University "Politehnica" of Bucharest

Automatic Control and Computers Faculty,
Computer Science Department

BACHELOR THESIS

STP/RSTP implementation in LiSA

Scientific Adviser:
As. Drd. George Milescu

Author:
Andrei Faur

Bucharest, 2009

Algorhyme[1]

I think that I shall never see
A graph more lovely than a tree.
A tree whose crucial property
Is loop-free connectivity.
A tree which must be sure to span.
So packets can reach every LAN.
First the Root must be selected
By ID it is elected.
Least cost paths from Root are traced
In the tree these paths are placed.
A mesh is made by folks like me
Then bridges find a spanning tree.

Contents

List of Figures

Notations and Abbreviations

ASIC – Application-specific integrated circuit
CLI – Command Line Interface
DSAP – Destination Service Access Point
FSM – Finite state machine
IEEE – Institute of Electrical and Electronics Engineers
LAN – Local area network
LiSA – Linux Switching Appliance
LLC – Logical Link Control
LMS – Linux Multilayer Switch
MAC – Media access control
NIC – Network Interface Card
OUI – Organizational Unique Identifier
RSTP – Rapid Spanning Tree Protocol
SSAP – Source Service Access Point
STP – Spanning Tree Protocol
VLAN – Virtual LAN

Chapter 1

Introduction

The fundamental problem of any network is the way information is sent from one node to another. The issue's complexity is increased by the fact that the network may be divided into several sub-networks, thus creating the need for an inter-network communication mechanism.

Two major paradigms are used in telecommunications for describing the way user messages reach different nodes of a network : packet switching and circuit switching.

Baran developed the concept of packet switching during his research at the RAND Corporation for the US Air Force into survivable communications networks . He first presented the idea to the Air Force in the summer of 1961 and then published it as RAND Paper P2626 in 1962. The paper focuses on three key ideas: first, use of a decentralized network with multiple paths between any two points; second, dividing complete user messages into what he called message blocks (later called packets); then third, delivery of these messages by store and forward switching.[2]

The packet switching concept was a radical paradigm shift from the prevailing model of communications networks using dedicated, analog circuits primarily built for audio communications, and established a new model of digital systems that break messages into individual packets that are transmitted independently and then assembled back into the original message at the far end. The conceptual breakthrough advantage of packet switching was enabling the construction of data networks at much lower cost with greater throughput, flexibility, and robustness by routing multiple communications over the same wire at the same time.[3]

The second paradigm, circuit switching, consists of establishing a circuit between nodes and terminals before the users may communicate, as if the nodes were physically connected with an electrical circuit. The bit delay is constant during a connection, as opposed to packet switching, where packet queues may cause varying packet transfer delay. Circuit switched networks are still in use today but on a lower scale than packet switched networks.[4]

Packet switching is well-known for its use in today's Internet and local area networks. The Internet uses the Internet protocol suite over a variety of Link Layer protocols such as Ethernet and frame relay which are very common. The use of packet switching in local area networks is called LAN switching and the devices that implement it are called switches. Switches can operate on several layers of the OSI stack but the term usually refers to a device that processes and routes data at the Data link layer.

Layer 2 switching is hardware based, which means it uses the MAC address from the host's NICs to decide where to forward frames[1]. Switches use ASICs to build and maintain filter tables (also known as MAC address tables)[5]. A switch's implementation of different layer 2 protocols are usually a mix between hardware and software logic, the hardware being used in areas where speed is critical.

LiSA , short for **Li**nux **S**witching **A**ppliance, is an open-source software project built for the GNU/Linux operating system that aims to provide its users with all the necessary instruments for implementing and maintaining an efficient and reliable switching solution at a low cost. The project was designed to be used in medium and small-sized networks. As opposed to usual hardware-driven implementations, LiSA offers a software approach to switching by utilizing the Linux Kernel networking stack and adding its own switch-specific functionality.

Since LiSA is practically a software implementation of what normally is a hardware-based logic, it might seem that there is no advantage in using it. On the contrary, LiSA provides several benefits that hardware implementations do not have. Many of these advantages stem from the fact that it is based on the Linux operating system. First of all this means it is not tied to a specific hardware architecture. It is only limited to the architectures supported by Linux, of which we mention : ARM, x86, MIPS, PowerPC and many more[2]. Secondly, it is not dependant on the network hardware. As long as there is a Linux device driver for the specific NIC which LiSA is supposed to handle, no problems will be encountered. Finally, Linux is well-known for its ability to run well on older systems and on embedded devices, that is, devices that have important resource constraints. This provides an advantage since one of LiSA's main objectives is to run on such devices.

Whether a network designer chooses a hardware or software approach to switching, he has to be certain that the chosen solution addresses all of the layer's problems, especially those that are critical and that could render the network unusable. Usually, when a network is designed, multiple connections between different components are created, in order to ensure a certain degree of redundancy. Redundant links make the network more tolerant to faults, since a bad connection doesn't imply that the whole network goes down. In turn, this creates another problem because these links usually create loops in the network. Having

[1]Layer 2 packets are usually called frames

[2]As of version 2.6.30 the Linux kernel supports a number of 22 main architectures, plus different variations on them

layer 2 loops in a network can lead to serious performance problems since frames can consume a large part of the available bandwidth.

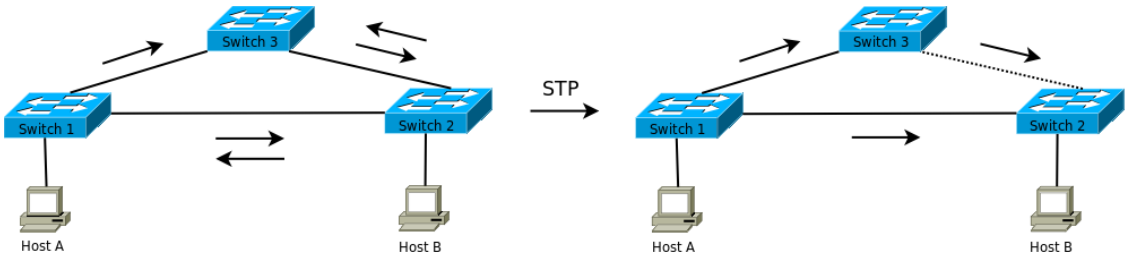

Figure 1.1: STP disables connections that form a loop

For example, Figure 1.1 shows a network made out of two hosts, A and B, and three switches labeled from 1 to 3. Suppose host A sends a broadcast message[1]. Switch 1 will not find the destination address in its filtering table, so it will forward the message on all links except the one that the message came from : the link to Switch 3 and the link to Switch 2. Switch 2 will receive the message and will react in the same manner as Switch 1, so it will forward the message to Switch 3. The result is that Switch 3 receives two copies of the initial message, and it will forward each copy both to host B and to the other switches. Even though the broadcast message has now reached all hosts (in our case, host B), there are copies of the message moving through the network that will spawn even more copies because of the existing loop. This is called a broadcast avalanche and it has a significant negative impact on network performance.

A solution to this problem is the layer 2 **Spanning Tree Protocol** which creates a logical tree topology, thus making sure that no loops are active in the network. One possible outcome of the protocol is shown in Figure 1.1, where the connection between Switch 2 and Switch 3 has been disabled so Switch 2 will now stop forwarding messages on that link and will also ignore[2] messages coming from it. Most switches offer a STP implementation, so it's only natural that a software approach to switching such as LiSA should have this capability too. The 802.1D Spanning Tree Protocol (STP) standard was designed at a time when the recovery of connectivity after an outage within a minute or so was considered adequate performance. Cisco enhanced the original 802.1D specification with features such as Uplink Fast, Backbone Fast, and Port Fast to speed up the convergence time of a bridged network. The drawback is that these mechanisms are proprietary and need additional configuration.

[1]Message that is sent to every device on the network

[2]It does not completely ignore them, since it adds the MAC address of the source to the filtering table. The exact details will be shown in subsequent chapters.

The 802.1w **Rapid Spanning Tree Protocol** can be seen as an evolution of the 802.1D standard more than a revolution. The 802.1D terminology remains primarily the same and most parameters have been left unchanged so users familiar with 802.1D can rapidly configure the new protocol comfortably. In most cases, RSTP performs better than proprietary extensions of Cisco without any additional configuration. 802.1w can also revert back to 802.1D in order to interoperate with legacy bridges on a per-port basis, but this drops the benefits RSTP introduces[6].

This thesis presents the implementation of the Rapid Spanning Tree Protocol in LiSA, describing the protocol's details and specific issues concerning the integration with LiSA.

Chapter 2

Spanning Tree Algorithms

A spanning tree algorithm is used in bridged networks to dynamically determine the best path from source to destination, while avoiding loops which can cause bridges[1] to continuously forward the same frames, as previously shown. An algorithm of this type creates a hierarchical tree that spans the entire network, including all switches. More than that, it determines all redundant paths and makes only one of them active at any given time.

2.1 Spanning Tree Protocol

The Spanning Tree Protocol (STP) is a link layer network protocol that ensures a loop-free topology for any bridged LAN. It is based on an algorithm invented by Radia Perlman while working for Digital Equipment Corporation. In the OSI model for computer networking, STP falls under the OSI layer-2. Spanning tree allows a network design to include redundant links to provide automatic backup paths if an active link fails, without the danger of bridge loops, or the need for manual enabling/disabling of these backup links. Bridge loops must be avoided because they result in flooding the network[7].

The purpose of the spanning-tree algorithm is to have bridges dynamically build a topology that is loop-free (a tree) and that still has connectivity between every pair of LANs. The algorithm accomplishes this by sending special messages containing information about the bridges. These messages allow the bridges to elect a single bridge as the root of the spanning tree and also compute the shortest path from themselves to that bridge.[8]

Before going into the details of the Spanning Tree Protocol, a few related terms have to be explained.

[1] The terms bridge and switch will be used interchangeably

2.1.1 Terminology

Tree Topology - The Spanning Tree Protocol creates such a logical topology in order to eliminate loops. Like any tree, it has a root, which in this case is a switch called the Root Bridge, and its role is to collect and distribute notifications about topology changes [1]. Another important observation is that the topology leaves no segment unreachable, so even if not all paths are active every node in the LAN has connectivity. Figure 2.1 depicts a simple loop-free network with active connections that span from the root (Root Bridge).

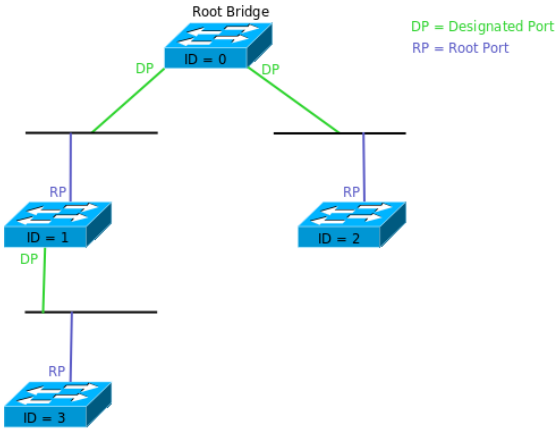

Figure 2.1: Loop-free network with active connections

Bridge Identifiers - Since bridges have to be properly accounted for in the algorithm, there needs to be a way to uniquely identify them in the network, including their ports. Thus, each bridge has a Bridge identifier which is a unique 64-bit field, representing the concatenation of a 16-bit priority value and a 48-bit MAC address. The MAC address is usually chosen as the address of the lowest numbered port. The priority field gives network administrators the ability to control the outcome of the algorithm by assigning a low priority value. If the outcome depended solely on the MAC address, a manufacturer whose OUI provided a numerical advantage over another's would automatically cause that company's bridges to become Root Bridges.

