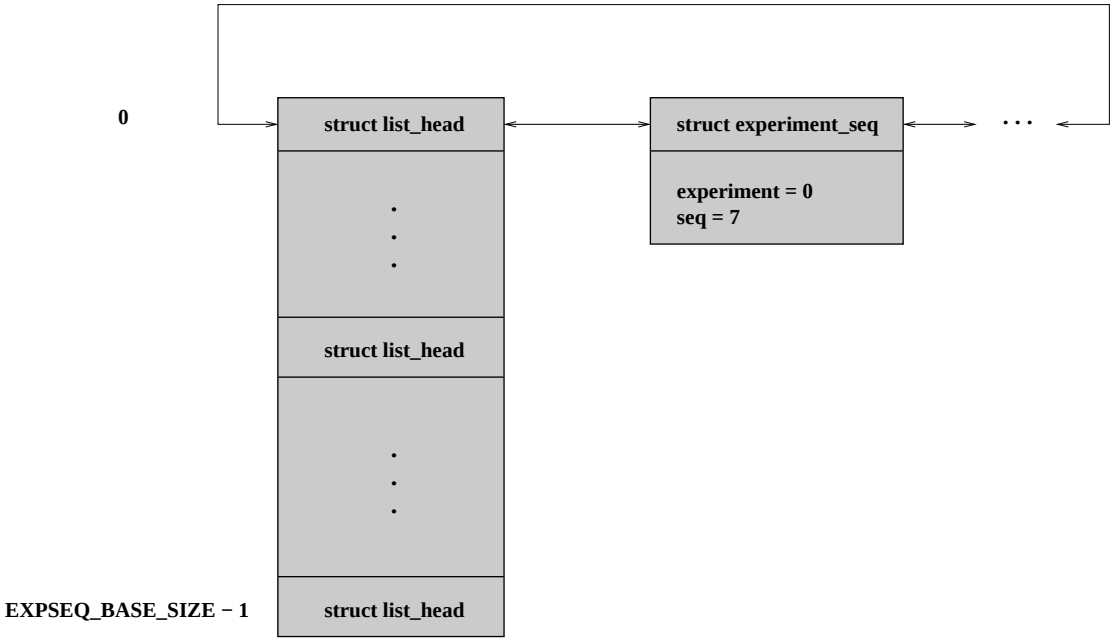


Figure 6.5.: expseq_base data structure

type unsigned integer. The parameter is equal to 0 by default.

prio The parameter prio is the generic protocol packet priority value used by the TDF sender module for PLACE packets. This parameter is not mandatory and is of type unsigned integer. This parameter is equal to 7 (highest priority) by default.

6.2.1.2. /proc Interface

This section describes the /proc interface to the TDF sender module.

/proc/tvee/tdf/sender/send_tdf The /proc/tvee/tdf/sender/send_tdf proc-file is a write-only file which allows to send PLACE packets from the user space of the Linux kernel. This file expects 2 integer numbers separated by the whitespace (space and tabs). The first number identifies the experiment to which a PLACE packet should be sent and the second parameter is a TDF value.

/proc/tvee/tdf/sender/stats The /proc/tvee/tdf/sender/stats proc-file is a read-only file which contains various statistical counters, such as number of packets sent etc.

6.2.2. TDF Receiver Module

This section describes the implementation details of the TDF receiver module.

The TDF receiver module implements the receiving instance of the PLACE protocol and is capable to receive PLACE packets belonging to only one experiment simultaneously. The TDF receiver module can join an experiment at loading time. Furthermore,

the TDF receiver module is also capable to join an experiment at run-time. The module uses the function experiment_add_protocol of the experiment module and registers a handler for the incoming PLACE packets destined to a specified experiment. When the TDF receiver module has already joined an experiment then first it has to unregister the PLACE packet handler with the experiment_del_protocol function in order to join another experiment at run-time.

The sequence number of every received PLACE packet is examined by the TDF receiver module and when the sequence number of a received PLACE packet is out of order then the TDF receiver module sends a warning message to the system logger of the cluster node.

For the purpose of debugging, the TDF receiver module provides the function tdf_receiver_get_stats which returns a pointer to the static variable of type struct tdf_receiver_stats. It provides statistical information to the users of the TDF receiver module: number of received packets and TDF changes etc.

6.2.2.1. Module Parameters

This section describes the module parameters of the TDF receiver module.

proto The parameter proto is the generic protocol value used by the TDF receiver module for packets of the PLACE protocol. This parameter is not mandatory and is of type unsigned integer. The parameter is equal to 0 by default.

experiment The parameter experiment is the experiment identifier used by the TDF receiver module to join this experiment at loading time. This parameter is not mandatory and is of type unsigned integer. The TDF receiver module does not join any experiment by default.

change_tdf The parameter change_tdf indicates if the TDF switching should be enabled or disabled at loading time of the TDF receiver module. This parameter is not mandatory and is of type integer. The TDF switching is enabled by default.

6.2.2.2. /proc Interface

This section describes the /proc interface to the TDF receiver module.

/proc/tvee/tdf/receiver/experiment The /proc/tvee/tdf/receiver/experiment proc-file is a read-write file. The file contains the current experiment which is joined by the TDF receiver module. By writing an experiment identifier to this file, it is possible to initiate the joining of the specified experiment by the TDF receiver module.

/proc/tvee/tdf/receiver/change_tdf The /proc/tvee/tdf/receiver/change_tdf proc-file is a read-write file. The file indicates if the TDF switching is enabled in the TDF receiver module. By writing an integer value to this file, it is possible to enable or disable the TDF switching by the TDF receiver module.

/proc/tvee/tdf/receiver/stats The /proc/tvee/tdf/receiver/stats proc-file is a read-only file which contains various statistical counters, such as number of packets received etc.

Chapter 7.

Evaluation

In this chapter, the evaluation of PLACE's protocol implementation which is described in the previous chapter is presented. First, the methodology of the protocol evaluation is explained. Second, the evaluation tools which are used for the protocol evaluation, are explained. Third, the evaluation scenarios are described. Fourth, the results of the evaluation are presented and briefly explained. And finally, these evaluation results are discussed.

7.1. Evaluation Goals

The goal of the evaluation is to examine if the implementation of the PLACE protocol described in the previous chapter fulfills the main requirements of PLACE which are described in Chapter 2.2.

The protocol properties which are researched in the evaluation are the performance of the sending and the receiving instance of the PLACE protocol, total packet delay, packet delay variation and packet delay in the ingress queue of the Cisco switches which build the control network of DTVEE. The protocol behaviour is evaluated under different CPU and network load conditions in cluster nodes of DTVEE.

7.2. Evaluation Tools

This section presents various programs and tools which are used in the evaluation of PLACE's protocol implementation. These evaluation tools are used in the evaluation in order to create various CPU load in cluster nodes and network load in the control and experiment network of DTVEE.

7.2.1. Network Load Generating

Network load generating programs are used in the evaluation in order to emulate various network load in the control and in the experiment network of DTVEE.

In the evaluation of the PLACE protocol two network performance measurement tools will be used: iperf and netperf. On the one hand iperf and netperf are the TCP/UDP network measurement programs that can be used to measure various aspects of networking performance (the bandwidth and the quality of a network link) and on the other hand

they can be used to generate network load. iperf and netperf use client-server architecture and can run in either server mode or client mode. In server mode, data is sent to the machine running one of these tools from other hosts and in client mode, the machine sends data to another host running one of these programs in server mode.

In order to create heavy network load in the control network and in the experiment network of DTVEE, iperf and netperf are used in UDP mode because TCP protocol uses flow and congestion control mechanisms which adjust transmission rate to the available bandwidth.

Furthermore, iperf and netperf allows in UDP mode to set the transmission rate and the size of packets.

iperf and netperf also support multicast communication in UDP mode. The multicast communication allows us to send UDP packets from one machine to several cluster nodes simultaneously and, therefore, to generate heavy network load on several cluster nodes with only a single machine. Furthermore, IGMP Snooping has to be enabled in switches of the control and experiment networks of DTVEE because the switches of the control and experiment network of DTVEE with disabled IGMP Snooping would forward the IP packets which are destined to a multicast group address to all switch outgoing ports and, therefore, would create high network load on all outgoing ports of the switches. And exactly this situation must be avoided because not every cluster node in an experiment must be flooded. In addition, we need to run a multicast routing daemon on one of the cluster nodes of DTVEE because without the multicast routing daemon, IGMP Snooping will not work in the control and experiment networks of DTVEE. The multicast routing daemon mrouted [mro08] will undertake this task. An finally, every cluster node which want to receive IP packets destined to a multicast group address must have IP Multicasting functionality compiled into the Linux kernel because the Linux kernel does not send IGMP packets into the network without that functionality. And IGMP Snooping does not work without IGMP messages.

7.2.2. CPU Load Generating

CPU load generating programs are used in the evaluation in order to emulate various CPU load on the cluster nodes of DTVEE.

In the evaluation of the PLACE protocol, two programs are used to generate CPU load: dd and openssl. The dd program is used in the evaluation to generate CPU load in kernel mode and the openssl program is used to generate CPU load in user space. It is not sufficient to generate CPU load in user space only in order to evaluate the behaviour of the PLACE protocol under high CPU load conditions because the PLACE protocol is implemented in kernel space. And therefore, the evaluation has to research the influence of high CPU load in kernel space.

The following shell command is used to generate high CPU load in kernel space:

```
dd if=/dev/zero of=/dev/null
```

This shell command transfers data from the /dev/zero Unix device to the /dev/null

device and , therefore, creates high CPU load in kernel space. And the following shell command is used to generate high CPU load in user space:

```
while true; do openssl speed; done
```

This shell command executes in an infinite loop the command openssl speed that tests every single cryptographic algorithm supported by the OpenSSL library and, therefore, creates high CPU load in user space.

7.2.3. Measurement of Packet Delay

To record the sending and the receiving time of a packet, the Linux kernel function do_gettimeofday is used in the evaluation. In order to make accurate packet delay measurements, the clocks of the cluster nodes which participate in the evaluation must be sufficiently synchronized. The clock synchronization is achieved by the NTP [NTP92] protocol.

One of the Cisco switches in the control network of DTVEE runs a NTP server and all cluster nodes of DTVEE synchronize their clocks with the NTP server of this Cisco switch. Before every experiment, the clocks of every cluster node participating in the experiment are synchronized with the clock of the Cisco switch so that the clock drift between a cluster node and the Cisco switch is equal to 0.000020 seconds. This guarantees sufficiently accurate measurements of the packet delay during an experiment. After an experiment is finished, the clock drift between cluster nodes participating in the experiment and the Cisco switch is recorded in order to assess the accuracy of the packet delay measurements.

7.2.4. Measurement of CPU Load

During the evaluation experiments, the CPU load of the dom0 and domU domains on the cluster nodes which participate in the evaluation has to be recorded in user space as well as in kernel space.

In user space, the function xenstat_domain_cpu_ns will be used to calculate the CPU load of the dom0 and domU domains on the cluster nodes over a specific time period. In order to be able to use this function, a program using this function has to be linked with the static C library libxenstat.a. The function xenstat_domain_cpu_ns returns how much CPU time has been used by a specific domain.

It is not possible to use the Xen C libraries in kernel space in order to get how much CPU time has been used by a specific domain. In kernel space, the hypercall getdomaininfo is used to get how much CPU time has been used by a specific domain. The hypercall getdomaininfo returns various statistics for a specific domain and CPU time.

The following equation is used to calculate the CPU load of a specific domain on a cluster node over the time period $T_{end} - T_{start}$:

$$CPU_{load} = \frac{CPU_{end} - CPU_{start}}{T_{end} - T_{start}} \tag{7.1}$$

CPU_{start} is the CPU time that was recorded at time T_{start} and CPU_{end} is the CPU time that was recorded at time T_{end}.

7.2.5. Protocol for Evaluation

In order to be able to measure delay of the packets of the PLACE protocol, either the implementation of the PLACE protocol described in the previous chapter has to be changed or a new protocol that will behave like the PLACE protocol but will allow us to measure delay of packets, has to be developed because the original PLACE protocol does not allow us to measure delay of its packets. The last option was chosen for the evaluation and the new protocol was called the EVAL protocol.

The EVAL protocol, like the PLACE protocol, consists of two modules: sending and receiving instance. The packets of the EVAL protocol contain only two fields: packet identifier and timestamp. The packet identifier allows us to detect a packet loss and to differentiate between several packets. The timestamp field contains the sending time of the packet and allows us to calculate the delay of the packet. Immediately before a packet of the EVAL protocol is sent, the current time is saved in the timestamp field of the packet. Directly after an EVAL packet is received by the receiving instance of the EVAL protocol, the receiving instance saves the receiving time of the packet and stores this information in memory which can be retrieved by user-space programs through a /proc interface. With the sending and the receiving time of a packet, the packet delay can be calculated.

