UNIVERSITY OF OSLO
Department of Informatics

Multicast traffic management and performance in Ethernet/Layer 2 Networks

Øystein Taskjelle

May 3, 2010

Abstract

The present thesis seeks to develop a better understanding of the Multiple MAC Registration Protocol (MMRP) and multicast in Ethernet. A theoretical study about mapping from IP version 4 multicast to Ethernet multicast, and bridge technology is presented in the first part. The main emphasis is the implementation of MMRP in the J-SIM simulation environment. Some performance aspects of MMRP are simulated. The results shows that the protocol timers may be altered to optimized MMRP's mechanisms for a given scenario.

Preface

This thesis is the final task of the fulfillment of my Masters degree at the Department of Informatics at University of Oslo and the University Graduate Center at Kjeller (UNiK).

I would like to tank my supervisors Fredrik Davik at Nera Networks and Knut Øvsthus at Bergen University College. They have helped me throughout the work, and given constructive criticism.

My family and friends, thank you for being supportive through the whole process of this Master's program. Espen and Thomas have made the days at school a lot of fun! Thank you guys for (all) the coffee breaks!

Øystein Taskjelle

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A cellular network includes different technologies from its wireless access network to the high speed core network. The part between the core network and the cellular base stations is called the backhaul. The aggregation network is the part of the backhaul, located in between the core and the part in the backhaul where multiple base stations are connected.

The access network has been the bottleneck, for bit rate, in 2G networks. The transition to 3G, 3.5G and 4G cellular networks gives the end user an increased bit rate in the access network, and the bottleneck is moved to the backhaul. Faster, in terms of bit rate, access networks push new services to the marked. Multimedia services increases the bandwidth need. Currently, users watch YouTube, surf on the internet, send e-mails and make calls with VOIP using their cellular network device. The first LTE network was made publicly available in December 2010. It was TeliaSonera that opened it in Norway and Sweden. A LTE network will most likely created an increased bit rate demand. In addition, due to the higher access network capacity, "mobile broadband" has gained popularity in Norway. The term "mobile broadband" means to use the cellular network to gain access to the Internet. The data volume sent and received by mobile broadband increased to 6.5 million gigabytes a year in 2009 for Norway. This is more than double of 2008 [14].

3G introduced packet switching in the backhaul, initial with ATM. ATM normally used bundles of E1s/T1s. However, the trend goes toward IP/MPLS/Ethernet[1] architecture, which may replace ATM. In addition, the LTE does not include ATM as a part of the standard, as the 3G network does. Bit rate gets cheaper for the operator with IP/Ethernet in the backhaul [33, 12], but with new services like TV, it will also be necessary to optimize the bandwidth usage in the backhaul. Aggregation networks based on radio relay systems is one of the reasons for the need of optimization. The frequencies in wireless technologies is a limited resource, and in addition physical properties of radio waves limit the capacity of wireless links. Multimedia-services with

[1] IEEE 802.3 standard.

one traffic-source and multiple receivers may use:

$$N \times B[bps] = T[bps]$$

Where N = users in the network, B = service bit rate demand [bps], and T = total-bit-rate usage in the network [bps]. IP Multicast is a solution to this problem. It will save resources by using logical routing of flows where there are multiple receivers.

The trend is that the access networks evolve toward IP and the backhaul toward Ethernet [31]. It may go parallel traffic of IP Multicast in the Ethernet, if Ethernet is not implemented with IP Multicast support. This removes the gain in terms of lesser load on the network by using IP Multicast. The challenge is to have a mapping between Ethernet and IP Multicast packets, that avoids parallel streams. There are two main approaches to this.

- The first is that the IP-layer must translate the logic so that Ethernet transmits the packets correct without changes in the protocol. Because Ethernet is below IP in the network architecture, Ethernet is invisible to IP. This makes it hard because the IP-layer does not know the topology and functionality of Ethernet.
- The other approach is that Ethernet adds functionality to understand the incoming IP Multicast packets and find the IP Multicast members on the network. With this, Ethernet is able to determine where to send and not to send the IP Multicast packets. This approach is the logical choice because Ethernet are able to snoop the IP IGMP messages, but not the other way around.

Multicast routing and group membership handling is the two main parts of IP Multicast. Group membership of the end users is handled by the Internet Group Management Protocol (IGMP) [5] in IPv4 or Multicast Listener Discovery[8] in IPv6. When a layer 2 device like Ethernet-bridges listen to the IP-packet to fetch IGMP-messages, it is called IGMP-snooping. With multiple Ethernet-devices in the aggregation network, the group membership must be shared with all devices in order to make the logical sending of IP Multicast packets through the Ethernet. IEEE Std 802.1ak [24] defines the Multiple Registration Protocol (MRP) and the Multiple MAC Registration Protocol (MMRP), which may be used for this purpose.

This introduction will further look at the problem statement, then work method and what tools that are used in the work. Finally it gives a short presentation of the thesis's outline.

1.1 The problem statement

The overall objective is to evaluate Multiple MAC Registration Protocol (MMRP) as a multicast solution in an Ethernet context. The convergence time of registrations and deregistrations are the most important parameters studied in this thesis. The following list presents the problems investigated in this thesis:

1. The problem of dynamic mapping from layer 3 (IP) multicast to layer 2 (Ethernet) multicast functions.

2. To evaluate potential bandwidth savings by mapping from layer 3 to layer 2 multicast.

3. Evaluate different aspects of MMRP theoretically and by simulation.

The first and seconds problem are solved theoretically. The third problem is evaluated theoretically, and followed by a simulation that tests the hypotheses. A implementation of current protocol(s) is required to complete the goals.

1.2 Work method

This section provides an overview of method and tools used to test (i.e. analyze and discuss) a set of working hyphoteses. After a brief overview of the different methods, and a conclusion on which methods to use — the tool(s) are introduced. I have used [30] as a source of information describing the working methods.

1.2.1 Approaches

The work method chosen should support a scalable and easy configurable test setup. Reasons backing this up are:

1. There are several scenarios of interest regarding network topology,

2. a topology may contain many devices, and

3. the time to complete the thesis is limited.

This part consider three approaches followed by a part that explains the choice taken.

Analytical methods In some situations a mathematical approach is relatively straight forward, and gives a good result when it is based on pre-developed algorithms and formulas, and known problems. But creating new algorithms and formulas requires a high level of knowledge both in mathematics and the content of the system itself.

Simulation Simulation software is widely distributed on the Internet with a multitude of frameworks, languages and environments available. Some are more popular and better developed than others.

Relatively detailed and realistic simulation models can be used — the challenge is to include all that is relevant for the evaluation but nothing more. The results from a simulation may have stochastic uncertainty, and it may be complex to analyze.

Measurement-based testing in real networks In order to create a scenario with real devices, a lot of equipment is required to realize a satisfactory starting point. The equipment needed is end users (like laptops or Ethernet/IP test instruments emulating end users), a number of layer 2/Ethernet switches, or some kind of a combination of these. With a given set of available hardware, there are a number of experiments. Furthermore, there may be several topologies applicable for each experiment. At any given time, a node is only a part of one topology, and switching to another topology requires reconfiguration. In addition, measurement-based testing may be error-prone. This is due to comprehensive and complex configuration of devices used to testing.

If the devices does not support the desired protocols, they need to be developed and implemented in the device, which depends an open OS or platform on the device . The cost in time exceeds the scope of this thesis.

There must be a way of collecting the data that are to be analyzed. This is necessary at every device in the network. It should be implemented in such a way that the collection is automated.

1.2.2 What approach to choose?

Real life tests require preparation of devices in order to generate results. Although the measurement based method is considered unfeasible as the main method for this thesis, it still presents an alternative that will be considered for limited parts. One example could be getting to know the packet-structure and -content through a transmission by sniffing of packets.

A full mathematical approach is to complex to solve for me. However, it might be useful for minor calculations. The obvious method is simulation, which is something I am familiar with and will produce the results needed to test my working hypotheses. An additional argument is to get a better knowledge of network simulation.

1.2.3 Tools

Nera Networks have knowledge of the J-SIM discrete event simulation environment. The choice of J-SIM is due to Nera Networks existing implementations in this environment. This gives a base for the creation of desired scenarios for the study. In addition, Nera Networks expands their tool base in J-SIM by including the implementation that is a product of this study. The J-SIM environment will be addressed in chapter 3 J-SIM.

`NetworkDesigner` is a little self developed Java program with a graphical user interface. The interface makes it easy to draw network topology designs for simulation. In addition, it has some primitive features to export the topology to the `pstricks` format for the report, and it is able to parse output from the simulator to draw the network topology and certain states of the nodes. A screenshot is shown in appendix C *Network designer screenshot*.

1.2.4 Simulation practice

In reference [30] a simulation study approach is presented which is summed up in figure 1.1. Each of the steps are not isolated, but connected as shown in the figure. One goes iteratively through the model, from development to validation and verification, throughout the process. The next paragraphs explains each of the steps in the model, as seen in figure 1.1. In later chapters of the thesis, these steps will be used to explain the work done.

Problem formulation and objectives This is the first step of the simulation study, and it starts off by defining the problem. A clear understanding of the problem is the starting point for further work. Based on this, the objectives of the study are prepared. In addition, a work plan that shows how to reach the objectives should be planned.

Model of conceptualization Model of conceptualization is also known as making an abstraction of the real system. In an abstraction process, one makes a simplified version of the real system. To simplify means that the abstracted model only includes the functionality that is important for the study. Note that in some cases, the abstracted model may be a complete copy of the real life system, which means that it is not simplified in terms of functionality.

Data collection In order to test realistic scenarios, one need to collect information regarding the systems behaviour from real life. This information is for example parameters like buffer size on network nodes or traffic pattern. More on how this data is collected in chapter 5.

Model translation The implementation is based on the abstraction model in J-SIM. As mentioned earlier, the work process goes iteratively through the phases mentioned in the start of this section, and it is natural to do the implementation step by step as the abstraction model is created. The step-by-step process of implementation ease the debugging since the code is tested piece by piece, and not all at once.

Testing When the model is implemented, it is time to verify that it works as intended, and validate that it imitates a real world execution. Going back to model conceptualization may be necessary if the verification or the validation reveals incorrect implementation.

Experiments A set of experiments testing the hypethesis(es) will form the bases for analysis. The simulation experiments may be time consuming due to both runtime of the simulation and repeatidly running simulations to get the output needed.

Post-processing When the simulator has produced some output, the output must be parsed and analyzed to produce statistics, graphs and save parameter values of interest. Due to large amount of data produced, it may be desirable with a tool that can parse the output. Taken the result of the output in consideration, it might be necessary to run new

experiments with different configurations. The last part is to present the results of the analysis in the thesis.

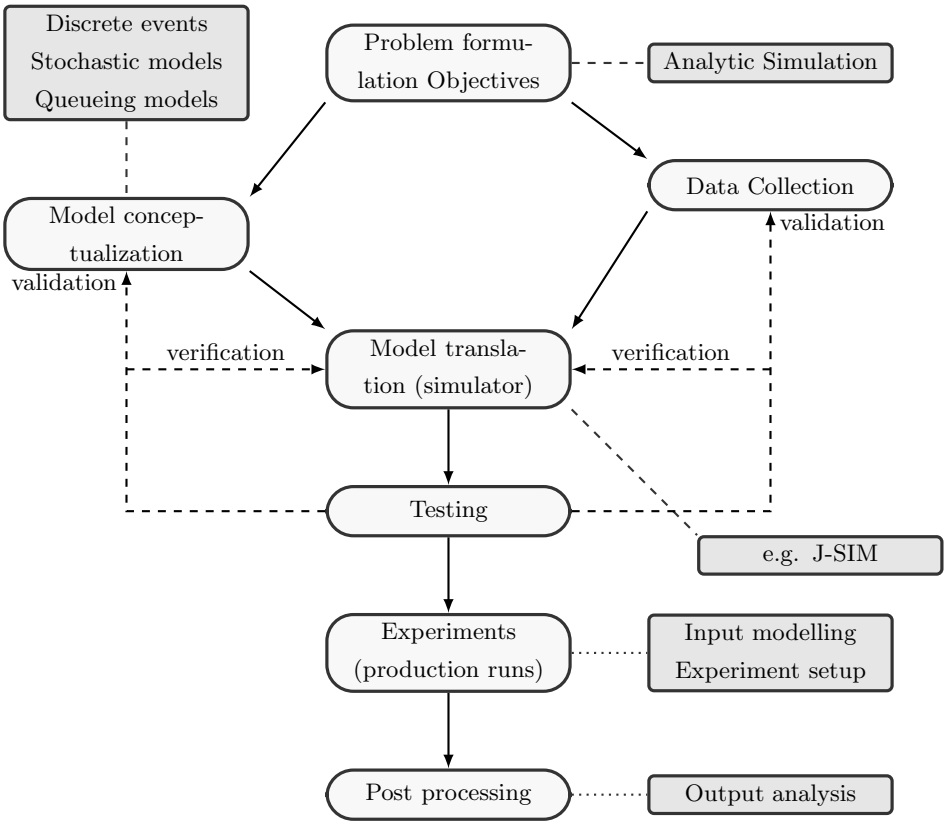


Figure 1.1: Flow chart of a simulation study [30].

Chapter 5 *Implementation* addresses study specifics for this thesis. The subparts of the section describes the points in the list above, e.g. 5.2 addresses the modeling specifics of this thesis's problem.

1.3 The scope of the thesis

The implementation is confined to Ethernet protocols, and no IP protocols are included. Consequently, discussion about IP protocols limits to a theoretical grounds. A real life testbed was not made, due to the time available to finish the thesis.

1.4 Outline

The thesis is divided in eight chapters, i.e. seven chapters after the introduction. Chapter 2 *Technology background* describes Ethernet and IP protocols that has relevance to the current problems. Next, chapter 3 *J-SIM* introduces the J-SIM simulation environment. Understanding the architecture and mechanisms of the simulation environment is a necessary base before implementing the Ethernet protocols. Next follows chapter 4 *Multiple MAC Registration Protocol*. This chapter introduces the MMRP protocol, before its described in more detail in the following chapters. The next chapter, 5 *Implementation* describes the implementation process from abstraction to implementation in the J-SIM environment. It is naturally followed by the chapter 6 *Testing* that verifies that implementation works as expected. Chapter 7 *Experiments and results* describes the simulation experiments, while chapter 8 *Discussion* does a discussion around the results from the simulation and the theory. Finally, chapter 9 *Conclusion* concludes this thesis.

Chapter 2

Technology background

2.1 Introduction

This chapter first introduces the OSI reference model, an abstract layered network communication model. The model is often used to describe what type of functionality a network protocol has, and what protocols it can communicate with. As the model is referred to in this thesis, and the model gives an overview of network protocol architecture, it is relevant to give an introduction. The next part of this section describes different types of routing in a network. One of the routing types is multicast – the type of routing that are used in Multiple MAC Registration Protocol (MMRP), the main subjective of this thesis. Finally network topologies are categorized and presented.

2.1.1 Layered network protocol design

The OSI reference is used to explain where in the network architecture a protocol belongs. This part gives a short introduction to the model. For further history and information read [27].

The computer networks gained success and popularity in the mid 1970's. The ARPANET that was developed in the USA lead the way of interconnecting networks. Packet switching had a commercial potential and a need for an international standard was important. That way it was possible to expand the networks reaching more people and services.

In 1978, the International Organization for Standardization (ISO) Technical Committee 97 on Information Processing, recognizing the need for a standard of interconnection between networks. A subcommittee, SC16, was created. The name of the subcommittee was "Open System Interconnection" or OSI. The term "open" means that by using this model one would be able to connect to other systems following the same model. The development of the model went through discussion and refinements and published as ISO standard 7498 in the Spring of 1983.

The model is based on a layered architecture to break up the network functionality in smaller

pieces. The top layers of OSI gives a description at a high level of abstraction which imposes few constraints and the further down in the layers, the more network details is defined. Seven layers are defined for OSI as seen in table 2.1. The implementation of a communication system may be based on layers where the end-host uses all the layers while the intermediate nodes in the network has support for bottom two or three layers to support the switching and routing of packets as needed.

Switches and bridges have generally support for Layer 1 (Physical) and Layer 2 (Data-link) functionality. Routers add more functionality with Layer 3 (Network) in addition to Layer 1 and 2. IP Multicast is a Layer 3 matter — technology mainly found in routers. The protocols referred to in this report are present in the Layers mentioned above, two and three. Layer 4 and higher will not be discussed.

The OSI model is only a reference model, and current networks are not implemented as the model, even though it is layered it uses the TCP/IP model as seen in table 2.1. As we can see in the comparison between the models. Layer 2 is divided in two pieces in Ethernet, LLC and MAC. Though, this layer might look different, e.g in optical networks.

Layer 3 – the network layer, handles relaying and forwarding of data packets through a network while it may maintain quality of service parameters. The Network Layer is at the same level as the IP-layer in the TCP/IP-model. **Layer 2 – Data link layer**, provides a reliable transmit of data across a physical network link. IEEE has splitted the Layer in three parts. The top-part called *Logical Link Control* (LLC), the middle part Bridging, and the bottom part *Medium Access Control* (MAC). LLC is concerned with the transmission of a LLC PDU (data) between two stations without the necessity of a intermediate switching node. Bridging may connect several LANs and it serves filtering functionality and creation of logical topologies. MAC controls the sending of the data between two stations. It is responsible of avoiding collision on the physical medium, grant access to send and physical addressing of devices.

2.1.2 Routing methods

Routing in computer networks is about choosing a path to through the network towards one or more receivers. In packet switching networks, routing schemes differ how they deliver packets to receiver(s). This sections presents the three schemes unicast, broadcast and multicast.

Unicast sends one data stream to a receiver, i.e. one transmitter (A) sends a single stream (S) to an user (B). If one more user (C) wants to receive the same stream, S, from A, then A must sends a duplicate stream of S towards C. For each station that wants stream S, node A needs to send a duplicate of stream S. In *broadcasting*, node A sends stream S to all stations in the network, but only one stream is transmitted from A, and intermediate devices in the network forwards stream S on all interfaces except from the one which stream S entered. The problem with this solution is that it consumes a lot of bandwidth and all stations in the network receives

<table>
<tr><th>#</th><th>OSI</th><th>TCP/IP</th><th>Application</th></tr>
<tr><td>7</td><td>Application</td><td rowspan="2">Application</td><td rowspan="3"></td></tr>
<tr><td>6</td><td>Presentation</td></tr>
<tr><td>5</td><td rowspan="2">Session</td><td rowspan="2">Transport (host-to-host)</td></tr>
<tr><td>4</td><td>Transport</td><td></td></tr>
<tr><td rowspan="2">3</td><td rowspan="2">Network</td><td>Internet protocol</td><td>IGMP</td></tr>
<tr><td rowspan="4">Network access</td><td></td></tr>
<tr><td rowspan="3">2</td><td rowspan="3">Data link</td><td>802.1 LLC</td></tr>
<tr><td>802.1 Bridging</td></tr>
<tr><td>802.3 MAC</td></tr>
<tr><td>1</td><td>Physical</td><td>Physical</td><td>802.3 PHY</td></tr>
</table>

Table 2.1: The OSI reference model compared to TCP/IP and IEEE [10, 9].

the stream, even those who may not want to receive it. E.g. station C can not choose to receive or not receive. See figure 2.1.

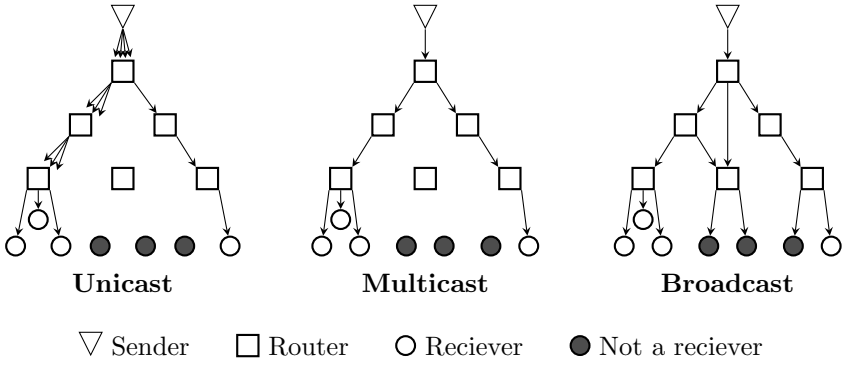


Figure 2.1: Different routing methods. Scenario: there are several receivers of a stream, for example a television channel, and some users who do not wish to receive the stream. The arrows illustrates how many duplicates of the streams that is sent with the different addressing methods, and where the streams are sent in the network. As we can see, in this scenario, multicast is the most efficient regarding bandwidth utilization.

Both unicast and broadcast is not good solutions to transmit a bandwidth demanding stream towards several users. From Steve Deering's work in the 1980's came IP *multicast*. The principle of multicast is that no matter how many users that are receiving the stream, only one stream is transmitted from the sender. Intermediate nodes in the network keep track of which interfaces that has a receiving user, and forwards only on these interfaces. As seen in fig. 2.1, it

serves a controlled distribution of the stream (improvement from broadcast) and saves bandwidth (improvement from both unicast and broadcast).

2.1.3 Network topologies

Network topologies can be categorized as the list below. Topologies may be composed by one more of these patterns.

Point-to-point Two end-users are connected directly. Due to full-duplex support there will not be collisions.

Bus Several users can be connected to a shared medium. Users are for example end-stations like a laptop, a printer or a server.

Star Several users are connected to a switch or hub, and by placing the switch in the middle of a topology-map it looks like a star. Hubs are not used much anymore due to its inefficient nature.

Hierarchy Several star-topologies gives a hierarchical built network.

Mesh In a full mesh, all nodes are connected directly to each other.

2.2 Ethernet switching

A short presentation of the origin follows. The standard IEEE Std 802.3 [2] has more information on the subject. Xerox developed the first Ethernet and it had a bit rate of 2.94 Mb/s. This work was the basis of the first IEEE Std 802.3 published in 1985, with an improved bit rate of 10 Mb/s. Since 1985, many projects has added or improved features of 802.3. Today Ethernet supports up to 10 Gb/s at full duplex, and the 100 Gb/s is imminent.

Ethernet defines a protocol for the physical- and the data link layer, and it is a wired LAN and WAN technology. Each station has a physical address, also called MAC-address. This is used to identify the device in the other end of the medium. MAC wraps up the LLC PDU, and sends the data over the link. The packet format of MAC is shown in table 2.4.

IEEE Std 802.1Q [23, 11] defines MAC bridging. A bridge has three main components relevant to this thesis. The first is the part that creates a logical topology based on the physical, and is described in the following section 2.2.1 *Rapid spanning tree*. Next, a bridge has mechanisms to forward data logically. This is done by the MAC relay entity component described in section 5.2.1 *Other bridge functionality*. The last component is the Multiple MAC registration protocol (MMPR), which is described further in section 4 *Multiple MAC Registration Protocol*.

0 3	4 7	8 15	16 18	19 24	31
Version	IHL	ToS	Total length		
Identification			Flags	Fragment offset	
Time to Live		Protocol	Header checksum		
Source address					
Destination address					
Options				Padding	

Table 2.2: IP version 4 packet Header [6]

0 3	4 11	12 15	16 23	24 31
Version	Traffic class	Flow label		
Payload length			Next header	Hop limit
Source address (128 bit)				
Destination address (128 bit)				

Table 2.3: IP version 6 packet header [7]

7 octets	Preamble
1 octet	SFD
6 octets	Destination address
6 octets	Source address
2 octets	Length/Type
4-1500 octets	Mac Client Data
	Pad
4 octets	Frame check sequence
	Extension

Table 2.4: MAC Frame format [2]

2.2.1 Rapid spanning tree

A logical topology in Ethernet is traditionally created on the basis of a spanning tree protocol (STP). Standard [22] introduced rapid STP (RSTP) and forms the basis for Multiple STP (MSTP) defined in standard [23]. In switched networks without VLANs, RSTP should be used. MSTP present more opportunities in forms of traffic engineering in bridged networks, but this is not relevant in this thesis context.

RSTP makes a logical topology on top of the physical topology. The logical topology has no loops, consequently there is only one logical path between to nodes. Every tree must have only one root bridge. The root bridge is, as the name suggests, the root of the logical tree. The root bridge is the bridge with the lowest bridge ID. The ID is calculated based on several parameters, but in this thesis, only the difference of bridge MAC address is the base for the root bridge calculation.

The logical path between a root bridge and one of the other is always the best considering a certain path cost. The cost of a path is the sum of all costs for each link in the path. The cost of a link is by default a number representing the data rate capacity of the link.

Each active port in a RSTP topology has a state and a role. The state could be either "discarding" or "forwarding". The ports in forwarding state are able to forward data, while those in discarding state do not transmit or process data exept RSTP signaling data.

The role of the port should be either "root", "alternate" or "designated". All the ports of the root bridge are designated ports. The other bridges must have a root port, which is the port that leads to the path with lowest cost to the root. If the bridge has an alternate physical way to the root bridge, the port may be set to alternate to avoid loops. The other ports that are connected to other devices are set to a designated role. A port role named "backup" also exists, but it is not relevant for this thesis.

2.2.2 Virtual LAN

In a Virtual LAN, or VLAN, end stations are connected virtually together as if they were on the same physical domain, independent of their physical location. Each VLAN forms its own logical topology, and a Bridge [23] filter traffic based on destination and VLAN. VLAN is used to segment users from each even though they may be connected to the same physical Bridge.

2.3 Mapping from IP multicast traffic to Ethernet

This section introduces the internet protocol (IP), followed by a introduction to IP multicast. The IP multicast part also explains how to map IP multicast addresses to Ethernet addresses. The section 2.4 *Current solution* presents current solutions in a cellular network. We see that the end user devices uses the internet protocol. The part of the network relevant for the problem

usese Ethernet, and is between end user and a core network with IP. Thereof, the internet protocol is relevant, and mapping of IP multicast traffic to Ethernet is relevant.

As shown in table 2.1, the internet protocol is above the Ethernet in reference to the OSI model. Internet today is built around te IP protocol, meaning that each internet-participant uses an IP-address to communicate. Data transmitted through the internet keeps the IP, but it might encounter different layer 1 and layer 2 devices during the transmission. IP version 4 (IPv4) is the most common protocol today, but the small address space is a critical limitation. Thus it was developed a new protocol to deal with the limited amount ofaddresses, IP version 6 (IPv6). This protocol enhance the functionality from IPv4 and has an address range between 0 and 2^{128}. IPv6 was developed to replace the IPv4, without a transition stage, and is not backwards compatible with IPv4. Most new devices has support for IPv6, but some older devices may not support this. Backward compatibility to IP version 4 must be supported in order to ensure connection while devices still has IPv4.

The IP layer splits data from the above layers into packets, and forwards these to the data link layer beneath. The packet format of IP version 4 and 6 is shown respectively in table 2.2 and 2.3, page 13.

A sender with a IP multicast stream sends the data with a IP multicast address as the destination in the IP packet. The source address is the normal IP unicast address of the sender. The multicast address space is from 224.0.0.0 to 239.255.255.255, also known as the Class D addresses. However, some of the addresses are reserved [1]. IP multicast creates a tree of users where the sender is at the top. Users that is not a member of a multicast-stream is not a part of the tree, and does not receive any multicast-streams. In contrast to unicast, only one stream is transmitted from the sender regardless of how many users are subscribed. This in contrast to unicast transmits that transmits n streams for n users.

IP multicast is splitted in two parts — the Internet Group Management Protocol (IGMP) and multicast routing. The latter part is developed in several directions, and there are several protocols available IGMP is the protocol which enables hosts to join a multicast group. Current version is IGMPv3 (version three) that is standardized as RFC 3376. The following description is based on this standard. A multicast group is identified by the destination address in the IP-packet. It can also be based on a combination with the source IP address, to be able to identify different streams if two or more senders use the same multicast-group as destination. IGMP-messages are originated from the end-user to the first multicast-supported router. Users join the multicast-stream by sending an IGMP join-message towards the router. There is no limit on how many multicast-streams a user can subscribe to, or how many subscribers there are to a single multicast-stream. IGMP is designed for IPv4, while a similar protocol called Multicast Listened Discovery (MLD) is standardized for IPv6. MLD version 2, standardized in RFC 3810,

[1] Internet Assigned Numbers Authority (IANA) coordinates the address-space: `http://www.iana.org/assignments/multicast-addresses/`

is quite similar to IGMP v3.

When a user terminate a subscription, it may send a IGMP leave-message. However, the router that received the leave-message, does not know how many subscribers are connected to the interface, hence it sends an Group-Specific Query and starts a time-out. If no Reports are received before the time-out, the router assumes that there no members left on the interface. IGMP messages are encapsulated in IPv4 datagrams, with an IP protocol number of 2. IGMP v3 supports the following five message types:

0x11 Membership query: are sent by IP multicast routers to query the multicast reception state of neighboring interfaces.

0x22 Version 3 Membership report: are sent by an IP system (to neighbor router) to report the current multicast reception state, or changes in the multicast reception state of their interfaces.

0x12 Version 1 Membership report: are sent by an IP system (to neighbor router) to report active membership to a multicast stream.

0x16 Version 2 Membership report: are sent by an IP system to report active membership to a multicast stream.

0x17 Version 2 Leave group: are sent by an IP system to report that it wants to leave a multicast group.

The hex-number before each message-type is the type number. Messages with other type numbers should be ignored. Specifications of the messages can be found in [3], [4] and [5].