Port Identifiers - The same way individual bridges have unique identifiers, each port on a bridge is assigned a Port Identifier locally unique for that bridge. It is composed of an 8-bit configurabile priority field and an 8-bit port number. Port numbers range from 1 to N, where N is the number of ports on the device. The priority field in the Port Identifier is used in a manner identical to the priority field in the Bridge Identifier, so it can be configured in such a manner that the resulting topology is independent of the numbering of the ports within a bridge.

[1] This is only valid for STP, since RSTP decentralizes this process

Root Bridge - This is the bridge that will represent the root of the tree, after the algorithm has finished. Only one Root Bridge can exist. The Root Bridge is elected according to the Bridge Identifier of each switch. Like mentioned before, there are two parts to the Bridge ID: a user selected priority and the MAC address. The switch with the lowest numerical value of the priority component becomes the Root Bridge. When more than one switch has the same priority value, the one with the lowest MAC address becomes the Root Bridge. An example can be found in Figure 2.1 where the switch with the lowest ID has been elected as the Root Bridge.

Designated Bridges - A simple way to prevent loops in the network is to ensure only one bridge is responsible for forwarding traffic from the direction of the root into any given link (branch). As long as only one active path from a root to any end node (leaf) exists, there will be no loops in the topology. The bridge responsible for forwarding traffic in the direction from the root to a given link is known as the Designated Bridge of that link. For example, in Figure 2.1 the bridge with ID equal to 1 can be considered Designated Bridge for the link between itself and the bridge with ID 3.[8]

Port States - There are five operational states assigned to each port by STP: Disabled, Blocking, Listening, Learning, and Forwarding:

- *Disabled* - A disabled port is administratively shut down. This state is not officially part of the 802.1 standard but most network equipment suppliers take it into consideration.
- *Blocking* - This is the first state a port enters after it has been opened. The only difference from the disabled state is that now the port accepts BPDUs. Still, the port does not do anything useful with them since frame forwarding and MAC address learning are still disabled in this state.
- *Listening* - From blocking mode a port will reach this state. Here, it will receive and accept BPDUs, just like the blocking state, and, in addition to that, it will also send out BPDUs. Frame forwarding and MAC address learning are disabled.
- *Learning* - Once a port reaches this state it enables MAC address learning. The port starts adding entries in the MAC address table by looking at the source address of incoming frames. Frame forwarding is still disabled.
- *Forwarding* - A port in this state forwards and receives data frames, sends and receives BPDUs, and places MAC addresses in its MAC table.

The states and the corresponding transitions are shown in Figure 2.2.

Path cost - The Spanning Tree Protocol attempts to configure the network such that every station is reachable from the root through the path with the lowest cost. The cost of a path is the sum of the costs of the links attached to the Root Ports in that path.

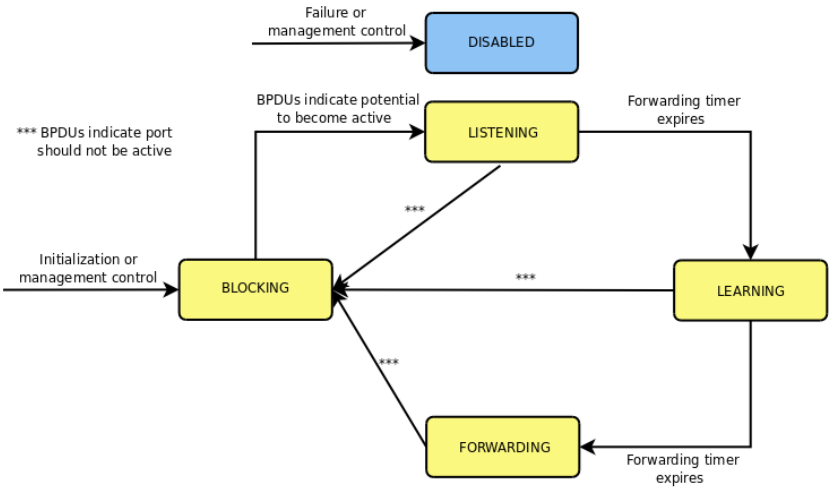

Figure 2.2: Port states

Port roles - A port role is a function STP and RSTP assign to each port. STP assigns one of the following roles: Root Port, Designated Port, or Blocking Port. These ports can be seen in Figure 2.3.

- *Root port* - A root port is the port closest to the root bridge in terms of path cost. When a switch has multiple paths connecting it to the root, the best path is determined based on the message priority vector carried inside the Bridged Protocol Data Unit (BPDU) and the receiving port ID. The port with the best priority vector becomes a root port while the remaining ports become alternates.
- *Designated Port* - A port is designated if it can send the best BPDU on the segment to which it is connected. All bridges connected to a given segment listen to each other's BPDUs and agree that the bridge sending the best BPDU is the designated bridge for the segment.
- *Blocking port* - A blocking port does not forward user data. A port is assigned this role if by adding the link to which the port is connected to ,to the topology, a loop would be created.

2.1.2 BPDU Formats

Bridges learn and exchange information about each other in order to calculate spanning tree by sending small packets called Bridge Protocol Data Units (BPDUs). STP uses two different BPDUs: Configuration BPDUs and Topology Change BPDUs.

- *Configuration BPDU*s originate from the root bridge every hello time and carry all information required to calculate spanning tree topology. Other

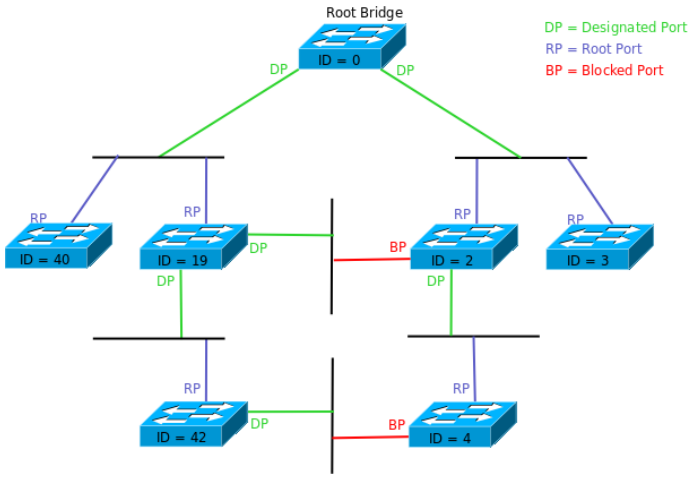

Figure 2.3: Network depicting all STP port roles

bridges listen for Configuration BPDUs on their root ports and forward them on their designated ports.

- *Topology Change BPDU*s (TCN BPDUs) are sent in the direction of the root by the bridge which detected a topology change. When the root bridge receives a TCN BPDU, it must then inform other switches a change has occurred in the current topology. It does so by setting a Topology Change (TC) flag in every BPDU it sends for a period of time (specified as Forward Delay + Max Age). When a switch receives a BPDU with a TC flag set,it switches its aging time from long to short in order to age out Filtering Database entries more rapidly.

When transmitted on a LAN, BPDUs are further encapsulated in MAC frames using LLC Type 1, with a DSAP and SSAP of 0x42. The MAC Source Address is the MAC address of the port through which the frame is being transmitted. The MAC Destination Address is the multicast address `01-80-C2-00-00-00`. The use of a multicast address as the destination for all spanning tree BPDUs is important. This allows bridges to send BPDUs to all other bridges without having to know the unicast address of the bridge ports that will receive the BPDU, or whether there are any bridges on the link to hear the message at all. This address is within the range of addresses reserved by IEEE 802.1D for link-constrained protocols. Frames with destination address in the range of `01-80-C2-00-00-00` through `01-80-C2-00-00-0F` are never forwarded by an IEEE 802.1D-conforming bridge. This prevents the unwanted propagation of BPDUs beyond the link on which they are significant[8].

2.1.3 Mode of operation

The spanning tree topology for a given set of links and bridges is determined by the Bridge Identifiers, the link costs, and (if necessary) the Port Identifiers associated with the bridges in the network. Logically, we need to perform three operations:

Determine (elect) a Root Bridge. Since the desired topology is a tree, only one Root Bridge must be selected. The election algorithm is involves selecting the the bridge with the numerically lowest Bridge Identifiere. If a new bridge is added to the network, having a smaller Bridge Identifier, then it will become the Root Bridge. Similar, if the current Root Bridge disconnects from the network then the bridge with the next smallest Bridge Identifier will become the Root Bridge. Administrators have control over which bridge will be the default root, through the priority filed in the Bridge Identifier.

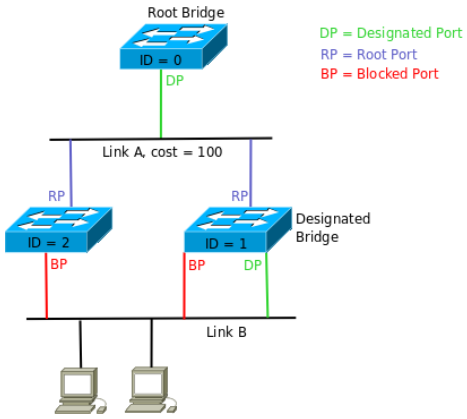

Figure 2.4: Designated bridge and ports

Determine the Designated Bridges and Designated Ports for each link. After the Root Bridge has been determined, there has to be identified, for each and every link in the network, a single bridge responsible for forwarding traffic from the root to that link. This is the Designated Bridge for the link in question. The Designated Bridge for each other link will be the bridge that offers the lowest-cost path back to the root. The path cost is simply the sum of the link costs over the path, with the link costs determined by the data rate of each link. It is possible that two bridges can offer the same path cost back to the root. In Figure 2.4, both bridge 1 and bridge 2 offer a path cost back to the root of 100 for link B and are thus equally qualified to serve as Designated Bridge for link B. In the event of such a tie, the bridge with the lowest-numbered Bridge Identifier will become the Designated Bridge. Similarly, it is possible that a Designated Bridge can have two ports on the same link, such as bridge 1. Only one of the ports can be the Designated Port for the link; the chosen port is the port with the lowest-numbered Port Identifier. Again, network administrators can decide in advance exactly which ports will become designated through manipulation of the priority

field in the Port Identifiers. The spanning tree is completely defined by the set of Designated Bridges (including the Root Bridge) and Designated Ports.[8]

Maintain the topology over time. Now that the spanning tree is complete, care must be taken to ensure that changes don't cause loops to form or portions of the network to become separated from the tree. The events that might instigate a change in the topology are the removal, addition, failure, or recovery of a bridge or link, or the reconfiguration of any of the spanning tree election parameters through network management. There may be a lot of bridges and/or links being added at once; the operation of the protocol accommodates this without invoking any special start-up mechanisms. The STP operates on the principle that all Designated Bridges (including the root) advertise their current understanding of the spanning tree and their internal state by emitting, on a regular basis through their Designated Ports, Configuration Messages (encoded as Bridge Protocol Data Units, or BPDUs). All bridges listen to these Configuration Messages and compare the advertised information to their own internal information. When a bridge's internal information (for example, Bridge Identifier or path cost) indicates that it has a better claim to become the Root Bridge, Designated Bridge, and so on, than that being advertised, it takes action to change the topology appropriately. When the spanning tree has converged to the proper topology, the regular emission of Configuration Messages maintains that topology, keeping inactive bridges and ports from becoming active and creating undesirable loops. In the event of a link or bridge failure, the lack of regular Configuration Messages from the now-failed link or bridge (i.e., a timeout) may cause previously inactive bridges and ports to become active in order to maintain maximum connectivity. The spanning tree topology will then re-converge to the best topology now available [8].