The EVAL protocol uses the same priority for its packets that is used by the PLACE protocol in order to emulate the behaviour of the PLACE protocol. The EVAL protocol, like the PLACE protocol, also uses the generic protocol module and the experiment module to send and to receive its packets.

7.3. Scenario Description

In this section, the scenarios for the experiments in the evaluation are described.

Different set-ups are required in order to separately study the specific aspects of the behaviour of the PLACE protocol. First, a scenario is used to study the performance of the sending and the receiving instance of the PLACE protocol. Second, a different scenario is used to study packet delay variation properties of the PLACE protocol. And third, a scenario is set-up to study packet delay in the ingress queue of the Cisco switches which build the control network of DTVEE.

7.3.1. Scenario: Performance

The performance evaluation of the sending and the receiving instance of the PLACE protocol studies how much packets can be sent by the sending instance of the PLACE protocol per time unit, how much packets can be received by the receiving instance of the PLACE protocol per time unit and how much CPU time it will cost.

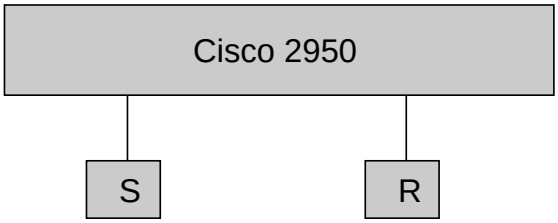


Figure 7.1.: Topology for scenario "Performance"

The Figure 7.1 shows the network topology which is used in this evaluation scenario. The scenario requires only two cluster nodes of DTVEE. The cluster node S plays the role of the sending instance and the cluster node R is the receiving instance of the PLACE protocol. As shown in the Figure 7.1, both cluster nodes are connected to the same Cisco switch but they also could be connected to different Cisco switches.

During this scenario, the sender node sends a specific number of PLACE packets to the receiver and CPU utilization is measured and recorded on both nodes. The sender node executes a loop of 10000 iterations and sends a specific number of PLACE packets during each iteration. During the experiment, the number of PLACE packets sent per iteration is set to 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096 and 8192. After each loop iteration, the sender sleeps 1 millisecond.

7.3.2. Scenario: Packet Delay and Packet Delay Variation

First, the evaluation of total packet delay studies how long it takes to send and to receive a packet of the PLACE protocol under different network and CPU load conditions on cluster nodes of DTVEE. Second, the evaluation of packet delay variation researches if packets of the PLACE protocol destined to an experiment are received by all receiving instances which participate in the same experiment simultaneously or not. This evaluation is also performed under different network and CPU load conditions.

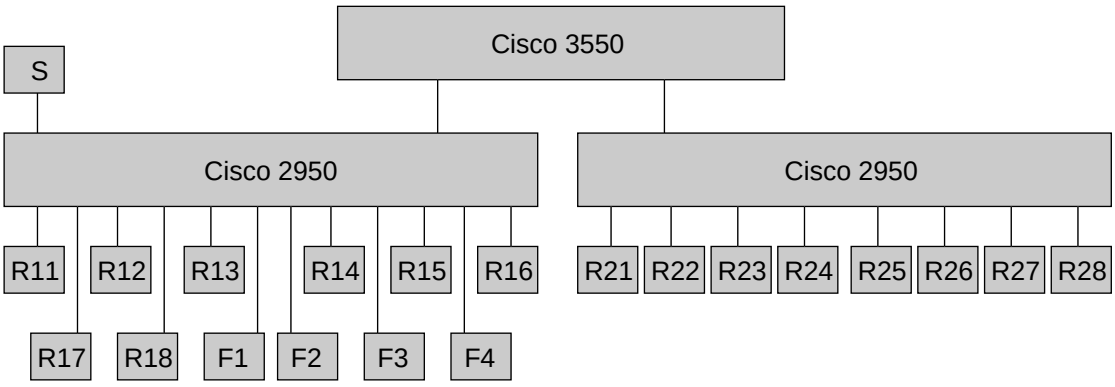


Figure 7.2.: Topology for scenario "Packet delay and packet delay variation"

The Figure 7.2 shows the network topology that is used in this evaluation scenario.

The scenario requires 21 cluster nodes: 1 sender node, 16 receiver nodes and 4 cluster nodes to create network load for receiver nodes. The sender node S sends PLACE packets to the receiver nodes R11 – R18 and R21 – R28. The receiver nodes R11 – R18 are connected to the one Cisco switch. And the receiver nodes R21 – R28 are connected to the other Cisco switch. The cluster nodes F1 – F4 create a specific network load in the control and experiment networks of DTVEE. These cluster nodes use IP multicast communication to create network load for several receiver nodes simultaneously. The cluster nodes F1 and F2 create network load in the control network of DTVEE and the cluster nodes F3 and F4 create network load in the experiment network of DTVEE.

Subscenario	Network load (% of bandwidth)	
	control network (100 Mbit)	experiment network (1 Gbit)
0	100	100
1	10	10
2	20	20
3	30	30
4	40	40
5	50	50
6	60	60
7	70	70
8	80	80
9	90	90
10	0	0

Table 7.1.: Subscenarios for scenario "Packet delay and packet delay variation"

The evaluation consists of 11 subscenarios: Subscenario 0 – 10. These subscenarios study the behaviour of the PLACE protocol under different network and CPU load conditions on receiver nodes. The Table 7.1 shows all subscenarios and the network load which will be created in the control and experiment networks of DTVEE during these subscenarios.

And the Table 7.2 shows the configuration of the receiver nodes for the subscenarios presented in the Table 7.1. The Table 7.2 shows which receiver nodes have heavy CPU load or are under heavy network load during the experiment. CPU load can be created in the domain dom0, the domain domU or in both domains simultaneously. And network load also can be generated for the domain dom0, the domain domU or for both domains simultaneously.

This evaluation scenario uses 16 receiver nodes, 8 receiver nodes connected to the one Cisco switch and the other 8 receiver nodes connected to the another one. As shown in the Table 7.2, the 8 receiver nodes of the first switch have the same configuration as the 8 receiver nodes connected to the other switch in order to research how large is the difference between delays of PLACE packets received by the cluster nodes connected to two different switches.

Cluster nodes	Configuration of cluster nodes							
	Subscenarios 0 – 9				Subscenario 10			
	cpu0	net0	cpuU	netU	cpu0	net0	cpuU	netU
R11/R21	-	-	-	-	-	-	-	-
R12/R22	x	-	x	-	x	-	-	-
R13/R23	-	x	x	-	-	-	x	-
R14/R24	x	-	-	x	x	-	x	-
R15/R25	-	x	-	-	not used			
R16/R26	-	-	-	x	not used			
R17/R27	-	x	-	x	not used			
R18/R28	x	x	x	x	not used			

Table 7.2.: Configuration of receiver nodes for scenario "Packet delay and packet delay variation" (- – no load, x – load)

7.3.3. Scenario: Packet Delay in Ingress Queue of Switch

The evaluation searches for the optimal parameters of the TBF [TBF08] queueing discipline which is installed on the outgress queue of every cluster node in DTVEE in order to avoid large packet delays in the ingress queues of the Cisco switches which build the control network of DTVEE.

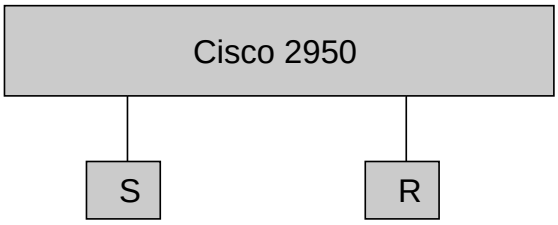


Figure 7.3.: Topology for scenario "Packet delay in ingress queue of switch"

The Figure 7.3 shows the network topology which is used in this evaluation scenario. The scenario requires only two cluster nodes of DTVEE. The cluster node S plays the role of the sending instance and the cluster node R is the receiving instance of the PLACE protocol. As shown in the Figure 7.3, both cluster nodes are connected to the same Cisco switch but they also could be connected to different Cisco switches.

During this scenario, the sender node sends 100000 PLACE packets to the receiver node, the delay of sent packets is measured and recorded by the receiver node. Furthermore, the client node runs an UDP client which sends UDP packets with full bandwidth to the UDP server running on the receiver node. The UDP data packets sent from the sender node to the receiver node cause the ingress queue of the ingress port to which the sender node is connected to grow very large. The TBF queueing discipline should prevent this situation.

During this experiment, the parameter burst of the TBF queueing discipline is varied

and the packet delay is measured.

7.4. Evaluation Results

This section presents the results of the evaluation.

7.4.1. Scenario: Performance

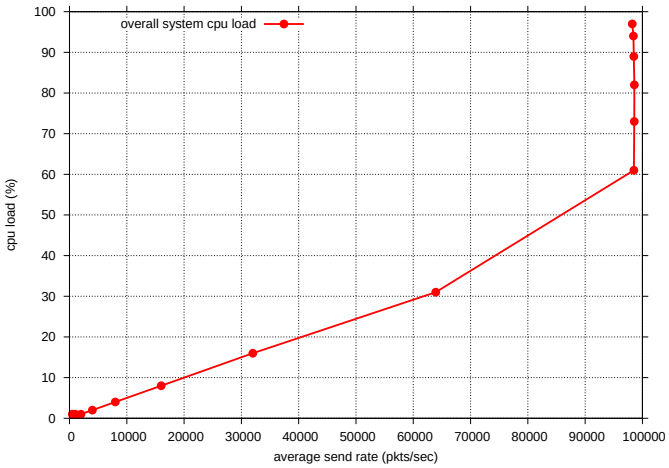


Figure 7.4.: Sender performance

The Figure 7.4 shows the average amount of PLACE packets sent by a cluster node of DTVEE per second and how much CPU usage it costs. As expected, an increase in the number of packets sent per second results in an increase in the CPU usage of the sending cluster node. During the experiment, the sender node began to drop several outgoing PLACE packets when the average sending rate reached 100000 packets per second because the outgoing queue of the control network NIC reached its limit. The Figure 7.4 shows that the average sending rate of 1000 packets per second does not cost much CPU power to the sending node.

The Figure 7.5 shows the average amount of PLACE packets received and processed by a cluster node of DTVEE per second and how much CPU usage costs the receiving and the processing of these PLACE packets. As expected, an increase in the number of packets received per second results in an increase in the CPU usage of the receiving cluster node. During the experiment, the receiver node received all PLACE packets which were successfully sent by the sender node and did not lose any PLACE packets. The Figure 7.5 shows that the average receiving rate of 1000 packets per second does not cost much CPU power to the receiving node.

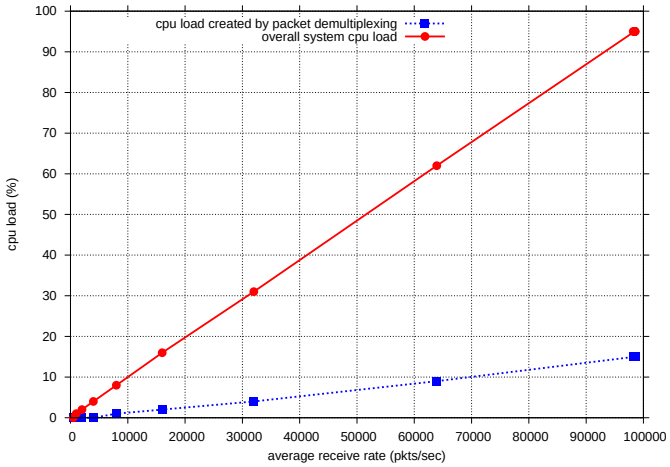


Figure 7.5.: Receiver performance

7.4.2. Scenario: Packet Delay and Packet Delay Variation

During the first run of the experiment, a problem was encountered that was caused by the default Xen domain scheduler – credit scheduler [xen08a] with 30 milliseconds time slices. The Figure 7.6 shows the problem caused by the default Xen credit scheduler. It shows delays of the first 1000 packets sent to the cluster node R12 from the Subscenario 0. This problem occurs every time the domU domain of a cluster node has high CPU usage. The Figure 7.6 shows spikes of length 30 milliseconds and of height 30 milliseconds which are caused by the Xen credit scheduler with 30 milliseconds time slices.

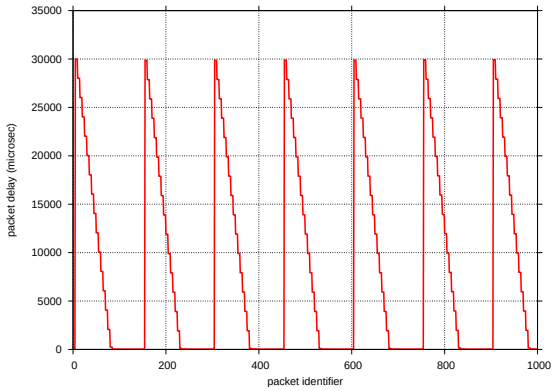


Figure 7.6.: Packet delay variation in Xen 3 with credit scheduler and 30 ms time slices

In order to partially solve the problem and to reduce the height of the spikes in the Figure 7.6, the default Xen credit scheduler was adjusted. The three constants in the Xen source file sched_credit.c were changed: CSCHED_TICKS_PER_TSLICE, CSCHED_TICKS_PER_ACCT and CSCHED_MSECS_PER_TICK. By default, these con-

stants are equal to 3, 3 and 10, respectively. The constants were changed to 1, 1 and 1, respectively. The length of one time slice for a domain is equal to the product of CSCHED_MSECS_PER_TICK and CSCHED_TICKS_PER_TSLICE. Therefore, the default Xen credit scheduler works with 30 milliseconds time slices. After the adjustments of the Xen credit scheduler, the time slice for a domain is 1 millisecond.