IGMP can be illustrated with the following. IGMP creates small branches to the leaves (the subscribers) of the multicast-tree, while the multicast routing protocols makes the main branches and the stem. Building the tree is the opposite of what we might think — from a leave (a multicast receiver) toward the source. I.e. a client uses IGMP to join a multicast group.

Multicast routing is sort of the opposite of unicast routing where unicast is about where the receiver is — multicast is about the source of the packet. Routing protocol can be defined in two categories, shortly mentioned due to its relevance here. The first is dense mode protocols which floods the entire network, prunes (removes) branches of the tree with no receivers. Well suited for networks with a high density of subscribers, and not suited with scattered subscribers. The other category is sparse mode protocols which is designed for thinly populated multicast groups over a large region.

IP Multicast is not supported in a layer 2 network. This leads to problems. Use figure 2.2 as an example. If all five computers were to subscribe to different multicast-streams, all of the streams using 10 Mbit/second. The switch forwards all multicast streams on all interfaces by default, and each computer receives 40 Mbits/seconds that they do not want. A workaround

is developed for switches and bridges called IGMP-snooping (and MLD-snooping for IPv6). No standard exists hence the implementations rely on the IGMP-protocol itself. The switch needs to be layer 3 aware and read the IP-packet headers.

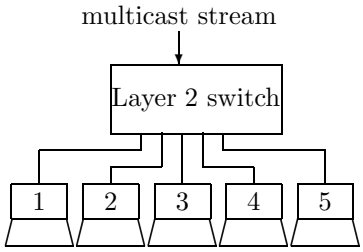


Figure 2.2: IP multicast on Ethernet with default group addressing behavior. The node with number 1 to 5 illustrate end users.

A switch can determine whether a MAC-frame that has entered on an interface is multicast or unicast based on the destination address. See the MAC-frame format in table 2.4. The 25 MSBs of the MAC-address is equal for MAC-frames containing a multicast IP-packet 0x0100.5Exx.xxxx [1, 3, 25]. The first octet, 01, is stated by the IEEE Std 802.3 and tells us that it is a "group-address" [2]. The rest of the destination address in the MAC-frame is mapped from the layer 3 multicast-IP. All multicast-addresses is class D, which means that the four MSBs is 1110 and the 28 LSBs determine a given multicast-stream. The 23 LSBs of the destination IP-address is set as the 23 LSBs of the MAC-frame destination address. This mapping ignores the five MSBs of the multicast-IP. For each mapped MAC address possible, there are $2^5 = 32$ possible IP multicast addresses – hence an end host can not be sure that the frame belongs to one of its subscribed multicast-streams before it decapsulates the MAC-frame and reads the full destination address in the IP-header. The switch should send a frame of a given multicast-stream only out interfaces where a subscriber is connected. In order to implement this, the switch must know what interfaces that has an active multicast-subscription, and this is where IGMP-snooping come in handy.

An IGMP-snooping switch must be able to "listen" to layer 3 packets and read those who contains IGMP-messages. When a switch reads one of these messages, it must also keep track of the memberships and send the correct information to the users or router. For example, based on figure 2.2, if both laptop 2 and 3 are members of the same multicast group and laptop 2 sends a message that it wants to leave the group. Because the switch knows that laptop 3 also is a member, it does not forward the message towards the router, but stops forwarding the multicast packets on the interface connected to laptop 2.

The latter example is really simple, and in a bigger Ethernet topology the layer 2 devices, like MAC bridges, may need to signal each other to keep track of the different multicast mem-

berships in an effective way. This dynamic registration and deregistration approach in Ethernet is supported by the IEEE Std 802.1ak [24] protocol, which defines Multiple MAC Registration Protocol (MMRP). MMRP is described in chapter 4.

2.4 Current solution

This section describes the current backhaul network technology and where the evolution is heading in near future. My references are [17], [21] and [12].

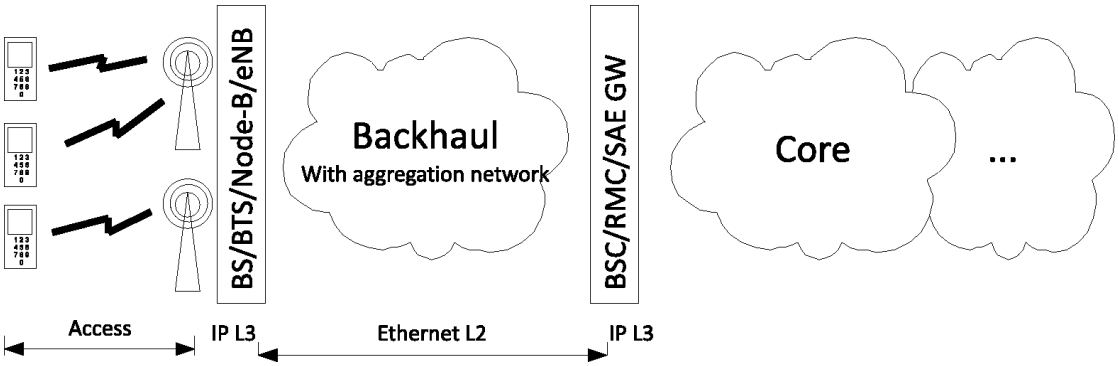


Figure 2.3: Mobile network topology IP endpoints marked.

A typical topology of current mobile networks is illustrated in figure 2.3. However, the Ethernet and IP labels at the bottom of the illustration is not typical for current networks. ATM technology has been the dominant technology for connecting the access network to the core network (in Europe) and served the cellular network well. GSM and UMTS networks has limited bandwidth demands and many backhaul networks has been designed to handle voice-centric traffic. However, the cellular networks and its terminals are changing, and this is a ongoing development. It is anticipated that terminals with improved performance and support for new services will become available, this will put pressures on the performance of the network. It is quite interesting to see the change in network traffic as new terminals are introduced, as illustrated in fig. 2.4. It is clearly seen that as the fourth generation of cellular networks are emerging, there will be even more demand for network capacity. E1 is not scalable enough [17]. Bandwidth-problem will be solved and the total cost of ownership (TCO) will be reduced by 20-30% replacing the existing technology with an IP/Ethernet solution [12]. As current backhaul network is migrating towards the topology in figure 2.3, the bottleneck is moved from the access to the backhaul network. There are two alternatives to Ethernet technology - MPLS-TE or a connection oriented Ethernet (COE). The industry has a ongoing discussion which to use in the backhaul. A comparison of the TCO between these two technologies over a five year period in [13], concludes that the COE alternative is 43% less expensive than the MPLS-TE alternative.

This could be a winning factor for the COE alternative in the long run.

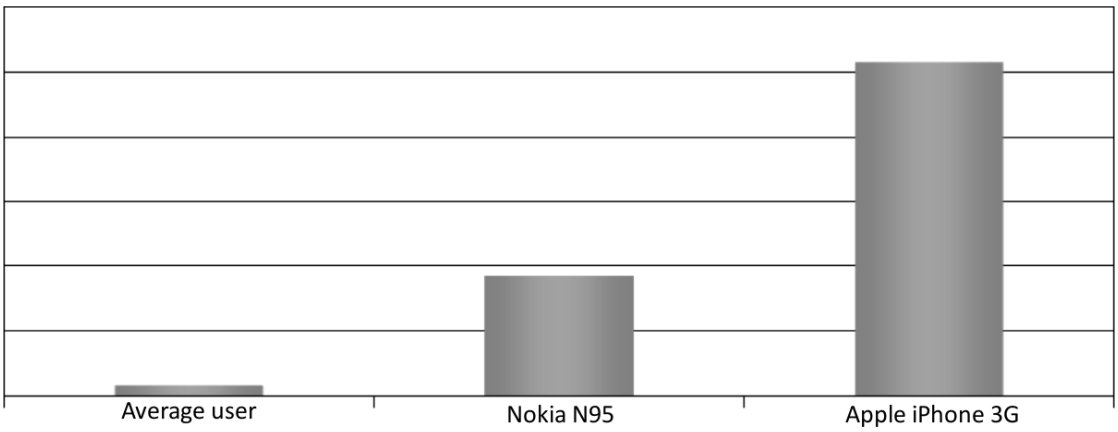


Figure 2.4: Statistics on data traffic per month generated by the average user, Nokia N95 users and iPhone 3G users in Netcom's cellular network [36].

Chapter 3

J-SIM

This chapter is divided into four parts. The first part is a introduction of J-SIM, and followed up by a section that describes the architecture. The third part concerns implementation of the architecture in Java, and finally a section that treats TCL and Java together to "glue together" elements implemented and run simulations.

3.1 Introduction

There are two main approaches on how to simulate the behavior of a system. **Time-driven simulation:** At a periodic fixed time-interval, the simulator engine checks the system state and eventually makes changes (e.g. every 5 time units). **Event-driven simulation:** The simulator engine checks the system at each event that occurs in the system.

A drawback with the time-driven approach is that there is no guarantee that the simulator will register all states of the system through the lifetime of the simulation. The most used approach is the event-drive simulation [28].Each event has a time stamp, and the events must occur in chronological order based on the time stamp. The time between each event is skipped, so the processing of events has no dead time[1] — there is always an event that is processed. In a system with a lot of events, the time consumed running the simulation may exceed the time used with time-driven simulation of the same system, but the fidelity of output from the simulator may be far better on the event-driven. The time and output fidelity only "may" be better, because, if the time-driven simulator have a period less than the smallest time between two events, the fidelity of output information turns out to be just as good, but then the processing time would increase beyond what is used by the event-driven. The same is true the other way around, where the time-driven approach has a period larger than the smallest time gap between two events. Time-driven simulation will have a faster processing time, but worse fidelity of the

[1]Dead time: here, time when the simulator does no processing.

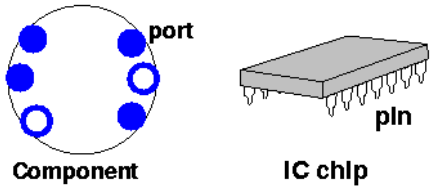


Figure 3.1: A J-SIM component illustrated to the left with the blue circles as ports, and an integrated circuit to the right [26].

output information (may have skipped some states in the system) [30].

J-SIM is a simulation environment developed by Hung-ying Tyan in his Ph.D. thesis project [20]. It is a component-based architecture built upon the Autonomous Component Architecture (ACA), also introduced in his Ph.D. thesis. ACA is implemented in Java, thereof the name "JavaSim" which later was changed to what it is known per date as, J-SIM. This was due to trademark conflicts with Oracle and Suns Java [2] [26]. The basic design idea of the J-SIM architecture is to mimic integrated circuit (IC) chips. A component in J-SIM is a software implementation of an IC. That means the ports of a J-SIM component is like the pins of an IC. J-SIM uses an extended version of a discrete-time event-driven simulation approach, called real-time process-driven simulation. Basically, it has the same features as event-driven simulation.

3.2 The autonomous component architecture

The difference between hardware design with ICs and software design with some object oriented programming (OOP) language is the component (object) binding. This is the motivation for the ACA, and Tyan describes this in [20]. The following section starts off with the background and motivation of the ACA. The architecture itself is explained in some detail at the end.

Bindings between objects in software often becomes a problem. In order to call a method from another object in OOP, we need to know the exact naming of the function and its parameters. This makes the binding to "strong", because the component that calls the functions needs to know more than necessary. The binding is so strong that it is impossible to extract a object because it is depended on other objects. Extracting an object for solely testing is impossible because of the dependence of other objects in the system. One of the original ideas when OOP was designed vanishes in the so called hyper spaghetti objects and subsystems phenomenon. This is contrary to hardware, because the ICs are clearly separated and each IC has a given interface. It can be tested and developed of its own, separate from the rest of the hardware in the system before it is implemented. This makes the design modular and easier to keep the overview of design and in debugging. Based on the issues above, Tyan proposed the autonomous component

[2] Oracle and Sun's Java: `http://java.sun.com/`, last visited 24.02.2010.

architecture, for software, that mimics IC design.

The basic entity in ACA is a component. A component may have one or several end points called ports. Each port has an incoming and an outgoing connection. When a component is designed and implemented, it defines how it reacts to incoming signals on a port, and what signals that are propagated through a port. There are no relations to other components it communicates with. The handling of signals on a port is called a contract. For example when a signal arrives at a port, it has to check the type of signal and perform some action based on evaluation. E.g. it sends a new message out on an other port, or register the value in a table. The functionality of the component and its port are defined before system integration time, while the components are bound together at the system integration time. The system are integrated by the use of the tool command language (TCL) bridged together with Java. How this works will be addressed in section 3.4.

Ports in a component can be organized into groups. Each group is identified by a group ID, and each port has it own port ID. Group ID and port ID is used to identify a specific port.

Two ports can be connected in a simplex or a duplex manner. A simplex manner is illustrated in figure 3.3*a*), and joins the output of port A with the input of port B together. In a duplex manner, illustrated in 3.3*b*), also connects the input of port A with the output of port B together.

3.2.1 Composite components and server ports

A component may be composited by several child components, as shown in figure 3.4. The composed component is a "parent" to its sub components. The sub components are often referred to as "child components". In the figure, port A of component exp_a is connected to port B. When data arrives at port A it actually arrives at port B. Port A may be viewed as, and data is going straight through port A both ways. Port A is called a shadow port, and must be established so the composite component has an port toward the rest of the network.

Several components may be connected to the same port of a component at system integration time. If they depend on a return value based on what it transmitted to the port, it is necessary

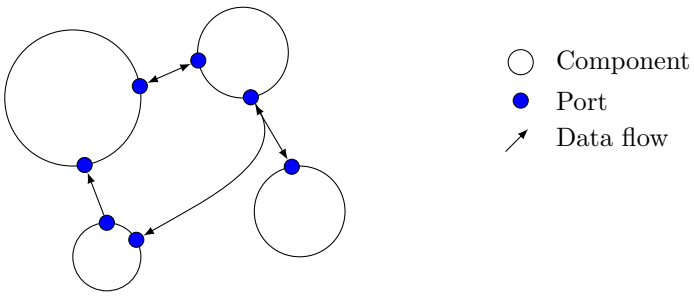


Figure 3.2:]
Components communicates through ports [26].

that the right component receives the right return value. To solve this issue, ACA defines a special type of port that is named server port. No matter how many components that are connected to the server port and request an answer, the answers are only returned to the querying component.

ACA also specifies that the runtime[3] shall handle the different execution contexts, which makes the components autonomous. To comply each component are executed with its own Java Thread. More about the runtime and component execution is presented in section 3.3.1.

3.3 Java implementation of the ACA

J-SIM is actually a Java implementation of the ACA. That means that it has used the principles of ACA to prevent the OOP-issues with component-binding addressed in 3.2 page 22. In this section, we start by describing the the J-SIM runtime in section 3.3.1 and 3.3.2, then in section 3.3.3 we proceed with an introduction of some basic J-SIM components.

3.3.1 The runtime

To handle all the incoming data in the different components, J-SIM uses a background thread manager. This manager is known as the runtime, and is implemented to comply with the ACA so that the components are autonomous. J-SIM uses Java Threads for execution context. Two Java-classes are used, WorkerThread and ACARuntime. When a component is about to transmit something on one of its ports, a WorkerThread is either created or waked up from a pool of sleeping threads to perform the handling of the data sending. Before the handling is finished, the thread notifies the ACARuntime. ACARuntime then uses the same thread to handle the incoming data on the other end. This achieves "one thread per message". WorkerThread has wrapped the Java Thread class that includes execution context information. The ACARuntime is a pool of all the WorkerThreads started. J-SIM has a limit of how many threads to be active at once. When a component wants to send something, but the maximum number of threads is reached, ACARuntime puts the request in a queue. If a thread is available, ACARuntime either starts a new WorkerThread or wakes up a WorkerThread that is in sleep state.

J-SIM has real-time process-based simulation as an extension to the runtime. The basic idea

[3]Runtime is a collection of background processes. In ACA it handles all the different execution context. In the realization of ACA, J-SIM, the runtime handles the different threads. [20]

(a) (b)

Figure 3.3: Ports connected in a simplex manner in (3.3a) and duplex in (3.3b) [26].

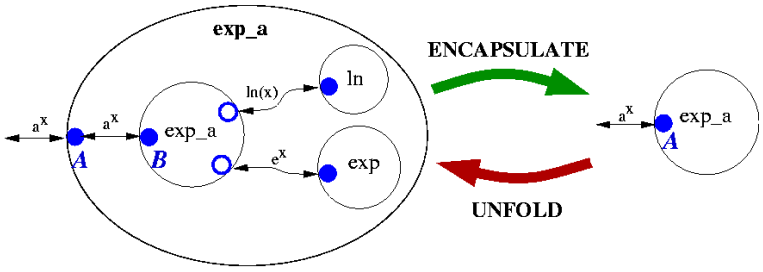


Figure 3.4: Illustration of capsulation of components [26].

isto always keep the simulation active with at least one WorkerThread. This means if there is no events in the system, J-SIM skips that time and starts processing the next event in the timeline. Therefore, the way J-SIM execute scripts is with high fidelity and still optimized. The simulation results must have the correct system time, so J-SIM keeps track of all the time skipped to calcute the system time through the simulation process.

3.3.2 Exporting information at runtime

To get diagnosis information at runtime, ACA defines a special port called infoport. Four types of information may be exported at this port, error message, garbage message, debug message and trace message. The are all of the same format:

1. Time of export (`double`).
2. Path of component/port in subject (`String`)
3. Data in subject (`Object`)
4. Detailed description (`String`)

There is also a port called event port, which export event messages. These messages has an additional parameter compared to the inforport messages, the event name (`String`). The list below has a short description of the different types of messages.

- Infoport, the following information is exported:
  - Error message - exported when a component can not handle incoming data.
  - Garbage message - exported when a component discards data.
  - Debug message - exported when the component writer would like to export debugging data.
  - Trace message - is a special debugging message and is exported for all incoming and outgoing data.
- Event port, triggered of some event.

3.3.3 Base classes

J-SIM comes packed with a range of components. It starts out with base classes like Component and Port, and has built several layers of protocols on top of it. An overview of the classes is shown in figure 3.5 at page 26. When it comes to the ACA; Component, Port and Wire are the exact counterpart of ACA as described in 3.2. The ACARuntime, WorkerTherad and Forkmanager classes includes the necessary mechanisms for the Java thread-handling in J-SIM, to realize the design of autonomous components.

The Component-class is the foundation for all components like the Module-class (figure 3.6). The Module-class is used as a foundation for my implementation due to its "timerport". The timerport receives events (of type `ACATimer`) that times out. This is needed in situation where a statemachine has timed events and timeouts. The Module-class has also a "downport" and an "up-port" which is indicates the dataflow down or up in a protocolstack.

It is mainly the two bottom layers of figure 14 that are relevant to my work. The exception is the Link- and Queue-classes which includes mechanisms to handle propagation-delay and queuing, respectively.

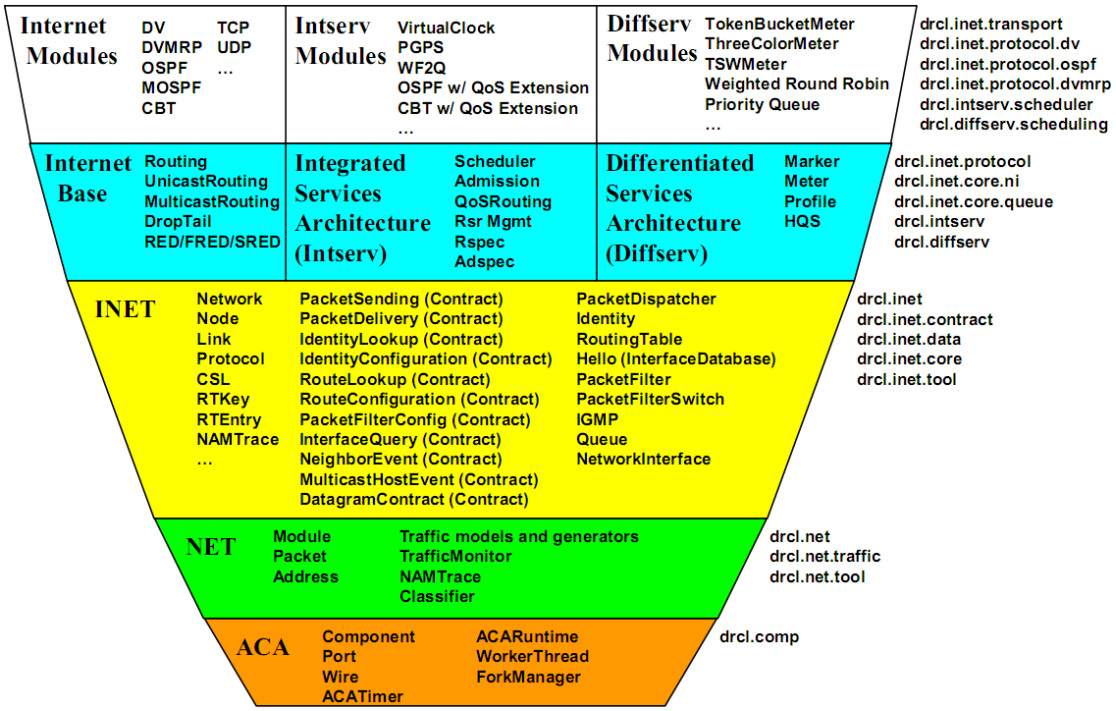


Figure 3.5: An overview of J-SIM packages with the basis at the bottom, and higher layers at the top. The corresponding Java-packages are listed to the right of the figure [20].

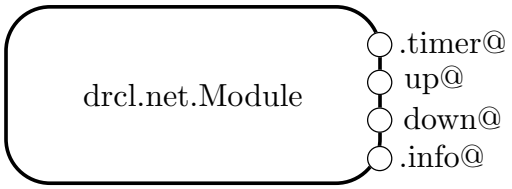


Figure 3.6: A simple overview of the Module-component. In addition to the ports that are illustrated, it contains several elements inherited from the `drcl.comp.Component` class. White circles are ports.

3.4 TCL and Java together in one system

Java is used to develop components, while TCL is used to connect components, access component methods and run the simulations. The TCL-version implemented in J-SIM is from the TCL/Java-project found at [15], which has integrated the Java-platform and TCL. Reference [15] also includes a manual of commands which comes in handy writing the scripts. In addition to the default TCL-commands, the J-SIM developers have added extra functionality in what they call the runtime virtual system (RUV) [20]. RUV will be addressed in 3.4.2 after a description of the component hierarchy in section 3.4.1.

3.4.1 How components and ports are identified

When using the terminal in J-SIM, we ?nd structure that has similarities of a UNIX-like filesystem and command set. To differ from components and port, ACA structures the component in a folder-subfolder way. A component is identified by the path, where parent component comes first and concatenated with a forward slash, "/", and then the child component. A software system forms a component hierarchy with it self as the root. The root component has only a forward slash, "/", as path. Example below is based on figure 3.7. The system name is "simulationTest" and the composed component is named "parent". The path for "parent" is:

```
/simulationTest/parent/
```

The component named "parent" has two sub components named "childA" and "childB". Their paths are respectively

```
/simulationTest/parent/childA/
/simulationTest/parent/childB/
```

Ports are, as mentioned, categorized in different groups inside of a component. The default group is `null`. Each port is identified by the path to its component concatenated with a "/", the port name, "@" and the group name. Thus, the port named "port1@group1" at the component "childA" is identifieed by the following path:

```
/simulationTest/parent/childA/
```

The port named “out@transmit” of the component named “parent” is identified by the following path:

```
/simulationTest/parent/out@transmit
```

While in the terminal, use `ls` to list the available components in your path, `cd [component_name]` to enter the component and `cat` for more information about the current path.

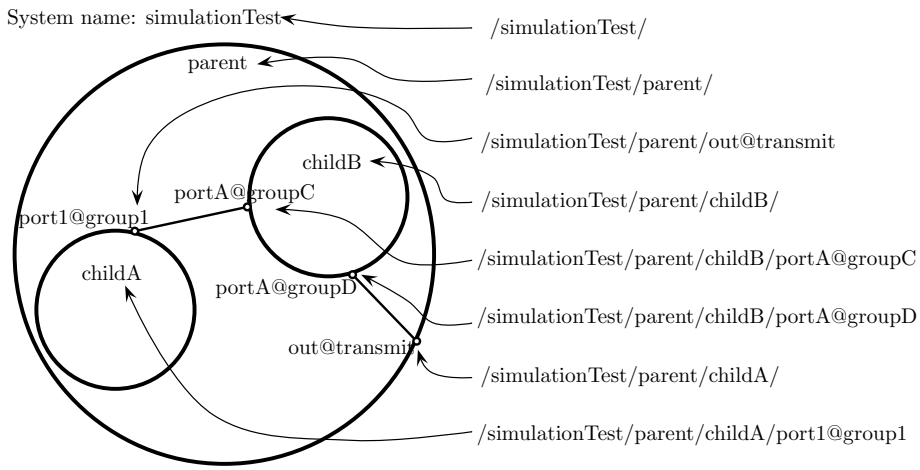


Figure 3.7: An example of a system with components. The system consists of a composed component named *parent* and two child components named *childA* and *childB*. Each component has one or two ports. The right part of the figure lists the path identifying each component and port in the system. Note that component *childB* has two port with the same name, but they are still unique due to different port group names.

3.4.2 The runtime virtual system

The J-SIM user interface is a terminal window. It is possible to execute TCL-files, and write TCL/RUV commands right into the terminal. After a simulationscript is processed, the RUV commands are useful to gather information about the state of the system. If it is desired to continue the simulation, it is possible by calling a method of the simulator.

If the system has many components, it is awkward to use TCL / Java. The reason is that TCL must save a TCL variable for all the components. Alternatively use methods like `getParent()` and `getComponent()` (for the child components). To make the scripting easier, Tyan developed a reference system which is called the runtime virtual system RUV. Based on component-hierarchy described in 3.4.1, one can easily create a reference to a Java Object. In addition to reference

!	Converting the path to the reference of the Java-object.
cat	Print the internal state (the info() method of the Java-object).
cd	Change the current working directory
connect	Connect to components/ports.
cp	Copy the component or port.
disconnect	Disconnecting components or ports.
exit	Closes the terminal.
ls	List the child components and ports in the current directory.
mkdir	Create a component or port.
mv	Move or rename a component or port.
pwd	Print the current working directory.
rm	Removes a component or port.

Table 3.1: Some of the RUV commands [20]. To list all commands, type `man` and hit enter in the terminal.

management, RUV has commands that we know from UNIX systems. See Table 3.1 for a brief overview. Reference handling is based, as mentioned on the component hierarchy. Assume the following path to a port with the name "portId" in group "groupId":

```
/parentComponent/childComponent/portId@groupId
```

To access a method such as `toString()`, in the Java-object and print the return value, type:

```
puts [[! /parentComponent/childeComponent/portId@groupId] toString]
```

While in the terminal, use ls to list the available components in your path, `cd [component_name ]` to enter the component and cat for more information about the current path. Some of the RUV-commands are listed i table 3.1.

3.5 Considerations

Simulation is a practical tool for analyzing systems. Each element[4] added in a scenario, a real life study would require more configuring and more resources when it comes to equipment, time and money. The cost of adding another element in a simulation tool, is just a few lines of scripting. You are able to add only the elements you want, contrary to real life where you need include whatever elements that the devices have. A real scenario with more then a few devices will soon require demanding time-resources to configure parameters on every device. Additional time is

[4]Element: here, e.g. a protocol/standard or some functionality.

required to tweak the performance, while in a simulation tool you only need to change some variables in a script.

J-SIM has some drawbacks not mentioned yet the most severe is the lack of an active community. That causes no bug-updates, no evolution of the environment and less resources helping you out when facing an issue. An other point worth mentioning is the lack of a good GUI, which would simplify the set-up of larger network-topologies and collect and analyze output.

Chapter 4

Multiple MAC Registration Protocol

Multiple MAC registration protocol serves a dynamic registration and deregistration scheme for individual and group MAC addresses. This chapter presents some of the architecture of MMRP. It serves as a base for the abstraction and implementation described in the next chapters.

The frames transmitted in Ethernet uses MAC addresses as destination and source addresses as described in [9]. The addresses may be either an individual address, or group address. An individual address points to a unique device or port in the network, while group addresses may have several end destinations. The MAC-group address is defined in [9], and it is identified by the least significant bit being 1 in the address, while the least significant bit of an individual MAC address is 0.

A Ethernet bridge mechanism for forwarding frames is based on destination MAC address. Depending on the type of filtering service the bridge uses, there are different ways to handle frames addressed to a group addresses. Two filtering services are listed in the standard – *basic filtering service* and *extended filtering service*. If a bridge uses basic filtering service, it will relay frames with a group address as destination on all ports in forwarding state, except the port the frame entered. An exception is if it has static entries in the filtering database, telling otherwise. On the other hand, extended filtering service will be able to make dynamical entries in the filtering database with the group addresses. Upon reception of a frame with destination address that is a group address, the bridge could send out on the ports where there are some who would like to receive frames addressed to this group address. The entries are saved in the filtering database by some signaling protocol.

IEEE Std 802.1Q [23] specifies the architecture and functional requirements of an Ethernet VLAN bridge. In an amendment published in [24], IEEE defined two additional protocols called Multiple Registration Protocol (MRP) and Multiple MAC Registration Protocol (MMRP). The

latter protocol is responsible for registering the group addresses in the filtering database if the bridge supports extended a filtering service. A correction to the two protocols was published in 2008 [11]. The corrections were minor (in terms of bit) adjustments to the packet format used by MRP. However it is important corrections that enables parsing of the PDU bit by bit. The reason is that an octet was added to the MRP signalling frame (MRPDU). The octet identifies the data size of the PDU payload.