Steady-state operation. In normal operation, the protocol operates as follows:

- Once every Hello time (usually 2 seconds), the Root Bridge transmits a Configuration BPDU. This indicates that the sender is the Root Bridge and that the path cost is 0, because it is being sent by the root.
- All bridges connected to the Root Bridge receive the BPDU and pass it to the STP entity within the bridge. BPDUs are never forwarded through the bridge as are data frames from end stations.
- Designated Bridges use the received information and create a new Configuration BPDU, updating the values of the Bridge Identifier, path Cost, Port Identifier appropriately and send it on each Designated port.
- In turn, bridges connected to these, go through the same process of receiving the BPDU, analyzing it and sending it out on all Designated ports. This process repeats until the message has been sent to every node in the spanning tree.
- When a bridge receives one the mentioned BPDUs, it compares the infor-

mation from that BPDU with the locally stored information. If a bridge's own identifier is numerically lower than the currently advertised Root Identifier, then this bridge should attempt to become the root. It would so by initiating a topology change and then sending Configuration Messages with its own Bridge Identifier as the Root Identifier. Another possibility is that the bridge might have a lower-cost path to a given link than the cost currently advertised by a Designated Bridge. Again, if that is the case, it would initiate a topology change and attempt to become a Designated Bridge for that link.

2.1.4 Disadvantages

The main disadvantage of the Spanning Tree Protocol is its convergence time. The convergence time is the time it takes the network to recover after a topology modification. For STP this usually is somewhere between 30 to 50 seconds, depending on how well the network is designed and the quality of the protocol's parameters configuration.

Another disadvantage is that old MAC addresses are still present in the MAC table long after there has been a change in the network's topology. This can cause temporary loops since frames are sent on the wrong ports, due to invalid information found in the MAC table. This table needs to be relearned every time the network's topology changes or frames may be sent to the wrong ports. A topology change such as a link failure can cause some nodes to become connected to different switches. Even though no stations have been physically moved, it can appear to the switches as though stations have been lifted from one part of the network and reconnected into another. In order for traffic to reach these stations, switches need to age old information and relearn new node locations. STP bridges do not flush their MAC tables when they detect a topology change. Instead, they send Topology Change Notification BPDUs (TCN BPDUs) in the direction of the root bridge, which then informs all bridges a topology change has occurred. It may take several seconds before the TCN BPDU reaches the root bridge and several more seconds before the BPDUs (with the TC flag set) reach other bridges on the network. Even then, STP switches do not flush old information immediately.Instead, they switch their aging timer from long to short (a default value for the short timer is Forward Delay which equals 15 seconds). After this time, if entries are not refreshed they are removed from the database[9].

In conclusion, we can say that the bulk of the protocol's decision-making logic is handled by the root bridge. This is somewhat of a centralized solution, since all other bridges have to wait for the root bridge to make a decision, before making one themselves. This leads to increased convergence times and overall bad performance.

2.2 Rapid Spanning Tree Protocol

The Rapid Spanning Tree Protocol was introduced by the IEEE as 802.1w. This technology was then absorbed by IEEE 802.1D-2004. RSTP is based upon the older STP standard and is backward compatible. RSTP was created to provide faster recovery (convergence time) from topology changes and shares most of STP's characteristics.

2.2.1 Terminology

All the terminology from the Spanning tree Protocol is valid for the Rapid Spanning Tree Protocol too. The differences that exist will be pointed out in the following paragraphs.

Port States - Just as STP does, RSTP defines a set of operational sets that can be assigned to each port. These states are: Discarding, Learning and Forwarding. The Discarding state shows that a port does not participate in the active topology and does not learn MAC addresses. RSTP replaces STP's Disabled, Blocking and Listening states with the Discarding state. The other two states, Learning and Forwarding have the same function as their correspondents from the Spanning Tree Protocol.

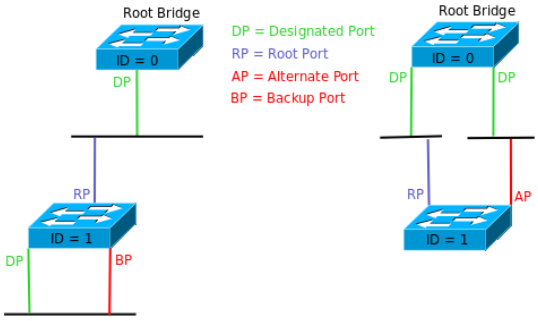

Figure 2.5: Alternate and backup ports

Port roles - RSTP can assign one of the following roles to a port : Root Port, Designated Port, Backup Port, Alternate Port. The Root and Designated roles have the same functionality as the roles with the same name from STP. The difference in this area between the two protocols is that RSTP splits the Blocked role into two different roles : Backup and Alternate. Ports having one of these roles do not forward user data, but serve as backups for the root and designated ports.

- *Backup port* - A backup port is connected to the same LAN as a designated port. Spanning tree blocks backup ports since the designated port provides a better path from this LAN to the root bridge. Figure 2.5 shows a Backup port.

- *Alternate port* - An alternate port provides a redundant connection to the Root Bridge and can become a new Root Port in the event the current Root Port loses its connection to a Root Bridge. In many cases, the alternate port can become a new root port and transition into the Forwarding state without a delay. Figure 2.5 shows an Alternate port[6].

2.2.2 BPDU format

RSTP uses only one type of BPDU called RSTP BPDUs. They are similar to STP Configuration BPDUs with the exception of a type field set to "version 2" for RSTP and "version 0" for STP, and a flag field carrying additional information. The STP BPDUs use only two flags: Topology Change and Topology Change Acknowledge. RSTP uses six additional bits to encode the role and the state of the port originating the BPDU, and two flags to handle the proposal/agreement mechanism. In 802.1D, a bridge that is not the root will produce a BPDU only when it receives one on the Root Port. In RSTP, a bridge will send a BPDU based on the hello time that is configured on the bridge (this is every two seconds by default). It no longer needs to receive a BPDU from the root bridge in order produce a BPDU and send it out[9].

The exact format of each type of BPDU can be seen in Appendix B.

2.2.3 Mode of operation

The Rapid Spanning Tree Protocol functions in almost the same manner as STP, with a few key differences. Concerning the algorithm, it follows the same ideas described in the Spanning Tree Protocol's mode of operation . This sub-chapter shows how RSTP runs on an example network, pointing out the areas where it differs from STP.

The initial RSTP convergence time is similar to that of STP. This is the time that passes between the moment the switches are powered up and connected and the moment when the topology has been computed and the network is stable. RSTP shows its advantages when subsequent changes occur, that is after the network has stabilized and all switches agree on the topology. Link failures occurring after that, cause the change in topology to be rapidly propagated throughout the network, much faster than in STP. Thus, in the given example, the focus lies on a scenario where the topology is stable, but a link has failed (the one marked with the red X).

The Spanning Tree Protocol acts in the following manner, when the link fails: after the link has failed, bridge 1 and bridge 3 continue to wait for the duration of the max age timer (which has a default value of 20 seconds) before deciding that their path to the root bridge is no longer functioning. During that time, bridge

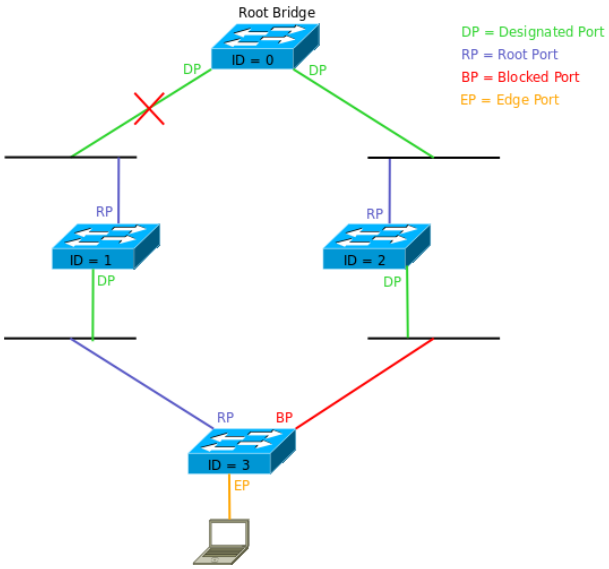

Figure 2.6: Example network for RSTP's mode of operation

3 does not accept BPDUs from bridge 2, marking them as inferior. After the max age timer has expired, bridge 3 ages out protocol information on the port connected to bridge 1, deciding that it has a path to the root bridge through the other port. It elects that port as the new Root Port and sends BPDUs containing this information to bridge 1. In order to ensure that all switches on the LAN agree with the new topology, bridge 3 will not forward user data on its ports for an additional 30 seconds, instead taking the ports through the listening and learning states, before reaching the forwarding state. Eventually, bridge 1 will figure out that it has a path to the Root Bridge through bridge 3 so it will mark the port connecting it to that bridge as a Root Port.

On the other hand, RSTP goes through the following steps. When bridge 1 loses connectivity, it decides that it is the new root bridge and starts advertising that to bridge 3. Bridge 3, recognizes that BPDUs received from bridge 1 are inferior to that from bridge 2, so that means that it has no longer a path to the Root Bridge. Thus, it immediately activates the secondary path through the Backup Port making it the new Root Port and placing it directly in the forwarding state. After that, it makes the previous Root Port a Designated Port and starts sending BPDUs containing all this information. Bridge 1 accepts the information and chooses the new Root Port while bridge 2 goes through a process known as a sync operation with bridge 3 in order to move the port connecting them to the forwarding state. This sync does not involve any timers, only BPDUs and thus it is faster than the STP method[9].

2.2.4 Improvements over STP

As mentioned in the previous sub-chapter, STP makes the topology change notification go through the Root Bridge first, and only after that it is distributed in the rest of the network. The 30 second wait in STP takes into account that the BPDU has to reach the Root Bridge before it reaches all other bridges. Thus, STP switches do not generate their own BPDUs, they wait to receive them on their root ports, and then they relay them to their designated ports. If the STP bridge does not receive a BPDU for max age time (usually 20 seconds), it declares the root bridge dead. RSTP differs in this respect, because every switch sends its own BPDU whether it received one on its root port or not. Instead of waiting for a max age time, an RSTP switch expects a BPDU within three hello times, which means 6 seconds. This time is the time it takes for the protocol to detect that a link has failed. Recovery after this detection has been made, is a lot faster in RSTP due to these improvements. Thus, RSTP manages to decentralize the protocol, as Figure 2.7 shows[6].

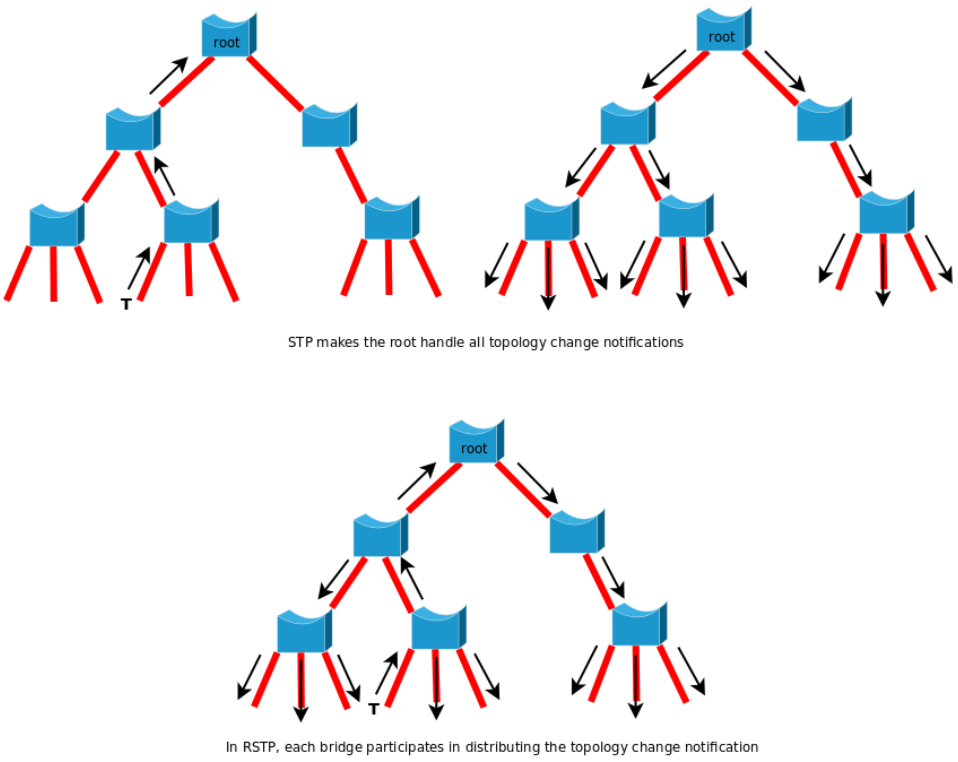

Figure 2.7: Comparison of topology change notifications

RSTP recognizes that there are situations where ports connect directly to end stations and thus cannot create loops. Taking this into consideration, RSTP allows ports to be configured as edge ports. This means that these ports do not connect directly to other switches. Such ports do not go through the usual spanning tree states, they go directly to forwarding. If a switch detects a BPDU

on the edge ports, it declares that port as a non-edge port. One example of an edge port can be seen in Figure 2.6.