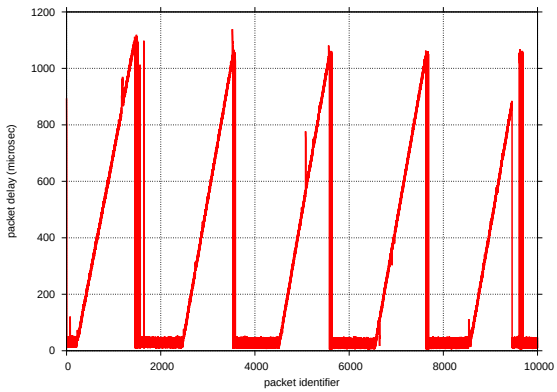


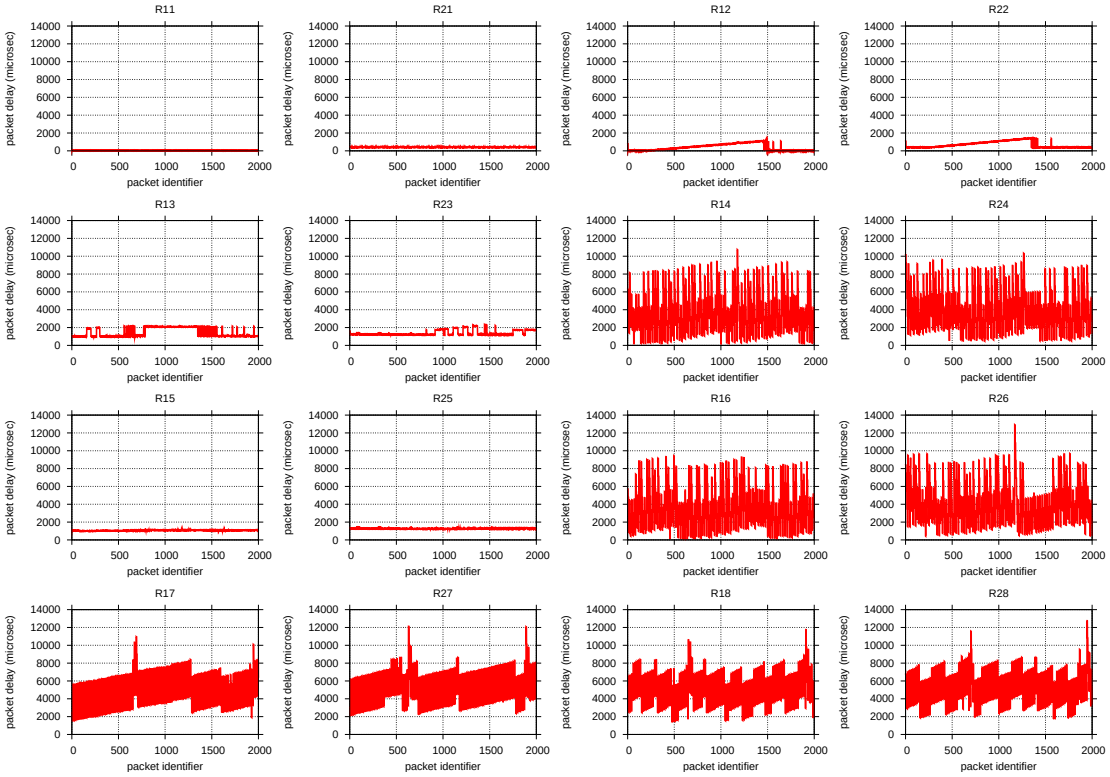
Figure 7.7.: Packet delay variation in Xen 3 with credit scheduler and 1 ms time slices

The Figure 7.7 the delay of the first 10000 packets sent to the cluster node R12 that runs the adjusted Xen credit scheduler with 1 millisecond time slices. As it can be seen in the Figure 7.7, the height of the spikes is slightly larger than 1 millisecond and far below the 30 milliseconds mark. And the length of a spike is about 1 second.

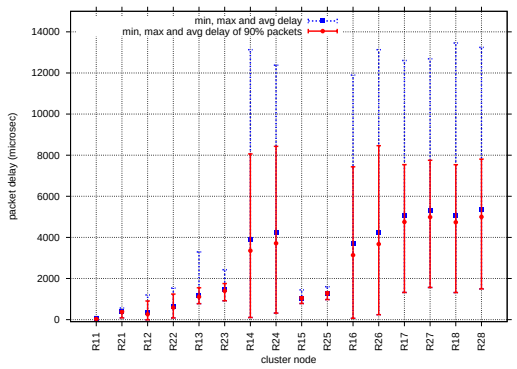
Furthermore, the default Xen credit scheduler with 30 milliseconds time slices caused a lot of packet loss on cluster nodes with high CPU and network load in the domain dom0 and domU during the evaluation. The adjusted Xen credit scheduler with 1 millisecond time slices solved this problem and there was no packet loss during the evaluation.

As mentioned before, the NTP protocol is used before the beginning of every experiment to synchronize the local clocks of all cluster nodes participating in the experiment. Because the local clocks of the cluster node are not perfect and the local clocks of the cluster nodes begin to drift after some time, the packet delays which are presented later in this section have the measurement error in the range from -0.000100 to 0.000100 seconds.

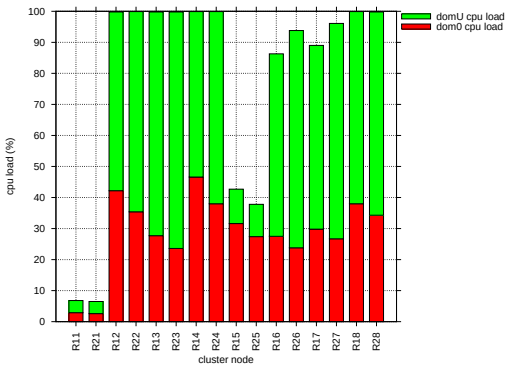
In this section, the results of some subscenarios shown in the Table 7.1 will be presented because the results of other subscenarios, which will not be shown, are very similar. The results of the following subscenarios will be presented: Subscenario 0, Subscenario 3, Subscenario 5, Subscenario 8 and Subscenario 10. For every subscenario, the delay of the first 2000 packets sent to every receiver node shown in the Table 7.2 will be presented. Furthermore, the minimum, the maximum and the average packet delay for every receiver node and the CPU usage of every receiver node will be also presented for these subscenarios. The results of all other subscenarios can be found in Appendix A.



(a) Packet delays



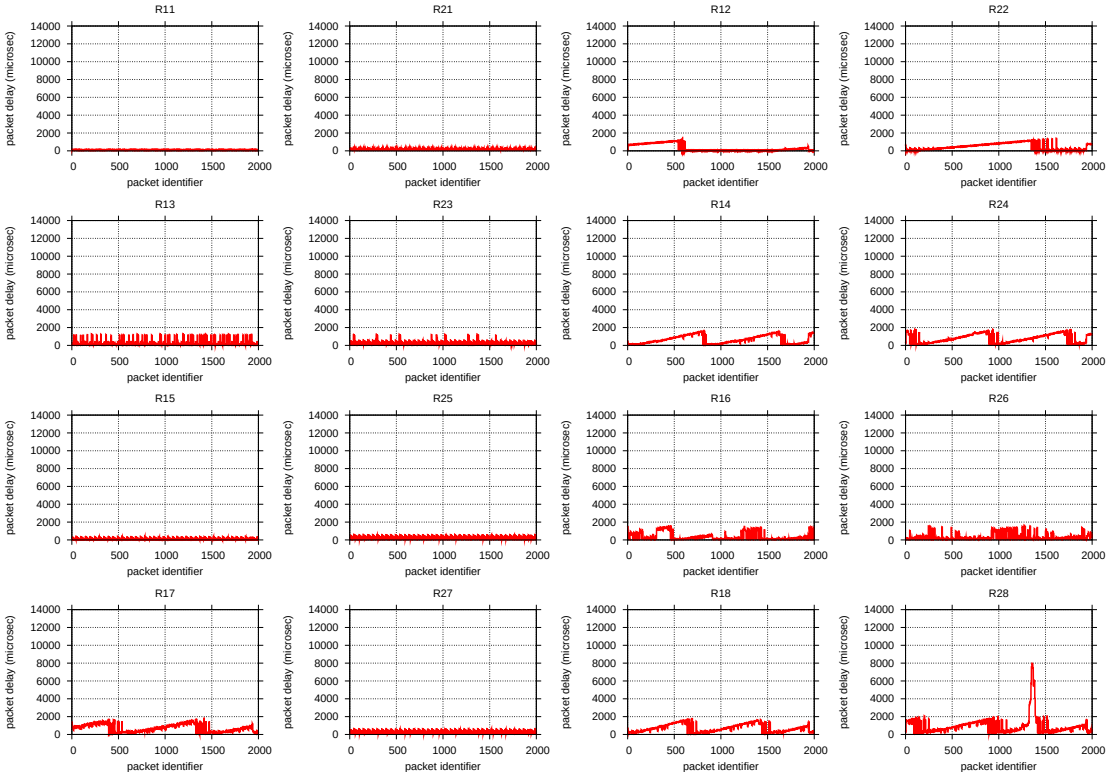
(b) Minimum, maximum and average packet delays



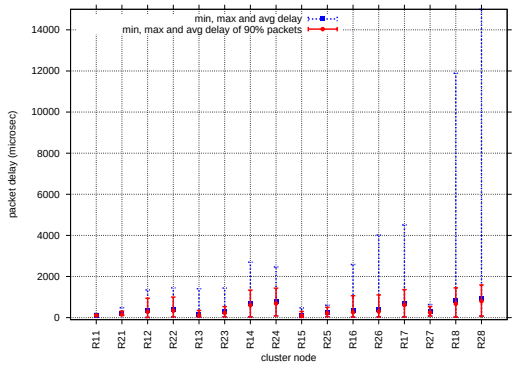
(c) CPU load of cluster nodes

Figure 7.8.: Subscenario 0 results

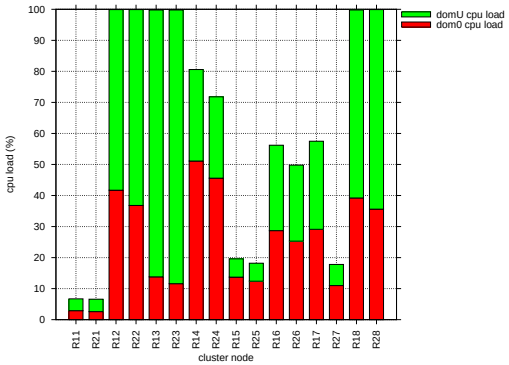
The results of the Subscenario 0 are shown in the Figure 7.8. As it can be seen in the Figure 7.8, high network load in the experiment network causes very large packet delays on the cluster nodes R14, R24, R16 – R18 and R26 – R28 which are flooded in the experiment network. As expected, high network load in the control network does not cause large packet delays on cluster nodes R13, R23, R15 and R25. Furthermore,