4.1 Group management in Layer 2

A bridged network has a physical topology based on the wired/wireless connections between end users and bridges. There may be more than one path between nodes in the network, which is good for several reasons. Alternative links makes the network reliable as there are redundant connections. In addition, alternate links enable load balancing (by using e.g. MSTP described in [23]). However, redundant links demands more administration and costs extra money for equipment and configuration/maintenance. A problem with loops in a layer two network is the way bridges propagates traffic with unknown destination. Reference [23] states that if the given destination address of a Ethernet frame [10] is not registered on any port in the filtering database, it should be transmitted on each other than the entered port. Needless to say, that frame could cause a duplicate storm through the network that will not end. In order to solve this problem, bridges exchange information based on an algorithm. By this information, they create a logical topology upon the physical. This topology is called the active topology and the protocol used to create this topology could be STP, RSTP or MSTP ([22] and [23]). Figure 4.1 gives an overview of the structure of a bridge.

The multiple MAC registration protocol (MMRP) introduces a way for participants in the network to dynamically register and deregister attributes. Dynamically registration means that the signaling message (e.g. a registration) from a participant is propagated throughout the network so that each other participant receives (and registers) the given message. Without the dynamical

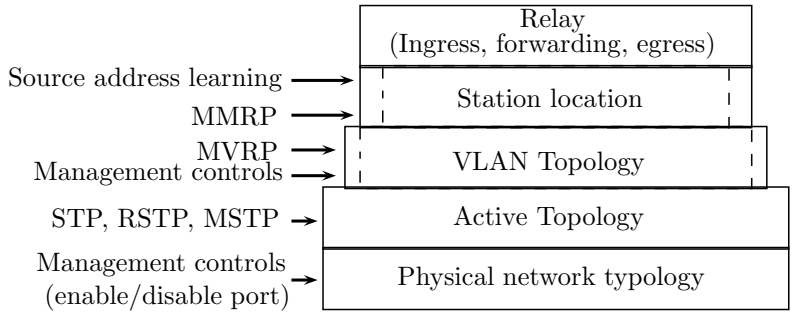


Figure 4.1: Bridge overview [24] as described in 4.1.1.

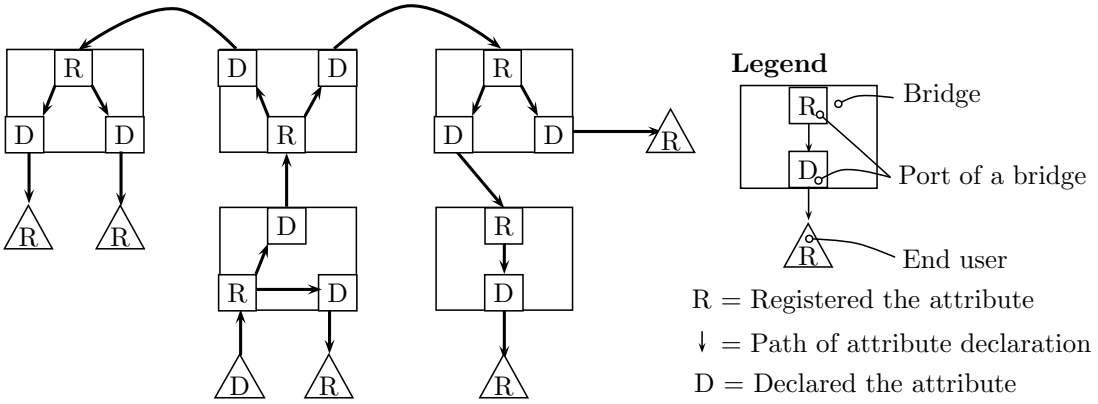


Figure 4.2: Attribute value propagation, in IEEE 802.1ak MRP, from one station [24].

registration mechanism, statical entries in needs to be added to all affected filtering databases to create a multicast topology. Multiple MAC Registration Protocol (MMRP) attributes are either an individual MAC address, a group MAC address or group service requirement information. This thesis limits the use of attributes in MMRP to group MAC addresses.

A practical example of MMRP is a network with video streams destined for different MAC group addresses. In order to avoid congestion in the network with widespread propagation of all the video streams, each participant that wants to receive a video stream, needs to subscribe to the stream by sending a registration through the network. See figure 4.2. The end station marked with "D" declares the MAC group address of the desired video stream. The declaration is propagated through the network by the help of MMRP, this is illustrated with arrows on the figure. As can be seen, the ports where the attribute enters the bridge, gets registered, while the outgoing port is gets declared. That feature gives the network the necessary information to send the video stream back to the subscriber — every bridge in the network now propagates the video stream on the ports which the attribute is registered. A more technical description is given in the next sections.

MMRP allows participants to register or withdraw attributes with other participants in a bridged LAN. A *participant* of a MMRP application is a port on a Bridge or end user supporting a given MRP application. It is maximum one participant per application per VLAN per Port combination on a Bridge, and one participant per application per end user. When a participant makes a *declaration* of an *attribute*, the declaration (or withdrawal) results in a *registration* with other MMRP participants. The declaration is fetched by an "Applicant State Machine", and triggers signaling of MRPDUs in the network towards other participants. Upon receiving of a declaration MRPDU, a "Registrar State Machine" registers the attribute, and eventually declares the attribute on the other participants of the Bridge. The attribute is not removed until all participants of the connected LAN withdraw the declaration. Figure 4.2 shows how

an attribute from a single station propagates through a network. Notice that the attribute is registered on the incoming port and declared on the outgoing port. Going the opposite way of figure 4.2 produces a "reachability three". If a multicast stream was present in the network, MMRP would be a good way of managing the streams to subscribers. End users that wants to subscribe may declare an attribute saying they are apart of the multicast group. MMRP gives end users the opportunity of tracking those attributes that have subscribers. A content provider (server) has information of which attributes that are declared in the network. Therefor, the server could choose to send or not send the stream out on the network. By stop sending a data stream when there are no subscribers, saves network resources. The latter principle is called "source pruning", meaning that the data is pruned at the source while there are no subscribers.

4.1.1 MRP architecture

The following section gives a more detailed description of the MMRP architecture. FIgure 4.3 illustrates the architecture.

In a layered protocol design, MMRP is situated above the logical topologies of RSTP and VLAN-configurations as illustrated in Figure 4.1. Its responsibilities are mainly to register the MAC addresses in the filtering database. MMRP establishes a new logical topology upon of the VLAN-topology, for each declared MAC address. The foundation is, as mentioned, the topology defined by the RSTP and the VLAN-configuration, as shown in figure 4.1.

MMRP has different components that will be presented in the following text. The presentation will start at the bottom of figure 4.3.

Port Every Bridge port has an instance of a Periodic State Machine. As illustrated in figure 4.3, a port may have several MMRP participants – at most one per port and VLAN combination.

Periodic State Machine Associated with each port, there is a state machine. The machine generates timer events every second. These events triggers resending of join messages for all applicant state machines related to the current port. This is to ensure that all registrations are propagated through the network in case of packet loss.

Participant A single participant is composed of a variable number of sub-components. The following state machines must be included: The LeaveAll State Machine, MRP Attribute Declaration (MAD) component and a MMRP-component.

LeaveAll state machine The LeaveAll state machine ensures that there are no unwanted permanent registrations in the network. These situations may occur when a part of the network has ceased to operate. Every 10 seconds, the leaveall timer will expire triggering a leaveall event in the applicant state machine.

MMRP component The MMRP-component handles the MMRP specific operations. These operations are for example the handling of different types of attributes or handle registrations request from other protocols.

MRP Attribute Declaration (MAD) MRP Attribute Declaration (MAD) handles each attribute at the given participant. There may be several attributes registered at a single MAD component as illustrated in figure 4.3. Each attribute has its own Registrar State Machine, and its own Applicant State Machine. The two state machines handles the state for each attribute at the participant.

Registrar state machine The registrar remember declarations done by other participants. It does not trigger sending of MRPDUs. The three states of the registrar state machine are:

- IN, the attribute is registered at this participant.
- LV, the attribute is leaving (being removed) from this participant.
- MT, the attribute is not registered on this participant (empty).

Applicant state machine Quote from the standard [24].

The job of the Applicant is twofold:

1. Ensure that this participant's declarations is correctly registered by other participant's registrars.
2. Prompt other participants to re-register after one withdraws a declaration.

End quote. The last point is completed by sending MAD primitives to participants on the same MAP context (see next paragraph about MAP), and send MRPDUs to the connected LAN.

MRP Attribute Porpagation (MAP) Signaling between MMRP participants on a bridge is realized with MRP attribute propagation (MAP). Participants do communicate among each other if they are on the same MAP context. A MAP context are Participants that are on the same VLAN. The signaling is based on MAD primitives, described in the following part:

The MMRP component has the information about the attribute values, registration, and semantics. It requests MAD to make or withdraw attributes by means of the primitives:

MAD_Join.request(attribute_type, attribute_value, new)
MAD_Leave.request(attribute_type, attribute_value)

The MAD handles reception of MRP messages and generates MRP messages for transmission to

other participants. It uses the two following primitives to notify the application component of a change in the attribute registration:

MAD_Join.indication(attribute_type, attribute_value, new)
MAD_Leave.indication(attribute_type, attribute_value)

In all four primitives, attribute_type specifies the type of the attribute declaration, attribute_value specifies the instance of the attribute, and new states if it is an explicit new declaration. The primitives are defined by MRP, but the new parameter is not used by MMRP. There are two possible values of attribute_type in MMRP:

The Service Requirement Vector Attribute Type Primitives with this attribute_type contains information about Group service requirements for the participant, and the attribute_value gives either "Forward All Groups" or "Forward Unregistered Groups". An example of usage for these requirements are network monitors that will listen to all traffic — this require the "Forward All Groups".

The MAC Vector Attribute Type The values of MAC Vector Attribute Types are MAC addresses, and depending on type of primitive, one wants to join or leave the group identified by the MAC address.

The architecture is illustrated in figure 4.3, and as shown the MRP Attribute Propagation (MAP) propagates information between the per-port and VLAN participants of a Bridge. If a port has seen a registration, MAP makes sure that the registration is propagated out all other ports. On the contrary when all ports has withdrawn an attribute it propagates the deregistration on the network. The figure also illustrates that the MRP protocol is located "above" the LLC layer.

To insure interoperability between MRP participants, each MRP application must use a MAC group address as destination address. MMRPs application address is defined in table 10-1 in reference [24] as 01-80-C2-00-00-20. Transmission and reception of MRPDUs between participants shall use LLC procedures. In order to identify which application the PDU belongs to, each application has its own EtherType number. The EtherType number of MMRP is according to [24, table 10-2]: "88-F6".

Topology change

If the underlying active topology of e.g. RSTP, MMRP, changes, MMRP must also reconfigure its subtrees for each attribute affected by the change. MMRP has two events that are triggered by topology change. If the active topology is based on one of the STP protocols mentioned above, the following is valid for MMRP: A "Flush!" event is deemed to have occurred if the Port

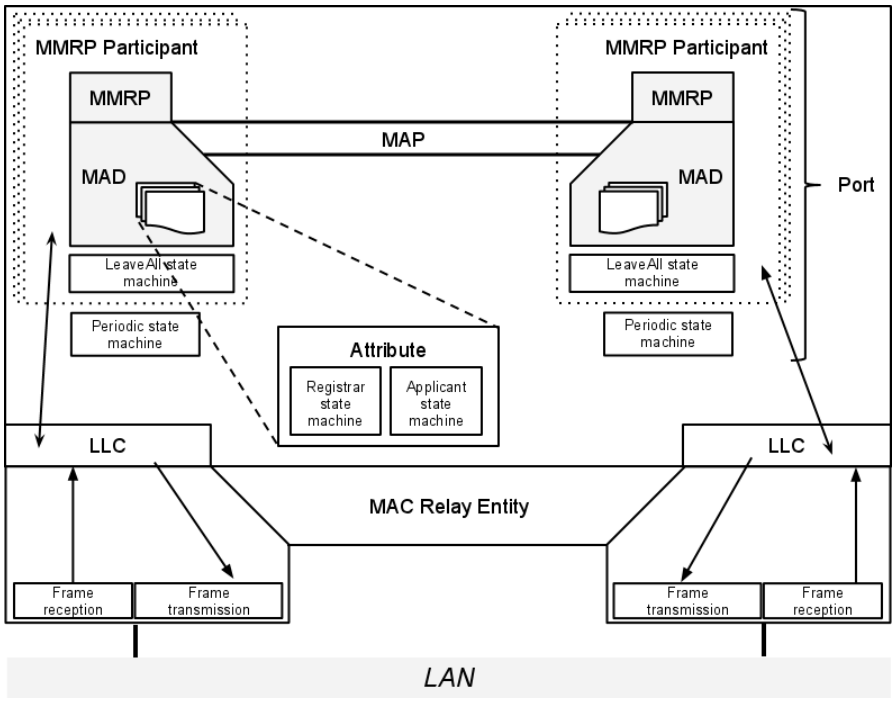


Figure 4.3: MRP architecture with a two-port Bridge. [24]

Role changes from either Root Port or Alternate Port to Designated Port. A "Re-declare!" event happens if a Port Role changes from Designated Port to either Root Port or Alternate Port.

MRP Protocol Data Units, MRPDU

MRP Protocol Data Units (MPRDU) are messages sent between MRP participants on different bridges. They have a special format in order to enable participants to recognize and parse the incoming messages, and generate the outgoing messages. Each MRPDU contains an the identifier of the MRP application that generated it. If a Bridge or end user does not support the listed MRP application, the MRPDU shall be forwarded on all ports that are in forwarding state, except from its arriving port.

A MRPDU can contain several MRP messages, which itself can contain one or more MRP events. The receiving MRP participant shall read the messages in the order they were sent.

As seen in figure 4.1 that contains the major components of an MRPDU, the first data field is *ProtocolVersion*. Supporting new enhancements of the application and making it possible to have different handling of different version, this is important. MMRP version as defined by reference [24] has hexadecimal value 0.

Upon parsing of an incomplete or corrupt MRPDU, the participant shall discard the whole

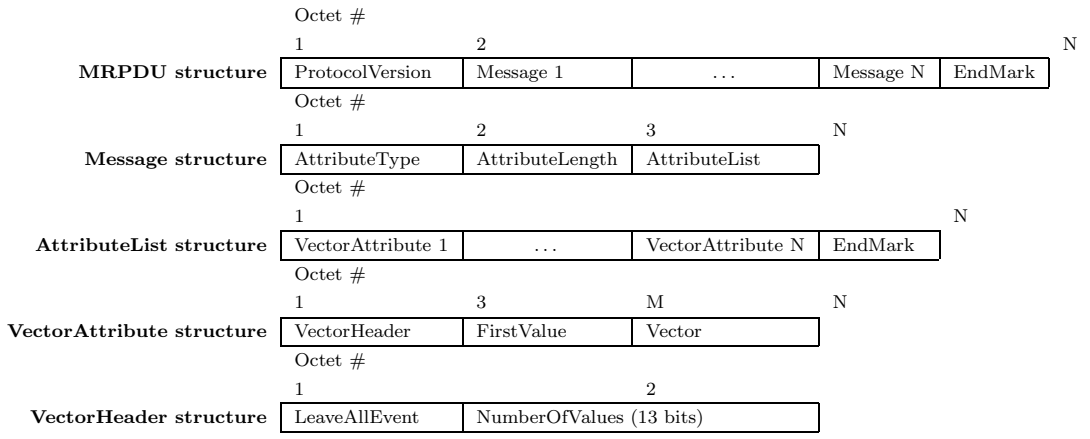


Table 4.1: Format of the major components of an MRPDU [11]

MRPDU and terminate the processing.

4.2 Type of MRP-implementation

There are four types of implementations of MRP:

Full participant This participant has all the state machines and the full set of states that are listed in [24, table 10-3].

Point-to-point subset of the full participant This type of participant includes all the state machines, similar to the full participant, but may rule out some of the states from [24, table 10-3]. This results in a applicant state table as shown in table 4.2. The bridge parameter `operPointToPointMAC` decides whether to use the full participant type, or the subset. If the parameter value is `TRUE`, the subset may be used, but when the value is `FALSE` the full participant should be used.

Applicant-Only participant Implements only the Applicant state machine with some states omitted, and a periodic State machine. These options are used by e.g. end-user that only needs the possibility to subscribe to traffic from group address, but are not itself a source of such traffic. In that case the participant do not need to know whether there are other user subscribing to the same group address(es).

Simple-Applicant participant A subset of the Applicant-Only participant. Implements the same as the one above, but has removed even more states from the applicant state machine. The difference between the Applicant-Only participant and the simple participants is the

same as the difference between the full participant and the point-to-point subset of the full participant.

		STATE									
		VO	**VP[6]**	**AA[6]**	**QA**	**LA[6]**	**AO**	**QO**	**AP[3,6]**	**QP**	**LO[6]**
EVENT	**Begin!**	–	VO	VO	VO	VO	VO	VO	VO	VO	VO
	Join!	VP	–	–	–	AA	AP	QP	–	–	VP
	Lv!	–	VO	LA	LA	–	–	–	AO	QO	–
	rJoinIn!	AO	AP	QA	–	–	QO	–	QP	–	–
	rIn!	–	–	QA	–	–	–	–	–	–	–
	rJoinMt! —— **rMt!**	–	–	–	AA	–	–	AO	–	AP	VO
	rLv! —— **rLA!** —— **Re-declare!**	LO	–	VP	VP	–	LO	LO	VP	VP	–
	periodic!	–	–	–	AA	–	–	–	–	AP	–
	tx!	[s] –	sJ AA	sJ QA	[sJ] –	sL VO	[s] –	[s] –	sJ QA	[s] –	s VO
	txLA!	[s] LO	s AA	sJ QA	sJ –	[s] LO	[s] LO	[s] LO	sJ QA	sJ QA	[s] –
	txLAF!	LO	VP	VP	VP	LO	LO	LO	VP	VP	–

Table 4.2: The applicant state machine for the Multiple MAC Registration Protocol (MMRP). See table 10.3 of [24]

		STATE				
		VO	**VP[6]**	**AA[6]**	**QA**	**LA[6]**
EVENT	**Begin!**	–	VO	VO	VO	VO
	Join!	VP	–	–	–	AA
	Lv!	–	VO	LA	LA	–
	rJoinIn!	AO	AP	QA	–	–
	rIn!	–	–	QA	–	–
	rJoinMt! —— **rMt!**	–	–	–	AA	–
	rLv! —— **rLA!** —— **Re-declare!**	VO	–	VP	VP	–
	periodic!	–	–	–	AA	–
	tx!	[s]	sJ	sJ	[sJ]	sL

Table 4.3: The simple-applicant state machine for the Multiple MAC Registration Protocol (MMRP). This is used at the end-user implementation of MMRP.

Chapter 5

Implementation

This chapter presents the first phases of the simulation study process described in chapter 1 *Introduction*. Implementation required both Java programming and TCL scripting. Finally the implementation ended at approximately 5300 lines of Java code, and 700 lines of TCL code. This chapter is broken down in three parts. The first is section 5.1 that presents an introduction to the chapter and what approach that is used for the implementation. Next, section 5.2 *Making the model* presents the abstraction of the system and data defining real life scenarios. The data defining real life scenarios are collected by contacting several ISP and content service providers, and from an article about usage behavior patterns in an IP television network. Finally, section 5.3 *Model translation*, presents the transition from the abstraction towards the complete implementation.

5.1 Introduction

The implementation requires functionality from several protocols that must be understood and implemented. This includes MMRP [24], RSTP [22], and bridge functionality and operations given in [10] and [23]. It is complex to understand the connection between variables and operations that a bridge consists of. The relation between standards may not be self-evident. The references may have several hundred pages, making study complex in itself. This has resulted in changes of the implementation as "new discoveries" of functionality or meaning of variables occurred from time to other.

There were several approaches of implementation considered in the beginning. They were not only considered as a solution at its own, but also as combination of some of them:

Multiple components As described in chapter 3, J-SIM's architecture is built upon autonomous components. That means that each component runs at its own. By designing a system with many components, each part of the implementation is reusable. The downside is that

the more components (a higher granularity of components), the more complex scripts are needed to glue these together, and more overhead is generated when sending data between components.

A single component A single component will avoid overhead between the components, and a single interface toward the system. The scripting would become easier, but the reuse ability would limit itself to the whole functionality of the component.

Several threads If a single component uses several threads, it is possible to make it receive packets at the same time as it handles other protocol events. This implementation design makes the composition of the functionalities complex and hard to control the concurrent actions toward variables in the component. A risk is two methods interfere with each others task by changing values affecting the other method's processing.

Due to multiple protocols, the choice became to use multiple components. Several threads for a component was avoided due to the complex handling of concurrent actions. Even though the design was based on multiple components, MMRP as a whole was based on one component. RSTP, which was based on a previous implementation, was also one component. The operations in a bridge that had an interface to more than one component, was chosen to be component of its own. The latter includes LLC, an interface component (a bridge port) and MAC relay entity. A more complete evaluation of the implementation design will be presented in the next parts of this sections.

5.1.1 Tools used

A most primitive tool, pen and paper, was actively used in the start trying to give an overview of the system. The implementation as J-SIM components could be realized in a simple text editor like Notepad in Windows or gEdit in Ubuntu, but there are also well developed Integrated Development Environments (IDE) available on the internet — for free[1]. My experience is limited to two IDEs, Eclipse [2] and Netbeans [3]. While Notepad is nothing more than an user interface where one can write the code in plain text and save it to a file, an IDE can do so much more. Some of the features of an IDE are connecting multiple files of a software project in a context, syntax highlighting, advanced search options, live debugging, built-in version-management clients and syntax error detection. For more features, enter the homesites of the IDEs mentioned above. The choice of IDE for the current task was Netbeans because of former experience with it.

Department of Informatics at the University of Oslo hosts a version-management service[4] based on Subversion[5]. This service enables one to continuously make a backup of its implemen-

[1] Free is often in terms of academic or personal use. Read the terms.
[2] Eclipse: `http://www.eclipse.org/`. Last visited 03.03.2010.
[3] Netbeans: `http://netbeans.org/`. Last visited 03.03.2010.
[4] Webservice: `https://wwws.ifi.uio.no/`. Last visited 04.03.2010
[5] Apache Subversion: `http://subversion.apache.org/`. Last visited 04.03.2010.

tation with a single click in Netbeans. In addition to backup, it provides history of the changes one have made to the code. That comes in handy if a part is deleted, saved, and later regrets the change.

5.2 Making the model

The work flow of a simulation study was described in 1.2.4, and was based on figure 1.1 on page 6. The first part is to figure out the problem introduced in chapter 1 *Introduction*. The problem provides a base for further work and it defines what to emphasize. The abstraction model must contain the elements from the real life model that make it possible to reach the objectives. Elements that does not have an impact should be omitted so that the model will be less complex. The more complex the model, the more difficult to debug, validate and analyze. If one has a thorough understanding of the system, it is easier to make an abstraction. During the modeling work, one can go on to testing to see if the functions implemented works as desired.

Data collection is important to prepare the simulation software to provide the necessary information. In J-SIM, there are two points to note. The information you want to get hold of for the analysis must be available in the Java class under certain commands and output messages. The configuration of your network is written in the TCL scripts, so the second point is that the simulation scripts must adapt what was found regarding scenarios and real life inputs.

Model translation is also divided in two, as mentioned in the previous paragraph. You have to transfer the model to Java, while some of the information should be used setting up the simulation in the TCL scripts.

One will go through the model several times with the validation and verification. In the beginning, empty shell of implementation are executed and validated. When adding more functionality, it is natural to test before too much is added. This is because it is easier to locate errors when there is less code to troubleshoot. It is important that one in advance of testing makes up a sense of what the outcome will be. By having an opinion on beforehand, one may avoid accepting the result even though it is not correct. After the validation and the implementation seems to work, the next step is to do the simulations that produces data to be analyzed.

This section is divided in two part, first a part about the abstraction called “Model of conceptualization”. The last part is about model translation, which means implementing the abstraction in the current programming language for the simulation environment.

5.2.1 Model of conceptualization

The challenge is to make a good enough abstraction to include only what is needed and discard what has no effect on the given objectives, and at the same time keeping it at a level of complexity that makes it possible to verify the model.

The main components of the systems is the MMRP, the RSTP for logical topology, and some bridge structure like filtering database. The next parts consider each of these main components:

Abstraction of MMRP

MMRP is described in chapter 4, and is based on the standard from IEEE found in references [24, 23, 11]. The standard presents MRP first, as a base for two protocols: MMRP and MVRP. The variable called "new" shows up in the description of MRP, both as a parameter in the MAD attribute propagation (MAP) primitives and as events in the state machines. This variable is not used in MMRP, which means that the Applicant state machine can be simplified from what is shown in [24, table 10-3].

Reference [23] defines a parameter called `operPointToPointMAC` for each port of a bridge. If its value is, quote, "*TRUE, the service is used as if it provides connectivity to at most one other system; if FALSE, the service is used as if it can provide connectivity to a number of systems*" [23, 6.4.3]. That means if the port is directly connected to one other port of another bridge, the value is `TRUE`. If the port is connected to e.g. a hub, and it may receive messages from several other ports and bridges, it is false. This thesis limits its systems to point-to-point links, and give grounds for that the value of `operPointToPointMAC` always is `TRUE`. When this parameter is `TRUE`, the participants applicant state machine may be a subset of the full participant as described in section 4.2 on page 38. The simplification based on the `operPointToPointMAC` and the "new" variable that is mentioned in the former paragraph, is shown in table 4.2 on page 40. As stated in [24, p34], "*This standard permits simple point-to-point subset implementations of MRP, but these will successfully operate on shared media albeit at reduced efficiency.*", resulting that the protocol will function correctly with hubs even though all state machines are based on the subsets shown in the table 4.2.

End nodes may also have MMRP implemented in order to make registration, or as a server, take advantage of source pruning. These nodes do not need the entire implementation of MMRP, and are able to use a simplified implementation. The source pruning and simplifications is explained in chapter 4 *Multiple MAC Registration Protocol*. If the **operPointToPointMAC** is `TRUE`, it uses the simple applicant state machines for these nodes as shown in table 4.3 on page 40.

MMRP is responsible, in addition to the MAC relay entity, for updating the filtering database of the bridge with MAC addresses and port maps. This database has a record of each known mapping between MAC-address and port, and uses it to decide forwarding port for a frame. MMRP will perform queries in the filtering database both to add, change and look at entries.

The two different types of attributes MMRP uses are the service requirement vector attribute type and the MAC vector attribute type. The service requirement vector set whether an participant wants to forward all groups or forward all unregistered groups. No support for the service requirement vector is added in the model, because it is not relevant for the given objectives for

this study. The MAC vector are the center piece of the model, and is included.

The MAP context described in section4.1.1, and each MAP context has its own identifier according to the standard. This identifier was excluded from the abstraction, because replaced by the VLAN and port identifier in the Java implementation.

Individual MAC address registration which is supported by MMRP following the standard is not included since the interesting subject is group traffic and performance aspects as convergence is not different for individual addresses and group addresses.

[24, Clause 12] introduces bridge administration, but implementing a specific interface with these methods are not important in a simulation aspect due to the nature of Java and method made public available.

The active topology

MMRP relies on an active topology — a logical topology without loops. It may be based upon protocols like RSTP or MSTP. Nera Networks had a finished implementation of RSTP for J-SIM at hand, which is used in this study. However, after some testing, it turns out that the RSTP is missing edge detection and has a bug when it comes to restoration handling. The missing edge detection means that it does not handle end-nodes (here a PC or server) without RSTP implementation (which end nodes normally do not use), so the solution was end nodes also needs to use RSTP component. That do no affect the objectives of this thesis and thereby the solution was the best compromise.

RSTP is short for Rapid Spanning Tree Protocol and is an improved version of the old STP. It is called rapid due to its fast convergence time compared to STP. Reference [32] presents results with convergence times around 10 milliseconds, and reference [18] states convergences times beneath 50 milliseconds. The former references do not take hardware delay into account, and the latter reference do not include hardware or BPDU processing delays in their simulation. Both conclude with convergence times in the order of centiseconds. The RSTP implementation for J-SIM, uses about 15 seconds to reconfigure its topology in a link brake, between the root bridge and a bridge, in a ring topology with ten bridges.

Other bridge functionality

Each bridge may have several ports and a port is a component so that each port may send and receive independent of each other. A sort of packet-filter is also needed, and the choice fell on a LLC-component which send incoming frames toward the right component above LLC based on parameters (EtherType and destination MAC address) given in the standards [23, 24]. The components that are to be placed above LLC in the layer stack are MMRP and RSTP.

The center piece of a bridge is the MAC relay entity which is responsible for learning, filtering database and forwarding. MMRP need the filtering database, and learning may come in

handy when simulation large networks topologies. VLAN-support is not fully added, but is near complete in the MMRP-implementation. It is the bridge implementation itself that lacks the VLAN support, together with a the MAD attribute propagation method of MMRP.

5.2.2 Data collection

Collection of data to form scenarios for simulation should ideally come from the real world. That way, the output data becomes realistic. There are several ways to get this kind of data:

Real system You may base your data from tests of real life systems, and get configuration parameters and features of a real life systems. This gives you trust able parameters of e.g. buffer size, when designing a simulation scenario. On the other hand, it is not within the scope of this study to collect information about user behavior in large systems.