RSTP bridges, connected by point-to-point links, use the agreement/proposal handshake mechanism instead of timers, to rapidly re-converge after a topology change. When the bridges detect that the topology has changed they try to transfer their new root and designated ports to forwarding and their alternate and backup ports into blocking as fast as possible. This can be done by using an agreement/proposal handshake mechanism which purpose is to avoid loops and to ensure consistent assignment of port roles across the network.

Switches listen for network traffic to learn which nodes (MAC addresses) are on which ports, then store these MAC-to-port entries in their databases. Later, when a frame arrives destined for one of these MAC address, it will be switched to the proper port. A database of these MAC-to-port entries is called a filtering database. This database needs to be relearned every time a network topology changes or frames may be sent to the wrong ports. A topology change such as a link failure can cause some nodes to become connected to different switches. Even though no stations have been physically moved, it can appear to the switches as though stations have been lifted from one part of the network and reconnected into another. In order for traffic to reach these stations, switches need to age old information and relearn new node locations.

STP bridges do not flush their filtering databases when they detect a topology change. Instead, they send Topology Change Notification BPDUs (TCN BPDUs) in the direction of the root bridge, which then informs all bridges a topology change has occurred. It may take several seconds before the TCN BPDU reaches the root bridge and several more seconds before the BPDUs (with the TC flag set) reach other bridges on the network. Even then, STP switches do not flush old information immediately. Instead, they switch their aging timer from long to short (a default value for the short timer is Forward Delay which equals 15 seconds). After this time, if entries are not refreshed they are removed from the database[9].

RSTP uses a more efficient mechanism to purge old information. First of all, every switch that detects a topology change sends BPDUs with the TC flag set. Secondly, the switch that detects a change purges old entries immediately. Finally, every switch that receives a BPDU with a set TC flag, purges old entries immediately and ask other switches to do the same.

The end result of all these improvements is that RSTP has a re-convergence time between tens of milliseconds to a few seconds. Compared to STP's 30 to 60 seconds, it is definitely a major performance gain.

Chapter 3

Architecture

The application's goal is to implement the Rapid Spanning Tree Protocol and integrate its functionality with LiSA, an open-source project that aims to bring switching capabilities to embedded devices. From an implementation point of view the application can be seen as having two components : the protocol's logic, which resides in user space, and a kernel space part that handles the frames, learning and forwarding processes[1]. The latter part requires modifications to the existing LiSA kernel code so STP/RSTP frames will be sent to our implementation. Also, new coded will be added to allow for learning and forwarding to be enabled and disabled.

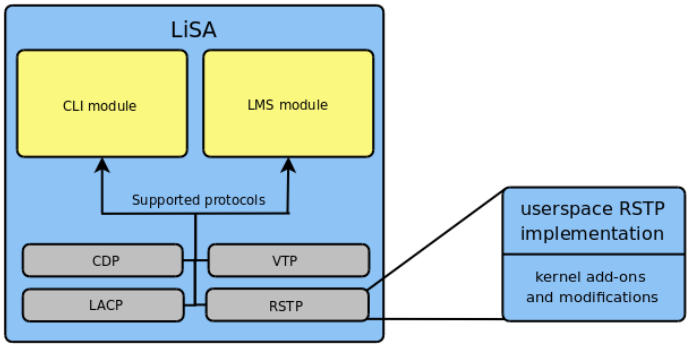

Figure 3.1: Top-down view

Figure 3.1 shows a generic view on the application and its relationship with the existing LiSA project. Our goal is to add RSTP to the existing list of supported protocols. The implemented protocols require interaction with the existing modules, namely CLI and LMS, which will be described in the chapters to come. The application tries to maintain compatibility with current RSTP implementations by testing against devices that present a fully functioning protocol, such

[1]The term process should not be confused with the Unix term with the same name. Here, and whenever it is in the same context as the learning and forwarding terms, it means *activity*

as modern Cisco switches. This will ensure that any errors in the program's logic will show up, because the testing environment is a network with a protocol implementation that is guaranteed to be correct.

Given that the chosen protocol for implementation is RSTP, this also guarantees backward-compatibility with the older STP protocol. The 802.1D standard is closely followed so that all aspects are covered, including the one previously mentioned.

3.1 LiSA

LiSA is an open-source application that started out as a graduate project in 2005 and aimed at:

- Layer 2 and layer 3 packet switching using a standard PC architecture
- Resolving Linux VLAN scalability issues
- Resolving performance with broadcast packets on both trunk and access ports
- Providing basic VLAN switching features: VLAN switching, VLAN tagging, inter-VLAN routing
- Cisco-like configuration and user interface

More than that, it also aimed at becoming a framework for layer 2 protocols prototyping and analysis. Layer 2 protocol implementation is not a trivial thing to do in the Linux kernel because the bridge module is not easily extendible. Thus, the framework should hide all of Linux's networking internals and provide a clean API allowing for focus on protocol design and less on implementation issues[10].

As mentioned earlier, LiSA's goal is to provide a unified switching platform for the Linux operating system. Linux already has several separate modules that provide some of the functionalities that LiSA currently offers, such as the bridge module that provides basic switching and the 8021q module providing VLAN support. Tests have proven though [11] , that a unified approach that delivers both functionalities in a single module yields better results in terms of performance. By eliminating the need for virtual interfaces in per-VLAN packet switching and reordering ports during packet flooding, thus providing a minimum number of socket buffer operations, LiSA manages to perform better than the two modules previously mentioned.

LiSA's architecture covers both kernel space and user space. A high-level view on LiSA would consist of a kernel module that implements the switching functionality, and the rest of the user space applications designed to add new protocols

or to present an interface to the user. The kernel module's main task is to implement the forwarding logic. In doing so, it has to interact with at least two additional components : the forwarding table and the VLAN table. Thus, from a high-point of view the kernel module takes as input packets from the network, makes a decision based on data from the two tables mentioned before and it outputs the packet on the corresponding port. Also, the kernel module has to provide an interface to the userspace, through which users can read and modify different switch-specific parameters, such as VLAN assignment, priority change, etc. Applications on the user side concern themselves with handling user input and implementing protocols using the interface provided by the kernel. Communication between them and the kernel module is done through the Linux-provided ioctl system call.

A more detailed view on LiSA's architecture would show that structurally, the project can be divided into three main components:

- a kernel module (LMS, or Linux Multilayer Switch)
- an userspace application for configuring the parameters (CLI, or Command Line Interface)
- a lightweight Linux distribution targeted at embedded systems

The kernel module is the one responsible for all switching-related logic, that is, frame handling, forwarding table and VLAN table manipulation, providing ioctl calls to userspace, and so on. Because most network administrators are familiar with the interface provided by Cisco IOS's CLI, a similar interface is also presented to the user through the CLI module. Thus, the CLI module acts as the link between the user and the actual switch functionality.

3.1.1 Linux Multilayer Switch

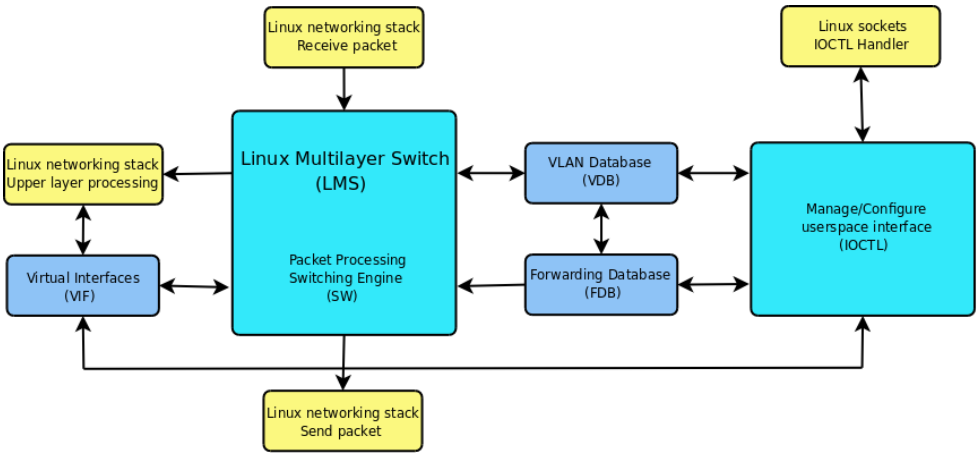

Figure 3.2: LMS architecture

Figure 3.2 shows the elements composing the Linux Multilayer Switch[12]:

- Switching engine (SW) - implements the following : packet receiving, switching decision, required algorithms for sending the packet on the appropriate port. When a frame is received by the network driver and passed to the kernel for processing, it is passed to the Switching Engine. If the incoming interface has been registered in the switch, the packet will be handled by the Switching Engine alone, and it will not be passed to the rest of the Linux packet reception code. If not, the packet will be handed off to the rest of the kernel reception routine.
- Forwarding Database (FDB) - contains all the routines required for accessing and modifying the data structure used for implementing the switch's forwarding database plus the data structure itself. This database is initially empty and entries are added as the SW receives frames. If an entry is not found in this database, then the received frame is sent on all ports except the one it came from. This database will have to be cleared when RSTP decides that a change in topology has occurred, or, in case STP is running, the aging times will have to be lowered.
- VLAN Database (VDB) - contains all the routines required for accessing and modifying the data structure used for implementing the switch's VLAN database plus the database itself. The database contains the list of all in-use VLANs and the specific configuration options pertaining to each VLAN, such as the name, aging time, etc. The VDB is useful for expanding the RSTP into an equivalent, per-VLAN protocol called MSTP.
- Virtual Interfaces (VIF) - the implementation of a generic net_device[1] along with its associated methods. Virtual Interfaces provide the easiest way of implementing inter-VLAN routing.
- Userspace configuration (IOCTL) - handles configuration commands received from userspace through the ioctl() call. This is a standard method for user-to-kernel communication used in many operating systems. Short for input/output control, ioctls are usually employed to allow userspace to communicate with hardware devices and kernel components. This method of using one system call for communication between userspace and device-drivers was chosen because of the large number of devices available, which would make creating system calls for each of them an impossible task. It is used in the RSTP implementation too, for enabling and disabling the learning and forwarding routines, since this requires access to the kernel part of LiSA.