(a) Packet delays



(b) Minimum, maximum and average packet delays

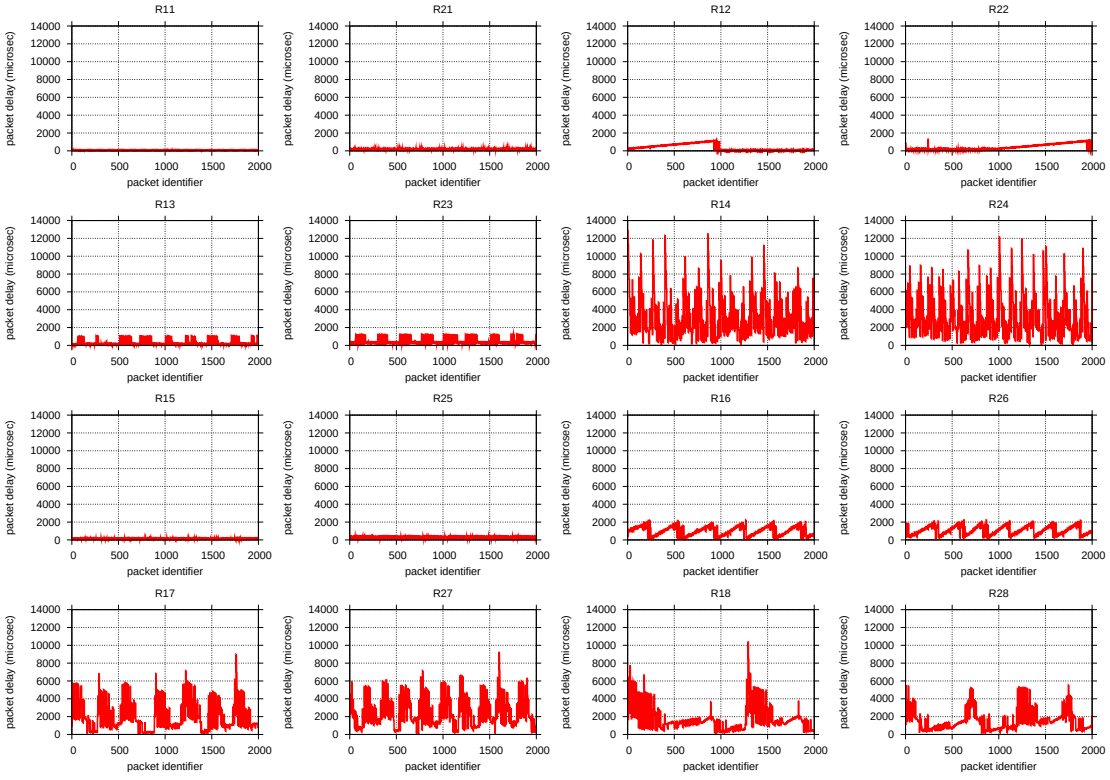


(c) CPU load of cluster nodes

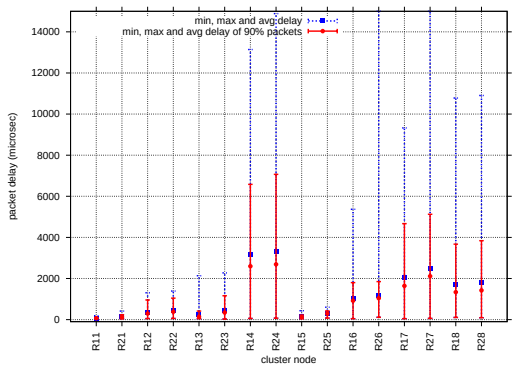
Figure 7.9.: Subscenario 3 results

high CPU usage in one of the domains or in both does not cause large packet delays on cluster nodes R11, R21, R12 and R22. But high CPU usage in the domain domU leads to spikes which were discussed earlier on cluster nodes R12 and R22.

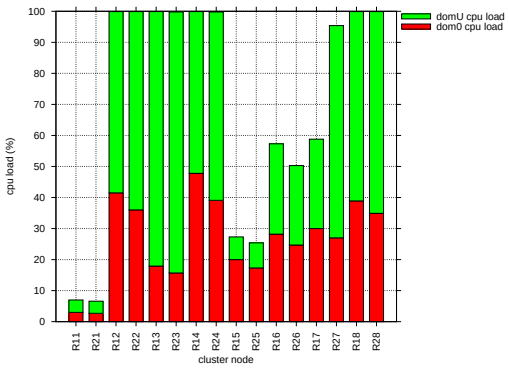
The graphs of the cluster nodes R11 and R21 in the Figure 7.8 also show us that the delay of packets sent to the nodes which are connected to the same Cisco switch to which



(a) Packet delays



(b) Minimum, maximum and average packet delays

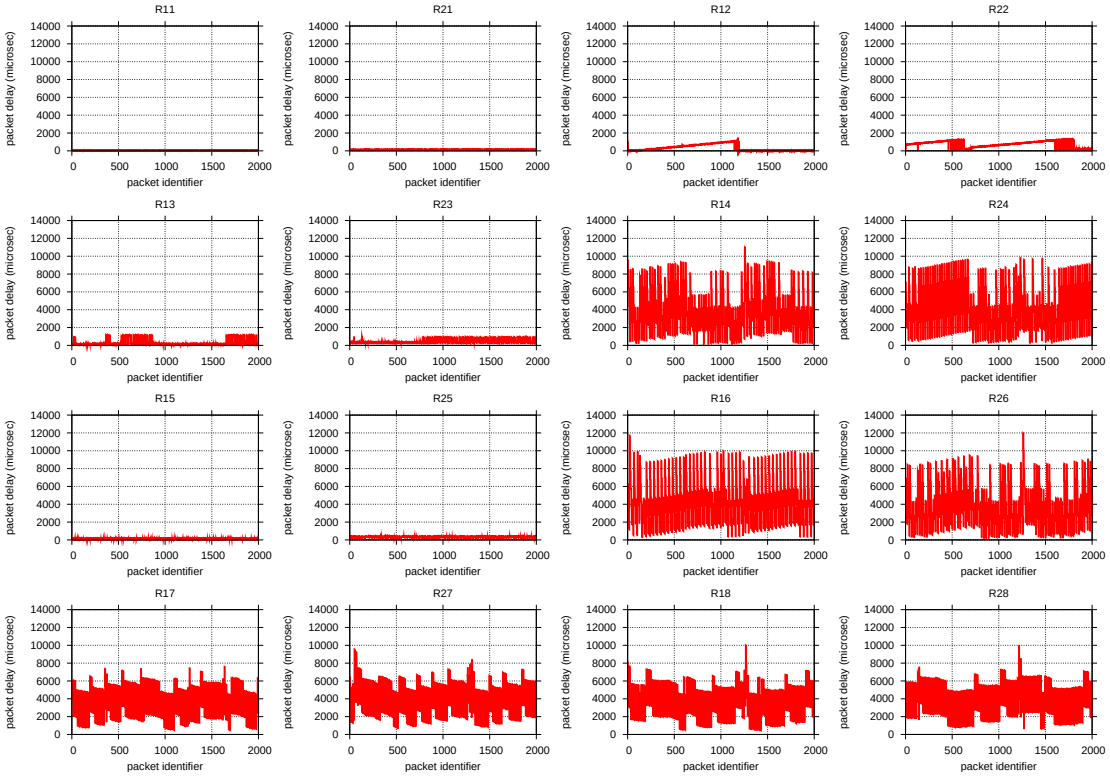


(c) CPU load of cluster nodes

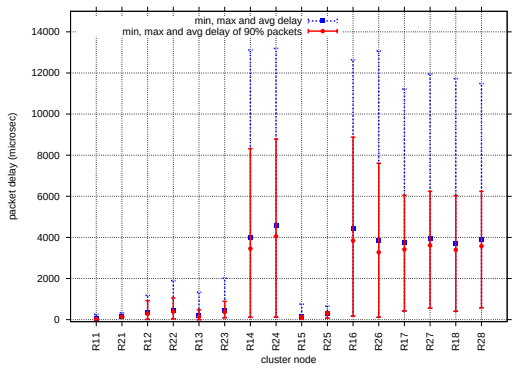
Figure 7.10.: Subscenario 5 results

the sender node is connected is slightly smaller than the delay of packets which are sent to the receiver nodes connected to the second Cisco switch.

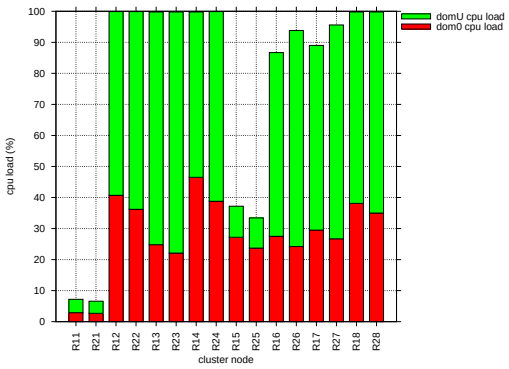
The graphs of the cluster nodes R15 and R25 in the Figure 7.8 shows us that high network load in the control network of DTVEE increases the total delay of PLACE packets to about 1 millisecond.



(a) Packet delays



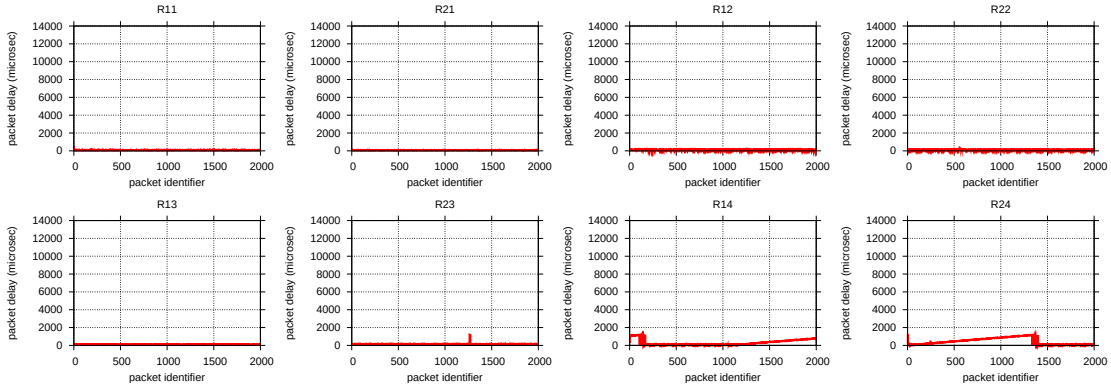
(b) Minimum, maximum and average packet delays



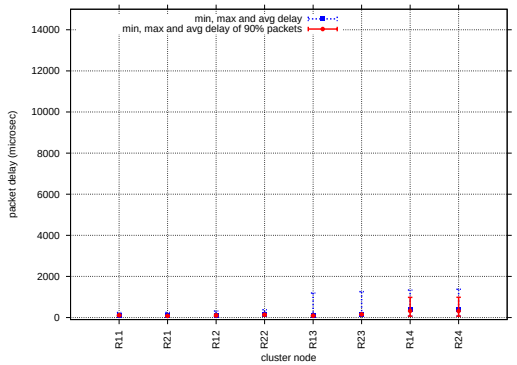
(c) CPU load of cluster nodes

Figure 7.11.: Subscenario 8 results

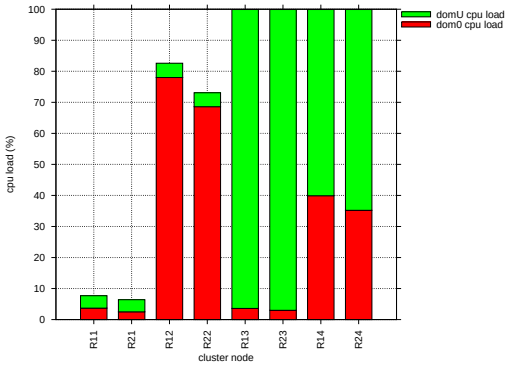
The results of the Subscenario 3 are shown in the Figure 7.9. The Figure 7.9 shows that the 30% network load in the experiment network does not cause large packet delays on the cluster nodes R14, R24, R16 – R18 and R26 – R28 as it was the case in the Subscenario 0. And the Figure 7.9 shows that the delay of the 90% of packets sent to the receiver nodes is smaller than 2 milliseconds. Unfortunately, the experiment results



(a) Packet delays



(b) Minimum, maximum and average packet delays



(c) CPU load of cluster nodes

Figure 7.12.: Subscenario 10 results

of the cluster node R27 were corrupted during the experiment.

The results of the Subscenario 5 are shown in the three Figures 7.10. The Figure 7.10 shows that the results of the Subscenario 5 are slightly better than the results of the Subscenario 0. Furthermore, the 50% network load in the experiment network does not lead to large packet delays on the cluster nodes R16 and R26 as it was the case in the Subscenario 0.

The results of the Subscenario 8 are shown in the Figure 7.11. The results of this experiment are very similar to the results of the Subscenario 0.

The results of the Subscenario 10 are shown in the Figure 7.12. During the Subscenario 10, no network load was created in the control network as well as in the experiment network. Therefore, the results of this experiment are very good. But as the Figure 7.12 shows, high CPU load in the domain dom0 and domU leads to the spikes of the height about 1 millisecond on cluster nodes R14 and R24.