Contacting the industry By contacting companies, one may collect useful data of user behavior, and real life infrastructure topologies. This is truly relevant information, but it may be hard to retrieve such information due to business and marked secrets.

Research articles For some topics there exists a lot of research papers on sites like ACM, Springer and IEEE [6]. Some universities has subscriptions at these libraries, and one may as a student download articles for free. It is not always trivial finding relevant articles, but some research fields are well documented and lots of useful data are available.

Statistical bureau Statistics Norway [34] and TNS Gallup [35] provides statistics and trends of a wide variety of topics.

The simulation part of this thesis requires data that specifies infrastructure details: The number of bridges between the server, end user and number of users connected to a single server and frequency of topology change. Link capacity and data rate may also be tweaked in order to simulate a backbone based on radio relay systems. The type of information described is available from companies hosting networks to day, like ISPs or TelCos. The work of receiving this data was done by contacting companies like Canal Digital, Get, BKK, Netcom and Lyse. These companies represents a wide range of different broadcasting technologies including cable-TV, IP-TV, cellular networks and satellite TV networks. Even though the scope of this thesis is limited to a specific protocol on Ethernet, the companies could bring useful information about topologies and user behaviour. The answers they came with was from little to nothing. But little is better than nothing, and the following information was provided (without a mapping to the company giving the information):

- Their core network consist of a fiber ring.

[6] ACM: `http://www.acm.org/`, Springer link: `http://www.springerlink.com/` and IEEE: `http://ieeexplore.ieee.org`. All visited last 07.03.2010

- From a content server to the end user it is normal with five or more nodes.
- In the areas with the greatest density of subscribers, there may be ten thousand customers per content server.
- The bit rate of standard definition (SD) MPEG4 streams are approximately 6 Mbps and for high definition (HD) approximately 10 Mbps.

An other sort of data of interest is user behavior (how subscribers use the services provided by the server) and user trends. This information may be given by the ISPs, a cable-TV company, some other content/service provider, research papers or statistical bureaus. The content-/service providers did not participate with any information. Statistics Norway [34] has some statistics about user pattern of television habits in [29]. The latter reference provides information about how many people is watching television through the different hours of a day in Norway. This could be used to calculate the amount of users simultaneously watching television at e.g. peak hour of the day. Let us say a content server has 10 000 customers connected as stated above, the peak time of the day is between 1900 and 2400 hours with 71 percent of the population watching television (in 2008). That means 7100 simultaneous users watching television at those hours, connected to a single content server.

TNS Gallup has more detailed, than Statistics Norway, information about television habits in Norway presented in [16]. These statistics are of higher granularity than what Statistics Norway has made public. The peak hour (the hour with most subscribers watching television) is between 2100 and 2200 hours, and counts approximately 42 percent of the total population. Consider the latter statistics, the peak hour has about 4200 simultaneous users at the peak hour, almost 3000 fewer users compared with the example given in former paragraph.

Reference [19] has done a study of user behavior and network traffic in an IP TV network with 250 000 households over a six month period. They have measured several parameters, including some of interest for this thesis. The analyzed data comes from a single provider with a given set of TV channels available as a foundation for its research. The pattern of user-behavior found, may not be realistic in networks from other countries and with another set of channels available. The pattern of user behavior is also dynamic and changes from year to year as seen as from the annual statistics from [29, 35]. Considering this, it is desirable all the data collection is from a specific network, and as new as possible. However, the output data is realistic and of current interest in the given environment. The fact that this type of statistical data is unavailable in Norway, reference [19] is based on data from 2007 and in the measurement is done in a network based on IP technology makes it relevant for the problem of this thesis. The following chapter presents the data from this reference.

Channel switching 60% of channel holding time is beneath 10 seconds and approximately 70% is beneath one minute. All channel holding times less than one minute are categorized as

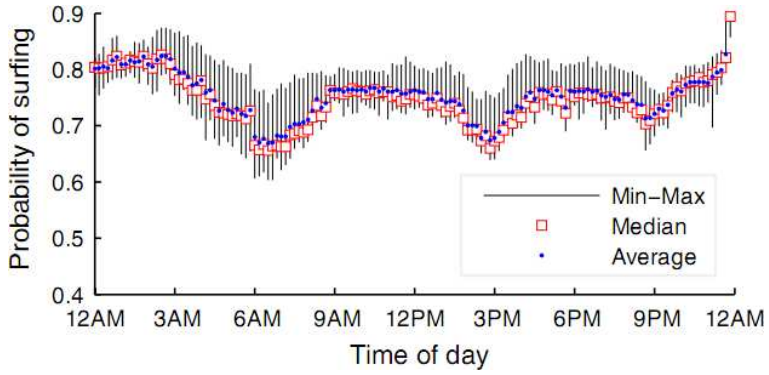


Figure 5.1: Probability of surfing across the day varies [19].

"surfing channels" — going through available channels just to find something to watch. Most channel changes happen four seconds after the previous channel change. The possibility of channel switching varies across the day as seen in figure 5.1. If the channel holding time is between one minute and one hour, it is categorized as "viewing" — the subscribers follows a program or show.

To test the performance of MMRP, it is useful to look at the hour of day with the heaviest load on the network. As stated above, that hour is between 2100 and 2200 hours. The following points gives a base for the simulation:

- $N_{users} = 4200$ simultaneous users.
- About 72 percent probability of channel switching [19]. That means 43.2 out of 60 minutes with channel switching. The remaining time is valued as "viewing time".
- When surfing channels, mean holding time per channel is ten seconds. This gives a total number of channel switches during the time categorized as "switching" in the peak hour S_{surf}.

$$S_{surf} = \frac{43.2 \text{ min} \cdot 60}{10 \frac{\text{sec}}{\text{channel}}} = 259.2 \text{ channel changes}$$

  .
- While viewing a channel, the holding time is 10.7 minutes. This gives a total number of channel switches during the time categorized as "viewing" in the peak hour S_{view}.

$$S_{view} = \frac{(60 - 43.2) \text{ min}}{10.7 \frac{\text{min}}{\text{channel}}} = 1.57 \text{ channel changes}$$

- A total of $S_{user} = S_{surf} + S_{view} = 260.77$ channel switches per user per hour.
- $S_{tot} = N_{users} \cdot S_{user} = 1095234$ channel switches total per hour.

- $S_{tot}/3600 = 304.23$ channel switches per second.

The probability of switching may seem high. Without knowing the whole truth, parts of the reason may be:

- People finished activities around these hours. Then they sit down in front of the TV, switching channels to see what is on.
- Programs are short, in terms of 30 minutes or less.
- The channels has frequent advertisement breaks.

Channel switching delay time The reference [19] mentions an infrastructure based IPTV system from Telco, guarantee a switching delay less than one second. However, users will not consider a switching delay as instantaneous until the delay decrease to the order of 100 to 200 milliseconds. The latter value will serve as a benchmark when analyzing the output data from simulations.

Radio relay topology Radio relay systems are used as backhaul transmission technology in cellular networks. The backhaul connects the access networks with the core network. There are also infrastructures solely based on radio relay system, and a real life topology is given by Nera Networks. This topology is quite complex, and may be hard to verify and analyze.

Verification topologies To verify that the implementation functions correctly, some base topologies should be used. The first is a simple topology. The standard presents an illustration of port roles in [24, figure 10.3].

5.3 Model translation

The functionality needed in each node to reach the desired objectives are divided in three main parts, the `MMRP` class, the `RSTP` class and the classes regarding bridge functions. The bridge functions are taken care of by the following classes. The `EthernetMAC` class for each interface (port) of a bridge, LinkLayerControl (`LLC`) class and the `MACRelayEntity` class which includes learning, forwarding, port states and filtering database. An illustration is shown in figure 5.3, where connections between the components are shown.

The `MMRP` class includes the functionality described in chapter 4 MMRP and the abstraction section above. The `Scheduler` class is special designed to control the timers and events in MMRP and make it work in the Autonomous Component Architecture.

This section is further broken down in three parts. First, the classes implementing the basic bridge functionality like LLC and the MAC Relay Entity. Next, a section about the RSTP. RSTP

was a preimplemented package, but some functionality was added. Finally, a section about the MMRP class.

5.3.1 The base bridge functionality classes

Some basic functionality is needed in the nodes so they act like a bridge. Each interface (or port) of a bridge is its own component, taken care of by the `EthernetMAC` class. Each component of this type has a down port where all incoming data from another node arrives, and an up port where data from the other components in the bridge arrives. The data from the network is of the `EthernetFrame` class. The `EthernetMAC` decapsulate the `EthernetFrame` that arrives at the down port and forwards it to the components above encapsulated in a `UnitDataIndication` object. The data arrived at the up port are decapsulated from the `UnitDataIndication` and transmitted on the network as an `EthernetFrame`. If the incoming data is not of valid format, it discards the data. Both of the sending formats mentioned are described later.

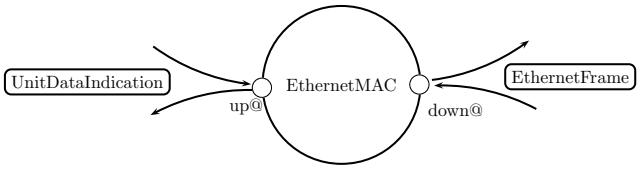


Figure 5.2: Illustration showing the data types used toward EthernetMAC throught the downport (EthernetFrame) and up port (UnitDataIndication).

The `LLC` acts like a packet filter which decides which of the higher layer components to send each incoming packet to. The standard [23] illustrates a `LLC` component for each port of the bridge, but instead the implementation has one `LLC` component per bridge. This component has one down port for each port of the bridge. This way the component can differentiate between the ports of the bridge. Based on the content of the incoming data, in relays it to either `RSTP` or `MMRP`. When data arrives at `LLC` from the components above, it simply looks at the destination interface and forward the data through the right port of its component.

`MACRelayEntity` has several tasks to take care of. It receives data from the ports of the bridge and must do address learning and frame forwarding. In addition it serves some ports where MMRP can query for port states and do filtering database configurations. The functions are based on the standard [23] and standard [24]. Some of the functions are discarded in the abstraction process, but the relevant functions are kept.

Figure 5.4 shows the flow when the `MACRelayEntity` receives an `UnitDataIndication`. The learning process takes note of where the data was sent from (MAC address) and if it not already registered in the `FilteringDatabase`, it maps the MAC address to the incoming port. That is, by learning, the bridge will know at which port to forward an incoming packet that has the

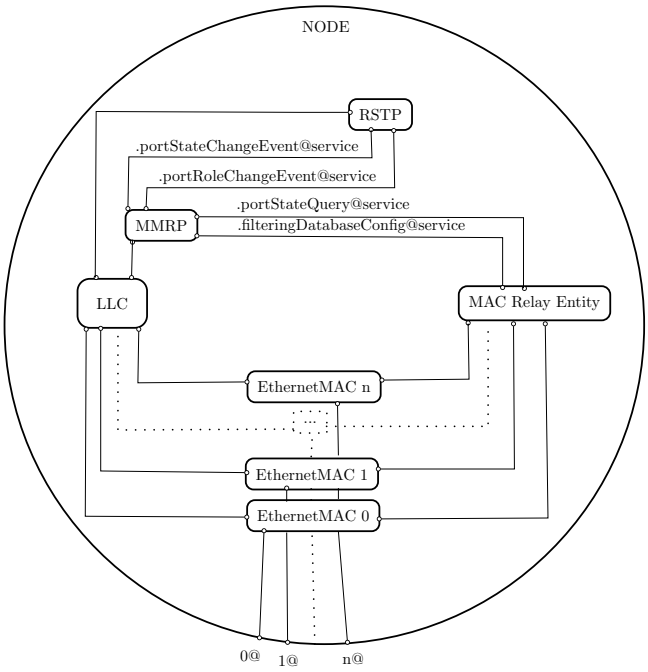


Figure 5.3: Overview of a the components and connections inside a node. A node is the abstraction of as ethernet bridge, and the ports named 0@, 1@ etc. at the bottom are the same as ports on a bridge. Each port is connected to an EthernetMAC component which sends the data to the next components of the node. The higher up on the illustration, the higher layer in the network stack the components belongs to. RSTP and MACRelayEntity are not connected, even though MACRelayEntity uses the port states of RSTP. Due to an early implementation approach, the MACRelayEntity has the RSTP as a private data member. This breaks with the ACA principle, and should be changed in the future.

previously learned MAC address as its destination address. This saves network resource, and makes the main difference between a hub and bridge. After the learning, the bridge will try to forward the data out the right port. If the destination MAC address is mapped to a port, it forward it to the given port or else it transmits the data out all ports in forwarding state except the port which it arrived at.

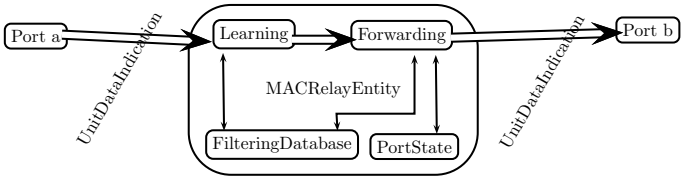


Figure 5.4: Overview of the MACRelayEntity flow when receiving an `UnitDataIndication`.

Other components may do changes in the filtering database through the `FilteringDatabaseConfig@service` port of the `MACRelayEntity` component. `MMRP` is the one component which makes use of this service in this implementation. The message used to communicate on this port is of datatype `FilteringDatabaseConfig`.

5.3.2 RSTP class

The implementation of RSTP was done in advance of the thesis by a former Nera Networks employee, Bård Henriksen. By using that as a base, the implementation has an active topology which was dynamically and correctly created by the RSTP implementation. The RSTP implementation is almost complete, but some minor addition had to be made so that it would play with the other components that would populate a bridge component. The addition made is:

State change event port Added a port called `.portStateChangeEvent@service` where a `PortEventMessage` is sent when a port changes state from forwarding to discarding. The changes influence both `Rstp` and `PortStateTransitionStateMachine` classes.

Role change event port Added a port called `.portRoleChangeEvent@service` where a `PortEventMessage` is sent when a port either changes from designated to alternate or root, or changes from alternate or root to designated. The changes influence both `Rstp` and `RstpPerPortParameter` classes.

These ports gives the support of events that MMRP needs to react to topology changes as defined in reference [24]. The RSTP class is the only class in the RSTP implementation that is a J-SIM component, hence it serves as the interface toward RSTP in J-SIM.

5.3.3 The MMRP implementation

The MMRP class extends the drcl.net.Module class and it is the only class of the MMRP implementation that is a J-SIM component. This class serves as an interface to all MMRP functions that was deemed necessary to implement from the abstraction process described in section 5.2. Instead of having one MMRP component for each port, the one MMRP component per bridge handles all MMRP related actions for the bridge. The MMRP class uses several other classes to complete its tasks. An overview of the composition is shown in figure 5.5. Extending the `drcl.net.Module` class gives MMRP the access to methods and data members needed to perform as a component in J-SIM. It is also possible to extend the `drcl.inet.Protocol` class, which itself is extended from the `Module` class. The `Protocol` class includes complete methods to link the component with the services from `CoreServiceLayer`[7] component, hence not relevant for MMRP.

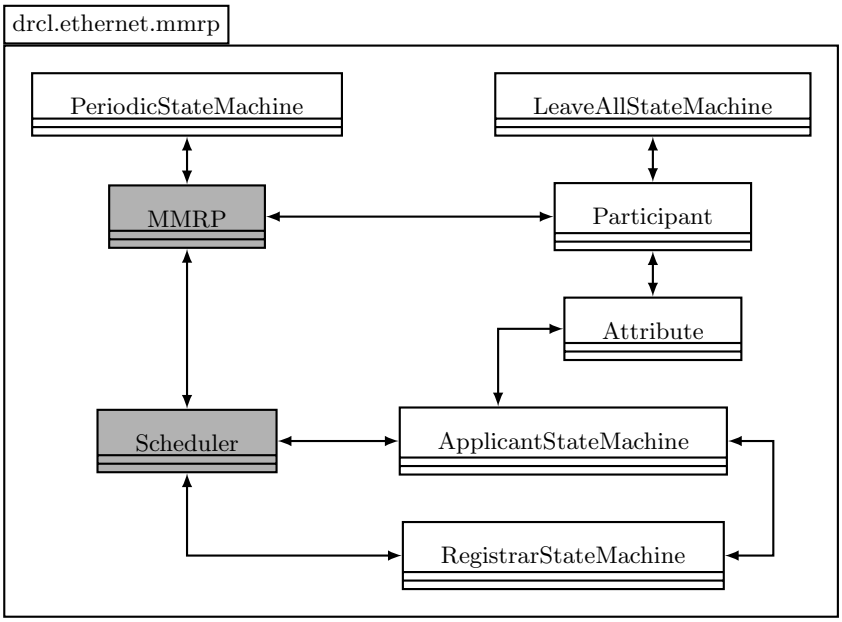


Figure 5.5: Overview of the main classes in the MMRP implementation and the flow of information in their collaboration. The fact that some relations are one-to-many is not illustrated here. Those relationships are shown in the figure 4.3, and are also true for the relationships on this illustration. The MMRP and Scheduler classes are colored gray. This is because they are two important classes when it comes to the architecture of the implementation. The MMRP class is the only J-SIM component and the interface uses building the scripts, while the Scheduler class handles all timers and action regarding the state machines.

[7] `CoreServiceLayer` is not mentioned here because it is not used. It is a component which contains the base of creating an IP routing node, which is not of interest.

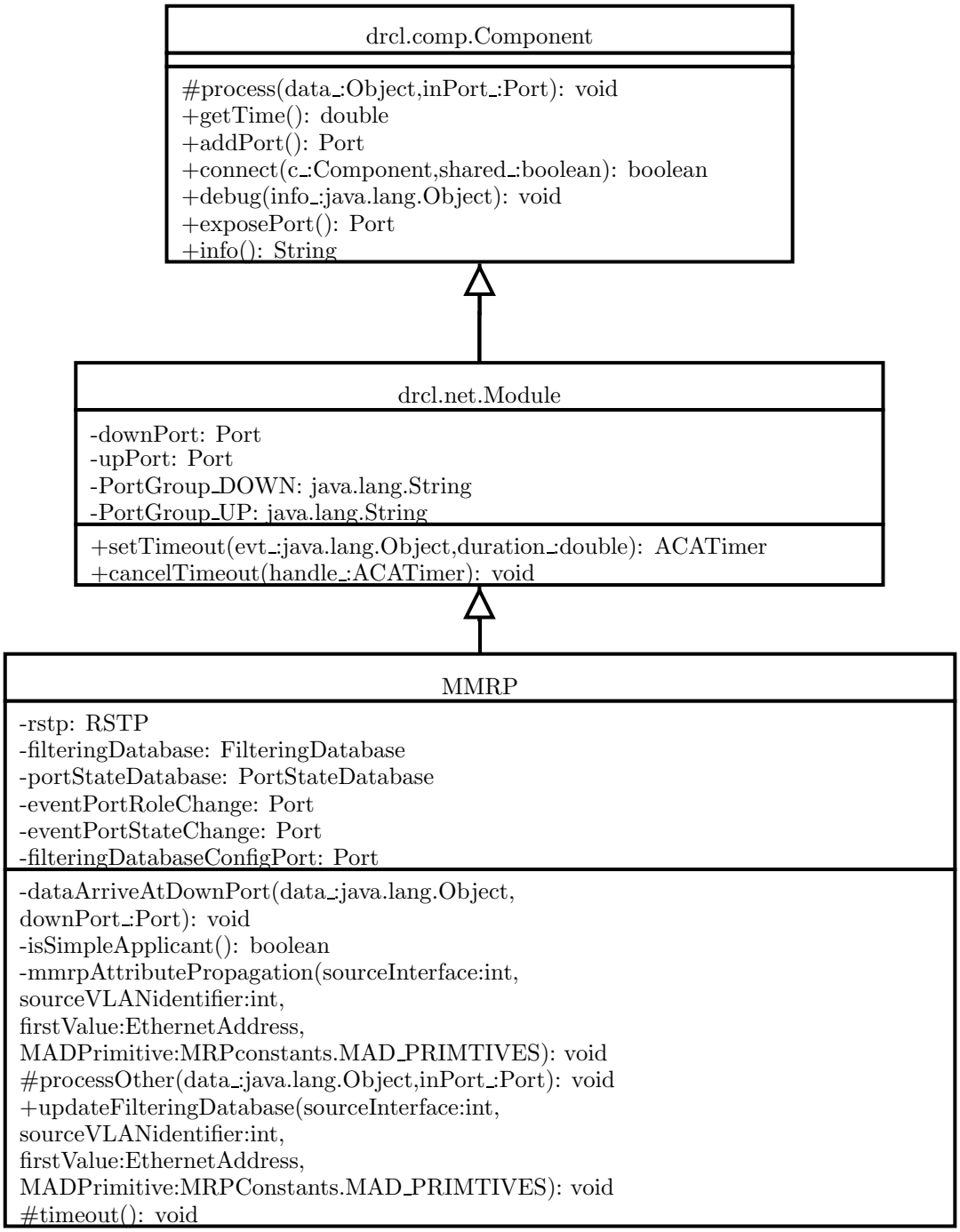


Figure 5.6: The MMRP class heritage. Some important methods and data members are shown, but not all are included.

The MMRP class

A component in J-SIM has some basic ports and methods to comply with the ACA. First, all events and messages that arrives at a J-SIM component are handled by a method called `process(Object data_, drcl.comp.Port inPort_)`. By default from the `Module` class it handles the incoming data based on which port it arrived at. `MMRP` do not override the `process` method itself, but the methods it calls. There are four methods that the `process` methods uses, depending on which port the message arrived at. The next part will describe how each method is implemented in `MMRP`.

`dataArrivedAtUpPort(...)` is not used by the `MMRP` class because it has no components communicating with it from a higher layer.

`dataArrivedAtDownPort(...)` handles the data which comes from the `down@` port and ports in the `@down` port group. MMRP receives data from the `.llc@down` port. The data should be MRPDUs from LLC, so other types of data is discarded. When MMRP recives a MRPDU, it notices which port it is received from and parses the data as described in the standard [24].

`timeout(...)` is called when data arrives at the `.timer@` port of the component. Data arrives at this port whenever a timeout event has happened. To register such an event, one need to call the `setTimeout(Object evt_, double duration_)` or `setTimeoutAt(Object evt_, double time_)` commands of the component. The data that arrives at the `.timer@` port is the `evt_` object. The set timeout methods returns an `ACATimer` object. By keeping the object, one may cancel the timeout by calling the `cancelTimeout(ACATimer handle_)` method, with the ACATimer object as input parameter. The `MMRP` component has the methods available for adding and removal of timer events, and the `Scheduler` object uses them. All data received at the `.timer@` port is sent directly to the `timeout` method of the `Scheduler` object.

`processOther(...)` receives all data which do not arrive at a down port, up port or timer port. The MMRP component has some ports which belong to the `service` port group, and those are identified by the group name and port id. The following ports are private data members of `MMRP`:

`.portStateChangeEvent@service` receives events from the active topology (in this case the `RSTP` component) when a port state changes from discarding or alternate to forwarding, or the other way around. This triggers an update of the set of ports in MMRP, and depending on the registrar state for attributes on the current port, certain message must be forwarded either to other ports on the bridge or to the network according to the MAP definition in [24, 10.3]. The data type of the event message is `PortEventMessage`.

`.portRoleChangeEvent@service` receives events from the active topology like the port mentioned above, but only when the port role changes from root or alternate to forwarding or vice versa. The former mentioned role change triggers a `Flush!` event in the Registrar state machine for the influenced MMRP participants. The vice versa role change triggers a `Re-declare!` event on both the Registrar and Applicant state machines of the influenced participants. It uses the same data type for sending as the port above.

`.portStateQuery@service` is used by MMRP attribute propagation (MAP). The MAD primitives that MAP disseminates are only to the ports that are in forwarding state. By sending a query on this service port, it gets a response with the states for all ports on the bridge. The data type for sending is of the `PortStateQuery` class.

`.filteringDatabaseConfig@service` is used to change the filtering database of the MAC relay entity. MMRP may add, change or delete entries in the database, and sends these commands in a `FilteringDatabaseConfigRequest` object.

Sending messages The MMRP uses MRPDUs to signal its state to other bridges and end user on the network and the MMRP Attribute Propagation share information between participants on a bridge. The next paragraphs describes the two types of signaling.

MMR Protocol Data Units It is only the Applicant and LeaveAll state machines that may trigger sending of MRPDUs. The PDUs are sent with a method called `MMRP.forwardPDU(...)`, which demands at least three input parameters. The first is the `MRPDU` object itself, which contains the information to be spread across the network. The two other parameters are port and VLAN which the PDU identifiers representing where the PDU should be forwarded. A `UnitDataIndication` is sent to the LLC which forward it to the correct port (an `EthernetMAC` component).

The method is called by MMRP itself when it receives a `PortStateChangeEvent` and have attributes declared. The `Scheduler` class uses it when it receives one of the send events from the applicant state machine.

MMRP attribute propagation (MAP) The MMRP Attribute Propagation shares informations between participant on a bridge, but does not send PDUs on the network. MAP is handled by a method with the same name: `public void mmrpAttributePropagation(int sourceInterface, int sourceVLANidentifier, EthernetAddress firstValue, MRPconstants.MAD_PRIMTIVES MADPrimitive)`. The parameter descriptions follow.

sourceInterface is the port which the MAD primitive is sent from.

sourceVLANidentifier is the VLAN identifier which the sending participant belongs to.

firstValue is in this implementation the MAC address (the attribute) which the primitive concerns.

MADPrimitive is the type of primitive sent. It can be a join indication, join request, leave indication or leave request. Basically, its either a join or leave primitive. The rule of how to handle the primitive differs a bit from a join and leave, as described in the standard [24].

Constants The standards [24, 23] defines a number of constants. These are implemented with `public final` accessibility in the interfaces `EthernetConstants` and `MRPconstants`. Some constants worth mentioning follows:

Ethernet address constants Like the Bridge PDUs (BPDUs) of RSTP, MRPDUs has a specific destination address which identifies that it should be delivered to the MMRP participant of the port – or as this implementation handles it, to the bridge MMRP component.

Timer constans The MMRP defines a set of timers to be used with the state machines, these are defined in the `MRPconstants` interface.

The attribute event types These types are used in the MRPDUs defining which event the PDU concerns. These are stated in the same interface as the timer constants.

The Scheduler class

The `Scheduler` class is, as the name suggests, responsible for the scheduling of events and actions of MMRP and its state machines. Each time a state machine does an action, it sends it to the `Scheduler` object so it may or may not schedule the action (some are instantly executed).

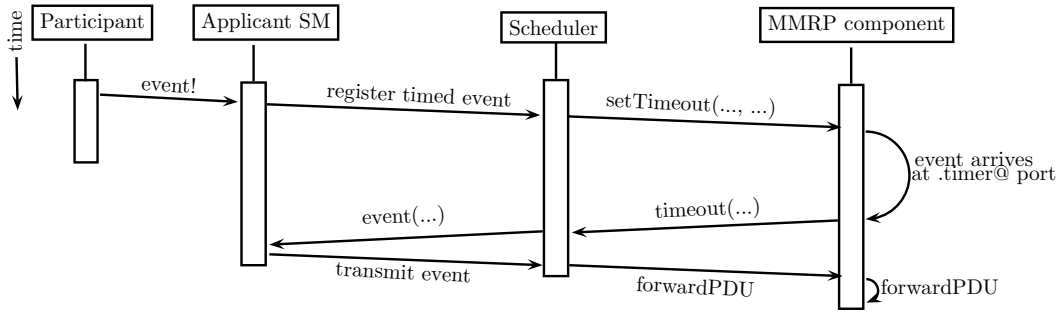


Figure 5.7: A sequence diagram showing an event registered at the applicant state machine resulting in forwarding of a PDU. The Scheduler are the center piece which decides what to do and communicates with the MMRP-class.

Due to the single J-SIM component architecture of the MMRP implementation, scheduling is done by using the ACA methods of the `MMRP` class – `setTimeout(...)` and `setTimeoutAt(...)`.

The `MMRP` component do not keep track of the time itself. It is handled by the ACATimer object of J-SIM. When a timer expires, the `ACATimer` sends the event object on the `.timer@` port of the `MMRP` component. `MMRP` does nothing with the event object other than sending it to the `Scheduler` object which handles all timer tasks. Both of the methods for setting a timer returns an `ACATimer` object. This object is saved because it may be used to cancel the event at a later time. When the Scheduler registers a timeout event at the MMRP component, it adds the returning ACATimer object in a list. The leavetimer and the leavealltimer may be canceled due to incoming messages. When this happens, the Scheduler looks up the current event in the list, fetches the ACATimer object (here named `handle_`) and cancels the event by calling the method `MMRP.cancelTimeout(ACATimer handle_)`. By letting J-SIM keep track of all the timers with its `ACATimer` – it complies with the discrete event driven simulator principle. As soon as nothing happens in the simulation world time, it can skip the time until next event in the simulation. This saves computer resources and speeds up the simulation process. The different timers handled by the Scheduler class is described in the following part.

The leave timer The registrar state machine holds the LV state as long as the duration of the leave timer. Default value of the timer is between 0.6 and one seconds, and the implementation has set it to 0.6 seconds. The leave message is not sent from the current MMRP participant before the timer expires.

The LeaveAll timer There is one LeaveAll state machine per MMRP participant. The LeaveAllTimer goes continuously, with a period of T. The time T is calculate each round, and it is a random value between $LeaveAllTime < T < 1.5 \cdot LeaveAllTime$. When the timer expires, a MRPDU with LeaveAll is sent from the current participant.