[1]The net_device data structure stores all information regarding a network device. There is one such structure for each device, both real ones (such as Ethernet NICs) and virtual ones (such as bonding). The data structure is defined in include/linux/netdevice.h)

3.1.2 Command Line Interface

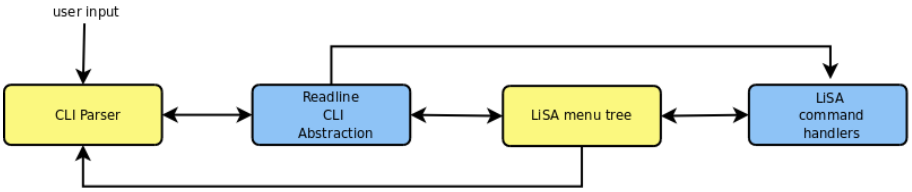

Figure 3.3: CLI architecture

Figure 3.3 shows the elements composing the Command Line Interface:

- CLI Parser - API for parsing Cisco-like commands; provides basic tokenizing functions and validation against menu tree structures. The tokenizing function is initially called with the whole command as input and the root menu tree node as context. As it extracts the first token from the input, it is iteratively called on the remaining input. A CLI command is formed from a succession of words separated by a variable number of white spaces. Commands can be in full or abbreviated formats, just like their Cisco counterparts, and the CLI is capable of handling both situations.
- Readline CLI abstraction - Integrates the CLI Parser with readline library, providing Cisco-like CLI behavior. The GNU Readline library provides a set of functions for use by applications that allow users to edit command lines as they are typed in. The Readline library includes additional functions to maintain a list of previously-entered command lines, to recall and perhaps reedit those lines, and perform csh-like history expansion on previous commands.
- LiSA Menu Tree - Data structures for all LiSA CLI commands. These data structures are defined using C's dot notation to refer to each of the structure's fields. This allows definitions to be chained in a tree-like structure, starting from the root node, going all the way down to the complete commands. Different trees are shown in the CLI, depending on the user's privilege level. Details will be given in the following chapter.
- LiSA Command Handlers - Functions that actually execute the CLI commands. These are called when a user enters a valid command, and they usually interact with the shared memory area, where all switch configurations are stored. Depending on the command, they also might have to resort to interprocess communication mechanisms in case the command's target resides in another process.

One last thing to mention about the CLI component is that it is in fact, generic and it is not necessarily tied to LiSA. To do that, a layered approach has been taken when designing the CLI. Just like any other layered stack, such as the OSI networking stack, the main idea is that lower layers don't have to know

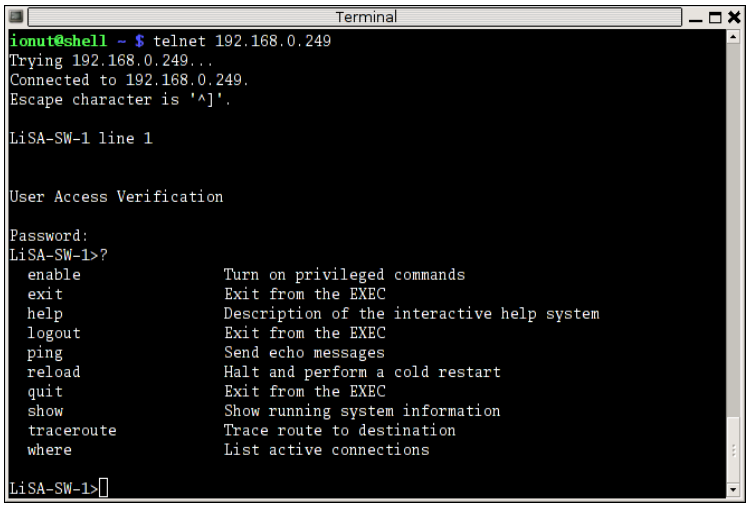

Figure 3.4: CLI snapshot

anything about the upper layer implementation, while the upper layers use the whole functionality provided by the lower layers. Thus, the bottom-most layer is a generic layer, that defines simple required data structures in order to properly function. The next layer is known as the readline shell, or rlshell, and it extends, in an object-oriented manner the data structure defined by the lower layer. Doing something like this in C, which is not an object-oriented programming language, requires that the upper structure has a field containing the lower structure. The final layer of the CLI is the actual LiSA-bound component, and it encapsulates structures from both lower layers. Such an approach means that the CLI could easily be extendible to another application, using the lowest layer as a starting point.

3.1.3 Linux distribution

The idea behind the Linux distribution is to provide a minimalistic operating system that can run on an embedded device. The result was a distribution that occupied a little over 12MB of physical disk space and contained all the packages required for the operating system to boot and run, plus the libraries required to support LiSA. These libraries are mostly tied to the Command Line Interface, such as the GNU Readline library.

Figure 3.5 shows the general architecture of a system that is running Linux compiled with Linux Multilayer Switch support, and has the Command Line Interface set up. The system can be configured both through a serial-line connection and a remote telnet session. The serial connection is important in the initial setup phase when the switch is not properly configured. During this stage, the network administrator has to set passwords and define a virtual interface through which remote connections can be set up. For remote connections the system has a dae-

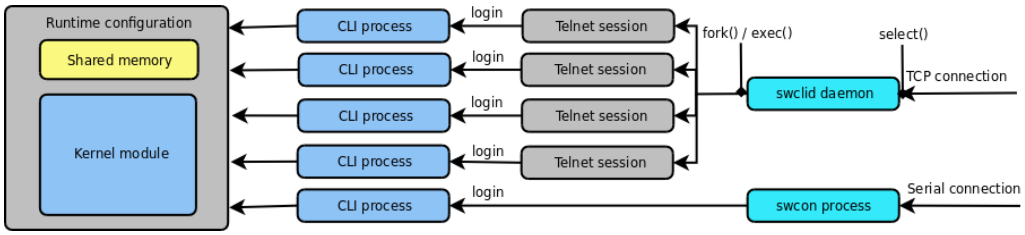

Figure 3.5: Multiplexing multiple CLI connections on a Linux system

mon listening on port 23 that multiplexes all incoming connections. For every connection a new process is created that initially executes an authentication program called swlogin. If the user is properly authenticated then the login program will successfully launch the CLI. For a serial-line connection the swcon program will launch the authentication routine only if the system has passed through the previously mentioned setup phase.

The runtime configuration of the system contains the kernel module and a shared memory area. This area is accessed indirectly by all CLI processes, when they issue commands. As mentioned before, most commands act upon configuration parameters of different protocols or even the switch itself. These parameters are stored in this shared memory. Naturally, this area has to be protected from concurrent accesses, that is, a mechanism for stopping two different processes from modifying the same variable at the same time is required. Such a mechanism is already implemented, but new variables defined in this area will have to implement their own wrappers on top of this mechanism, so processes can safely access them.

3.2 RSTP module architecture

The architectural design of the RSTP implementation has to answer the following questions:

- Where does the input come from, and how many types of input are there?
- What are the constraints imposed by the protocol's logic?
- How can the implementation be integrated with LiSA?

The above figure was the result of putting the answers to all the questions together.

First, there are two types of input that have the capability of triggering changes in the execution of RSTP's logic. There is user input, manifested when the user modifies RSTP's running parameters and there are received BPDUs, which, depending on their contents, might trigger a topology change and put the protocol to work. User input is handled by the CLI module, which offers a interface

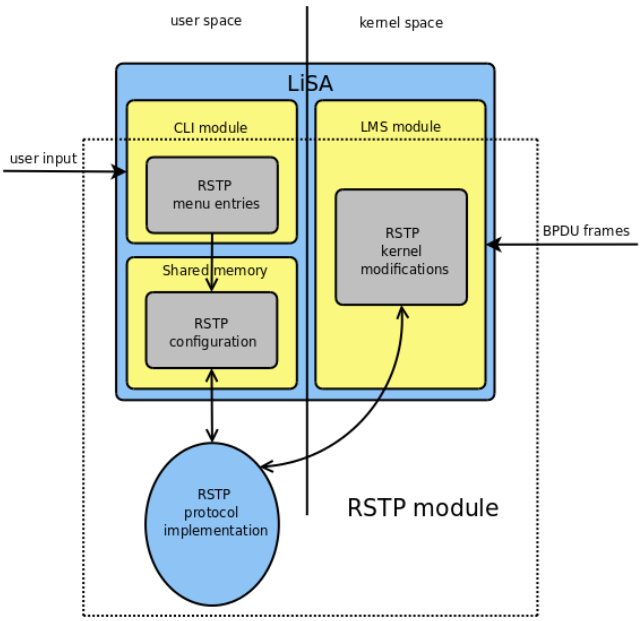

Figure 3.6: RSTP implementation architecture

through which users can alter the protocol's configuration at any time. That means the module has to have entries that specifically alter RSTP's parameters. That is one the first tasks our module has to accomplish. The arrow from the RSTP menu entries to the RSTP configuration means that the entries are capable of altering variables existing in that shared memory area[1].

The other type of input are received BPDUs. These will end up all the way up to the protocol implementation since the latter depends on their content. The part of the implementation handling this type of input is the LMS module. More specifically, it is the RSTP kernel modifications to the LMS module that allow it to handle the BPDUs. Without these modifications, BPDUs would simply be lost, since no STP entity exists in LiSA. With the mentioned modifications in place, the BPDUs are now sent to the protocol implementation, where their contents will be interpreted and acted upon. The arc between the RSTP kernel modifications and the protocol has an arrow at both heads, since the protocol can call upon routines defined in that area. These routines implement the enabling and disabling of the forwarding and learning actions.

The second question finds answers in the previous two paragraphs. The protocol's logic dictates that when a BPDU is received, one of the state machines shall be triggered and will start executing. This will have an avalanche effect, since all other state machines will find that the conditions they are waiting upon are now

[1]Actually, one might think looking at the picture, that the module contains the whole shared memory area, but that is not the case. It only contains the part of the memory where RSTP's configuration resides. The picture would have been uselessly complicated if I were to draw the box so that only that part were contained

fulfilled, as a result of the received BPDU. Not only BPDUs can have this effect, since the protocol depends on a number of timers to do its job. When these timers expire, some of the state machines might have to execute as a result. Most of these timers are set with values that can be changed by the user, when a corresponding entry from the menu tree is used.

So far, the constraints have been only about input, but what about output? In order for the protocol to operate correctly, we have to make sure that the implementation is fast enough to be able to process received BPDUs at an acceptable rate[1]. This is a very important constraint that will become obvious when we discuss the protocol's implementation. The decisions on how to implement the protocol can have a great impact on its speed, and it can go as far as by making a bad choice during the program's design process, the implementation will not give expected results. Other constraints relate to the internals of the protocol and will be mentioned in the following sub-chapters.

As for the final question, related to integration with LiSA, it also finds an answer in the figure. The protocol's implementation is relatively independent of LiSA, since it only needs to receive BPDUs as input, and has to have access to the protocol's configuration parameters. We've mentioned earlier that LiSA provides a good framework for user space protocol implementation, so the final choice was to implement the protocol entirely in user space, and communicate with the kernel side when the switch's ports would change their state. Thus, the integration with LiSA consisted of adding some kernel code so communication between the user space part and the kernel space part would be possible through the ioctl interface. Another step towards integration is adding menu entries that respect the format and implementation guidelines described earlier in the CLI sub-chapter.

3.2.1 Kernel sub-module

The kernel sub-module actually refers to the kernel modifications presented in Figure 3.6. A more detailed description can be seen in the following figure.

The sub-module has two important roles to play:

- Identify incoming BPDUs and send them to the user space protocol implementation.
- Allow the user space protocol implementation to enable or disable the learning and forwarding process.

Identifying BPDUs is only a matter of comparing the incoming frames' DSAP, SSAP and LLC types against the correct values. Once a frame matches these values it gets sent to the upper layers, where our protocol implementation resides.

[1]The 802.1D standard mentions that a bridge will send no more than txCount BPDUs per second out of a port. The variable txCount's value defaults to 6.

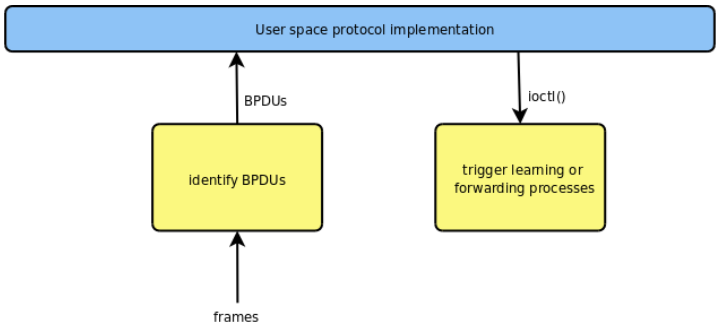

Figure 3.7: RSTP kernel sub-module

Concerning the enabling and disabling of the learning and forwarding processes, this is done by using ioctl calls from the protocol implementation.