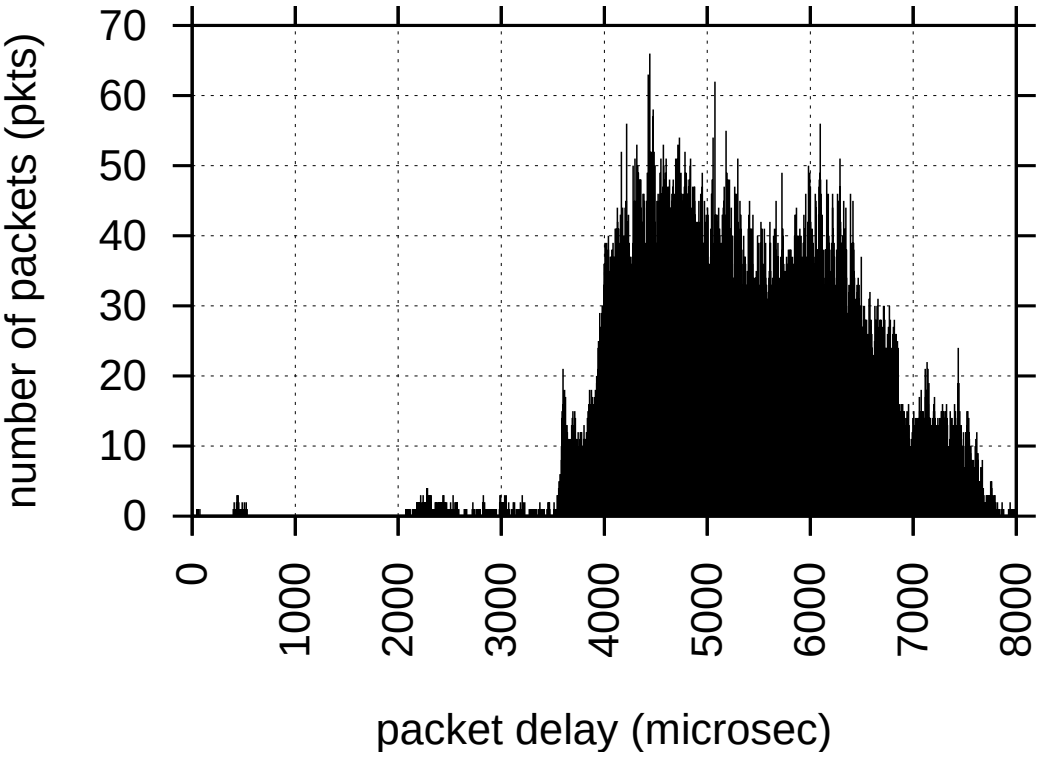


Figure 7.13.: Packet delay distribution without TBF egress qdisc

7.4.3. Scenario: Packet Delay in Ingress Queue of Switch

The Figure 7.13 shows the packet delay distribution in the situation when the sender of PLACE packets does not have the TBF queueing discipline installed. As expected, when the sender node heavily utilizes the outgoing bandwidth of the control network then the packet delay increases drastically.

The Figure 7.14 shows the packet delay distribution for different burst values of the TBF queueing discipline. As expected, an decrease in the burst size results in an decrease in the delay of PLACE packets.

The smaller the value of the burst parameter, the smaller is the average outgoing network bandwidth that will be provided for packets of protocols other than the generic protocol. The generic protocol has no network bandwidth limitations because the packets of the generic protocol are not shaped by the TBF queueing discipline. With the burst of 1 kb, the average outgoing network bandwidth which will be provided for packets of protocols other than the generic protocol is about 50 MBit/s and it increases when the value of the burst parameter increases.

It is important to note that the delay of the PLACE packets in the switch ingress queue does not negatively affect simultaneous reception of PLACE packets on receiver nodes because the PLACE protocol uses multicast communication to transport its packets.

Therefore, it is unimportant how large is the delay of a PLACE packet in the switch ingress queue in respect to simultaneous packet reception.

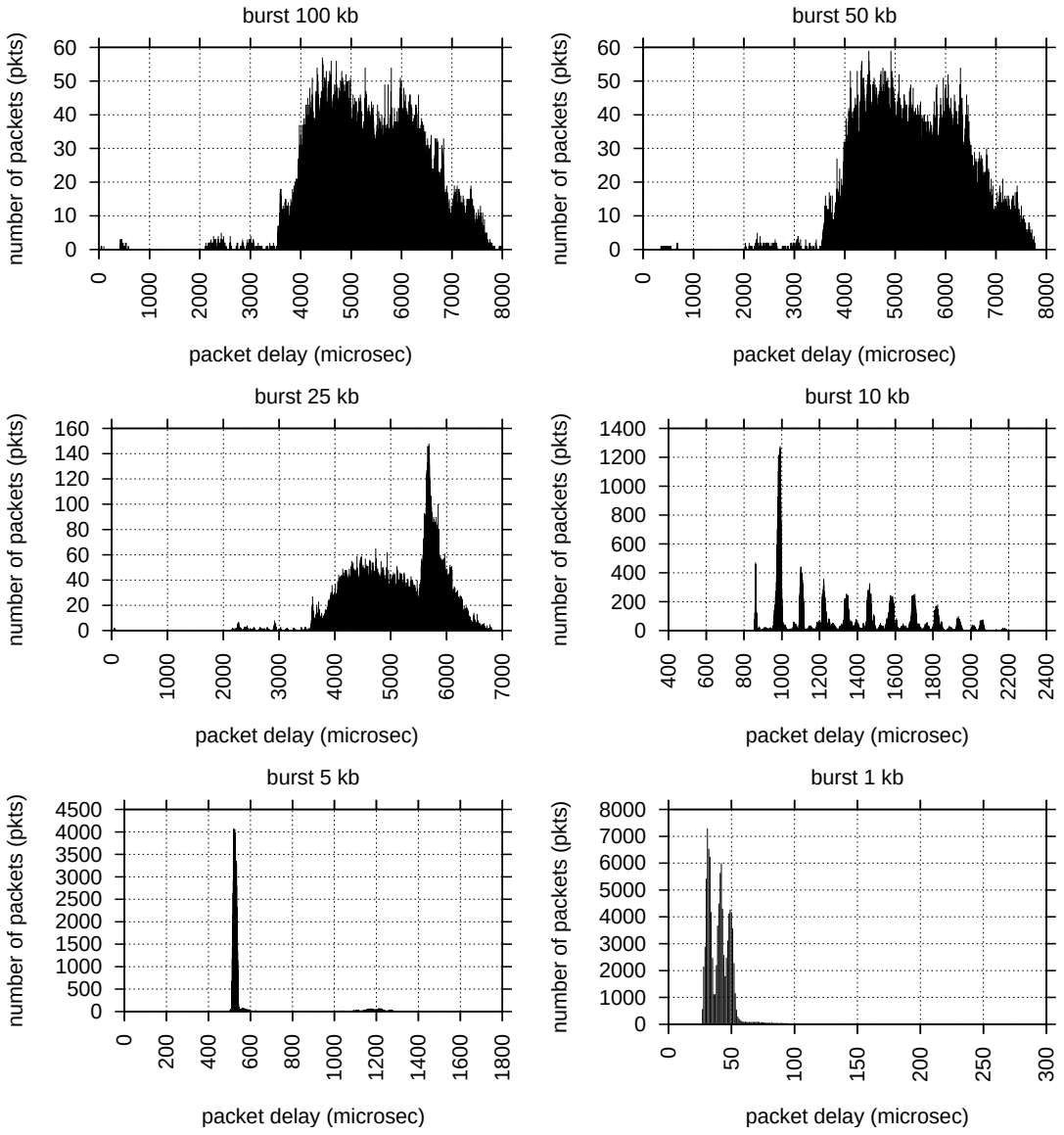


Figure 7.14.: Packet delay distribution with TBF egress qdisc

7.5. Discussion of Results

This section discusses the properties of the PLACE protocol and uses the evaluation results of the previous section as a base for discussion about the PLACE protocol, its advantages and its disadvantages. The focus of this section is to sum up the properties of the PLACE protocol and how good the implementation of the PLACE protocol discussed in the Chapter 6 fulfills the protocol requirements stated in the Chapter 2.2.

The evaluation of the PLACE protocol has shown that not a single packet of the PLACE protocol has been lost which is one of the most important protocol requirements.

The evaluation results of the Scenario "Performance" show us that the performance of the PLACE implementation surpass the protocol requirements. Both, the sender instance and the receiver instance of the PLACE protocol, are capable to send and to receive 1000 packets and much more per second with very low CPU usage.

The evaluation results of the Scenario "Packet Delay in Ingress Queue of Switch" show us that the network infrastructure also plays a very important role in the overall delay of PLACE packets. The Cisco switches Cisco 2950 do not support ingress packet scheduling and, therefore, it was necessary to remedy this situation with the TBF egress queueing discipline on the sending cluster node. By using more advanced Ethernet switches with ingress packet scheduling, it will not be any more necessary to install the TBF queueing discipline on the sender node. And the switch will undertake the task of prioritizing the PLACE packets over other unimportant packets in the ingress queue of a switch ingress port.

The evaluation results of the Scenario "Packet Delay and Packet Delay Variation" show us that the PLACE protocol performs very poor when the receiver nodes are under a very high network load in the experiment network. Furthermore, high CPU usage in the domain domU introduces spikes in packet delays because of the Xen default domain scheduler – credit scheduler. With the default Xen credit scheduler which use 30 milliseconds time slices, the spikes are about 30 milliseconds high. With the adjusted Xen credit scheduler which uses 1 millisecond time slices, the spikes are only 1 millisecond high but nevertheless we could not get rid of them.

The receiving instance of the PLACE protocol was placed in the Linux kernel space of the domain dom0 during the protocol design and the results gained during the evaluation showed that this design decision is the main cause of large packet delays on the receiving side when the domain domU has high CPU usage or when the receiving side is under high network load in the experiment network. The problem lies in the scheduling of both domains by the Xen hypervisor so that while the domain domU is running the domain dom0 is blocked and, therefore, the PLACE packets which arrive in this moment cannot be processed by the Linux network stack of the domain dom0 where the receiving instance of the PLACE protocol runs. And it leads to spikes in packet delays discussed before.

High CPU usage in the domain dom0 and high network load in the control network do not cause the problem with very large packet delays.

But a very high network load in the control and the experiment networks is not a realistic situation in DTVEE and, therefore, this situation is not so significant. The

scenarios with a very high network load in both networks of DTVEE are only meant to research the behaviour of the PLACE protocol under the worst possible conditions in DTVEE. As the evaluation results show, the PLACE protocol performs well under 30% network load in the control and experiment networks of DTVEE.

As mentioned earlier, very high network load in the control network of DTVEE do not cause very large packet delays but the packet delay increases to about 1 millisecond and stays relatively constant near 1 millisecond while the control network of DTVEE is under very high network load. The reason for this increase of packet delay probably lies in the hardware properties of Cisco 2950 switches which build the control network.

Chapter 8.

Conclusion

This chapter concludes this diploma thesis concerning protocol for epoch switching in Distributed Time Virtualized Emulation Environment. At first, the hole topic is summarized and the most important results are recapitulated. Finally, limitations and open questions are discussed.

8.1. Summary

The goal of this diploma thesis was the development of a protocol to switch epochs in Distributed Time Virtualized Emulation Environment. In Chapter 1, the motivation and a detailed problem description was provided with a description of the main goals of this diploma thesis.

Chapter 2 presented a detailed description of DTVEE and its characteristics. The description covers network, node and operating system characteristics. Afterwards, the challenges introduced by the the special characteristics of DTVEE were discussed. This was followed by a detailed description of the main protocol requirements.

Chapter 3 introduced related work of this diploma thesis. The focus of the diploma thesis was developing and evaluating an efficient and low-latency network protocol for one-to-many communication to change TDF of cluster nodes simultaneously. Therefore, network protocols and projects on related topics were discussed in this chapter. At first, a short introduction to real-time systems was presented. This was followed by approaches to real-time process scheduling and hardware- and software-based approaches to real-time communication.

Chapter 4 introduced design issues of this work. It was subdivided into two main sections. First, basic concepts of the Linux kernel 2.6 network stack were presented. These fundamentals of the Linux network stack are crucial in order to understand possible design approaches introduced in the second part of design issues. Furthermore, the concepts of the Linux kernel 2.6 network stack are important for understanding the protocol design and implementation which are discussed in Chapter 5 and Chapter 6, respectively. The second part of design issues discussed several design approaches for every main protocol requirement with its advantages and its disadvantages.

The protocol design was introduced in Chapter 5. The overall protocol architecture, the most important parts of the protocol and the protocol behaviour were described in detail and illustrated by data structures, finite state machines and sequence diagrams. The role of each protocol component and its external interface were introduced.

Chapter 6 presented the implementation details of the protocol PLACE and explained in detail the data structures used in each component of the PLACE protocol. Additionally, the interfaces of the components were presented. Furthermore, the cluster node and switch configuration was explained.

The evaluation of the PLACE protocol was presented in Chapter 7. The goal of the evaluation was to examine how good the implementation of the PLACE protocol presented in Chapter 6 fulfills the protocol requirements explained in Chapter 2.2. In three steps, different parts of the PLACE protocol were separately evaluated, namely the performance, the total packet delay, the packet delay variation and the packet delay in the switch ingress queue. The behaviour of the protocol was studied under increasing CPU and network load conditions.

The evaluation had showed that the Xen default domain scheduler (credit scheduler) with 30 milliseconds time slices has a very negative effect on packet delay and causes packet delays upto 30 milliseconds when receiver nodes have high CPU usage in the domain domU and low network load in the experiment network of DTVEE. By changing the length of scheduler time slice to 1 millisecond, this problem could have been partially solved. Also high network load in the experiment network of DTVEE causes a large delay of packets on receiver nodes. The evaluation results also had showed that the behaviour of the PLACE protocol is tolerable when the network load is only 30% in the experiment network of DTVEE. Based on the results gained during the evaluation, it can be stated that the main goals of the protocol were fulfilled.