When a participant receives a LeaveAll message, its LeaveAllTimer is restarted without any further action. This suppresses multitple LeaveAll messages from Participants connected to the same LAN.

The Join timer The join timer controls the interval between transmit opportunities that are applied to the Applicant and LeaveAll state machines. Default value of the timer duration is 0.2 seconds. If the **operPointToPointMAC** is `TRUE`, a transmission opportunity is given as soon as practicable. No more than three opportunities may be given during a $1.5 \cdot JoinTime$ interval.

The Periodic timer The periodic timer triggers a `periodic!` event in the applicant state machine which results with a JoinMt or JoinIn message being sent from the applicants which are declared. The JoinMt means that the current attribute is declared but no registered, while the JoinIn means that the attribute is both declared and registered. This ensures that registrations are successful without using to much bandwidth. The default PeriodicTime is one second, and

the timer goes in a loop as long as it is not turned off by management.

Messages

There are two kinds of messages described in the following. The first type is message defined by the standards [23, 24]. This includes PDUs (bridge to bridge) and UnitDataIndication (in-bridge signaling). These messages are explained first in the following paragraphs. The other type of message is the one sent between components in the J-SIM composition, like the one MMRP sends to RSTP to ask for PortStates. These messages are explained at last, starting with `PortStateQuery`.

MRPDU The format of the PDU is described with figure 4.1. MRPDUs are used by MMRP for signaling between bridges. The implementation uses one class for each of the subelements of variable size shown in the figure. The main class `MRPDU` has constructors for creating a MRPDU with correct format.

UnitDataIndication The unit data indication is described in the standard [23], and is a message primitive used in the ISS (Internal Sublayer Service) in bridges. The implementation do not follow the standard, because of a difference in the architecture building the bridge. The standard has an architecture where each port has an application of a protocol such that in J-SIM, all ports would have its own MMRP component. Since the implementation uses only on single MMRP component in a bridge, it needs some more information included in the parameters of the primitive.

type_ Type of UnitDataIndication. Not used with the current implementation.

dst_ The destination of the PDU given by an `EthernetAddress` object.

src_ The origin of the PDU given by an `EthernetAddress` object.

pkt_ The data itself, also known as payload, of the UnitDataIndication.

payloadSize_ The size (in bytes) of the payload.

initDestinationInterface The destination interface of the PDU inside a bridge. This was added in addition to the parameters given by the standard. It is needed when e.g. the MMRP component sends an UnitDataIndication to the LLC, or else the LLC do not know at which port to forward the message.

initSourceInterfac The source interface of the PDU inside a bridge. This was also added in addition to the parameters given by the standard. It is needed when e.g. the MACRelayEntity component receives a message, and the learn function must map the address to

correct port. Another example is when the forward EthernetAddress has no mapping, the frame is supposed to be forwarded on all ports except the incoming port.

EthernetFrame The EthernetFrame or "MAC Frame" as it is named in the standard is described in [10, 23]. Every messages that are sent between the bridges are encapsulated in a EthernetFrame. For VLAN-networks, a special type of header is used which includes a VLAN identifier and a few other bytes. These additional parameters are not implemented in the current version.

PortStateQuery The PortStateQuery class is used between the MMRP component and the MACRelayEntity component. It should go to the RSTP component, but this solution was used in an early implementation and is not fixed yet. The message contains two parameters as follows.

request Shall be `TRUE` if the message is a request for port states, and `FALSE` if it is a response to a request.

portList A list with an item for each port and state of the port. All ports of a bridge is added to the list. The state of the port is either forwarding, learning or discarding.

PortEventMessage The PortEventMessage class is used to create objects sent by the RSTP when one of the state or role of a port changes as described earlier in this chapter. When MMRP receives such a message it triggers events on the applicant and registrar state machines. The PortEventMessage has the following set of parameters:

portId The identificator representing the port number (unique on a bridge).

oldValue A number representing the old state or old role of the port.

newValue A number representing the new state or new role of the port.

FilteringDatabaseConfig MMRP is responsible of creating, changing and deleting entries in the filtering database of a switch. In this implementation, the filtering database is placed in the MACRelayEntity. To be able to access the filtering database, MMRP sends these FilteringDatabaseConfig objects to the `.filteringDatabaseConfig@service` port of the MACRelayEntity component. A response is also given indicating a successful or unsuccessful action. The following parametes are included in the message type:

portId The port identifier which is unique at a bridge.

identifier Identifier for the VLAN.

ok Shall be `TRUE` if the action was completed without any faults, `FALSE` otherwise.

EthernetAddress The address which the entry concerns.

forward Shall be `FALSE` if frames destined to the EthernetAddress should be filtered from this VLAN and port combination, `TRUE` else.

Chapter 6

Testing

This chapter describes the testing of the implementation. This is done to verify that the components handles as desired. The chapter is broken down in four parts, each part representing one test. First, a test of the RSTP package to confirm that the logical topology was created correctly. Next, a test of the basic bridge functionality that emphasize the learning and forwarding. Finally, the two last parts test the MMRP implementation.

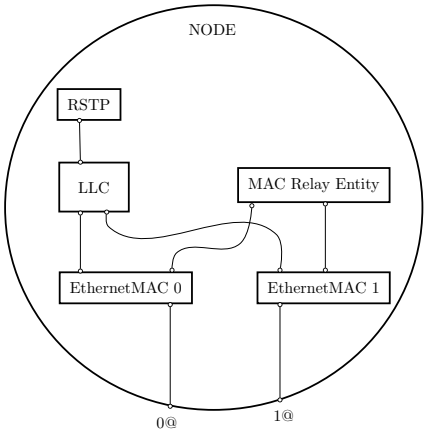
6.1 Test one – RSTP

This part will test the RSTP implementation. The version of RSTP used is not implemented by me. I have used a version provided by Nera Networks. The RSTP is responsible of creating an active topology with no loops. In addition, it should restore a broken topology. Initial tests showed that the RSTP implementation for J-SIM, uses about 15 seconds to restore the logical topology on a link brake. This was in a ring topology with ten bridges. On the other hand, reference [32] presents results with convergence times around 10 milliseconds, and reference [18] states convergences times beneath 50 milliseconds. The former references do not take hardware delay into account, and the latter reference do not include hardware or BPDU processing delays in their simulation. Both conclude with convergence times in the order of centiseconds.

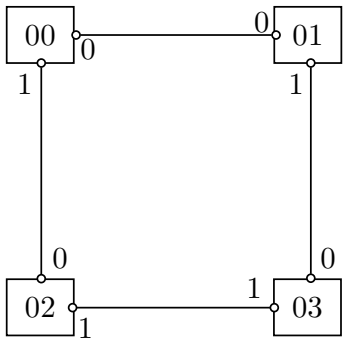
Furthermore, the port roles and states must be correct. Upon an event that should trigger topology change, RSTP must change to a new correct spanning tree. Furthermore, the topology change process should be completed in certain time interval.

6.1.1 The test set up

The components of the bridges are illustrated in figure 6.1a. Each bridge has two ports handled by the EthernetMAC Java class. Each port is connected to a LLC component and a MACRelayEntity component. The RSTP component is connected to the top of the LLC component.



(a) The subcomponents of a bridge node.



(b) The physical topology of four brigdes connected in a ring.

Figure 6.1: The two illustration represents test number one. Figure 6.1a shows the internal structure of the four bridges in shown in the topology, figure 6.1b.

The bridges has their own unique MAC address, which the ports also use. The MAC address is made out the number printed in the middle of each bridge in figure 6.1b. Where ## is representing the number, the MAC address is 00:11:22:33:##:00. Each part of this address separated by ":" is of hexadecimal format.

The physical topology is illustrated in figure 6.1b. Each link between the bridges has a propagation delay representing approximately 15 meters. There is no processing delay or sending delay (bandwidth limit). The simulation is running for 10 seconds to be sure that the bridges are in a stable state.

6.1.2 The expected result

RSTP makes a logical topology on top of the physical topology as described in 2.2 *Ethernet switching*. The root bridge is the bridge with the lowest bridge ID. The ID is calculated from the MAC address, so in the test scenario it is reasonable to believe that bridge "00" becomes the root bridge.

The cost of the links are set to the same, so the best path would be the path with the least number of hops to the root bridge. Bridge "03" is two hops away from the root bridge from both of its ports. The port that receives the best bridge ID from its connect bridge gets the role "root" and state "forwarding". The other port gets the role "alternate" and state "discarding". In this setup, the port 0 of bridge 03 going trough bridge 1 becomes the "root" port, while port 1 becomes the "alternate". Furthermore, port 0 of bridge 01 and port 0 of bridge 02 should become root and port 1 of bridge 01 and port 1 of bridge 02 should become designated. The

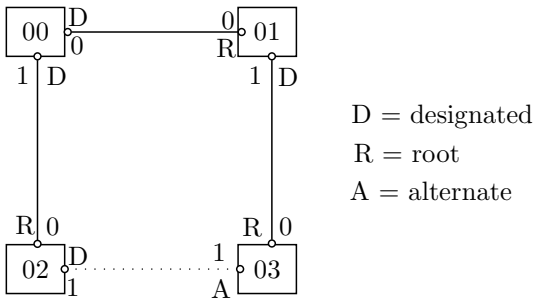


Figure 6.2: The expected logical topology with role ports. The dotted link between bridge 02 and 03 is not a part of the topology. It may be used if the physical topology changes.

expected roles are shown in figure 6.2.

The filtering database in the MACRelayEntity do learning of all incoming packets except signaling the BPDUs sent by RSTP. Because RSTP sends packets on the entire physical topology, learning from the packets would risk creating loops. Therefor, the filtering database should not contain any dynamic entries after the simulation is done.

6.1.3 The result

The states of RSTP are read at the end of the simulation and the states and roles are as predicted. This means that the RSTP implementation sets up a correct spanning tree from this physical topology compared to what the standard states [22].

The filtering databases contains no dynamic entries, which also reflects the expected result. The first test of basic bridge functionality is passed.

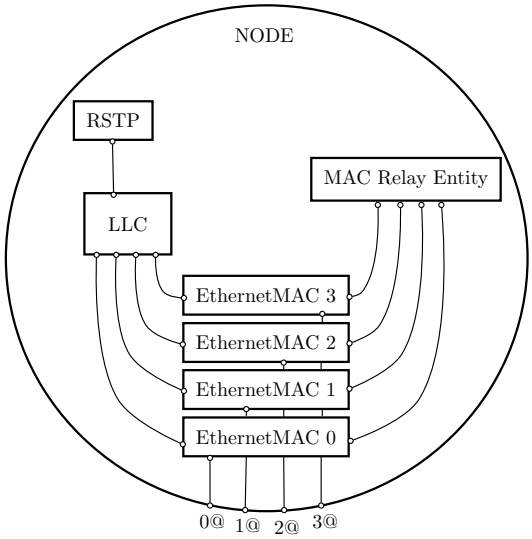
However, the restoration mechanism is not implemented correctly. The reason is that it depends on timers on each link instead of propagating the new states through the network.

6.2 Test two – MAC relay entity

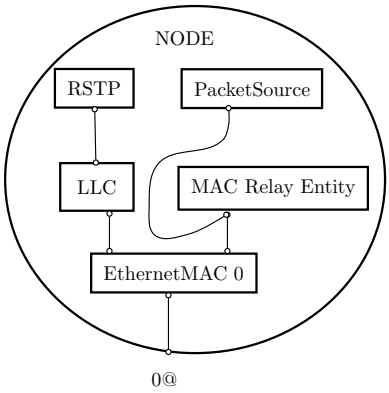
This part will test the functionality of the MAC relay entity. The MAC relay entity is responsible for the mapping between port, VLAN and MAC address in the filtering database, and forwarding of frames.

6.2.1 The test set up

Five bridges are connected in a star formation with bridge 00 in the center. Bridge 01, 02, 03 and 04 are only connected to the center bridge. As seen in figure 6.3, the center bridge has four ports while the edge nodes has one port. The MMRP component is not included because it is not of relevance for this test.



(a) The structure of the central node, bridge 00.



(b) The structure of the edge nodes, number 01, 02, 03 and 04. The "PacketSource" generates packets to be sent on the network. It is connected to the MACRelayEntity.

Figure 6.3: Node structure of the two different nodes in test number two.

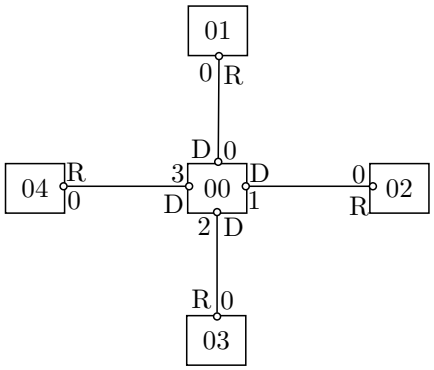
The physical topology is illustrated in figure 6.4a. The links have a delay according to 15 meters of propagation. The bandwidth cost parameter of RSTP is equal on all ports. The bridge MAC address are based on the number printed in the middle of the bridges in figure 6.4a, as described in test number one (6.1).

The edge nodes, number 01, 02, 03 and 04, has a source component called "PacketSource" as seen in figure 6.3b. The PacketSource is connected to the down port of the MACRelayEntity. Data entering the down port of the MACRelayEntity are supposed to come from a port of the bridge. The PacketSource manipulates this functionality by sending data with source port as "-1". The MACRelayEntity treats data with this source port as any other, except that it does not call the learning method.

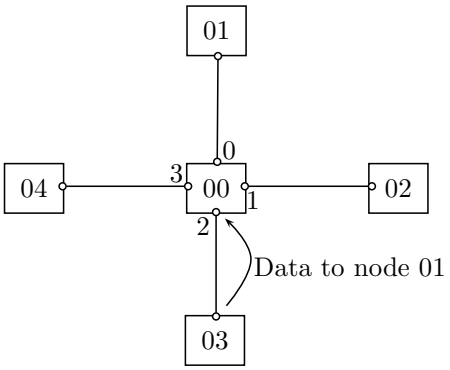
The test has three phases. The first is the initial actions from RSTP with setting the right roles and states for the ports. The second phase shall send data from node 03 towards node 01. In the last phase, node 01 sends data the other way towards node 03.

6.2.2 The expected result

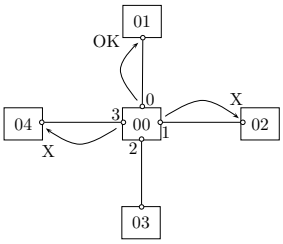
The logical topology has only one outcome based on the given physical topology. This is because the physical topology represents no loops. The center bridge has the lowest bridge ID because



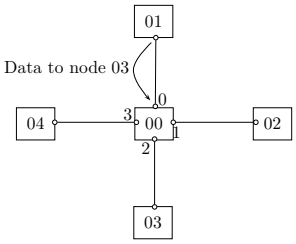
(a) The topology of test number two. A center bridge connected to four nodes. Port numbers and port roles are shown next to the ports. The abbreviations of port roles are the same as in figure 6.2.



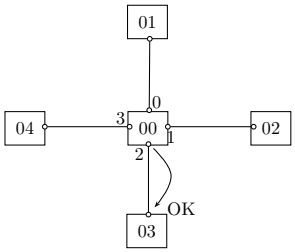
(b) The first part of phase two. Node 03 sends data towards node 01. First it arrives at port 2 of the center node.



(c) The second part of the second phase. The filtering database has no entries with the destination address for the data. Therefore, the data is duplicated and transmitted out all ports except the incoming port.



(d) The first part of the last phase. Node 01 sends data towards node 03. The data enter port 0 of the center bridge.



(e) The last part of the last phase. The filtering database has a entry for the destination address, and the data is only forwarded out port 2 towards node 03.

Figure 6.4: Test number two. Figure 6.4a shows the physical topology and the expected port roles given by RSTP. Figure 6.4b and 6.4c shows the traffic flow of the second phase of this test. The last two figures, 6.4d and 6.4e, shows the traffic flow for the last phase of the test.

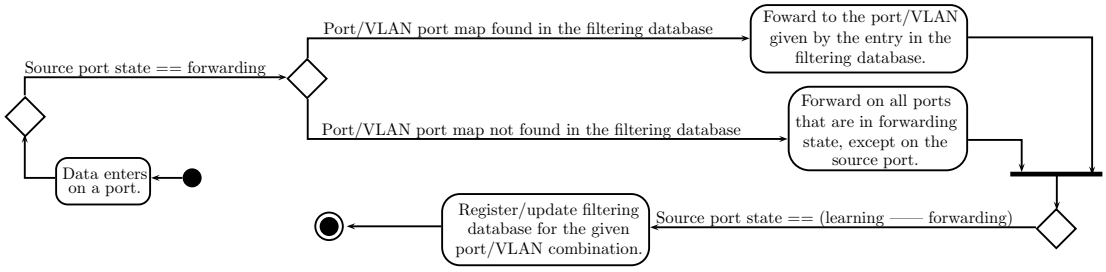


Figure 6.5: An activity diagram illustrating how the MAC relay entity of the bridge 00 forwards data it receives.

of its MAC address, therefor it will become the root bridge. When the first phase is completed, the port roles are expected to look as in figure 6.4a.

The second phase starts after ten seconds. Data shall be sent from node 03 towards node 01. The data sent is a String saying "DUMMY PAYLOAD OF IP PACKET". The String is encapsulated in an IP packet, and furthermore encapsulated in an EthernetFrame. The "PacketSource" component of node 03 sends a packet out on its port.

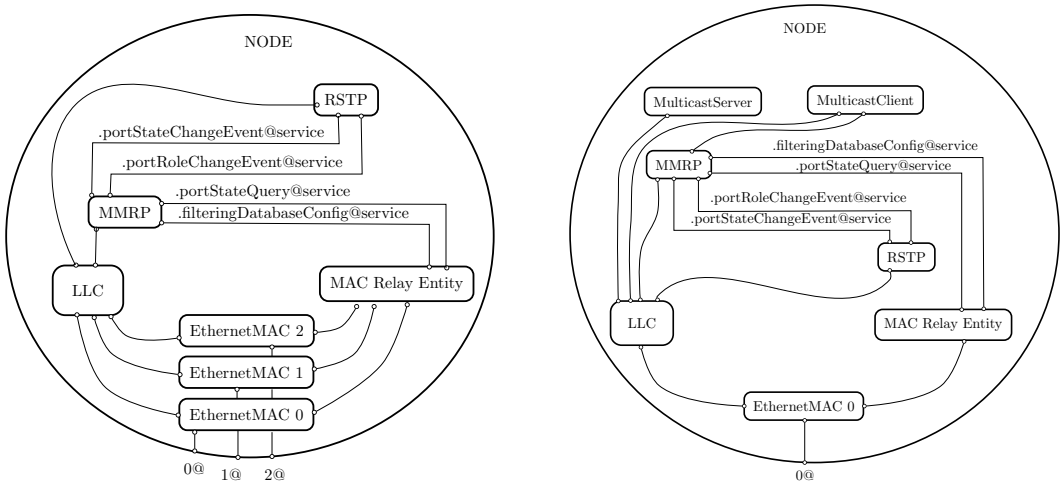
The data arrives at port 2 of bridge 00, as shown in figure 6.4b. Because this data is the first data that is sent in the network, the filtering database is empty (except from the static entries). As shown in figure 6.5, the data is expected to be forwarded on all ports except from the port it arrived. This is further illustrated in figure 6.4c. In addition to the forwarding, the MAC relay entity calls the learning method which should create an entry in the filtering database.

The third phase starts after 15 seconds. Data shall be sent from node 01 towards node 03, as shown in figure 6.4d. It is an IP packet containing the same String as in phase two. The difference from phase two is illustrated in figure 6.4e. The data is not forwarded on all ports, because bridge 00 finds an entry in the filtering database. The forwarding process is shown in figure 6.5. The entry from the filtering database contains a port/VLAN mapping of the destination address. The port map is pointing on port two. Because of this, the data is only forwarded out on port two, and it then reaches node 03 correctly.

6.2.3 The result

The results are as expected. Phase one gave a correct logical topology. The ports of the edge nodes was in forwarding state and the root port role. The center bridge was chosen as root, and all its ports was in the designated role and forwarding state.

In the second phase, the data was forwarded out port 0, 1 and 3 from bridge 00 as predicted. When data was sent the other way in phase three, it was only forwarded out port 2. At the end, the filtering database of bridge 00 had two entries. The first was node 03's address with port 2, and the last was 01's address with port 0.



(a) The bridge component. In test three, the bridges have three ports. Further, the bridge has a LLC, MMRP, RSTP and MACRelayEntity component. All these components are described in chapter 5.3.

(b) Illustration of an end user node. An end user has only one port. The MulticastServer is used to send traffic addressed to MAC group addresses. The MulticastClient sends registration and deregistration primitives to the MMRP component. Furthermore, it receives traffic addressed to MAC group addresses.

Figure 6.6: Component composition of simulation test number three.

To conclude, the forwarding and learning functions of the MAC Relay Entity work as expected.

6.3 Test three – basic test of MMRP

This part will test registration and deregistration of attributes with the MMRP. The states of the state machines in MMRP will be compared with the expected result.

6.3.1 The test set up

The physical topology is illustrated in figure 6.7a. The triangles in the figure illustrate end users, while the squared node illustrate a bridge. The system consists of one bridge connected to three end users. The links between the nodes have a delay according to 15 meters. The bandwidth cost parameter of RSTP is equal on all ports. The bridge MAC address are based on the number printed in the middle of the bridges in figure 6.7a, as described in test number one (6.1).

The composition of the end nodes is illustrated in figure 6.6b. They have one port connected to the bridge. The “MulticastServer” component may send EthernetFrames destined to a group MAC address on the network. This may be used to see that data addressed to group addresses is forwarded correctly. The “MulticastClient” may send register and deregister commands to the

MMRP client.

The bridge component is illustrated in figure 6.6a. It has three ports, one for each end node. Compared to the previous tests, the MMRP component is added. All traffic uses the same VLAN identifier.

6.3.2 The expected result

This test consists of three parts. The first test registration of attributes from several clients. The second part tests deregistration and the last part sends data addressed to one of the registered group addresses to see if the data is forwarded correctly.

Part one

This part starts after 10 seconds. By that time, the RSTP should be finished with the logical topology. No MMRP signaling is done before 10 seconds, therefor no MMRP participants are created in the system. In fact, no other traffic than RSTP's BPDUs has been sent, so the filtering databases should be empty.

The standard [24] defines two primitives. One for registration and the other for deregistration of attributes. Compared to the OSI reference model, the primitives shall be sent from a layer above MMRP. In the J-SIM implementation, these primitives are sent by the "MulticastClient" component. When the MMRP component receives a primitive, it should start the process of registering or deregistering the attribute. This process is explained step-by-step in the following.

1. The registration starts at end user 01. When the MMRP component receives the registration primitive, it does the same as when it receives a MAD_Join.request(...). Because this is the first time the attribute is being declared, a MMRP participant is created. The state machines are set to initial states. The applicant state machine is in the VO state, and the registrar state machine starts in the MT (empty) state. A MAD_Join.indication(...) triggers a "Join!" event on the applicant state machine (from now "applicant"). The applicant asks for a transmission opportunity and changes state to VP. The transmit event happens as soon as possible (as described in 5.3), and triggers a PDU to be sent at from the port. The PDU shall contain the state "JoinMt". This state means that the participant is declaring the attribute but the attribute is not registered. After the PDU is sent, the applicant changes state to AA and asks for another transmit opportunity. The transmit opportunity occurs as soon as possible and then the applicant changes state to QA. The reason why it send two message is to ensure that participants on the same LAN receives the registration (as described in 4.1).

2. The messages sent by end user 01 is received by port 0 on number 00 bridge. The message type is "JoinMt", which means that the sending participant has declared but not

registered the attribute. This triggers a "rJoinMt!" event in registrar and applicant state machines. The registrar state machine performs a "Join" action, which sends a MAD_Join.indication(...) to the other participants on the bridge. Finally, the registrar updates the filtering database and changes state to IN. The update of the filtering database is done by adding an entry with the given MAC group address, saying that it should be sent out on the current port. In this case it is port 0. The applicant's state is VO, because nothing has happened yet. The "rJoinMt!" event triggers nothing at when the applicant is the VO state. When the second "JoinMt" message from end user 00 arrives, the states is not changed on the applicant nor the registrar. Table 4.3 and the registrars state table in [24] shows all states, events and actions as described here.

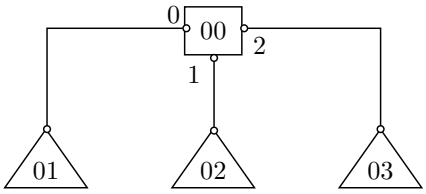
3. The MAD_Join.request(...) that arrives at the other participants triggers a "join!" event on the applicants. The state changes to VP and then they follow the same actions and state changes as described in the first point from end user 01. This means that the participant for port 01 and 02 sends two"JoinMt" messages each on the connected LAN. They are received by, respectively, end user 02 and end user 03. The actions and state transitions at end user 02 and end user 03 is the same as described in point 2 where bridge 00 receive the "JoinMt" message from end user 01. Because end user 02 and 03 is not bridges, they have only one port. This means that there is further messages sent on the network at this point.

The stable state is shown i figure 6.7b and 6.7c. The former figure shows the states of the applicant and registrar. The latter figure is simplified by saying that the participant has either declared or registered the attribute. It also shows the propagation MRPDUs by arrows. The time from end user 01 receives the registration request to the information is converged to the rest of the net should be $2 \times PropagationDelay = 1 \times 10^{-12}$. This is because the model lacks the processing and bandwidth delay. The state of the network is stable until the leaveAll timer expires. The leaveAll! event triggers re-registration from the participants.

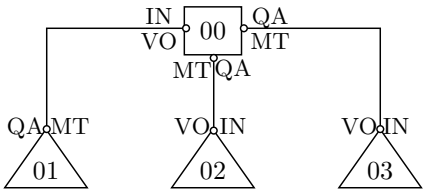
End user number 02 is also registering the same MAC group address as end user number 01. End user 02 starts the registration at time 15.0 seconds. The propagation of the subscription from end user 02 will go approximately the as with end user 01. The different is that most of the participant in the network no are both declaring and registering MAC group addresses. The exception shall be port 02 of bridge 00. Because end user number 03 has not declared the attribute, the port 02 of bridge 00 only declares the attribute. The stable state after end users to registration is shown in figure 6.7d.

Part two

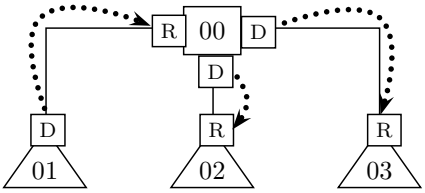
This part starts after 15 seconds. A deregistration shall be sent from end user 01 and propagate through the network. At the end, the only registered user should be end user 02. It starts



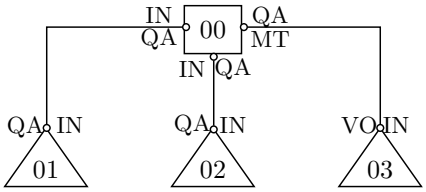
(a) The physical topology of test number three. The node number 00 which is squared is a bridge. The triangles are end users. The number around the bridge are port numbers.



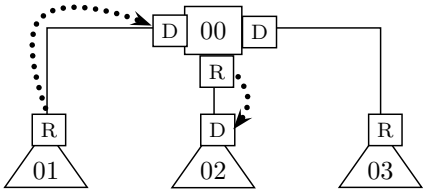
(b) The states after the first registration in part one of test three. Each port has states for the applicant and the registrar state machines. QA and VO are applicant states while MT and IN are registrar states. Figure 6.7c shows whether the participants have registered or declared the attribute.



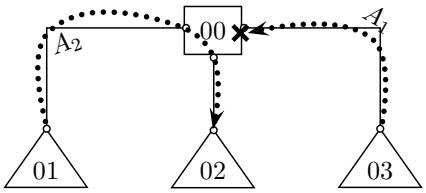
(c) The same as figure 6.7b, but with simplified annotations. Shows whether the participants have registered or declared the attribute. It also illustrates the way of MMRPDU propagation by arrows. One can also get this information from the states of the applicant and registrar state machines as shown in figure 6.7b.



(d) The state after part one of test number three. The ports with QA and IN has both registered the attribute and is declaring it. The VO state of the applicant means that the attribute is not declared, while the MT of the registrar means that the value is not registered.



(e) The state after part two of the test. The states represent whether a participant has declared or registered (or both). The arrows shows how the leave message should propagate.



(f) The expected propagation of data addressed to A_1 and A_2 at the end of test number three. If source pruning was enabled, traffic addressed to A_1 would have been truncated at end user 03.

Figure 6.7: Flow charts and topology of simulation test number three.

with the MmrpClient component of end user 01 sending an deregistration primitive to the Mmrp component. This triggers a MAD_Leave.indication(...) that further triggers a Lv! (leave) event at the applicant. The applicant changes state from QA to LA. Then it sends a leave message (PDU) to the participants on the LAN. Finally, it changes state to VO. The participant is no longer declaring the attribute.