3.2.2 Userspace RSTP implementation

This part of the application handles the protocol's logic. This is the part where all the decisions are made concerning port states and roles, and it represents the core of the entire implementation. Without going into the exact details, and viewing it as a black box, the implementation takes as input either BPDUs or user modifications to the protocol's configuration, and produces as output other BPDUs and makes decisions related to the ports' behaviour. This can be seen in Figure 3.8.

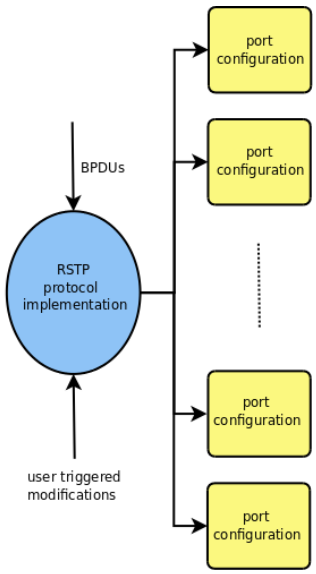

Figure 3.8: High-level view on the implementation and its effects

BPDUs are received because of the kernel modifications that allow frames to be identified and correctly forwarded to the implementation. Of course there has

to be a mechanism that allows BPDUs from different ports to reach the implementation, and to remember from what ports the BPDUs came since part of the protocol's logic relies on this information. User input is handled by the CLI entries we previously talked about. Through these entries, a user is able to modify both the global configuration of the protocol and the individual ports' configuration. Thus, indirectly, the protocol implementation takes these modifications as input, since it accesses these configurations when executing the logic.

The protocol makes modifications on each port's configuration in order to get the port in a different state, or role. More than that, the implementation also handles sending new BPDUs on each port. These BPDUs have to be created and their fields have to have the right values. For that, the protocol requires storage for each port, and this is the first sign that the logic has to be split on a per-port basis.

Going into the actual protocol's details, as Appendix A shows, the protocol's logic is implemented with the help of finite state machines. There are a total of 10 state machines, one of which is not shown in the picture. These are:

- *Port Timers state machine* (PTI)- This state machine's sole job is to decrement all the timers related to a specific port. There are nine timers per port that have to be decremented. This is the state machine not shown in the picture.
- *Port Receive state machine* (PRX) - This machine handles incoming BPDUs, by checking their version and updating variables as necessary.It marks the BPDU as being a STP-type BPDU, that is either a Configuration BPDU or a Topology Change BPDU, or an RSTP-type BPDU. The format of these BPDUs has been described earlier in the paper. This machine also signals that a BPDU has been received to the Port Information state machine.
- *Port Protocol Migration state machine* (PPM) - The job of this FSM is to fall back on the STP implementation in case an STP-type BPDU is received. If a switch that does not support RSTP is added to the network, then this machine will see that the received BPDU is correlated to an STP implementation, so it will signal all other FSMs that the RSTP logic is no longer in effect, and that the switch will run STP logic.
- *Bridge Detection state machine* (BDM) - RSTP introduces a new role for ports, called Edge port. This role can be assigned to a port by a network administrator when the port is sure to be connected to an end station and not to another switch. Thus, the port is immediately moved in the forwarding state, and does not count when the spanning tree is built. This machine enables and disables the flag that signals if a port is an edge port.
- *Port Transmit state machine* (PTX) - This FSM implements the sending of BPDUs. It collects information created by the other FSMs and puts it into a new BPDU after which it sends the BPDU on the port. The type

of BPDU it sends is given by a few flags that are set by the Port Protocol Migration machine.

- *Port Information state machine* (PIM) - This handles the proposal/agreement mechanism discussed earlier on. It also updates the port's configuration with the information received from the BPDU, if the latter is better than the one stored. For example, when a bridge auto-proclaims to be the Root Bridge, it might receive a BPDU that says that another bridge, having a better Bridge ID, is the Root Bridge. If that is true, then the auto-proclaiming bridge shall update the internal configuration, acknowledging that another bridge is the Root Bridge.
- *Port Role Selection state machine* (PRS) - This machine's task is clear from its name. It has to select the roles of every port from the bridge.
- *Port Role Transitions state machine* (PRT) - This machine implements each role's logic. That is, if a port has the Disabled Role, then it means that the port must block messages. The machine makes sure that the port behaves according to the role it has. It is comprised of four different state machines, one for each role : Disabled, Root, Designated, Alternate or Backup.
- *Port State Transition state machine* (PST) - The FSM handles transitions from the blocked and learning states to the forwarding states, and vice-versa. It is also the machine that uses the ioctl calls defined in the kernel modifications for enabling and disabling learning and forwarding.
- *Topology Change state machine* (TCN) - Lastly, this implements the protocol's logic when a change has occurred in the network's topology. It helps rebuild the spanning tree if necessary. A single Port Role Selection state machine is implemented for the whole bridge, and one instance of each of the other state machines are implemented per port.

Now that we have a look on the way the protocol is implemented it is time for the architecture to be updated as well. The RSTP protocol implementation has the architecture shown in Figure 3.9.

Each port shall have its own set of running state machines that have access only to that port's variables. The only machine that is shared between ports is the Port Role Selection machine. The machines run asynchronously per port and on a bridge level. In a port, the machines are connected only through the variables they share. The lines between the FSMs are there to show that they share some variables, and are connected that way. The full diagram remains the figure from Appendix A.

BPDUs are sent to the appropriate Port Receive machine using a mechanism that will be detailed in the implementation chapter. Each FSM group shall work independently of another and will send BPDUs based only on that port's variables and received BPDU. The PTX machines only have access to the port they belong.

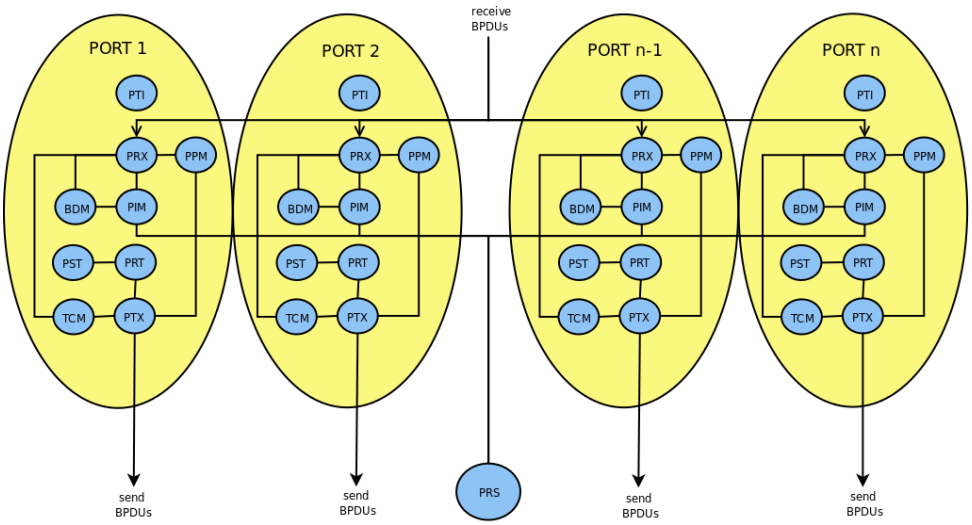

Figure 3.9: RSTP implementation

The Port Role Selection machine monitors variables in all ports and makes decisions based on them. These decisions will influence the outcome of the respective port's logic. As it can be seen, there is a high degree of asynchronism both on a port level, and on a bridge level. Inside a port, different FSMs share that port's configuration variables, and may try to access them simultaneously. More than that, the FSMs wait for conditions to be true before jumping to the next state. Looking at the switch as a whole, each port's FSM group has to synchronize with the PRS machine.

Figure 3.9 does not show the shared memory area where the protocol's configuration resides. Sometimes, the ports have to access this area because they need information that is bridge-specific, such as the bridge's ID and timer values. This is yet another level on which the ports' FSM groups have to synchronize. Similarly, a user might decide to modify the RSTP's running parameters for a specific port, therefore the port's variables are not only accessed by the FSMs belonging to the port, but also by the CLI module, with the help of the entries we've mentioned before.

3.2.3 CLI sub-module

The CLI sub-module's main task is to accept user input and change the protocol's running parameters accordingly. Also it has to offer the option of disabling the RSTP protocol on all ports or for each individual port.

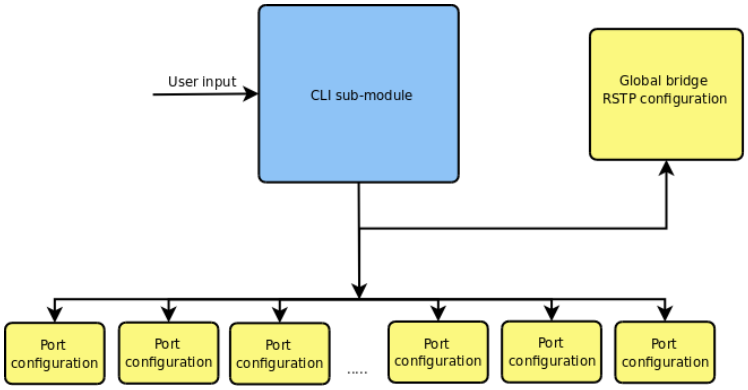

Figure 3.10: CLI sub-module

Moreover, the sub-module has access to the shared memory area where the global bridge-specific configuration resides. Since this part is not the only one that needs to access that area, synchronization issues appear.

Fortunately, LiSA contains an implementation of a mutually exclusive shared memory area, so the implementation will only have to worry about locking and unlocking the associated mutex.

Chapter 4

Implementation

Until now, we have described the architecture of the application. In this chapter, we present the actual implementation, focusing on the low level, programming-related components, and departing ourselves from the high-level view previously employed. Thus, there are three steps that we've followed in implementing the application, steps that are closely related to the architecture itself:

- Integration of the protocol's implementation with LiSA
- Adding CLI entries for the RSTP protocol
- Implementing the RSTP protocol

Before presenting the three steps, let us focus on the implementation idea of the whole application. This is presented in Figure 4.1. The application will run as a daemon[1] and will interact with the CLI process and the LMS part of LiSA.

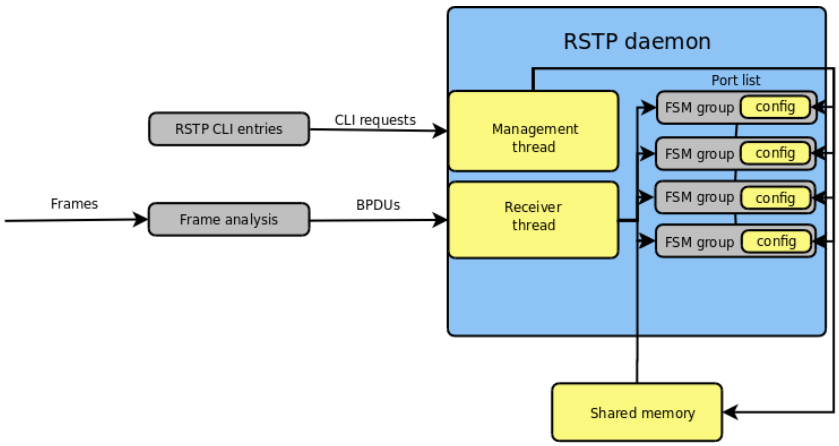

Figure 4.1: Implementation architecture

[1]In Unix and other computer multitasking operating systems, a daemon is a computer program that runs in the background, rather than under the direct control of a user

The daemon starts by creating two threads:

- A *management thread* whose role is to communicate with the CLI process and receive input from the user
- A *receiver thread* that implements the multiplexing of incoming BPDUs. It will forward the received BPDU to the corresponding port. Each port has a circular buffer where BPDUs are stored. This thread will put received BPDUs in the respective buffer and signal the Port Receive state machine that a BPDU has arrived.