8.2. Limitations and Future Work

This section discusses limitations of the current protocol implementation and possible future work.

The first limitation of the PLACE protocol is its heavy dependence on the QoS functionality of the control network of DTVEE. Without the QoS support of the Cisco switches, the PLACE protocol is unusable and could not guarantee small packet delay in the control network. Furthermore, it also could not prevent packet loss under high network load in the control network of DTVEE without QoS support of Cisco switches.

The main limitation of the PLACE protocol is large delay of packets on receiving side of the protocol caused by the scheduling of domains when the domain domU has high CPU usage or high network load in the experiment network of DTVEE. This problem could be partially solved by decreasing the length of time slice used by the Xen default domain scheduler (credit scheduler).

The problem with the Xen domain scheduler could be completely solved by placing the receiving instance of the PLACE protocol not in the Linux kernel space of the domain dom0 but directly in the Xen hypervisor and, therefore, avoiding the problem with the domain scheduling completely.

Another advantage of placing the receiving instance of the PLACE protocol in the Xen hypervisor is that the receiving side does not have to make a TDF hypercall in order to change the TDF of the domain domU and, therefore, it would avoid expensive

context switches between the dom0 domain and the Xen hypervisor.

But placing the receiving instance of the PLACE protocol in the Xen hypervisor has also the disadvantage of less efficient and more complex communication between the the receiving instance of the PLACE protocol and the domain dom0 from where the receiving instance of the PLACE protocol is controlled.

Bibliography

[Adm98] Administratively Scoped IP Multicast. IETF RFC 2365, July 1998.

[AH06] G. Apostolopoulos and C. Hasapis. V-eM: A Cluster of Virtual Machines for Robust, Detailed, and High-Performance Network Emulation. *Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation (MASCOTS06)*, pages 11–14, September 2006.

[ASJS96] T. Abdelzaher, A. Shaikh, F. Jahanian, and K. Shin. RTCAST: Lightweight multicast for real-time process groups. *IEEE Real-Time Technology and Applications Symposium, Boston, Massachusetts*, pages 250–259, June 1996.

[BC05] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel, Third Edition*. O'Reilly Media, Inc., November 2005.

[Ben05] Christian Benvenuti. *Understanding Linux Network Internals*. O'Reilly Media, Inc., December 2005.

[Bro05] Eduard Broese. ZeroCopy: Techniques, Benefits and Pitfalls, 2005.

[Cas04] Matthew J. Castelli. *LAN Switching first-step*. Cisco Press, July 2004.

[Cis08a] Cisco Catalyst 2950 Series Switches. `http://www.cisco.com/en/US/products/hw/switches/ps628/index.html`, 2008.

[Cis08b] Cisco Catalyst 3550 Series Switches. `http://www.cisco.com/en/US/products/hw/switches/ps646/index.html`, 2008.

[CKHR05] Jonathan Corbet, Greg Kroah-Hartman, and Alessandro Rubini. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., February 2005.

[CSM] IEEE Std 802.3, 2000 Edition: IEEE Standard for Information technology–Telecommunications and information exchange between systems–Local and metropolitan area networks–Common specifications–Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications.

[DFH+03] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the Art of Virtualization. *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003.

[DSC98] Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. IETF RFC 2474, December 1998.

[ELA+02] Mustafa Ergen, Duke Lee, Roberto Attias, Stavros Tripakis, Anuj Puri, Raja Sengupta, and Pravin Varaiya. Wireless Token Ring Protocol. Master Thesis at UC Berkeley, July 2002.

[Eth08] Ethernet Multicast Addresses. `http://www.cavebear.com/archive/cavebear/Ethernet/multicast.html`, 2008.

[Fal99] K. Fall. Network Emulation in the VINT/NS Simulator. *Proceedings of the fourth IEEE Symposium on Computers and Communications*, July 1999.

[GRL05] Shashi Guruprasad, Robert Ricci, and Jay Lepreau. Integrated network experimentation using simulation and emulation. *Testbeds and Research Infrastructures for the Development of Networks and Communities*, pages 204–212, February 2005.

[GW88] R. Mangala Gorur and ALfred C. Weaver. Setting Target Rotation Times in an IEEE Token Bus Network. *IEEE Transactions on Industrial Electronics*, 35, August 1988.

[GYM+06] D. Gupta, K. Yocum, M. McNett, A. C. Snoeren, A. Vahdat, and G. M. Voelker. To Infinity and Beyond: Time-Warped Network Emulation. *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation*, pages 87–100, May 2006.

[HHH+02] F. Hanssen, P. Hartel, T. Hattink, P. Jansen, J. Scholten, and J. Wijnberg. A Real-Time Ethernet Network at Home. *Proceedings Work-in-Progress session 14 Euromicro international conference on real-time systems (Research report 36/2002, Real-Time Systems Group, Vienna University of Technology)*, pages 5–8, June 2002.

[ICM81] Internet Control Message Protocol. IETF RFC 792, September 1981.

[IGM97] Internet Group Management Protocol, Version 2. IETF RFC 2236, November 1997.

[IMQ08] Intermediate Queuing Device. `http://www.linuximq.net`, 2008.

[Ind08] The Industrial Ethernet Book. `http://ethernet.industrial-networking.com/`, 2008.

[int08] Intel 8255x 10/100 Mbps Ethernet Controller Family Open Source Software Developer Manual. `http://download.intel.com/design/network/manuals/8255X_OpenSDM.pdf`, 2008.

[IP81] Internet Protocol. IETF RFC 791, September 1981.

[KaZB05] Jan Kiszka, Bernardo Wagner andYuchen Zhang, and Jan Broenink. RTnet - A Flexible Hard Real-Time Networking Framework. *10th IEEE International Conference on Emerging Technologies and Factory Automation*, pages 19–22, September 2005.

[KR05] James F. Kurose and Keith W. Ross. *Computer Networking: A Top Down Approach Featuring the Internet*. Pearson Education, Inc., 2005.

[Lin08a] Intermediate Functional Block Device. `http://www.linux-foundation.org/en/Net:IFB`, 2008.

[Lin08b] Linux kernel preemption project. `http://kpreempt.sourceforge.net/`, 2008.

[MH05] José María Martínez and Michael González Harbour. RT-EP: A Fixed-Priority Real Time Communication Protocol over Standard Ethernet. *Proceedings of the International Conference on Reliable Software Technologies*, June 2005.

[MHG03] José María Martínez, Michael González Harbour, and J. Javier Gutiérrez. RT-EP: Real-Time Ethernet Protocol for Analyzable Distributed Applications on a Minimum Real-Time POSIX Kernel. *Proceedings of the 2nd Intl WORKSHOP ON REAL-TIME LANS IN THE INTERNET AGE*, July 2003.

[mro08] Linux-Mrouted-MiniHOWTO. `http://www.jukie.net/~bart/multicast/Linux-Mrouted-MiniHOWTO.html`, 2008.

[NET08] NET-Project. `http://net.informatik.uni-stuttgart.de/`, 2008.

[NTP92] Network Time Protocol (Version 3) Specification, Implementation and Analysis . IETF RFC 1305, March 1992.

[Ope08] OpenVZ. `http://openvz.org`, 2008.

[PD03] Larry L. Peterson and Bruce Davie. *Computer Networks: A Systems Approach, 3rd Edition*. Morgan Kaufmann, 2003.

[pro08] Protocol Numbers. `http://www.iana.org/assignments/protocol-numbers`, 2008.

[RTA08] RTAI - Real-Time Application Interface for Linux. `http://www.rtai.org/`, 2008.

[RTL08] RTLinux. `http://www.rtlinuxfree.com/`, 2008.

[RTP96] RTP: A Transport Protocol for Real-Time Applications. IETF RFC 1889, January 1996.

[RV92] Jose Rufino and Paulo Verissimo. A Study on the Inaccessibility Characteristics of ISO 8802/4 Token-Bus LANs. *INFOCOM (2)*, pages 958–967, 1992.

[SGG04] Avi Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts, 7th Edition*. John Wiley & Sons, 2004.

[SGZ+02] Srikant Sharma, Kartik Gopalan, Ningning Zhu, Gang Peng, Pradipta De, and Tzi cker Chiueh. Implementation Experiences of Bandwidth Guarantee on a Wireless LAN. *ACM/SPIE Multimedia Computing and Networking*, January 2002.

[SMK+01] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer to Peer Lookup Service for Internet Applications. *Proceedings of the 2001 SIGCOMM*, August 2001.

[SNM90] Simple Network Management Protocol (SNMP). IETF RFC 1157, May 1990.

[Spu00] Charles E. Spurgeon. *Ethernet: The Definitive Guide*. O'Reilly Media, Inc., February 2000.

[TBF08] TBF queuing discipline. `http://www.opalsoft.net/qos/DS-24.htm`, 2008.

[TCP81] TCP: Transmission Control Protocol. IETF RFC 793, September 1981.

[Tzi99] Tzi-cker Chiueh. RETHER: A SoftwareOnly RealTime Ethernet for PLC Networks. *Proceedings of the Embedded Systems Workshop*, pages 29–31, March 1999.

[UDP80] UDP: User Datagram Protocol. IETF RFC 768, August 1980.

[Ven96] Chitra Venkatramani. The Design, Implementation and Evaluation of RETHER: A Real-Time Ethernet Protocol. *PhD thesis, State University of New York at Stony Brook*, December 1996.

[VYW+02] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. *Proceedings of the Fifth Symposium on Operating System Design and Implementation (OSDI)*, December 2002.

[WPR+04] Klaus Wehrle, Frank Pählke, Hartmut Ritter, Daniel Müller, and Marc Bechler. *The Linux Networking Architecture: Design and Implementation of Network Protocols in the Linux Kernel*. Prentice Hall, August 2004.

[xen08a] Xen Credit Scheduler. `http://wiki.xensource.com/xenwiki/CreditScheduler`, 2008.

[Xen08b] Xenomai: Real-Time Framework for Linux. `http://www.xenomai.org/`, 2008.

Appendix A.