Next, bridge 00 receives the leave message on port 0. This triggers a "rLv!" (received leave message) event at both the applicant and registrar. The registrar starts the leave timer and changes state from IN to LV. The applicant changes state from QA to VP and sends a "JoinMt" message. Furthermore it changes state from VP to AA. Then the applicant sends another "JoinMt" message and changes state to QA. The leave timer that the registrar started, expires after leave time. The leave time is 0.6 seconds. When the timer expires it triggers a MAD_Leave.indication(...) primitive to be sent. The MAD attribute propagation (MAP) sends the MAD leave primitive to another participant if the other participant is on the same bridge and uses the same VLAN identifier as the source participant. There are two additional rules for the MAD_Leave primitive. The other participant receives the MAD_Leave primitive if

- the receiving participant and the sending participant have registered the attribute, but no other participant on the same context have a registration, or
- the only participant that has registered the attribute is the one who sends the primitive.

This is because participants that have registered the current attribute shall not loose their registration in the network. In the current test, the MAD leave primitive should not be sent to the participant at port number 2. If this is done, end user 02 will loose its registration at end user 03. On the other hand, it should be sent to port number 1. This removes the declaration from port number 1, and forwards a leave message towards end user 02.

The participant on port 1 should receive the MAD_Leave.request(...), which should further trigger the same actions explained for port 0 at end user 01 at the beginning of this part. The participant should finally be in the VO state (not declaring the attrbiute). Figure 6.7e shows the state at the end of this part. Additionally, it shows the propagation of the leave message by stippled arrows. The total convergence time of the deregistration should be two times the propagation delay and a leave time delay. This results in a convergence time of 0.600000000001 seconds.

Part three

This parts test whether traffic are sent correctly based on registration from the end users. Default filtering behavior is to filter all unregistered groups. That means that if a group MAC address is not registered by any end users, the data with this address will not be forwarded by the bridges. If a group address is registered, it will be forwarded to the ports in the port map.

Abbreviation	Description	Value
Δ_{prop}	Propagation delay	5×10^{-13}
Δ_{join}	Join time	0.2s
Δ_{tx}	Transmission opportunity delay	$0 \leq \Delta_{tx} \leq 1.5 \times \Delta_{join}$
Δ_{leave}	Leave time	0.6s
A_1	First address	`ff:ff:11:11:11:ff`
A_2	Second address	`ff:ff:11:22:11:ff`
T_1	End user 03 starts sending data to the A_1	25.0
T_2	End user 01 starts sending data to the A_2	25.1
T_d	End user 02 deregister the A_1	26.0
T_{rx_d}	Bridge 00 receives deregistration and leave timer starts.	$26 + \Delta_{prop}$
T_{tx_j}	End user 02 registers the A_2	26.1
T_j	Register the A_2 at bridge 00	$26.1 + \Delta_{prop} + \Delta_{tx}$
T_{rx2}	End user 02 receives traffic to A_2	$T_j + \Delta_{prop}$
T_l	Leave timer expires at bridge 00	$26 + \Delta_{prop} + \Delta_{leave}$
T_{endA1}	End user 02 do no receive traffic to A_1 no more	$T_l + \Delta_{prop}$

Table 6.1: The different times used in part three of test three.

At the start of this part, end user 02 has registered the group address “ff:ff:11:11:11:ff”. At the time of approximately 25.1 seconds, the end user 01 and 03 will start sending data on the network. End user 01 sends traffic with destination address “ff:ff:11:22:11:ff”, while end user 03 sends to “ff:ff:11:11:11:ff”. These are both group addresses. The body of the EthernetFrames that are sent, is the string “DUMMY IP-packet” encapsulated in an IP-packet. Both end users 01 and 03 will send these frames every 0.1 seconds.

At approximately 26 seconds, end user 02 will deregister from the “ff:ff:11:11:11:ff” group address (named A_1). Later, at approximately 26.1 seconds, end user 02 will register the group address “ff:ff:11:22:11:ff” (named A_2). The propagation delay is named Δ_{prop}. The bridge 00 receives the leave message for A_1 at 26 seconds + Δ_{prop}. Immediately, the leave timer of 0.6 seconds starts. Next, the bridge receives the join message for the second address, and adds this address to the current port and VLAN mapping entry in the filtering database. The variables are listed in table 6.1.

End user 02 should receive data addressed to A_1 until the time T_{endA1}, where $T_{endA1} = 26 + (2 \times \Delta_{prop}) + \Delta_{leave}[s] \approx 26.6[s]$. The time it deregistered should be $T_d = 26.0$. The delay between deregistration and when the traffic no longer is sent to end user 02 is about 0.6 seconds. In this time space, the MulticastClient component of end user 02 should print an error message

Node	0						1		2		3	
Port	0		1		2		0		0		0	
Attribute	A_1	A_2	A_1	A_2	A_1	A_2	A_1	A_2	A_1	A_2	A_1	A_2
Applicant	VO	QA	VO	VO	VO	QA	VO	VO	VO	QA	VO	VO
Registrar	MT	MT	MT	IN	MT	MT	MT	IN	MT	MT	MT	IN

Table 6.2: The expected state machine states after part three of test three. "A" is short for applicant state machine, while "R" is short for registrar state machine. A_1 and A_2 are taken from table 6.1, and represents the attributes. The VLAN identifier is not mentioned, because all traffic uses the same identifier.

.

saying that the received data is not registered any longer.

The time end user sends a registration primitive for attribute A_2 is T_{tx_j}. Further, the earliest it may receive the data addressed to A_2 is T_{rx2}. The delay between T_{tx_j} and T_{rx2} is variable because of the transmission opportunity described in 5.3. At a minimum time, the first data may arrive at $T_{rx2} + (2 \times \Delta_{prop}) + 0[s] \approx 26.1[s]$. At maximum time, the data may arrive at $T_{rx2} + (2 \times \Delta_{prop}) + (1.5 \times \Delta_{join})[s] \approx 26.3[s]$. The propagation times are so small compared to the timers that they are not included in the answer – thereof the use of $\approx$. End user 02 has a maximum of 0.3 seconds ($T_{tx\_j} - T_{rx2}$) from registration until it actually receives the data stream.

End user 02 will receive data from both end user 01 (addressed to A_2) and end user 03 (addressed to A_1) at a certain time interval. This is due to the leave timer being larger than the join timer ($\Delta_{leave} > \Delta_{tx}$). Based on the calculations of the two latter paragraphs, end user 02 will receive two data streams in the time from 26.1 or 26.3 seconds to 26.6 seconds.

At the end, attribute A_1 should have no subscribers in the network, but A_2 should be declared by end user 02. The expected states at the end is listed in table 6.2.

6.3.3 The result

Part one

The result was almost as expected. The difference was in the convergence time when the second user (end user 02) sent a declaration. After two times the propagation delay, the applicant state of port 1 on the bridge 00 were AA instead of QA. The transition from AA to QA should happen instantly after a Join message is sent. The transmission was delayed for 0.3 seconds, thereof the transition from AA to QA was also delayed. The reason is as described in 5.3. There can be no more than three transmission during any period of $1.5 \times JoinTime = 0.3$ seconds. Because there is no processing delay or bandwidth delay, the three last messages was sent at the same time.

The next transmission opportunity occurred at 0.3 seconds later, and the stable state, QA, is reached.

Part two

The deregistration was completed successfully, consequently the expectations matches the result. The states of the participants ended as illustrated in figure 6.7e. The leave message was propagated as shown with the stippled arrow. The convergence time was two times the propagation delay and leave time as expected.

Part three

The states of the participants at the end was as expected. The deregistration propagated as described, and the time delays was satisfying. The time delay of the transmission opportunity when end user 02 sent a declaration for attribute A_2, was a minimum. End user 02 received traffic addressed to A_2 at ≈ 26.1 seconds.

```
DEBUG| 26.100000311001015| /test3/node02/multicastClient/| Received data addressed to
    group address: ff:ff:11:22:11:ff which is correct. Data type: IP packet. Content:
    DUMMY IP-packet
```

After the leave timer expired, the data addressed to A_1 and A_2 propagated as illustrated in figure 6.7f. If source pruning had been enabled, the traffic to A_1 would have been stopped at end user 03. This would, however, not show whether the bridge would perform as expected.

6.4 Test four – advanced test of MMRP

This part will test a more advanced topology of MMRP than the basic test in 6.3. The topology is taken from the standard [24, figure 10.2]. The objective is to get the same result as the standard illustrates.

6.4.1 The test set up

An overview of the topology with numbers on the nodes is shown in figure 6.8. Two nodes, number 16 and 1a, will declare an attribute. This is marked with a "D" and a double triangle in the figure. The arrows between the nodes illustrate how the declaration propagates through the network. The links have the same propagation delay as the previous tests. There are no loops in the physical topology.

There are three different types of nodes. The squared nodes are bridges, and are composed like figure 6.6a. The figure shows a component with three ports. The number of ports vary from bridge to bridge in the current test. Further, the triangles in figure 6.8 illustrate the end users.

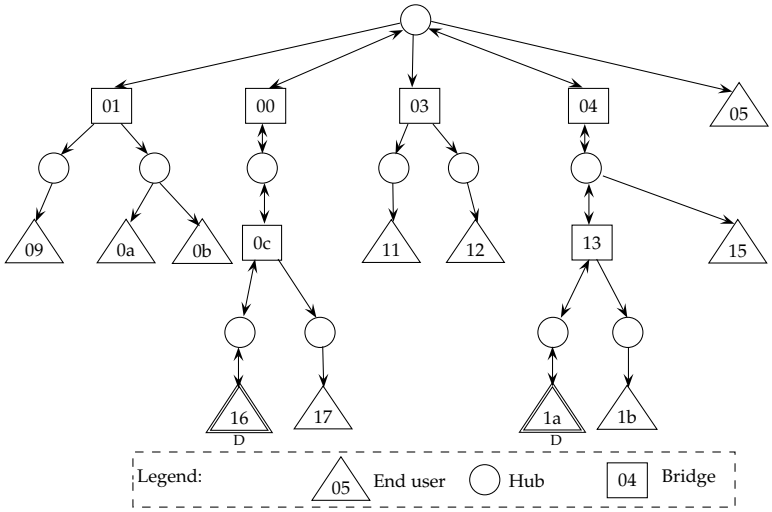


Figure 6.8: Topology of test number four. The arrows between the nodes illustrate both the logical topology, and the propagation of attribute declaration. End user 16 and 1a are marked with a "D" and double triangle. This means that it is declaring the given attribute.

These nodes are composed as illustrated in figure 6.6b. The last node type is illustrated with circles, and they represent hubs. The hubs are simple components that sends incoming data out all other ports than the port the traffic entered.

6.4.2 The expected result

The final states of each participant in the network should be as illustrated in [24, figure 10.2]. The applicant and registrar stable states for are shown in table 6.3. The details of transitions between states are described in the previous tests, and will no be repeated here.

The convergence time for the declarations are affected by the number of jumps and the transmission opportunity mechanism. The nodes which have the largest distance to 16 in terms of number of hops, are 1a and 1b. The number of hops is ten. The largest distance from 1a is also ten hops towards nodes 16 and 17. The transmission opportunity mechanism says that must not be sent more than three MRPDUs from a participant in an interval of $1.5 \times JoinTime = 0.3$ seconds. End user number 16 will declare the attribute at time ten seconds. At that time, no other declarations exists. Thereof, the participants are crated at the same time as the declaration propagates through the network, and the first messages sent are of the propagation if node 16's declaration. The convergence time is therefor only affected by the link propagation delay of 10 hops $\times$ 5 $\times$ 10^{-13} seconds $=$ 5 $\times$ 10^{-12} seconds. When end user number 1a declares the attribute, some traffic may be generated by the existing declaration due to the periodic timer or leave all timer. Worst case scenario is that each participant has just sent three messages. For

each hop, a delay of $1.5 \times JoinTime = 0.3$ seconds will be added. This results in a convergence time of 10 hops $\times$ $(0.3 + 5 \times 10^{-13})$ seconds $\approx$ 3 seconds.

	D	R	DR
Applicant state machine	QA	VO	QA
Registrar state machine	MT	IN	IN

Table 6.3: Stable states of the applicant and registrar.

The `operPointToPointMAC` parameter in a bridge should be set to `FALSE` when a port is connected to more than one other port. This is described in section 5.2. In the implementation, `operPointToPointMAC` is always `TRUE`, but the standard [24] states that the result should be the same. The difference is that the performance is not optimized.

6.4.3 The result

Some lines from the simulation output is pasted below. These messages are called debug messages. Each message has several parts divided by a "|". The first word of each line is "DEBUG", saying that it is a debug message. The second part of a line is a number, which is the time. The first line was printed at 10 seconds. The third part is says which component printed the message, and the last part is the debug message itself. The output below shows how the attribute declaration from end user 16 propagated to end user 0c. As said earlier, the propagation time of each link is 5×10^{-13}. From the lines belows, we see that each message is printed in $2 \times 5 \times 10^{-13}$ seconds interval. This is because there is a hub between each bridge or end user, and thereof two links. The reason why $2 \times 5 \times 10^{-13}$ seems to equal 9.98×10^{-13} instead of 1×10^{-12}, is the way Java calculates its Double data type. I.e. the offset is not due to the implementation. The last message coming from component `/test47node1a/macRelayEntity/` was printed when the attribute declaration arrived at end user 1a. The time it arrived is as expected 5×10^{-12}.

```
1 DEBUG| 10.0| /test4/node16/mmrp/| MMRP forwarded a unitDataIndication at the downPort if
      (0). Type: MRPDU (ff:ff:11:11:11:ff, event: JOIN\_MT, leaveAll: false).
2 DEBUG| 10.000000000000998| /test4/node0c/mmrp/| MMRP forwarded a unitDataIndication at
      the downPort if(0), state(FORWARDING). Type: MRPDU (ff:ff:11:11:11:ff, event: JOIN\
      _MT, leaveAll: false)
3 DEBUG| 10.000000000001997| /test4/node00/mmrp/| MMRP forwarded a unitDataIndication at
      the downPort if(0), state(FORWARDING). Type: MRPDU (ff:ff:11:11:11:ff, event: JOIN\
      _MT, leaveAll: false)
4 DEBUG| 10.000000000002995| /test4/node04/mmrp/| MMRP forwarded a unitDataIndication at
      the downPort if(1), state(FORWARDING). Type: MRPDU (ff:ff:11:11:11:ff, event: JOIN\
      _MT, leaveAll: false)
```

```
5 DEBUG| 10.000000000003993| /test4/node13/mmrp/| MMRP forwarded a unitDataIndication at
      the downPort if(2), state(FORWARDING). Type: MRPDU (ff:ff:11:11:11:ff, event: JOIN\
      _MT, leaveAll: false)
6 DEBUG| 10.000000000004992| /test4/node1a/macRelayEntity/| RECEIVED from port 0 with
      address 00:11:22:33:13:00 to address 01:80:c2:00:00:20 body type: drcl.ethernet.mmrp
      .MRPDU
```

The next output shows the attribute declaration from end user 1c towards end user 16. The result shows that there were no delay considering the transmission opportunity mechanism. The propagation time was the same as above.

```
1 DEBUG| 15.0| /test4/node1a/mmrp/| MMRP forwarded a unitDataIndication at the downPort if
      (0). Type: MRPDU (ff:ff:11:11:11:ff, event: JOIN\_MT, leaveAll: false)
2 DEBUG| 15.000000000000998| /test4/node13/mmrp/| MMRP forwarded a unitDataIndication at
      the downPort if(0), state(FORWARDING). Type: MRPDU (ff:ff:11:11:11:ff, event: JOIN\
      _IN, leaveAll: false)
3 DEBUG| 15.000000000001997| /test4/node04/mmrp/| MMRP forwarded a unitDataIndication at
      the downPort if(0), state(FORWARDING). Type: MRPDU (ff:ff:11:11:11:ff, event: JOIN\
      _IN, leaveAll: false)
4 DEBUG| 15.000000000002995| /test4/node00/mmrp/| MMRP forwarded a unitDataIndication at
      the downPort if(1), state(FORWARDING). Type: MRPDU (ff:ff:11:11:11:ff, event: JOIN\
      _IN, leaveAll: false)
5 DEBUG| 15.000000000003993| /test4/node0c/mmrp/| MMRP forwarded a unitDataIndication at
      the downPort if(1), state(FORWARDING). Type: MRPDU (ff:ff:11:11:11:ff, event: JOIN\
      _IN, leaveAll: false)
6 DEBUG| 15.000000000004992| /test4/node16/macRelayEntity/| RECEIVED from port 0 with
      address 00:11:22:33:0c:00 to address 01:80:c2:00:00:20 body type: drcl.ethernet.mmrp
      .MRPDU
```

At the end, all state machine was in the correct states. This concludes a functioning implementation of MMRP and its cooperative components.

Chapter 7

Experiments and results

This chapter uses the simulation environment and the MMRP/Bridge implementation to test some performance parameters of MMRP. The first section described the experiment and the expected result, while the last section looks at the results of the simulations.

7.1 MMRP timer experiment

As described in chapter 6 *Testing*, the convergence time of a MMRP attribute registration is the propagation, processing and bandwidth delays. However, when the participant for the current attribute has sent more than three messages in the last $1.5 \times JoinTime = 0.3$ seconds, a delay is added before sending the next PDU as described in chapter 5 *Implementation*.

The more attributes registered at a participant, the more PDUs are sent. Thereof, a higher risk of delayed PDUs. When the `operPointToPointMAC` parameter of a bridge port is set to `TRUE`, a PDU is sent on request with only one attribute per PDU. This is described further in chapter 5 *Implementation*. The current implementation has `operPointToPointMAC` set to `TRUE`, thereof only one attribute per MRPDU.

Besides the registration and deregistration propagations, there are two timers that triggers propagation periodically. The LeaveAll Timer which ensures that deregistrations are fully propagated, so that no registrations are kept alive without reason. The second timer is the Periodic timer which ensures that registrations are kept alive in case of packet loss.

The LeaveAll timer has a random length between 10 and 15 seconds. When it expires, a "sLA" action is triggered. The sLA action sends a MRPDU with the LeaveAll-field set to `TRUE`. With only one attribute per PDU, the number of PDUs sent when the LeaveAll timer expires are the same as the number of attributes at the participant. Furthermore, the sLA event triggers a "rLA!" event at the applicant and registrar state machines for all the attributes of the participant. The rLA! event starts the leave timer of 0.6 seconds on the registrar state machines.

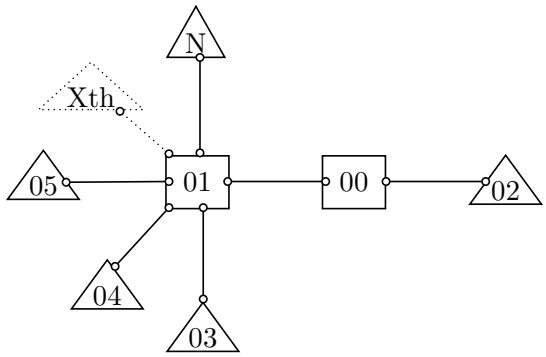


Figure 7.1: Topology of the experiment. Triangles illustrate clients, while the square nodes illustrate bridges. On the left side, bridge 01 is connected to $N-2$ clients. On the right side, bridge 00 is connected to one client.

The participant on the other side of the LAN will return a MRPDU with its current state. If the MRPDU is delayed from the participant due to the many attributes, this will cause an unwanted deregistration because the leavetime expires.

In addition to the PDUs generated by the LeaveAll event, the periodic timer expires every one second. If the applicant is in either the QA or the QP state, the periodic timer expiration triggers a single sending. The PDU contains the applicants current state.

This experiment will look at the timers in MMRP and how it affects the capacity of a participant. It is done by simulation. The capacity here is in terms of the number of attributes per participant.

7.1.1 The test set up

The topology is illustrated in figure 7.1. The left part of the figure shows $N-2$ clients connected to bridge number 01. $N-2$ is because the N is the total number of nodes in the network, an the first client connected to bridge 01 has number 03. On the right side, one client (number 02) is connected to bridge number 0. Client 02 will act as a content server sending multicast data traffic, simulation real time content. Each data stream has the same constant bit rate. All links has the same propagation delay, which is according to a 15 meter link. The bridges are composed as illustrated in figure 6.6a on page 69, expect from the number of interfaces. Bridge 01 has $N-2+1$ ports, and bridge 00 has two ports. The clients are composed as figure 6.6b on page 69. There are no processing delay, no bandwidth delay or queue on the interfaces.

The experiment executes several simulations. The parameters that is changed to see the effect on the timers are:

Number of clients. The clients connected to bridge 01 declares an attribute different from the other clients connected to the bridge. The number of clients connected to bridge 01 is

varied through the simulations.

LeaveTimer The standard [24] gives a default value of the LeaveTimer between 0.6 and 1.0 seconds. This value is adjusted to see what effect it has.

JoinTimer Based on the JoinTimer, MMRP decides how many MRPDUs may be sent per second. The default value is 0.2 seconds.

7.1.2 The expected result

The expirement is broken down in two parts. The first part of the experiment will use default values of the timers, but change the number of clients. The second part will alter the timer values, and change the number of attributes declared in the network.

Similarities of all the experiments are as follows. All the $(N-2)$ clients of bridge 01 will declare different attributes. In other words, there are $(N-2)$ different attribute declarations in the network. As explained above, when the "leavealltimer!" expires, the registrar changes state from "IN" to "LV" for all registered attributes at the participant. The registrar stays in the LV state for $LeaveTime$ seconds before the attribute is deregistered. All attributes declared from the other side of the LAN have $LeaveTime$ seconds to reregister at the participant by sending a JointMt or JoinIn message. The LeaveAll Timer is a random value between 10 and 15 seconds. The Periodic Timer triggers events that generates one PDU for each attribute with applicant state equal to QA or QP. The period of the Periodic timer is as mentioned, one second.

Part one – default timer values

Simulations executed in this part will have a JoinTimer with the default value 0.2 seconds, and the LeaveTimer is by default 0.6 seconds. The expected behavior is illustrated in figure 7.2. The figure shows an example where the clients already have registered their attributes. The first event illustrated in the figure is that the leavealltimer expires on bridge 00 that is linked to bridge 01. This triggers a "sLA" action which again starts the leave timer of all the attributes of the participant.

As seen in figure 7.2, it is expected that six MPRDUs is received by the participant at bridge 00 before the leave timer expires. Traffic addressed to those attributes that arrives after the leavetimer expires will no longer be forwarded on the port. In the illustration, this is the clients that subscribe to attribute number 4 and higher with a leave timer of 0.6 seconds. We can see from the illustration that with a leave timer of 1.0 seconds, two more attributes are reregistered.

In addition to what is illustrated in figure 7.2, the periodic timer expires every second for each port. This triggers sending of one PDU per applicant that is in the QA or QP state. With a maximum of 10 MRPDUs per second, a maximum of 10 applicants per participant will send their state on every periodic timer expiration.

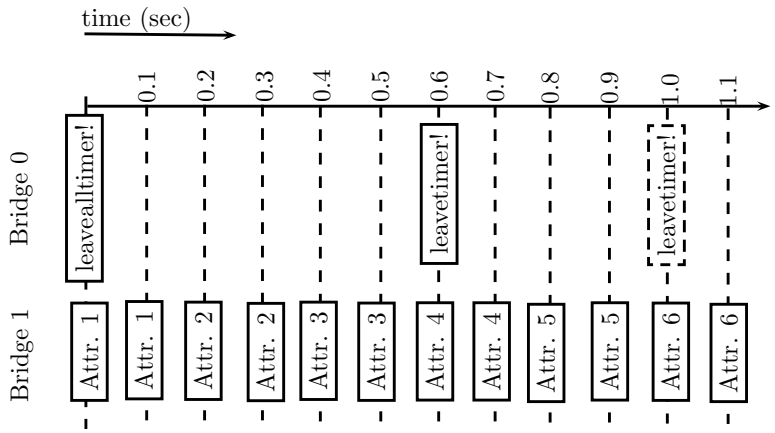


Figure 7.2: A leaveall timer expires. The leave timer is by default between 0.6 to 1.0 seconds. If the leave timer is 0.6 seconds, bridge 1 manages to send seven MRPDUs (four attributes), while with a leave timer of 1.0 seconds it sends 11 MRPDUs (six attributes). When the leavetimer is 0.6 seconds, the seventh MRPDU may not be received before the leave timer expires. The same is valid regarding the 11th MRPDU if the leavetimer is 1.0 seconds.

Four simulations will be executed. The difference is the number of attributes declared in the network. The first scenario has four clients, each declaring different attributes. The three other scenarios have five, six and seven clients where each client declares different attributes. The `MulticastServer` component in client 02 sends traffic towards all the registered attributes. The MulticastClient at the client side is connected to a `TrafficMonitor` component, which is further connected to a `Plotter` component. This way, the incoming data rate is registered an plotted.

The scenario with four different attributes declared in the network will handle the LeaveAll-Timer as illustrated in figure 7.2. The worst case scenario is if the participant has just sent three messages when the leaveall timer expires. Consequently, only two attributes manages to reregister bedore the registrars have transitioned to MT. A break in the traffic between 0.1 and 0.3 seconds for the clients which registered the attributes is expected in this case.

The second scenario has five attributes, the third has six and finally the fourth has seven attributes declared in the network. It is expected that all these scenarios will suffer from a faulty MT state in registrars during the leaveall timer expirations. This means that some attributes are deregistered on bridge 00. This will show on the data rate plot from the clients. Those attributes that are deregistered will reregister as soon as the MRPDU from bridge 01 arrives at bridge 00.

Part two – altered timer values

In this part, the value of the LeaveTimer and the value of the JoinTimer is altered. By this it is expected to achieve better scalability in terms of more attribute declarations per participant. A

`TrafficMonitor` component is connected to the `LLC/mmrp@up` port of bridge 00. This will show the data rate of incoming MRPDUs on the bridge. Three different configurations are used. The expected results for each of them are as follows.

JoinTime = 0.2s, LeaveTime = 1.0s, number of attributes = 7. This configurations differs from the last scenario of part one by an increased LeaveTime. Note that it is still a default value as the standard states a value between 0.6 and 1.0 seconds. By increasing the LeaveTime, we see by figure 7.2 that six attributes may be reregistered before the timer expires. Consequently, one client looses its data stream. As seen in figure 7.2, attribute number 7 will be reregistered at 1.2 seconds after the LeaveAllTimer expired.

JoinTime = 0.1s, LeaveTime = 1.0s, number of attributes = 10. The number of attributes declared in the network is increased to ten, and the JoinTime is decreased to 0.1 seconds. By reducing the JoinTime by 50%, the expected MRPDUs per second rate will double. This means that each participant may send 20 MRPDUs per second. Consequently, 20 MRPDUs may arrive before the LeaveTime expires. All attributes should be reregistered, and there should not occur a break of any of the data streams.

JoinTime = 0.1s, LeaveTime = 1.0s, number of attributes = 11. The JoinTime is still 0.1 seconds giving the 20 MRPDUs per second rate. The difference is that the number of attributes declared in the network is increased to 11. Eleven attributes produces 22 MRPDUs, which in turn requires 1.1 seconds to send with the given JoinTime. Consequently, one out of the 11 reregistrations should arrive after the LeaveTime expires.

When the JoinTime is reduced, more MRPDUs may be sent per second. This should lead to a higher bandwidth demand by MMRP.

7.2 Results

This section is broken down in two parts, representing the parts described in the last section. First, the results of simulation with default timer values, next the results of the simulations with altered timer values.

Note that the results is based on this specific implementation. The implementation may be error-prone due to wrong interpretation of the protocol. If this is the case, the reflections around the result may differ from the protocols intentions.

7.2.1 Part one

The figure 7.3 on page 87. show the results. The plots show the data rate of the received traffic at the clients. Where the curves drop, the clients looses its data stream due to the behavior

described in the expected results. The reason for the drops is the expiration of the LeaveAll timer.

Figure 7.3a shows the results with four attributes declared in the network. The result is as expected with no long breaks in the data streams. Figure 7.3b, on the other hand, shows that for each time the LeaveAll timer expires, one attribute is deregeistered. Consequently, the data stream for the given attribute is not forwarded from bridge 00, and it is shows by a drop in the received data rate in the plot. Further, both figure 7.3c and 7.3e indicates the same behavior. With six attributes declared in the network, two attributes are deregistered when the LeaveAll timer expires. Again, with seven attributes declared in the network, three attributes are deregistered. A detail of the received data stream is shown in figure 7.3d and 7.3f. The detail shows that each stream is regeregistered with approximately 0.2 seconds between, which is the time used to send two MRPDUs, i.e. one attribute. This matches the expectations shown in figure 7.2.

The simulation output confirms the results from the plots. A part of the output is shown below. Each single output message is on a new line. A message is further broken down in different parts divided by a "|" character. The following describes the first line of the output below.

GARBAGE The type of output message.

45.09000000000199 The current time in the simulation execution context.

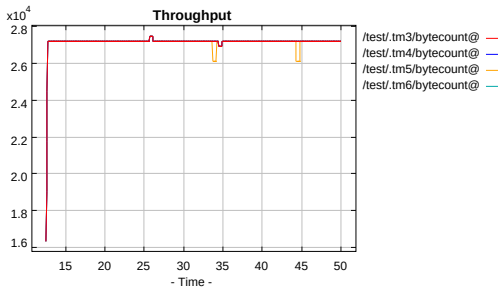
/test/node00/mmrp/ The component that prints the message.

drcl.ethernet.mmrp.Scheduler@16c1857 The Java object of message origin.