LiSA maintains a shared memory area where global configuration options are stored. The management thread and each port will be able to access it.

BPDUs are received with the help of the kernel modifications made in order to recognize BPDU frames. Frame analysis is done by code in the kernel part of LiSA.

The daemon also maintains a list of all monitored ports. Since it is not necessary that all ports have RSTP enabled on them, only the ports that do are included in the list. Each port has an associated structure where all variables and timers are stored. When a user enables RSTP on a port by issuing the *rstp enable* command in the *configure terminal* menu, memory is allocated for such a structure and the port is added to the list. Each port will have an associated group of finite state machines. The exact details of how these are implemented will be described in the following chapters.

4.1 RSTP integration with LiSA

It may seem odd to start with this since in an initial point, there is no implementation to speak of. Actually, an important port of the integration process is represented by getting the appropriate input from LiSA. LiSA only forwards to the upper layers those frames that it can identify, meaning that BPDUs would not reach our implementation if we didn't make the appropriate modifications. Frame identification is done in the kernel code of LiSA's implementation, in the *sw_socket_filter* routine. All frames are filtered through this routine. If a protocol is recognized then the socket buffer gets enqueued in the appropriate switch socket and the packet does not get in the forwarding algorithm.

The socket buffer is the most fundamental data structure in the Linux kernel networking code. Every packet sent or received is handled using this data structure. The modifications we make to the *sw_socket_filter* routine use this buffer to access the layer 2 data of the received packet. We've mentioned earlier that a BPDU is identified by having the DSAP and SSAP fields equal to 0x42 and an LLC Control field value of 0x03. Thus, the code that enables LiSA to send BPDU frames to our implementation is this:

```
1 if (skb->data[0] == 0x42 && skb->data[1] == 0x042 && skb->data[2] ==
      0x03) {
2         dbg("Identified␣RSTP␣frame␣on␣%s\n", port->dev->name);
3         list_for_each_entry_rcu(sw_sk, &port->sock_rstp, port_chain)
              {
4                 atomic_inc(&skb->users);
5                 handled |= sw_socket_enqueue(skb, port->dev, sw_sk);
6         }
7         goto out;
8 }
```

After this code has been inserted, the question that remains to be answered is how exactly does an user space application receive the frame? By creating a *struct list_head* field called sock_rstp in LiSA's *struct net_switch_port*, and choosing a globally unique constant by which we can identify our protocol in LiSA, a user space application can use the socket interface to receive frames. By setting the *ssw_proto* field of the *struct sockaddr_sw* defined in LiSA, to the value of the constant we previously defined, sending and receiving frames is just a matter of reading and writing on a socket.

After doing this modification we now have one of the inputs to our implementation, the BPDUs. The protocol implementation will have to open up a socket for each port, and listen to modifications on those sockets, so that it properly receives the BPDUs.

There is another type of input we require, before going into the protocol implementation : entries in the CLI menu.

4.2 CLI entries

Most protocols have a small number of configurable parameters that influence the protocol's behaviour. In our case, the RSTP parameters are stored in a shared memory area accessible by all userspace LiSA related processes. The next step in the implementation is to allow the user to modify these variables with the help of the CLI component of LiSA.

First thing to do is to add an entry in the Cisco-like menu, to allow the user to enable or disable RSTP support on all ports. This is done by adding an instance of the *menu_node* structure to the existing menu tree. The *menu_node* structure has the following definition:

```
1 /* Command tree menu node */
2 struct menu_node {
3         /* Complete name of the menu node */
4         const char *name;
5
6         /* Help message */
```

```
7         const char *help;
8
9         /* Bitwise mask for filtering */
10        uint32_t *mask;
11
12        /* Custom tokenize function for the node */
13        int (*tokenize)(struct cli_context *ctx, const char *buf,
              struct menu_node **tree, struct tokenize_out *out);
14
15        /* Command handler for runnable nodes; */
16        int (*run)(struct cli_context *ctx, int argc, char **tokv,
              struct menu_node **nodev);
17
18        /* Additional data that a custom tokenize function may use
              */
19        void *priv;
20
21        /* Points to the sub menu of the node */
22        struct menu_node **subtree;
23 };
```

The *name* and *help* fields are what the user sees in the menu and the *mask* field is used for filtering entries based on privilege levels. The next two routines are *tokenize* which allows for custom tokenizing of user input, followed by *run* which is the routine called by the user when he selects the corresponding menu entry. The *priv* field is used to store private data, and the *subtree* field allows multiple *menu_ node*s to be chained in a tree-like structure.

In the end, an RSTP entry in the menu structure, looks like this:

```
1  (struct menu_node){
2        .name     = "rstp",
3        .help     = "Global␣RSTP␣configuration␣subcommands",
4        .mask     = CLI_MASK(PRIV(15)),
5        .tokenize = NULL,
6        .run      = NULL,
7        .subtree  = (struct menu_node *[]) {
8          & (struct menu_node){
9            .name     = "run",
10           .help     = "",
11           .mask     = CLI_MASK(PRIV(15)),
12           .tokenize = NULL,
13           .run      = cmd_rstp_run,
14           .subtree  = NULL
15         },
16
17         NULL
18       }
```

In this example, we have the *rstp run* command given by the user when he wants to enable support for the RSTP protocol. This can be compared to a global switch that enables and disables the protocol on all ports. This is added in the

config_ main tree which offers the user diverse options of modifying the running configuration of the switch. The user enters this menu branch after typing the *configure terminal* command in the CLI. The final result looks like this:

Also, the user has to be able to turn RSTP on or off on a per-port basis. This is done by adding a similar entry in the config_if_main tree that the user accesses when selecting a specific interface to configure by giving the command *int Ethernet <0-24>*.

Before going into the details of how the routines from the *run* field are implemented , we have to go through the methods of communication between the CLI and external processes. There are two such methods : shared memory and message queues.

Shared memory is used to allow LiSA related processes to access global switch information, that is, information that is not directly related to any of the ports. As an example of such information we could give the switch's priority in the RSTP protocol that determines which switch will end up being the Root Bridge.

The second method uses software engineering components usually used for interprocess communication: message queues. Message queues provide an asynchronous communications protocol, meaning that the sender and receiver of the message do not need to interact with the message queue at the same time. Messages placed onto the queue are stored until the recipient retrieves them. These are used when the user wants to modify port information or makes queries that involve receiving an answer back from the RSTP daemon. This answer does not have to be stored anywhere since the user will only want it to show up on the screen.

The shared memory structure has the following format:

```
1  /* Switch shared memory structure */
2  struct shared {
3          /* Enable secrets (crypted) */
4          struct {
5                  char secret[SW_SECRET_LEN + 1];
6          } enable[SW_MAX_ENABLE+1];
7          /* Line vty passwords (clear text) */
8          struct {
9                  char passwd[SW_PASS_LEN + 1];
10         } vty[SW_MAX_VTY + 1];
11         /* CDP configuration */
12         struct cdp_configuration cdp;
13         /* RSTP configuration */
14         struct rstp_configuration rstp;
15         /* List of interface tags */
16         struct mm_list_head if_tags;
17 };
```

RSTP specific information is contained in the *rstp* field, of type *struct rstp_configuration*. This structure contains all bridge variables along with different flags that enable the protocol. The RSTP configuration from the shared memory structure is accessed using setter/getter methods that we define. Each protocol configuration, such as CDP and RSTP previously mentioned, is manipulated using such methods.

Concerning the message queue, there had to be a globally defined name so that all interested processes could access it. The task of defining such a name and creating the message queue was delegated to the management thread contained in the RSTP daemon. The queues used for communication between the CLI and the RSTP daemon are named */lisa-rstp-%d*, where %d is used to differentiate between different queues. The management thread opens a queue named */lisa-rstp-0*, and this queue will be enabled for the duration of the daemon's life. After creating the message queue, this thread loops forever, waiting for messages to be received on the queue. These messages follow a pre-defined pattern, implemented as a structure of the following format:

```
1 struct rstp_request {
2         int type;
3         pid_t pid;
4         int if_index;
5         char device_id[64];
6 };
```

The *type* field tells the management thread what the message represents, whether it is an enable/disable message, or a message that requests all interfaces that have RSTP enabled on them, etc. The *pid* field shows the process id of the process that sent the message. This is useful since some messages require giving an answer back to the process that sent them. Such processes create a queue with names that follow the previously mentioned pattern, */lisa-rstp-%d*, replacing %d with their process ids. Having the process id in the message allows the management thread to open the queue on which the CLI process listens for answers. Even requests that do not involve receiving an answer, such as enabling the RSTP protocol on a specific port, use this communication method as a way of acknowledging that the request has completed successfully. The *if_index* field is used to determine what interface the request refers to.

In conclusion, the steps taken by the CLI to communicate with the RSTP daemon are the following:

- open the queue that the daemon uses to accept requests
- for each CLI instance create a queue for the instance to receive answers on
- if the request modifies a global parameter then only access the shared memory

- if the request modifies a variable that is not stored in shared memory then send a message to the RSTP daemon

4.3 RSTP implementation

The core of the application is represented by the protocol's implementation. Until now, we have handled the input for our protocol, namely the BPDUs and the user commands. Looking at the problem from a layered perspective, we now have interfaces with the upper and lower layers.

The general problem of the implementation is the way each finite state machine will be executed and implemented. The standard specifies that each FSM will run independently from the others, but they do cooperate in the sense that data is shared between them. Ultimately, this data will be the one deciding the behaviour of the protocol. Naturally, we'll want to store this somewhere, so the *struct rstp_interface* has been defined. This structure will contain all the required information for a single port. Figure 4.2 shows the structure of a port, including the data it stores.

Each port has its own buffer where received BPDUs are stored. When the Port Receive state machine is ready to accept a new BPDU it checks this buffer for available data. It is the role of the receiver thread we previously mentioned to put BPDUs into the buffer as they arrive on the socket.

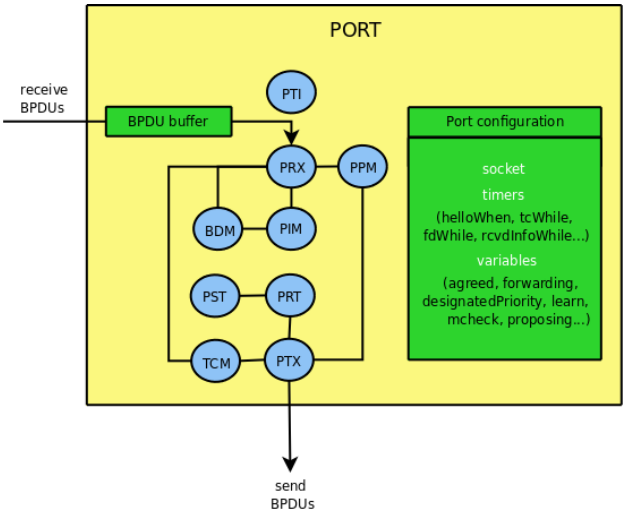

Figure 4.2: Port structure from an implementation point of view

The FSMs have been described earlier, the important thing to note here is that the port configuration is accessible from each state machine. No matter the model of implementation chosen for them, they will have access to this data. This is not shown explicitly in the figure because of space reasons.