Appendix

A.1. PLACE Use Cases

A.1.1. TDF Sender Module Use Cases

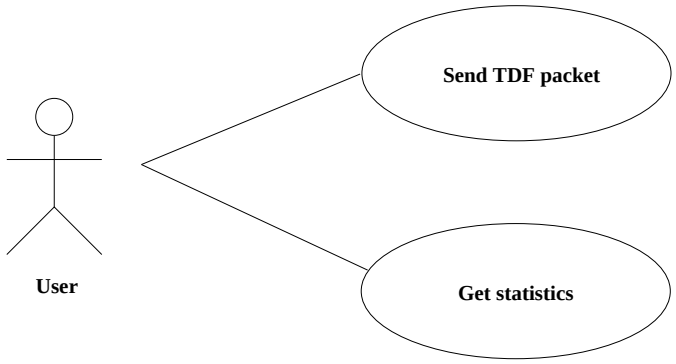


Figure A.1.: TDF Sender Module Use Cases

A.1.2. TDF Receiver Module Use Cases

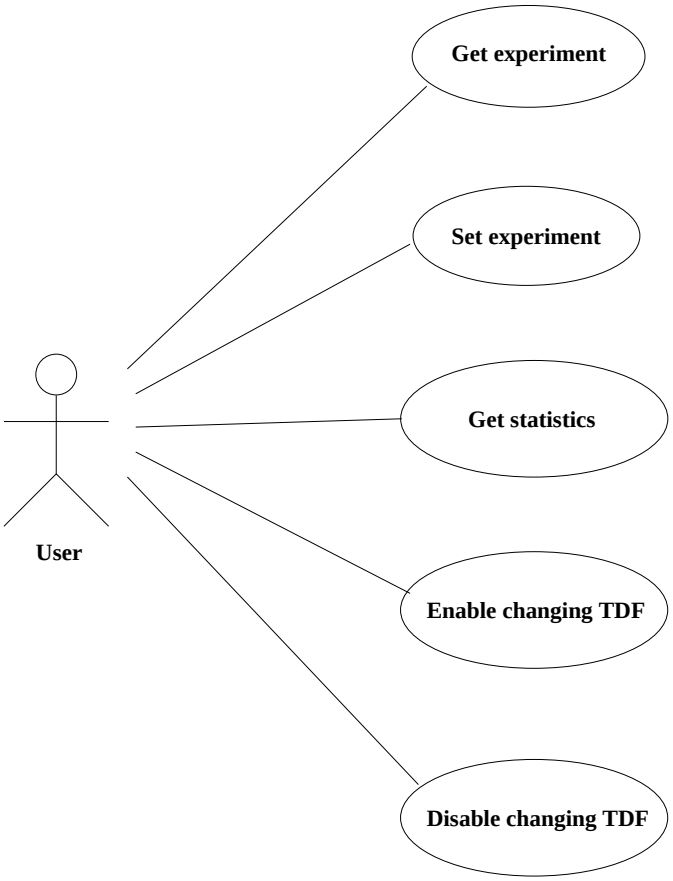
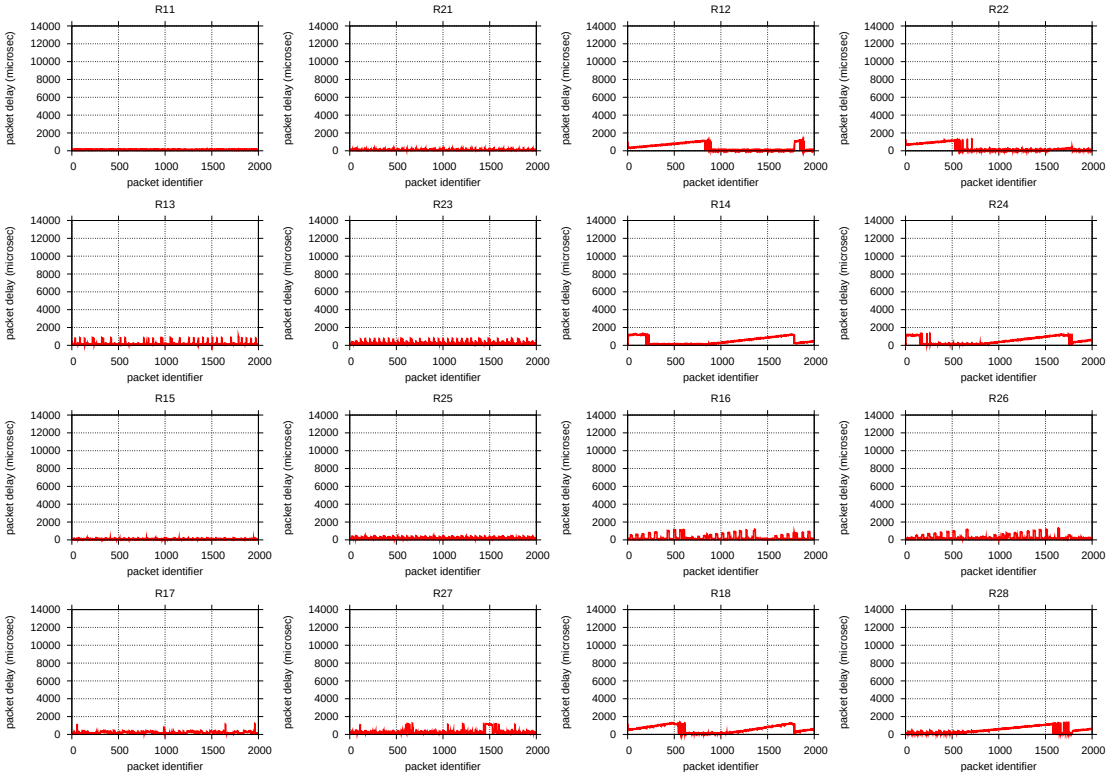
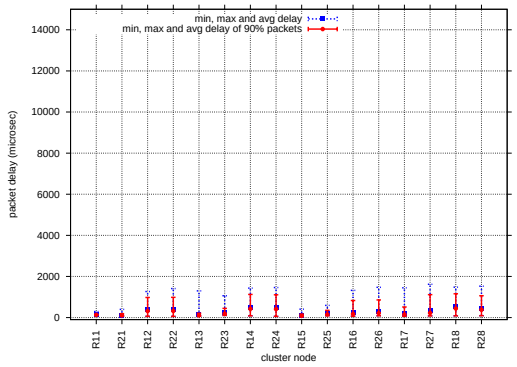