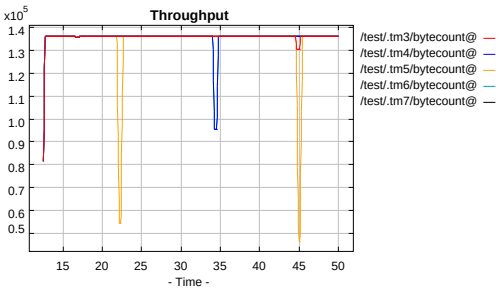
LeaveAllStateMachine port 0, vlan 1. Event = RLA The message itself. Here, the LeaveAllStateMachine has passed the message on a RLA event. Port and vlan is also printed.

```
1 GARBAGE| 45.09000000000199| /test/node00/mmrp/| drcl.ethernet.mmrp.Scheduler@16c1857|
      LeaveAllStateMachine port 0, vlan 1. Event = RLA
2 GARBAGE| 45.09000000000199| /test/node00/mmrp/| drcl.ethernet.mmrp.Scheduler@16c1857|
      LeaveAllStateMachine port 0, vlan 1. Action = START_LEAVEALLTIMER
3 GARBAGE| 45.09000000000199| /test/node00/mmrp/| drcl.ethernet.mmrp.Scheduler@16c1857|
      Registrar. Port 0, vlan 1, attribute: ff:ff:11:03:11:ff. currentState=IN, EVENT: RLA
4 GARBAGE| 45.09000000000199| /test/node00/mmrp/| drcl.ethernet.mmrp.Scheduler@16c1857|
      Registrar. Port 0, vlan 1, attribute: ff:ff:11:03:11:ff. switched to state: LV,
      EVENT: RLA
```

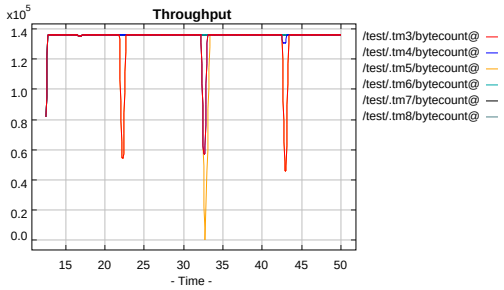
The output above shows the attribute "ff:ff:11:03:11:ff" transitions to registrar state LV due to a reception of a LeaveAll message. This further causes the leavetimer to start. The same results are found for every attribute declared in the network.



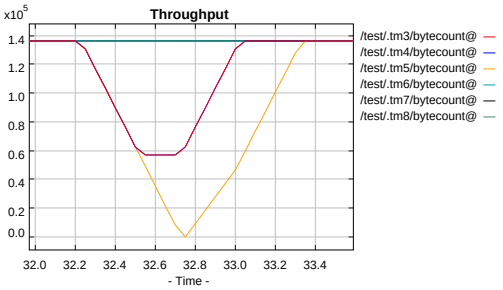
(a) A scenario with four attributes declared in the network.



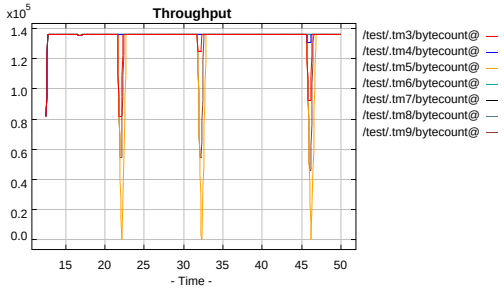
(b) A scenario with five attributes declared in the network.



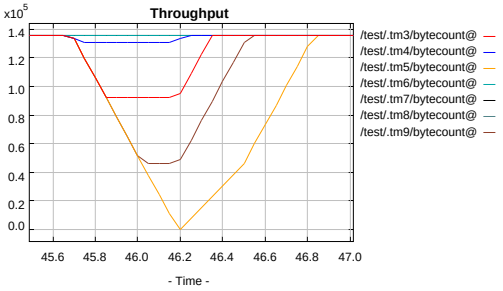
(c) A scenario with six attributes declared in the network.



(d) A detail of figure 7.3c between the time 32 and 33.5 seconds.

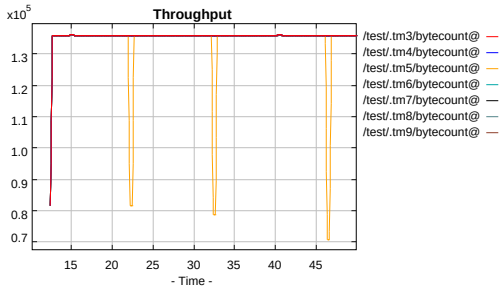


(e) A scenario with seven attributes declared in the network.

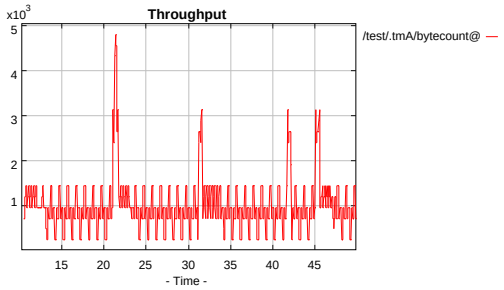


(f) A detail of figure 7.3e between the time 45.5 and 47 seconds.

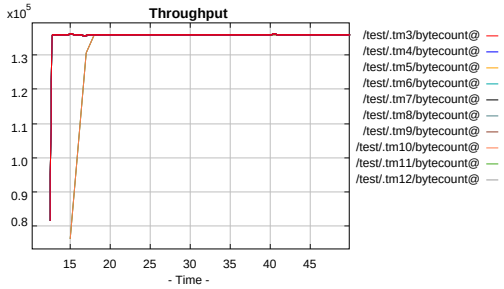
Figure 7.3: Plots from different scenarios. All scenarios have a JoinTimer of 0.2 seconds, and a LeaveTimer of 0.6 seconds. The difference between the plots is the number of attributes declared in the network. Figure 7.1 illustrates the topology of the scenarios. The plots shows the data rate of the received data stream (not MRPDUs) at the clients. Each client declares a single attribute, different from the other clients. Figure 7.3a and 7.3b have respectively four and five attributes declared in the network. Figure 7.3c shows the result of the whole simulation with six attributes, while 7.3d shows a detail of the plot between 32 and 33.5 seconds. Figure 7.3e and 7.3f does the same, where the detail is of the time 45.5 and 47 seconds.



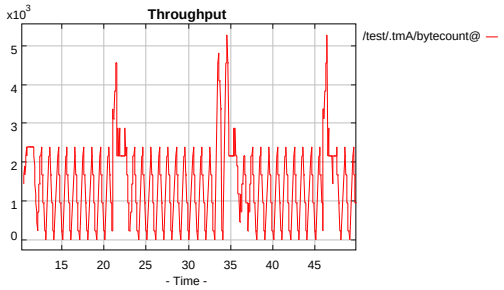
(a) JoinTime = 0.2 s, LeaveTime = 1.0s, number of attributes = 7. Data rate of the received stream at the clients.



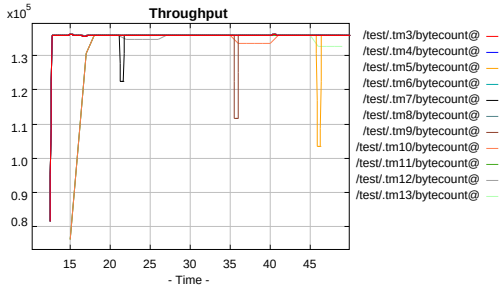
(b) JoinTime = 0.2 s, LeaveTime = 1.0s, number of attributes = 7. Data rate of received MRPDUs at bridge 00.



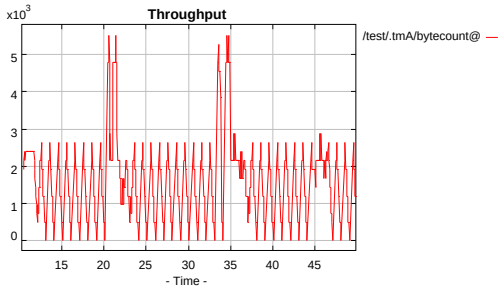
(c) JoinTime = 0.1 s, LeaveTime = 1.0s, number of attributes = 10 Data rate of the received stream at the clients..



(d) JoinTime = 0.1 s, LeaveTime = 1.0s, number of attributes = 10. Data rate of received MRPDUs at bridge 00.



(e) JoinTime = 0.1 s, LeaveTime = 1.0s, number of attributes = 11. Data rate of the received stream at the clients.



(f) JoinTime = 0.1 s, LeaveTime = 1.0s, number of attributes = 11. Data rate of received MRPDUs at bridge 00.

Figure 7.4: Plots from different simulation scenarios. The two plots that are on each line come from the same simulation. The on to the left shows the data rate of the received data stream at each client. The legend gives the path to the traffic monitor component. The right plot shows the data rate of the received stream of MRPDUs at the MMRP component at bridge 00. The difference between the scenarios is the number of clients, and the timer values. The topology of the scenarios is illustrated in 7.1.

```
1 GARBAGE| 45.690000000001994| /test/node00/mmrp/| drcl.ethernet.mmrp.Scheduler@16c1857|
      Registrar. Port 0, vlan 1, attribute: ff:ff:11:03:11:ff. currentState=LV, EVENT:
      LEAVETIMER
2 GARBAGE| 45.690000000001994| /test/node00/mmrp/| drcl.ethernet.mmrp.Scheduler@16c1857|
      Registrar. Port 0, vlan 1, attribute: ff:ff:11:03:11:ff. switched to state: MT,
      EVENT: LEAVETIMER
```

At 0.6 seconds (= LeaveTime) later, the registrar for the current attribute is triggered by the leavetimer! event. Consequently, the registrar transitions to the MT (empty) state. At this point, the bridge will not forward data to this address out the port.

```
GARBAGE| 45.850000000000975| /test/node00/mmrp/| drcl.ethernet.mmrp.Scheduler@1e0f790|
    Registrar. Port 0, vlan 1, attribute: ff:ff:11:03:11:ff. switched to state: IN,
    EVENT: RJOINMT
```

Next, at approximately 0.15 seconds later. The attribute is once again registered at the participant. From this point on, the traffic towards the current attribute is forwarded to the client. The output described above represents the findings of the incoming data rate on the client as shown in figure 7.3.

7.2.2 Part two

The plots from the simulation is shown in figure 7.4 on page 88. A total of three simulations were executed as described in the experiment section. In addition to the plots of received data rate at the clients, a plot of the received data rate at the MMRP component is presented for each simulation. The higher data rate at the MMRP components means more MRPDUs received. All three simulations clearly shows that the periodic timer expires with a peak of the MRPDU data rate each second. Further, when the LeaveAll timer expires, the plot shows a distinct increase in the amount of received MRPDUs. The results for this part is broken down to each simulation, i.e. three parts, presented in the following list.

JoinTime = 0.2s, LeaveTime = 1.0s, number of attributes = 7. The expectations was that one attribute should be deregistered when the LeaveAllTimer expires. Figure 7.4a confirms this.

JoinTime = 0.1s, LeaveTime = 1.0s, number of attributes = 10. Both the expected outcome of the attribute reregistration and the increased data rate of MRPDUs are confirmed respectively by figure 7.4c and 7.4d.

JoinTime = 0.1s, LeaveTime = 1.0s, number of attributes = 11. This simulation also confirms the expected results. Figure 7.4e shows the received throughput by the clients.

One attribute is deregistered when the LeaveTime expires. Figure 7.4f shows the rate of MRPDUs received at bride 00, which also follows the pattern explained in the expectations.

Chapter 8

Discussion

This chapter will discuss the results from the simulations, and discuss topics relevant to the problem. The first part looks at performance parameters of MMRP. The second part describes the problems around mapping IP multicast addresses to MAC multicast addresses.

8.1 MMRP performance parameters

This section looks at the performance parameters of MMRP. The section is divided in three, beginning with the scalability issues due to the timer mechanisms. Next, the convergence time of the attribute declaration is discussed and finally the MMRP bandwidth demand.

8.1.1 Scalability of a single Participant

Scalability here is in terms of the number of attributes a single participant can handle. As seen in the simulation results from chapter 7 *Experiments and results*, the timers constraints how many attributes a participant can handle. With a LeaveTimer of 0.6 seconds and a JoinTime of 0.2 seconds, a maximum of seven attributes is reregistered before the LeaveTimer expires, after a LeaveAllTimer expired. An applicant needs to send its current state two times upon a LeaveAll event. This is to ensure that the message is received on the other side of the LAN during situations with packet loss. Because the current implementation sends these two MRPDUs successively, only four attributes are reregistered. Due to the fact that one message is sufficient for a reregistration, the MMRP participant should delay the second message. Consequently, all applicants sends one message before any of the applicants have sent two. Given the scenario with LeaveTimer of 0.6 seconds and a JoinTime of 0.2 seconds, seven attributes will be reregistered before the LeaveTimer expires. This opposed to the four reregistrations with the current implementation.

To optimize the performance, one may alter the timer values. The results when increasing the LeaveTime to 1.0 seconds, was that six attributes reregistered before the timeout. And by

decreasing the JoinTime to 0.1 seconds, ten attributes was reregistered. The effects of increasing the LeaveTimer is first the obvious that more attributes may reregister. Next, it may also affect the bandwidth usage in a network. If a client frequently changes stream with registrations and deregistrations, several streams are forwarded to the client within the time of *LeaveTime* after the deregistration. On the other hand, the data streams of the different attributes may not require much bandwidth each, and therefore this may not become a problem.

The standard [24] states that one may alter the timer value, and still the MMRP will act correctly. Therefore, the timers may be changed in order to utilize for a certain scenario.

With five attributes in the network, one attribute was deregistered, and consequently traffic not forwarded, for 0.15 seconds before it was reregistered. For each successive attribute that fails to deregister before the LeaveTimer, it takes 0.2 seconds more before the attribute reregsiters. This is shown in figure 7.3f. For some applications, a break 0.15 seconds may accepted.

The periodic timer is by default 1.0 seconds. If the JoinTime is 0.2 seconds, each participant will only manage to send ten MRPDUs between each time the timer expires. However, it only request to send the current state if the applicant is the declaring the attribute and have sent or received two messages since the last Leave or LeaveAll message. Those applicants that are already triggered by the periodic timer will go over in another state. Consequently, only ten applicants will ask for transmission opportunity at each periodic event, and no congestion of periodic messages occurs. In addition, it do not lead to any problem regarding the LeaveAll event. This is because the messages from the periodic event sends the current state of the attributes applicant and registrar. If the participant is declaring, the Leave timeout will be canceled.

8.1.2 Convergence time

The convergence time of an attribute propagation depends on the number of nodes in the network and the number of attributes on the same VLAN. Given that a participant is declaring five attributes. If the periodic timer expires just before a declaration of a new attribute arrives, the declaration is delayed a time based on the JoinTime and number of attributes already declared in the network. This may happen through out the network, increasing the convergence time. On the other hand, it may propagate through the network without a delay.

If the attribute is already registered by another client on the same bridge, the attribute propagation do not need to forwarded longer than the first bridge. The MMRP component of the bridge will register the attribute at the incoming port. Therefore the convergence is complete in the time it takes to send the MRPDU to the bridge and process it on the bridge.

8.1.3 MMRP bandwidth demand

Given the MAC frame format in table 2.4 and the MRPDU format in figure 4.1, one MRPDU encapsulated in an EthernetFrame is 72 bytes. This is due to the minimum payload of a MAC Frame is 46 bytes. With the default timer value set, a maximum bandwidth usage of $46 \times 10 = 460$ bytes per second is used. Pure MRPDU rate is 112 bits $\times$ 10 = 140 bytes per second. With a JoinTime of 0.1, the MRPDU rate is 280 bytes per second. Relative to the links capacity in Ethernet, this is small data rate.

8.2 Bandwidth savings of mapping from layer 3 to layer 2 multicast

A IP multicast packet is mapped to a group MAC address. The default fowarding of MAC frames with a group MAC address is on all ports except the incoming port. With several end stations on a LAN subscribing to different multicast streams, each end stations receives all the different streams that are subscribed to. If the Ethernet Bridges knows where to send the Ethernet group addresses, this will be avoided. In addition, with the use of Multiple MAC Registration Protocol, an end station that sends content on the network may use the principle of source pruning discribed in chapter 4 *Multiple MAC Registration Protocol*.

8.3 Mapping of addresses from IP multicast to MAC multicast

As described in chapter 2 *Technology background*, there are more Multicast addresses available in the IP version 4 address range than the MAC address range. If the network manager also controls the content distributed on the network, one may avoid using two IP version 4 multicast addresses that maps to the same MAC multicast address. However, if the network manager do not control the content, it might be desirable to add a function which controls the mapping and avoids equal mapping.

Chapter 9

Conclusion

This thesis has presented the work of implementing the MMRP protocol in the J-SIM environment. In addition, a theoretical study and a simulation is presented to cover more aspects of multicast in Ethernet. This chapter presents the conclusion from the simulation, and a section about future work related to the problem.

The abstraction of MMRP and bridge functionality was implemented, and the tests of chapter 6 and experiments of chapter 7 verified the implementation. The results were in accordance with the expectations.

The IP version 4 multicast address range exceeds the size of MAC multicast address range. Therefor, it is desirable to either control the IP multicast addresses used, or control the mapping process.

Ethernet Frames that encapsulate IP multicast packets have group address in the destination field. The frames with group address in the destination field are broadcasted as long as no filtering service is enabled in the bridge. By enabling extended filtering services, and map the multicast users to the Ethernet topology, it could save bandwidth resources.

The performance of the Multiple MAC Registration protocol constraints the number of attributes per VLAN. However, timers may be altered to optimize for a given amount of attributes. MMRP uses small amounts of data rate relative to the link capacity it is design for.

9.1 Future work

This thesis has focused on the implementation of MMRP and some basic bridge functionality. Due to the time limitation of the thesis, the complete system for the problem could not be completed. I.e. the IGMP snooping functionality is missing. Further, more scenarios should be tested and the MMRP timers should studied to give some good arguments for optimization in different environments. The rest of this section presents more aspects that could be study in a

future work.

A study of the possibilities with MMRP in a VLAN environment should be enlightened. Adding VLAN to the current implementation requires changes in the MMRP, MACRelay, RSTP and EthertMAC components. This could limit the number of attributes per participant, but at the same time increase the bandwidth demand.

The RSTP implementation has a bug regarding restoration time on link break. The MMRP implementation supports handling of link break, but due to the RSTP bug, a MMRP restoration test is not completed.

Based on the data collection of a IP Televison network in chapter 5 *Implementation*, it would be useful to see the performance of MMRP in such an environment. There are several aspects with both dependability and performance that could be studied.

The study of IGMP snooping together with MMRP multicast utilization in a Ethernet network involves aspects like placement of the snooping node, differences between IGMP versions and CPU demands. In addition, the same aspects could be studied in a IP version 6 and MLD context.

Bibliography

[1] IANA Considerations and IETF Protocol Usage for IEEE 802 Parameters, RFC5342. http://tools.ietf.org/html/rfc5342.

[2] IEEE Std 802.3-2008 part 3: CSMA/CD access method and physical layer. http://ieeexplore.ieee.org/servlet/opac?punumber=4726157.

[3] Internet Group Management Protocol, version 1, RFC1112. http://tools.ietf.org/html/rfc1112.

[4] Internet Group Management Protocol, version 2, RFC2236. http://tools.ietf.org/html/rfc2236.

[5] Internet Group Management Protocol version 3, RFC3376. http://tools.ietf.org/html/rfc3376.

[6] Internet Protocol, RFC791. http://tools.ietf.org/html/rfc791.

[7] Internet Protocol version 6, RFC2460. http://tools.ietf.org/html/rfc2460.

[8] Multicast Listener Discovery, version 2, RFC3810. http://tools.ietf.org/html/rfc3810, last visited 12.12.2009.

[9] IEEE Standard for Local and Metropolitan Area Networks: Overview and Architecture. Amendment 2: Registration of Object Identifiers. *IEEE Std 802-2001 (Revision of IEEE Std 802-1990)*, page 0_1, 2002.

[10] IEEE Standard for Information Technology–Telecommunications and Information Exchange Between Systems–Local and Metropolitan Area Networks–Specific Requirements Part 3: Carrier Sense Multiple Access With Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications - Section One. *IEEE Std 802.3-2008 (Revision of IEEE Std 802.3-2005)*, pages c1 –597, 26 2008.

[11] IEEE Standard for Local and Metropolitan Area Networks Virtual Bridged Local Area Networks Corrigendum 1: Corrections to the Multiple Register Protocol. *IEEE Std 802.1Q-2005/Cor1-2008 (Corrigendum to IEEE Std 802.1Q-2005)*, pages C1 –6, 15 2008.

[12] IP and Ethernet solve mobile backhaul bottlenecks. Internet, 10 2008. http://www.ericsson.com/solutions/news/2008/q4/081029-mobile-backhaul.shtml.

[13] Connection-Oriented Ethernet vs. MPLS-TE: An Ethernet Transport Layer TCO Comparison. Internet, 3 2009. http://www.nspllc.com/.

[14] Mobildata per 3. kvartal 2009, 12 2009. http://www.npt.no/ . Last visited 26.04.2010.

[15] TCL/Java. http://tcljava.sourceforge.net/docs/website/index.html, 02 2010. Last visited: 20.02.2010.

[16] TV-seing 2009. http://www.tns-gallup.no/arch/_img/9090529.pdf Last visited 07.03.2010, 2010.

[17] Andy Sutton. Considering the evolution to packet backhaul over microwave. Presentation at conference, 9 2008.

[18] Carmichael, L.S. and Ghani, N. and Rajan, P.K. and O'Donoghue, K. and Hott, R. Characterization and comparison of modern layer-2 Ethernet survivability protocols. pages 124 – 129, March 2005.

[19] Cha, Meeyoung and Rodriguez, Pablo and Crowcroft, Jon and Moon, Sue and Amatriain, Xavier. Watching television over an IP network. In *IMC '08: Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*, pages 71–84, New York, NY, USA, 2008. ACM.

[20] Hung-ying Tyan. *Design, realization and evaluation of a component-based compositional software architecture for network simulation.* PhD thesis, The Ohio State University, 2002.

[21] D. Hunt and N. Shah. IP/MPLS in the mobile radio acces network (RAN). Presentation at conference, 2009.

[22] IEEE. 802.1DTM IEEE Standard for Local and metropolitan area networks Media Access Control (MAC) Bridge. Electronic, http://ieeexplore.ieee.com/, 2004.

[23] IEEE. IEEE Standard for Local and metropolitan area networks Virtual Bridged Local Area Networks. Electronic, http://ieeexplore.ieee.com/, 2005.

[24] IEEE. IEEE Standard for Local and metropolitan area networks-Virtual Bridged Local Area Networks Amendment 7: Multiple Registration Protocol. Electronic, http://ieeexplore.ieee.com/, 2007.

[25] Internet Assignet Numbers Authority. Ethernet numbers. http://www.iana.org/assignments/ethernet-numbers, 03 2010. Last visited 02.05.2010.

[26] J-SIM homesite. Tutorial. http://j-sim.cs.uiuc.edu/, 09 2009. Last visited 04.09.2009.

[27] John D. Day, Hubert Zimmerman. The OSI Reference Model. *Proceedings of the IEEE*, 71(12):1334–1340, 12 1983.

[28] Mehta, S. and Ullah, Niamat and Kabir, Md. Humaun and Sultana, Mst. Najnin and Kwak, Kyung Sup. A Case Study of Networks Simulation Tools for Wireless Networks. In *AMS '09: Proceedings of the 2009 Third Asia International Conference on Modelling & Simulation*, pages 661–666, Washington, DC, USA, 2009. IEEE Computer Society.

[29] S. Norway. Mediebruk til ulike tider. In *Norsk mediebarometer 2008*, page 70. Statistic Norway, Statistic Norway, 2009. http://www.ssb.no/emner/07/02/30/medie/sa106/sa_106.pdf Last visited 07.03.2010.

[30] Peder J. Emstad and Poul E. Heegaard and Bjarne E. Helvik and Laurent Paquereau. *Dependability and performance in information and communication systems - fundamentals.* Tapir Akademisk Forlag, 2008.

[31] Rajesh Chundury. Mobile broadband backhaul: Addressing the challenge, 8 2008. http://ericsson.com/ericsson/corpinfo/publications/review/2008_03/files/Backhaul.pdf.

[32] Sfeir, E. and Pasqualini, S. and Schwabe, T. and Iselt, A. Performance evaluation of ethernet resilience mechanisms. pages 356 – 360, may 2005.

[33] Stanislav Milanovic. Case study for unified backhaul performance optimization. *Journal of Computers*, 2(10):38–44, 12 2007.

[34] Statistics Norway. Official site. http://www.ssb.no/, 03 2010. Last visited 08.03.2010.

[35] TNS Gallup Norway. Official site. http://www.tns-gallup.no/, 03 2010. Last visited 08.03.2009.

[36] Toy Ove Nilssen. Vil mobilt bredbånd ta over for alt?, 1 2009. Norsk UMTS forum, http://www.umts.no/.

Appendix A

TCL methods

This appendix shows the method library developed for easy creation of new network topologies. There are three parameters needed to create a new topology:

Delay The delay on each link in seconds.

Name Name of the scenario.

Topology string A String representing the topology. Each node in the network is divided by an empty space. For each node, it must be character first then the node number of each neighbor separated by ":". The character can be **b** for bridge, **c** for an end station and **h** for a hub. An example of a four node topology with a bridge connected to three end stations is `b:1:2:3 c:0 c:0 c:0`.

When the three variables are set, call `create $runName $topology $delay` to create the scenario.

```
1       # ###################### #
2       # #### FUNCTIONS #### #
3       # ###################### #
4
5       proc createNode { number interfaces } {
6               set interfaces [expr $interfaces - 1];
7               set node$interfaces [mkdir drcl.comp.Component node$number];
8               cd node$number;
9               set bridgeAddress "00:11:22:33:$number:00";
10
11              mkdir [java::new drcl.ethernet.MACRelayEntity] macRelayEntity;
12              set mmrp [mkdir [java::new drcl.ethernet.mmrp.MMRP $bridgeAddress] mmrp];
13              set rstp [mkdir [java::new drcl.ethernet.rstp.Rstp [expr 0x$number]
                    $interfaces] rstp];
```

```
14              set llc [mkdir [java::new {drcl.ethernet.LLC int} $interfaces] llc];
15              for {set j 0} {$j < $interfaces} {incr j} {
16                      mkdir [java::new {drcl.ethernet.EthernetMAC int} $j] if$j;
17              }
18              [! macRelayEntity] setInterfacePorts $interfaces;
19              [! macRelayEntity] setBridgeAddress $bridgeAddress;
20              [! macRelayEntity] setRstp [! rstp];
21              cd ..
22              exposePorts $number $interfaces
23              connectNodeComponents $number $interfaces
24      }
25
26      proc exposePorts { number interfaces } {
27              for {set j 0} {$j < $interfaces} {incr j} {
28                      ! node$number exposePort [! node$number/if$j/down@] [java::field
                            [! node$number] PortGroup_DEFAULT_GROUP] $j
29              }
30
31      }
32
33      proc connectNodeComponents { number interfaces } {
34              for {set j 0} {$j < $interfaces} {incr j} {
35                      connect -c node$number/if$j/.mac@up -and node$number/
                            macRelayEntity/$j@down
36                      connect -c node$number/if$j/.llc@up -and node$number/llc/$j@down
37              }
38              connect -c node$number/rstp/down@ -and node$number/llc/.rstp@up
39              connect -c node$number/mmrp/down@ -and node$number/llc/.mmrp@up
40              connect -c node$number/mmrp/.portStateQuery@service -and node$number/
                    macRelayEntity/.portStateQuery@service
41              connect -c node$number/mmrp/.filteringDatabaseConfig@service -and
                    node$number/macRelayEntity/.filteringDatabaseConfig@service
42              connect -c node$number/mmrp/.portRoleChangeEvent@service -and node$number/
                    rstp/.portRoleChangeEvent@service
43              connect -c node$number/mmrp/.portStateChangeEvent@service -and node$number
                    /rstp/.portStateChangeEvent@service
44      }
45
46      proc createLink { number1 if1 number2 if2 delay } {
47              set hexNumber1 [format %x $number1];
48              set hexNumber2 [format %x $number2];
```

```
49              if [expr $number1 < 16] {
50                      set number1 "0$hexNumber1";
51              } else {
52                      set number1 "$hexNumber1";
53              }
54              if [expr $number2 < 16] {
55                      set number2 "0$hexNumber2";
56              } else {
57                      set number2 "$hexNumber2";
58              }
59              set link$number1$number2 [mkdir drcl.inet.Link link$number1$number2]
60              [! link$number1$number2] setPropDelay $delay; # 300 ms
61              [! link$number1$number2] attach [! node$number1/$if1@] [! node$number2/
                    $if2@]
62      }
63
64      proc createLinks { topology delay } {
65              for {set i 0} {$i < [llength $topology] } {incr i} {
66                      for {set j 0} {$j < [llength [lindex $topology $i]]} {set j [expr
                            $j+3]} {
67                              if {[expr $i < [lindex [lindex $topology $i] [expr $j+1]]]}
                                    {
68                                      createLink $i [lindex [lindex $topology $i] $j] [
                                            lindex [lindex $topology $i] [expr $j+1]] [
                                            lindex [lindex $topology $i] [expr $j+2]] $delay
                                            ;
69                              }
70                      }
71              }
72      }
73
74      proc createLinksFromString { topology delay } {
75              set tmpList [split $topology " "]
76              set tmpIfCounter [list]
77              for {set i 0} {$i < [llength $topology] } {incr i} {
78                      lappend tmpIfCounter 0
79              }
80              set tmpTopology [list]
81              set tmpNodeTopology [list]
82              set tmpCounter 0
83
```

```
84              for {set i 0} {$i < [llength $topology] } {incr i} {
85                      set tmpNeighborsList [list]
86                      set tmpNeighborsShortList [split [lindex $topology $i] ":"]
87                      for {set j 1} {$j < [llength $tmpNeighborsShortList] } {incr j} {
88                              set neighbor [lindex $tmpNeighborsShortList $j]
89                              if { [expr $neighbor > $i ] } {
90                                      lappend tmpNeighborsList [list [lindex $tmpIfCounter
                                             $i] $neighbor [lindex $tmpIfCounter $neighbor]]
91                                      set tmpIfCounter [lreplace $tmpIfCounter $i $i [expr
                                             [lindex $tmpIfCounter $i]+1]]
92                                      set tmpIfCounter [lreplace $tmpIfCounter $neighbor
                                            $neighbor [expr [lindex $tmpIfCounter $neighbor
                                            ]+1]]
93                              }
94                      }
95                      for {set j 0} {$j < [llength $tmpNeighborsList] } {incr j} {
96                              set tmpNodeTopology [concat $tmpNodeTopology [lindex
                                    $tmpNeighborsList $j]]
97                      }
98                      lappend tmpTopology $tmpNodeTopology
99                      set tmpNodeTopology [list]
100             }
101
102             createLinks $tmpTopology $delay
103     }
104
105     proc createSender { number destination } { # Creates a packet sender, has nothing
            to do with MMRP. Sends packet to a given MAC address
106             mkdir [java::new {drcl.inet.application.PacketSender String String} "00
                    :11:22:33:$number:00" "00:11:22:33:$destination:00"] node$number/
                    source
107             connect -c node$number/source/down@ -and node$number/macRelayEntity/down@
108     }
109
110     proc createMmrpClient { nodeNumber } {
111             [! node$nodeNumber/mmrp] setSimpleApplicant true; # OOOPS! Avoid calling
                    this when one want the registrar state machine to be included.
112
113             mkdir [java::new {drcl.inet.application.MulticastClient String} "00
                    :11:22:33:$nodeNumber:00"] node$nodeNumber/multicastClient;
```

```
114             connect -c node$nodeNumber/multicastClient/.mmrp@service -and
                    node$nodeNumber/mmrp/.client@service
115             connect -c node$nodeNumber/multicastClient/down@ -and node$nodeNumber/llc/
                    .ip@up
116
117             mkdir [java::new {drcl.inet.application.MulticastServer String String} "00
                    :11:22:33:$nodeNumber:00" "11:11:11:33:11:00"] node$nodeNumber/
                    multicastServer
118             connect -c node$nodeNumber/multicastServer/down@ -and node$nodeNumber/llc/
                    .ip@up
119     }
120
121     proc createMmrpServer { nodeNumber } {
122             mkdir [java::new {drcl.inet.application.MulticastServer String String} "00
                    :11:22:33:$nodeNumber:00" "11:11:11:33:$destination:00"]
                    node$nodeNumber/multicastServer
123             connect -c node$nodeNumber/multicastServer/down@ -and node$nodeNumber/if0/
                    up@
124     }
125
126     proc createHub { number interfaces } {
127             set interfaces [expr $interfaces - 1];
128             set node$number [mkdir drcl.comp.Component node$number];
129             cd node$number;
130             set hub [mkdir [java::new drcl.ethernet.Hub $interfaces] hub];
131             cd ..
132             for {set j 0} {$j < $interfaces} {incr j} {
133                     ! node$number exposePort [! node$number/hub/$j@down] [java::field
                            [! node$number] PortGroup_DEFAULT_GROUP] $j
134             }
135     }
136
137     proc createScenario { topology delay } {
138             set prefixBridge "b"
139             set prefixHub "h"
140             set prefixClient "c"
141             set tmpNodes [split $topology " "]
142             for {set i 0} {$i < [llength $tmpNodes] } {incr i} {
143                     set number "";
144                     set hexNumber [format %x $i];
145                     if [expr $i < 16] {
```

```
146                             set number "0$hexNumber";
147                     } else {
148                             set number "$hexNumber";
149                     }
150                     set tmpNeighbors [split [lindex $tmpNodes $i] ":"]
151                     if [string equal [lindex $tmpNeighbors 0] $prefixHub] {
152                             createHub $number [llength $tmpNeighbors]
153                     } elseif [string equal [lindex $tmpNeighbors 0] $prefixClient] {
154                             createNode $number [llength $tmpNeighbors]
155                             createMmrpClient $number
156                     } else {
157                             createNode $number [llength $tmpNeighbors]
158                     }
159             }
160
161             createLinksFromString $topology $delay
162     }
163
164     proc startRSTP { nodes runName topology } {
165             set prefixBridge "b"
166             set prefixHub "h"
167             set prefixClient "c"
168             cd /$runName
169
170
171
172             set k 1;
173             set tmpNodes [split $topology " "]
174             for {set j 0} {$j < [llength $tmpNodes] } {incr j} {
175                     set number "";
176                     set hexNumber [format %x $j];
177                     if [expr $j < 16] {
178                             set number "0$hexNumber";
179                     } else {
180                             set number "$hexNumber";
181                     }
182                     set tmpNeighbors [split [lindex $tmpNodes $j] ":"]
183                     if [string equal [lindex $tmpNeighbors 0] $prefixHub] {
184                             # puts "\tHubs do not contain RSTP, do nothing..";
185                     } else {
186                             # puts "Setting BEGIN ...."
```

```
187                             ! node$number/rstp setDisplayRstBpdu false;
188                             ! node$number/rstp setDebugEnabled false
189                             ! node$number/rstp setDebug false
190                             [! node$number/rstp getPerBridgeVariables] setBEGIN true;
191                             ! node$number/rstp invokeTaskScheduler;
192 
193                             # puts "Clearing BEGIN ...."
194                             [! node$number/rstp getPerBridgeVariables] setBEGIN false;
195                             ! node$number/rstp invokeTaskScheduler;
196 
197                             # puts "\tStarting Timers ...."
198                             ! node$number/rstp startTimersWithDelay $k;
199                             set k [expr $k+0.1];
200                     }
201             }
202     }
203 
204 
205     proc create { runName topology delay } {
206             # puts "Creating a simulation topology..."
207             cd /
208             rm $runName
209             cd [mkdir drcl.comp.Component $runName]
210 
211             createScenario $topology $delay
212             # puts "DONE creating the simulation topology!"
213     }
```

Appendix B

Experiment TCL script

This appendix shows one of the simulation experiments.

```
source "D:/Mi mappe/Dokumenter/My Dropbox/Master thesis/jsim-1.3_NB/script/mmrp/
    functions.tcl"

set nodes 8
set delay 0.0000000000005
set runName "experiment"
set topology "b:1:2 b:0:3:4:5:6 c:0 c:1 c:1 c:1 c:1"

cd /
rm $runName

create $runName $topology $delay

# Connect plotter and traffic monitor for graphs
set plot [mkdir drcl.comp.tool.Plotter .plot]
set tm_3 [mkdir drcl.net.tool.TrafficMonitor .tm3]
set tm_4 [mkdir drcl.net.tool.TrafficMonitor .tm4]
set tm_5 [mkdir drcl.net.tool.TrafficMonitor .tm5]
set tm_6 [mkdir drcl.net.tool.TrafficMonitor .tm6]

connect -c $tm_3/in@ -and node03/llc/.ip@up
connect -c $tm_4/in@ -and node04/llc/.ip@up
connect -c $tm_5/in@ -and node05/llc/.ip@up
connect -c $tm_6/in@ -and node06/llc/.ip@up

connect -c $tm_3/bytecount@ -and $plot/0@0
```

```
connect -c $tm_4/bytecount@ -and $plot/0@1
connect -c $tm_5/bytecount@ -and $plot/0@2
connect -c $tm_6/bytecount@ -and $plot/0@3

connect -c $tm_3/bytecount@ -and $plot/0@4
connect -c $tm_4/bytecount@ -and $plot/1@4
connect -c $tm_5/bytecount@ -and $plot/2@4
connect -c $tm_6/bytecount@ -and $plot/3@4

setflag garbagedisplay false /$runName/node*
setflag garbagedisplay true /$runName/node00/mmrp

! .tm? configure 0.5 0.05; # window size, update interval

# ####################
# # RUN SIMULATION ##
# ####################
set sim [attach_simulator .]
$sim stop

startRSTP $nodes $runName $topology
script {cat node*/rstp} -at 9.9999 -on $sim

puts "Creating registrations from each client.."
script {[! /test/node03/multicastClient] registerMacAddress "ff:ff:11:03:11:ff"} -at 10
    .3 -on $sim
script {[! /test/node04/multicastClient] registerMacAddress "ff:ff:11:04:11:ff"} -at 10
    .4 -on $sim
script {[! /test/node05/multicastClient] registerMacAddress "ff:ff:11:05:11:ff"} -at 10
    .5 -on $sim
script {[! /test/node06/multicastClient] registerMacAddress "ff:ff:11:06:11:ff"} -at 10
    .6 -on $sim

script {[! /test/node02/multicastServer] setDestinationAddress "ff:ff:11:03:11:ff"} -at
    12 -on $sim
script {[! /test/node02/multicastServer] setSendDelay 0.005} -at 12.1 -on $sim
script {[! /test/node02/multicastServer] run} -at 12.2 -on $sim

 script {cat node*/mmrp} -at 20 -on $sim

End user 03 sends data to the multicast group
```

```
62 script {[! $server03] setDestinationAddress "ff:ff:11:11:11:ff"} -at 25.00001221 -on
       $sim
63 script {[! $server03] setSendDelay 0.1} -at 25.0000402 -on $sim
64 script {[! $server03] run} -at 25.10012 -on $sim
65
66 $sim resumeTo 50;
```

Appendix C

Network designer screenshot

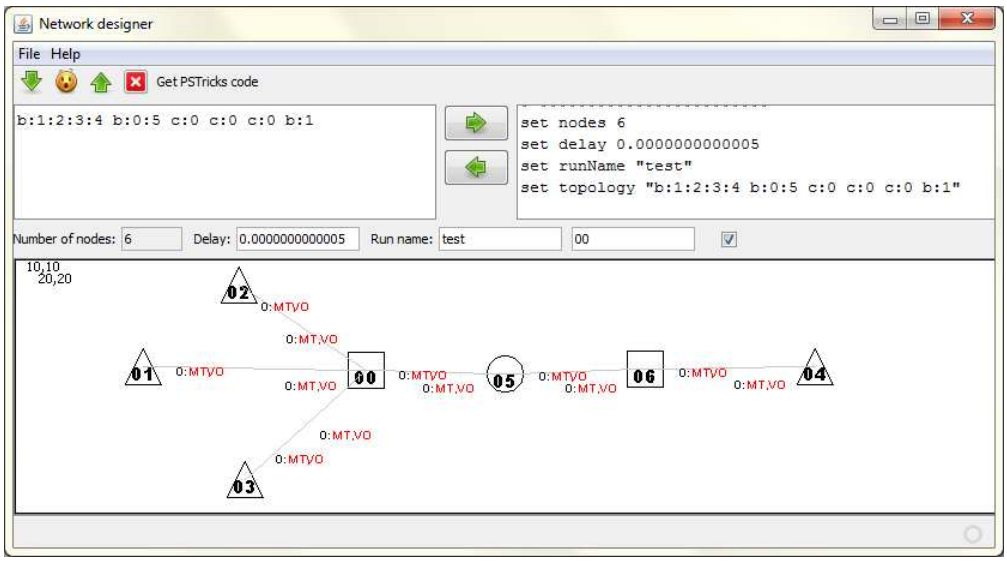


Figure C.1: Screenshot of the Network designer GUI

Appendix D

Javadoc for the MMRP component

The next pages includes the Javadoc for the MMRP component.

drcl.ethernet.mmrp

Class MMRP

```
java.lang.Object
  └drcl.DrclObj
      └drcl.comp.Component
          └drcl.net.Module
              └drcl.ethernet.mmrp.MMRP
```

All Implemented Interfaces:

ObjectCloneable, ObjectDuplicable, java.io.Serializable, java.lang.Cloneable

---

```
public class MMRPextends Module
```

From IEEE Std 802.1ak-2007

The Applicant for each Attribute implements states that record whether it wishes to make a new declaration, to maintain or withdraw an existing declaration, or has no declaration to make. It also records whether it has actively made a declaration, or has been passive, taking advantage of or simply observing the declarations of others. It counts the New, JoinIn, and JoinEmpty messages it has sent, and JoinIn messages sent by others, to ensure that at least two such messages have been sent since it last received a LeaveAll or Leave message, and at least one since it last received a JoinEmpty or Empty message. This ensures that each of the other Participant's Registrars for the Attribute have either received (assuming no packet loss) two Join or New messages or have reported the Attribute as registered. The Applicant state machine (Table 10-3) uses the following states:

VO Very anxious Observer. The applicant is not declaring the attribute, and has not received a JoinIn message since the state machine was initialized, or since last receiving a Leave or LeaveAll.
VP Very anxious Passive. The applicant is declaring the attribute, but has neither sent a Join nor received a JoinIn since the state machine was initialized, or since last receiving a LeaveAll or Leave.
VN Very anxious New. The applicant is declaring the attribute, but has not sent a message since receiving a MAD Join request for a new declaration.
AN Anxious New. The applicant is declaring the attribute, and has sent a single New message since receiving the MAD Join request for the new declaration.
AA Anxious Active. The applicant is declaring the attribute, and has sent a Join message, since the last Leave or LeaveAll, but has either not received another JoinIn or In, or has received a subsequent message specifying an Empty registrar state.
QA Quiet Active. The applicant is declaring the attribute and has sent at least one of the required Join or New messages since the last Leave or LeaveAll, has seen or sent the other, and has received no subsequent messages specifying an Empty registrar state.
LA Leaving Active. The applicant has sent a Join or New message since last receipt of a Leave or LeaveAll, but has subsequently received a MAD Leave request and has not yet sent a Leave message.
AO Anxious Observer. The applicant is not declaring the attribute, but has received a JoinIn since last receiving a Leave or LeaveAll.
QO Quiet Observer. The applicant is not declaring the attribute, but has received two JoinIns since last

receiving a Leave or LeaveAll, and at least one since last receiving a message specifying an Empty registrar state.
AP Anxious Passive. The applicant is declaring the attribute, and has not sent a Join or a New since last receiving a Leave or a LeaveAll but has received messages as for the Anxious Observer state.
QP Quiet Passive. The applicant is declaring the attribute, and has not sent a Join or a New since last receiving a Leave or a LeaveAll but has received messages as for the Quiet Observer state.
LO Leaving Observer. The applicant is not declaring the attribute, and has received a Leave or LeaveAll message.

See Also:
Serialized Form

Nested Class Summary

Nested classes/interfaces inherited from class drcl.comp.Component
Component.Locks

Field Summary

protected Port	**clientPort** clientPort belonging to the service port group.
protected Port	**eventPortRoleChange** eventPortRoleChange port belonging to the service port group.
protected Port	**eventPortStateChange** eventPortStateChange port belonging to the service port group
protected Port	**filteringDatabaseConfigPort** filteringDatabaseConfigPort belonging to the service port group
protected Port	**filteringDatabaseQueryPort** filteringDatabaseQueryPort belonging to the service port group
protected Port	**linkBrokenPort** linkBrokenPort belonging to the servicePortGroup.
protected static java.lang.String	**PortID_CLIENT** Port ID for the client port
protected static java.lang.String	**PortID_FILTERING_DATABASE_CONFIG** Port ID for the filteringDatabaseConfig port
protected static java.lang.String	**PortID_FILTERING_DATABASE_QUERY**

	Port ID for the filteringDatabaseQuery port
protected static java.lang.String	**PortID_LINK_BROKEN** Port ID for the .linkbroken port
protected static java.lang.String	**PortID_PORT_STATE_CONFIG** Port ID for the .portStateConfig port
protected static java.lang.String	**PortID_PORT_STATE_QUERY** Port ID for the .portStateQuery port
protected static java.lang.String	**PortID_ROLE_CHANGE_EVENT** Port ID for the portRoleChangeEvent port
protected static java.lang.String	**PortID_STATE_CHANGE_EVENT** Port ID for the portStateChangeEvent
protected Port	**portStateConfigPort** portStateConfigPort belonging to the service port group
protected Port	**portStateQueryPort** portStateQuery port belonging to the service port group

Fields inherited from class drcl.net.Module
downPort, PortGroup_DOWN, PortGroup_UP, timerPort, upPort

Fields inherited from class drcl.comp.Component
FLAG_COMPONENT_NOTIFICATION, FLAG_DEBUG_ENABLED, FLAG_DIRECT_OUTPUT_ENABLED, FLAG_ENABLED, FLAG_ERROR_ENABLED, FLAG_EVENT_ENABLED, FLAG_GARBAGE_DISPLAY_ENABLED, FLAG_GARBAGE_ENABLED, FLAG_PORT_NOTIFICATION, FLAG_STARTED, FLAG_STOPPED, FLAG_TRACE_ENABLED, FLAG_UNDEFINED_START, id, infoPort, locks, name, parent, PortGroup_DEFAULT_GROUP, PortGroup_EVENT, PortGroup_SERVICE, Root, Trace_DATA, Trace_SEND

Constructor Summary

MMRP(EthernetAddress bridgeEthernetAddress) MMRP constructor	
MMRP(java.lang.String bridgeEthernetAddress) MMRP constructor	

Method Summary

protected void	**dataArriveAtDownPort**(java.lang.Object data_, Port downPort_) The handler invoked when a packet arrives at a "down" port.
void	**deregisterMacAddress**(EthernetAddress ethernetAddress) This method is called on receipt of the DEREGISTER_MAC_ADDRESS primitive.
void	**forwardPDU**(MRPDU mrpdu, int sourceInterface, int sourceVLANidentifier) This method is called when the MMRP-participant receives a MAC_Join.
void	**forwardPDU**(MRPDU mrpdu, int sourceInterface, int sourceVLANidentifier, boolean checkState)
EthernetAddress	**getBridgeEthernetAddress**() Get the ethernet MAC address of the bridge
java.util.HashMap<java.lang.Integer,java.util.HashMap<java.lang.Integer,Participant>>	**getParticipantList**() This method returns the participantList
java.util.HashMap<java.lang.Integer,PeriodicStateMachine>	**getPeriodicStateMachineList**() Get method for the perodicStateMachineList
java.lang.String	**info**() Returns information regarding this component.
boolean	**isSimpleApplicant**() Get the state of the applicant
void	**mmrpAttributePropagation**(int sourceInterface, int sourceVLANidentifier, EthernetAddress firstValue, MRPconstants.MAD_PRIMTIVES MADPrimitive) The MMRP Attribute Propagation (MAP) function.
protected void	**processOther**(java.lang.Object data_, Port inPort_) The handler invoked when a packet arrived at a port other than the "up", "down" and timer ports.
void	**registerMacAddress**(EthernetAddress ethernetAddress) This method is called on receipt of the REGISTER_MAC_ADDRESS

	primitive.
void	**setSimpleApplicant**(boolean simpleApplicant) Sets whether the applicant is simpleApplicant or not.
protected void	**timeout**(java.lang.Object data_) The handler invoked when a timeout event occurs.
void	**updateFilteringDatabase**(int sourceInterface, int sourceVLANidentifier, EthernetAddress firstValue, MRPconstants.MAD_PRIMTIVES MADPrimitive) Updating the filtering database by sending a FilteringDatabaseConfig message out of the filteringDatabaseConfig port.

Methods inherited from class drcl.net.Module

cancelTimeout, dataArriveAtUpPort, deliver, duplicate, process, removeDefaultDownPort, removeDefaultUpPort, removeTimerPort, setTimeout, setTimeoutAt

Methods inherited from class drcl.comp.Component

_resume, _start, _stop, addComponent, addComponent, addEventPort, addEventPort, addForkPort, addPort, addPort, addPort, addPort, addPort, addPort, addPort, addServerPort, addServerPort, cancelFork, componentAdded, componentRemoved, connect, containsComponent, containsComponent, containsPort, debug, disconnectAll, disconnectAllPeers, disconnectAllPorts, drop, drop, error, error, expose, exposeEventPorts, exposePort, exposePort, exposePort, exposePort, exposePort, exposePort, findAvailable, findAvailable, findAvailable, finishing, fork, forkAt, getAllComponents, getAllPorts, getAllPorts, getAllWiresInside, getAllWiresInsideOut, getAllWiresOut, getComponent, getComponentFlag, getComponentFlag, getContract, getContractHT, getContractHT, getDebugFlagsInBinary, getDebugLevelNames, getForkManager, getID, getName, getParent, getPort, getPort, getRoot, getRuntime, getTime, iduplicate, isAncestorOf, isComponentNotificationEnabled, isContainer, isDebugEnabled, isDebugEnabledAt, isDirectlyRelatedTo, isDirectOutputEnabled, isEnabled, isErrorNoticeEnabled, isEventExportEnabled, isGarbageDisplayEnabled, isGarbageEnabled, isPortNotificationEnabled, isPortRemovable, isStarted, isStopped, isTraceEnabled, lock, notify, notifyAll, operate, portAdded, portRemoved, reboot, removeAll, removeAllComponents, removeAllPorts, removeAllPorts, removeComponent, removeComponent, removePort, removePort, removePort, reset, resume, run, sduplicate, send, sendAt, setComponentFlag, setComponentFlag, setComponentNotificationEnabled, setContract, setDebugEnabled, setDebugEnabled, setDebugEnabledAt, setDebugEnabledAt, setDebugEnabledAt, setDirectOutputEnabled, setDirectOutputEnabled, setEnabled, setErrorNoticeEnabled,

setErrorNoticeEnabled, setEventExportEnabled, setEventExportEnabled, setExecutionBoundary, setGarbageDisplayEnabled, setGarbageDisplayEnabled, setGarbageEnabled, setGarbageEnabled, setID, setID, setName, setPort, setPort, setPortNotificationEnabled, setPortRemovable, setRuntime, setTraceEnabled, setTraceEnabled, sleepFor, sleepUntil, stop, toString, unexpose, unlock, useLocalForkManager, useLocalForkManager, wait, yield

Methods inherited from class drcl.DrclObj
clone

Methods inherited from class java.lang.Object
equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Field Detail

PortID_PORT_STATE_QUERY

```
protected static final java.lang.String PortID_PORT_STATE_QUERY
```

Port ID for the .portStateQuery port

See Also:
Constant Field Values

PortID_PORT_STATE_CONFIG

```
protected static final java.lang.String PortID_PORT_STATE_CONFIG
```

Port ID for the .portStateConfig port

See Also:
Constant Field Values

PortID_FILTERING_DATABASE_QUERY

```
protected static final java.lang.String PortID_FILTERING_DATABASE_QUERY
```

Port ID for the filteringDatabaseQuery port

See Also:
Constant Field Values

PortID_FILTERING_DATABASE_CONFIG

```
protected static final java.lang.String PortID_FILTERING_DATABASE_CONFIG
```

Port ID for the filteringDatabaseConfig port

See Also:
Constant Field Values

PortID_LINK_BROKEN

```
protected static final java.lang.String PortID_LINK_BROKEN
```

Port ID for the .linkbroken port

See Also:
Constant Field Values

PortID_CLIENT

```
protected static final java.lang.String PortID_CLIENT
```

Port ID for the client port

See Also:
Constant Field Values

PortID_ROLE_CHANGE_EVENT

```
protected static final java.lang.String PortID_ROLE_CHANGE_EVENT
```

Port ID for the portRoleChangeEvent port

See Also:
Constant Field Values

PortID_STATE_CHANGE_EVENT

```
protected static final java.lang.String PortID_STATE_CHANGE_EVENT
```

Port ID for the portStateChangeEvent

See Also:
Constant Field Values

---

portStateQueryPort

protected Port **portStateQueryPort**

portStateQuery port belonging to the service port group

---

portStateConfigPort

protected Port **portStateConfigPort**

portStateConfigPort belonging to the service port group

---

filteringDatabaseQueryPort

protected Port **filteringDatabaseQueryPort**

filteringDatabaseQueryPort belonging to the service port group

---

filteringDatabaseConfigPort

protected Port **filteringDatabaseConfigPort**

filteringDatabaseConfigPort belonging to the service port group

---

linkBrokenPort

protected Port **linkBrokenPort**

linkBrokenPort belonging to the servicePortGroup. DEPRECATED.

---

clientPort

protected Port **clientPort**

clientPort belonging to the service port group. DEPRECATED.

---

eventPortRoleChange

protected Port **eventPortRoleChange**

eventPortRoleChange port belonging to the service port group.

eventPortStateChange

```
protected Port eventPortStateChange
```

eventPortStateChange port belonging to the service port group

Constructor Detail

MMRP

```
public MMRP(EthernetAddressbridgeEthernetAddress)
```

MMRP constructor

Parameters:
`bridgeEthernetAddress` - Setting the bridge ethernet MAC address

MMRP

```
public MMRP(java.lang.StringbridgeEthernetAddress)
```

MMRP constructor

Parameters:
`bridgeEthernetAddress` - Setting the bridge ethernet MAC address

Method Detail

dataArriveAtDownPort

```
protected void dataArriveAtDownPort(java.lang.Objectdata_,
                                    PortdownPort_)
```

Description copied from class: Module
The handler invoked when a packet arrives at a "down" port. Subclasses should override it to handle such an event.

Overrides:
`dataArriveAtDownPort` in class `Module`

---

timeout

```
protected void timeout(java.lang.Objectdata_)
```

Description copied from class: Module
The handler invoked when a timeout event occurs. Subclasses should override it to handle such an event.

Overrides:

timeout in class Module

See Also:

Module.setTimeout(Object, double), Module.setTimeoutAt(Object, double)

---

updateFilteringDatabase

```
public void updateFilteringDatabase(intsourceInterface,
                                    intsourceVLANidentifier,
                                    EthernetAddressfirstValue,
                                    MRPconstants.MAD_PRIMTIVESMADPrimitive)
```

Updating the filtering database by sending a `FilteringDatabaseConfig` message out of the filteringDatabaseConfig port. Checking wheter the response is ok or not. Error message if not ok.

Parameters:

`sourceInterface` -
`sourceVLANidentifier` -
`firstValue` -
`MADPrimitive` -

---

mmrpAttributePropagation

```
public void mmrpAttributePropagation(intsourceInterface,
                                     intsourceVLANidentifier,
                                     EthernetAddressfirstValue,
                                     MRPconstants.MAD_PRIMTIVESMADPrimitive)
```

The MMRP Attribute Propagation (MAP) function. Responsible for sender the MAD primitives between participants in the same MAP context in the bridge.

Parameters:

`sourceInterface` -
`sourceVLANidentifier` -
`firstValue` -
`MADPrimitive` -

---

registerMacAddress

```
public void registerMacAddress(EthernetAddressethernetAddress)
```

This method is called on receipt of the REGISTER_MAC_ADDRESS primitive. Only for hosts which wants to join a group.

Parameters:

`ethernetAddress` -

---

deregisterMacAddress

```
public void deregisterMacAddress(EthernetAddressethernetAddress)
```

This method is called on receipt of the DEREGISTER_MAC_ADDRESS primitive. Only for hosts which wants to join a group.

Parameters:

`ethernetAddress` -

---

forwardPDU

```
public void forwardPDU(MRPDUmrpdu,
                       intsourceInterface,
                       intsourceVLANidentifier)
```

This method is called when the MMRP-participant receives a MAC_Join. indcation or MAC_Leave.indication.

Parameters:

`mrpdu` - The packet to send. Of type `MRPDU`
`sourceInterface` - The sourceInterface.
`sourceVLANidentifier` - The source VLAN.

---

forwardPDU

```
public void forwardPDU(MRPDUmrpdu,
                       intsourceInterface,
                       intsourceVLANidentifier,
                       booleancheckState)
```

Parameters:

`mrpdu` - The packet to send. Of type `MRPDU`
`sourceInterface` - The sourceInterface.
`sourceVLANidentifier` - The source VLAN.
`checkState` - `true` if it is going to check the State in the topology (e.g. RSTP), `false` if else.

---

info

```
public java.lang.String info()
```

Description copied from class: Component
Returns information regarding this component. Subclasses should override this method to provide useful information at run time.

Overrides:

`info` in class `Component`

---

processOther

```
protected void processOther(java.lang.Objectdata_,
                            PortinPort_)
```

Description copied from class: Module
The handler invoked when a packet arrived at a port other than the "up", "down" and timer ports.

Overrides:
processOther in class Module

---

getParticipantList

```
public
java.util.HashMap<java.lang.Integer,java.util.HashMap<java.lang.Integer,Participant>>
getParticipantList()
```

This method returns the participantList

Returns:

---

getPeriodicStateMachineList

```
public java.util.HashMap<java.lang.Integer,PeriodicStateMachine>
getPeriodicStateMachineList()
```

Get method for the perodicStateMachineList

Returns:
HashMap<Integer, PeriodicStateMachine> The list.

---

setSimpleApplicant

```
public void setSimpleApplicant(booleansimpleApplicant)
```

Sets whether the applicant is simpleApplicant or not.

Parameters:
simpleApplicant -

---

isSimpleApplicant

```
public boolean isSimpleApplicant()
```

Get the state of the applicant

Returns:
true if the applicant is of simpleApplicant type, or false else.

---

getBridgeEthernetAddress

`public EthernetAddress` **`getBridgeEthernetAddress`**`()`

Get the ethernet MAC address of the bridge

Returns:

`EthernetAddress` of the bridge

~ To J-Sim Home ~