Figure A.2.: TDF Receiver Module Use Cases

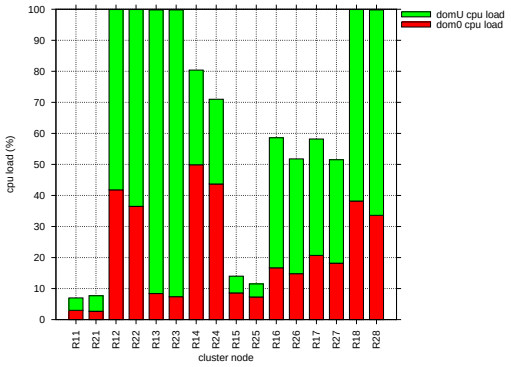
A.2. Evaluation Results



(a) Packet delays

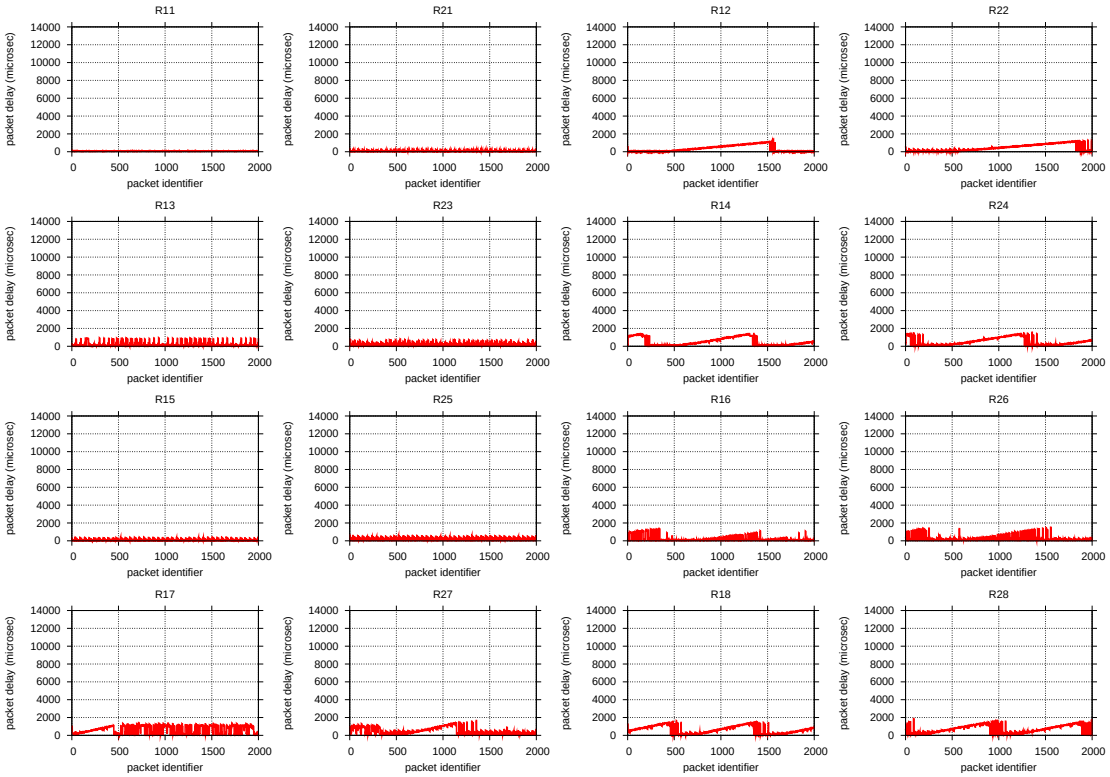


(b) Minimum, maximum and average packet delays

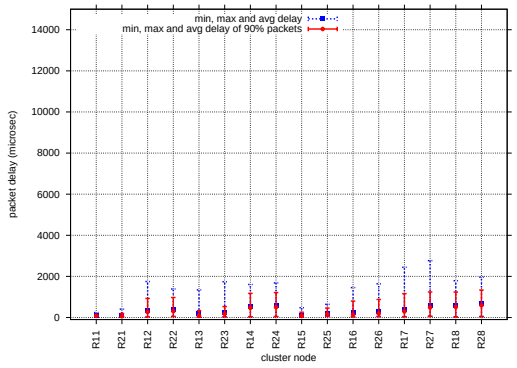


(c) CPU load of cluster nodes

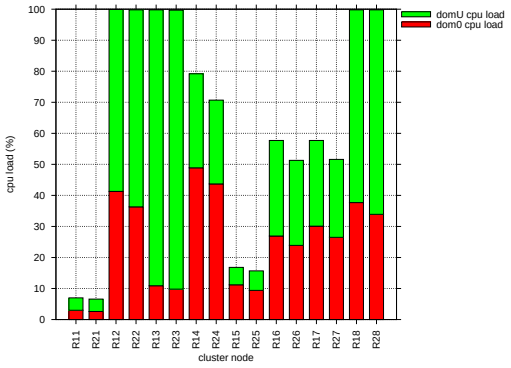
Figure A.3.: Subscenario 1 results



(a) Packet delays

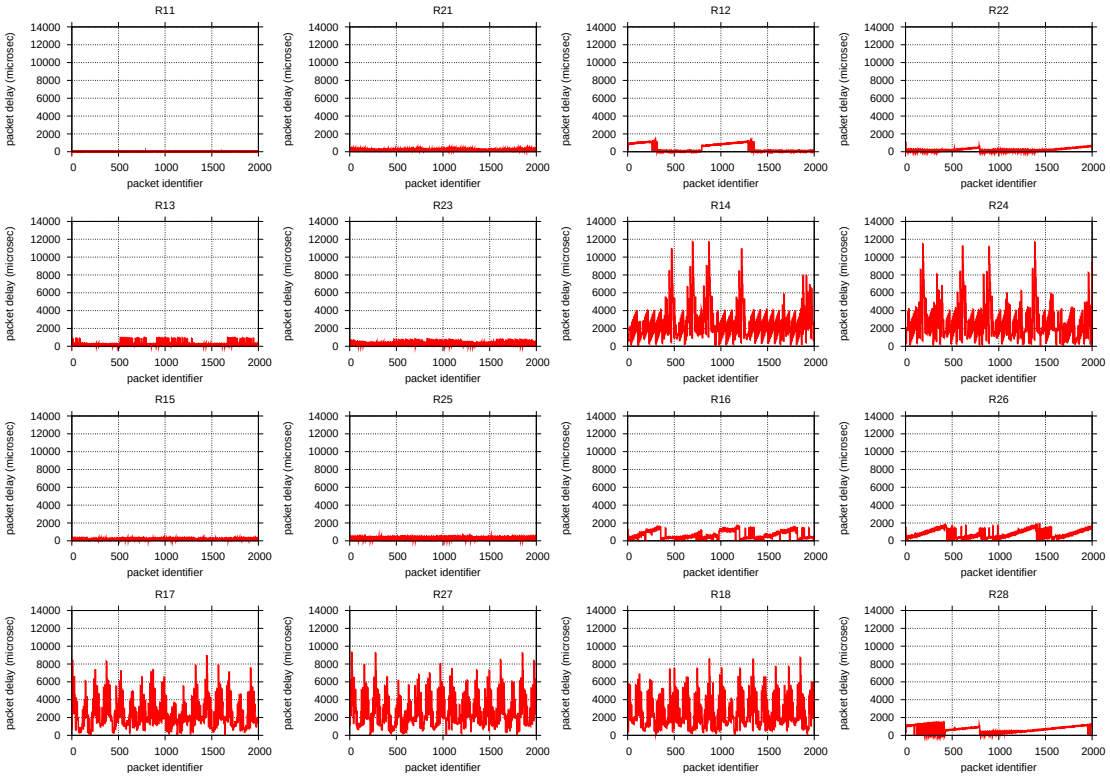


(b) Minimum, maximum and average packet delays

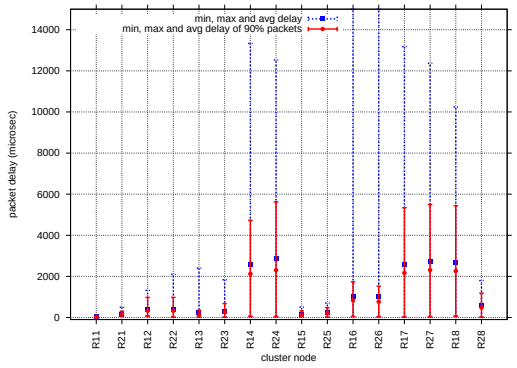


(c) CPU load of cluster nodes

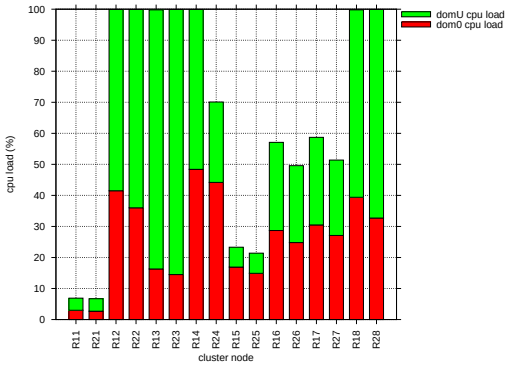
Figure A.4.: Subscenario 2 results



(a) Packet delays

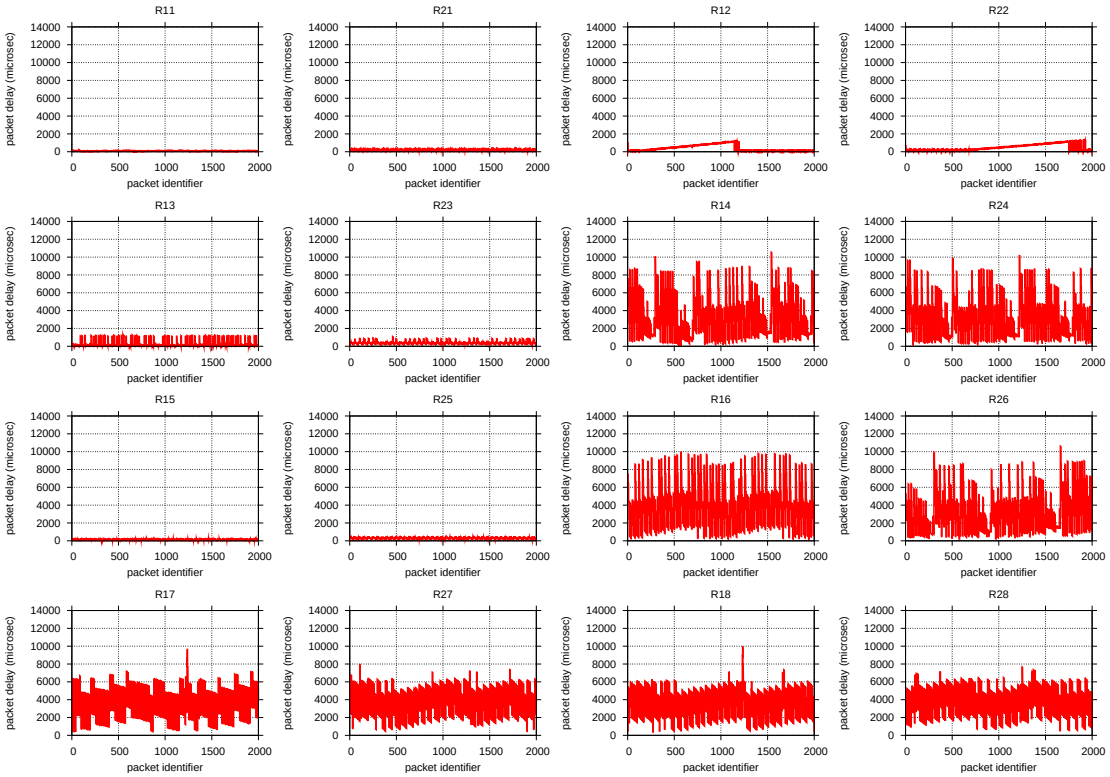


(b) Minimum, maximum and average packet delays

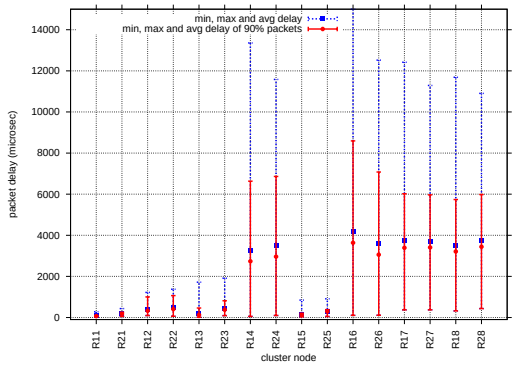


(c) CPU load of cluster nodes

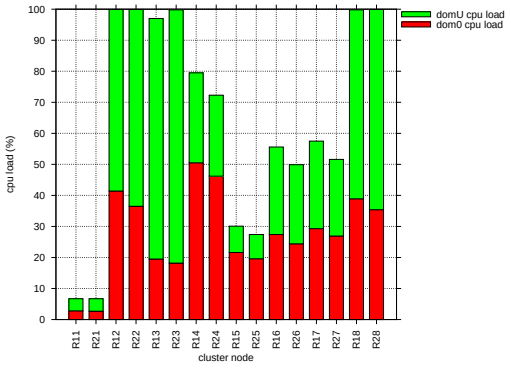
Figure A.5.: Subscenario 4 results



(a) Packet delays

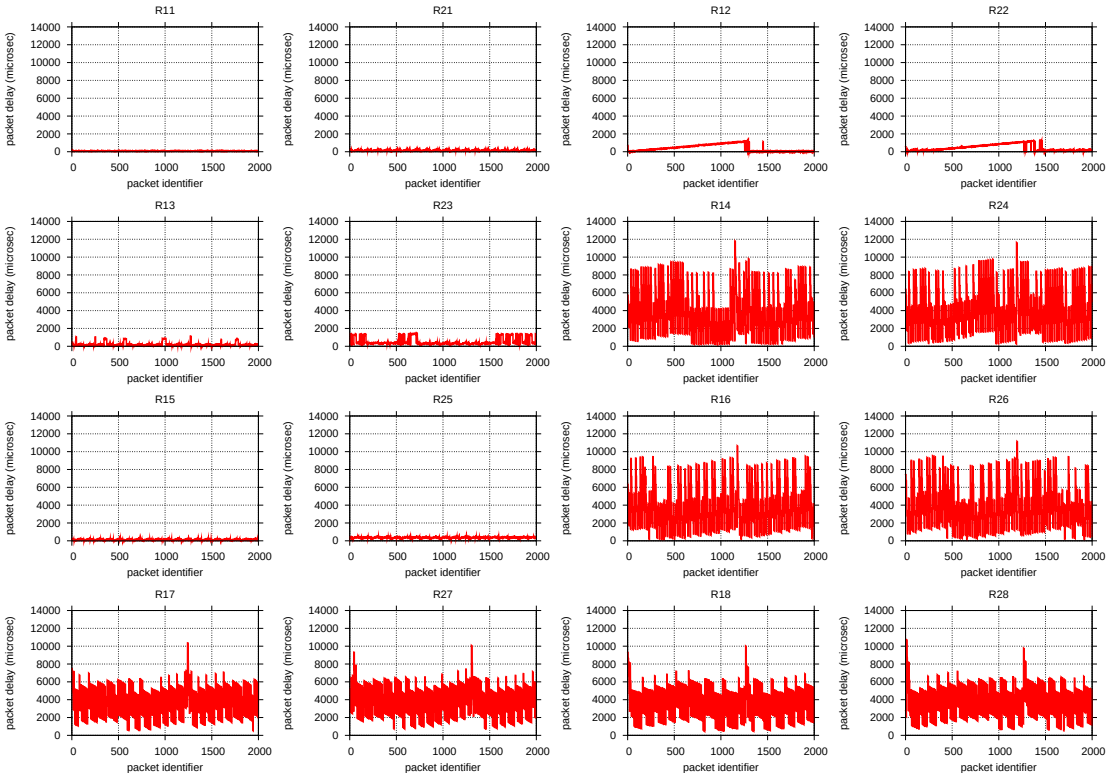


(b) Minimum, maximum and average packet delays

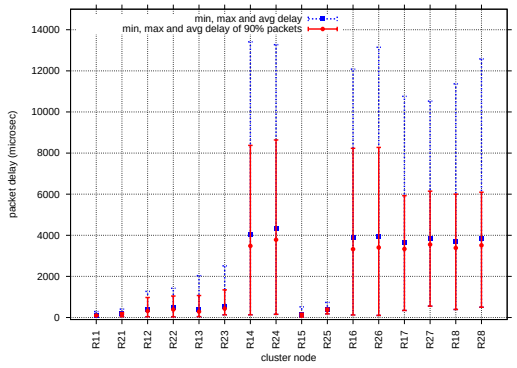


(c) CPU load of cluster nodes

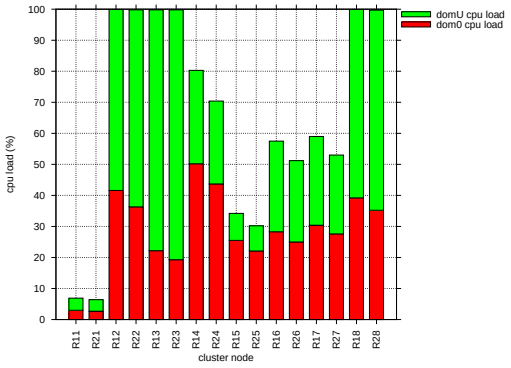
Figure A.6.: Subscenario 6 results



(a) Packet delays

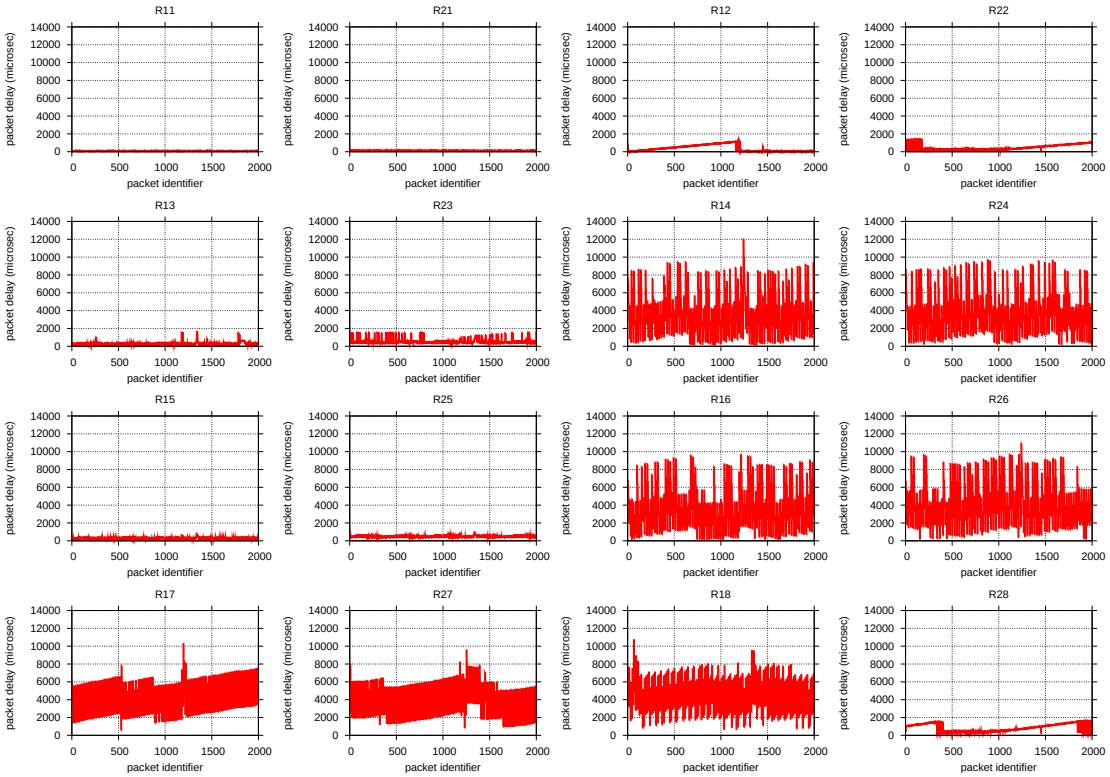


(b) Minimum, maximum and average packet delays

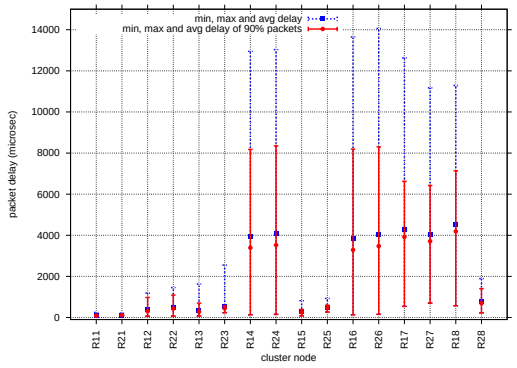


(c) CPU load of cluster nodes

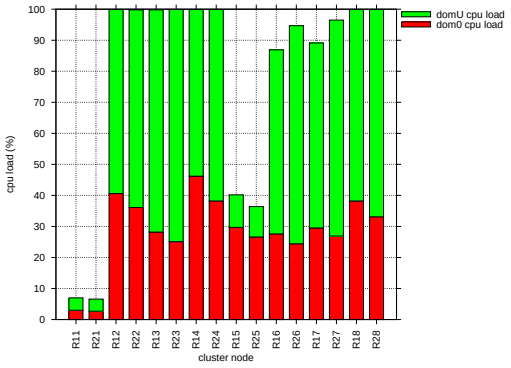
Figure A.7.: Subscenario 7 results



(a) Packet delays



(b) Minimum, maximum and average packet delays



(c) CPU load of cluster nodes

Figure A.8.: Subscenario 9 results

A.3. Statement

I ensure that I have created this document on my own and only used those external sources listed in the bibliography.

Stuttgart, _____

_____
Alexander Egorenkov