The port's configuration contains the following:

- A socket representing the connection to the actual physical port. This socket will be used both to receive and send data.
- Timers controlling different aspects of the protocol's logic. There are a number of eight timers that each port must implement. Of these, we mention fdWhile and helloWhen. The fdwhile timer delays a port transition into an other state in order for other bridges to receive spanning tree information. It is this timer that makes STP so slow compared to RSTP, since STP relies on it to move a port into the forwarding state. RSTP fixes this through the proposal/agreement mechanism we explained eariler. The helloWhen timer ensures that at least one BPDU is transmitted by a Designated port in each HelloTime period.
- Variables required by the RSTP implementation. The 802.1d standard defines 45 variables that should be assigned to each port. Most of them are used internally by the implementation, to synchronize the different state machines. Since the goal of the protocol is to change the roles and states of the ports, we mention the selectedRole and reselect variables as being among the most important ones. The selectedRole variable is self-explanatory, while the reselect variable is set to *true* when an event has happened such that a port might have to change its state. If one or more of the ports have the reselect variable set, then the Port Role Selection state machine activates and checks that each port is in the correct state.

With the general implementation in place, having all data structures and buffers ready, the state machines are now ready for implementation.

4.3.1 Finite state machine implementation

The initial approach concerning the FSMs was that each one should be represented by a thread. Since the 802.1d standard specifically mentions that each thread should run independent from the other, the first idea that comes in mind is a thread. Each FSM can be viewed as a collection of variables and procedures that execute simultaneously with other FSMs' procedures. Having a separate process for each thread would be inappropriate since the threads share a lot information, so the process idea is completely out of the question. The library used for providing the thread functionality is pthreads[1].

Having chosen the thread solution, each port would now have 10 associated threads that start running whenever a port receives the *rstp enable* command through the management thread. All these threads share the port's configura-

[1]POSIX threads, or Pthreads, is the POSIX standard that defines an API for creating and manipulating threads

tion. More specifically, each thread has a pointer to the location in memory where the *struct rstp_ interface* associated with that port resides.

Because multiple threads now point to, and access, the same area, synchronization issues appear. Two or more threads might modify the same variable at the same time, leaving it in an inconsistent state. For that, variables that are shared between threads, are protected using mutexes. The matter is complicated though, by the fact that FSMs test a certain number of variables before jumping into the next state. If the variables have the correct values, then the FSM goes into the next state, else it blocks waiting for the respective variables to have the necessary values. This is the issue that complicated the implementation, and made it go through several iterations and revisions before reaching its current state.

In order for a FSM to jump to the next state, the transition has certain conditions to be fulfilled. The simplest solution is for the thread to do a busy-waiting loop until the conditions were satisfied. That meant that the thread repeatedly checked each variable to see if it had the right value. This is the worst possible solution since the processor is kept busy at all times and the system's load average could go as far as 0.9 because of it.

Dropping the busy-waiting solution, the next idea was to use semaphores. When a thread[1] had to jump to another state it simply used the *down* operation on the semaphores associated with the variables from the condition. Each variable had an associated semaphore, so if a thread waited for variable A to equal 1 and variable B to equal 1 then it would simply call *sem_post* on sem_A, and after that do the same thing for sem_B. Unfortunately, the solution works for states that only have one transition to another state. If a FSM could go from a state to one of multiple states, then it had to wait on multiple sets of conditions, corresponding to each transition to those states. The semaphore solution was no longer viable, because while waiting on a semaphore, another set of conditions might have become true. There was no way for the thread to know that, since it was blocked waiting on the semaphore. Using non-blocking *down* operations on the semaphores amounted the same busy-waiting solution that we previously skipped.

Giving up on the semaphores leads to the next solution : condition variables. These are synchronization primitives that allow threads to wait until a particular condition occurs. They support two operations : *signal* and *wait*. Every variable from the *struct rstp_ interface* is associated a condition variable so threads can be notified when the variable is changed. The downside of using condition variables can be seen in Figure 4.3.

Thread 1 modifies the x variable and then uses the *signal* operation on the x_-changed condition variable to notify Thread 2 that variable 1 has changed. The problem with this is that Thread 2 might be doing something else when Thread

[1]Since each FSM is represented by a thread, the term *thread* can be replaced with *FSM*

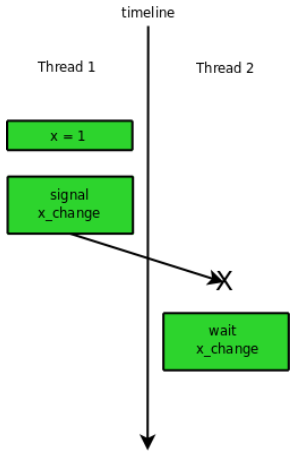

Figure 4.3: Lost signal using condition variables

1 sends the signal. That is, Thread 2 is not waiting at the condition variable when Thread 1 sends the signal. In that case, the signal is lost, since no thread is waiting at the condition variable. No matter what we do, there is still a possibility that Thread 2 is preempted right before waiting on the condition variable, and during that time, Thread 1 might send the signal. We have no way of knowing that a thread is waiting on a condition variable when another one is signaling it.

The conclusion up to this point is that semaphores could be a solution if we could monitor several of them without busy-waiting, and condition variables could be used if they somehow stored the signal even if there is no thread on the receiving end. What we need is a combination of both, something that saves the signal and supports monitoring multiple instances of its type. The answer to this has been introduced recently in the Linux kernel, in version 2.6.22 and support was provided in glibc since version 2.8 : event notifications using file descriptors, or short, eventfd.

Eventfd provides an event notification system through file descriptors, which is exactly what we wanted. The *signal* and *wait* calls have now been replaced with the usual *write*, and *read* calls. Multiple event monitoring is done with the help of the *select* routine. Let's give an example of how a thread, being in state A, might wait for conditions that could lead it to go in state B or C or D (Figure 4.4).

First, the thread has to make an initial check on the conditions to see if they are true or not:

```
1 if (!x) {
2         goto stateB;
3 } else {
4         if (y) goto stateA;
5         if (z) goto stateC;
6 }
```

Suppose none of the above is true when the thread checks the conditions so now

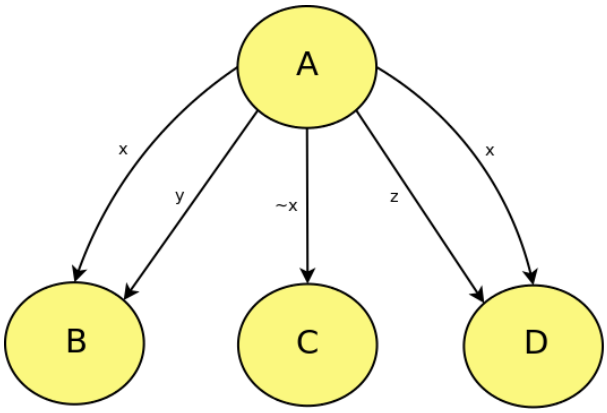

Figure 4.4: Example states and transitions

the thread has to wait for one of the three sets to become true. For that, it has to monitor variables x, y and z. To do that, it first creates a set with the associated condition variables, and then it goes in the select call.

```
1 FD_ZERO(&condition_set);
2 FD_SET(cond_x, &condition_set);
3 FD_SET(cond_y, &condition_set);
4 FD_SET(cond_z, &condition_set);
5
6 repeat_label:
7 select(MAX(cond_x, cond_y, cond_z) + 1, &condition_set, 0, 0, 0);
```

The thread is now monitoring all of the above three condition variables : cond_x, cond_y and cond_z, each corresponding to one of the three variables. What this implies is that whenever another thread assigns a value to x or y or z, it also has to signal the associated condition variable. When one of the three condition variables is signalled then the *select* unblocks and the waiting thread has to recheck all three condition sets again and clear the received signal by reading from the file descriptor:

```
1  if (FD_ISSET(cond_x, &cond_set))
2          read(cond_x, &u, sizeof(u));
3  if (FD_ISSET(cond_y, &cond_set))
4          read(cond_y, &u, sizeof(u));
5  if (FD_ISSET(cond_z, &cond_set))
6          read(cond_z, &u, sizeof(u));
7  if (!x) {
8          goto stateB;
9  } else {
10         if (y) goto stateA;
11         if (z) goto stateC;
12 } else {
13         goto repeat_label;
14 }
```

The thread checks the conditions again, and if no set is true, it goes back to the select call.

As it can be seen, the implemented code was for a simple case with only three variables and four states, but the RSTP finite state machines can have as many as 10 states and transitions with 8 tested variables. It became a tedious task to implement all states and transitions in this manner. A partial implementation of 4 of the FSMs allowed for some quick testing using the sub-modules previously defined. The Port Receive state machine got the BPDU and it signalled a few of the other FSMs to start working, using the synchronization mechanism described above. The proposal/agreement mechanism described earlier requires bridges to exchange BPDUs at a fast rate so that the process ends in at most 2 seconds, and propagates across the whole network. It is only normal the implementation must be fast enough to allow the exchange of packets at such a fast rate. Because of the many condition variables, the 10 threads per port, and all the other synchronization variables, the resulting implementation was too slow to allow rapid exchange of packets. A new approach was necessary.

The original idea of using threads appeared only because the standard mentioned the asynchronism characteristic of the finite state machines. On the other hand, it didn't mention that the FSMs should be scheduled in a random order, or any other order. That means we can apply our own order to the execution of the FSMs, one that is indeed in a pre-defined order, but it gives the illusion that they run in parallel. Instead of letting the operating system do the scheduling by using threads, we execute each FSM sequentially.

For this idea to be possible, we now have to monitor the exact state the finite state machine is in, since we have to know what routine to execute. The port keeps a bi-dimensional array of size number_of_FSMs x 2. Each FSM shall store in this array, the state it is currently in and if it had executed that state or not. Each state shall be executed only once, after which the FSM shall wait for a condition to become true, so it can jump in another state. It is the responsibility of each state to update the array.

Additionally, each FSM is assigned a one-dimensional array of routines, each routine representing one state. For example, the Port Receive state machine has two states, labeled *discard* and *received*. The array and the definitions look like this:

```
1 #define PRX_DISCARD 0
2 #define PRX_RECEIVE 1
3
4 void (*prx_state_table[2])(struct rstp_interface*) = {prx_discard,
    prx_receive};
```

First, the FSM checks to see if the current state has been executed or not by looking at the bi-dimensional array. If the state has not executed then it, does its computing. After that, the FSM checks all conditions to see if it can jump in

the next state. If it can't it keeps the current state and marks it as executed. If it can jump, then it changes the current state to the target state and marks it as not executed. The code looks like this:

```
1  if (!port->state[FSM_INDEX][EXECUTED]) {
2          // call routines
3          // update variables
4  }
5
6  if (cond1 && cond2) {
7          port->state[FSM_INDEX][STATE_FUNCTION] =
               TARGET_STATE_FUNCTION;
8          port->state[FSM_INDEX][EXECUTED] = NOT_EXECUTED;
9  } else {
10         port->state[FSM_INDEX][STATE_FUNCTION] = CURRENT_FUNCTION;
11 }
```

Finally, we have to make sure that all FSMs get to execute their states, so we have to iterate through all of them. This is done with a simple for construct that executes each FSM's current state, for each port:

```
1  for (;;) {
2  list_for_each_entry_safe(entry, tmp, &registered_interfaces, lh) {
3          prx_state_table[entry->state[PRX][FUNC]](entry);
4          ppm_state_table[entry->state[PPM][FUNC]](entry);
5          bdm_state_table[entry->state[BDM][FUNC]](entry);
6          ptx_state_table[entry->state[PTX][FUNC]](entry);
7          pim_state_table[entry->state[PIM][FUNC]](entry);
8          prt_state_table[entry->state[PRT][FUNC]](entry);
9          pst_state_table[entry->state[PST][FUNC]](entry);
10         tcm_state_table[entry->state[TCM][FUNC]](entry);
11         }
12
13 prs_state_table[ prs_func ]();
14 }
```

The Port Role Selection is executed only once since only one instance of it exists, as specified by the standard.

By doing the above for loop, we essentially got rid of all threads and more than that, there are no more synchronization issues. It is basically an user space deterministic scheduling scheme, where each FSM gets to execute each state in a pre-defined order. On the other hand, we are back at square one, since there is a chance that none of the FSMs pass in their next state. That is, each machine is checking their transitions, but none of them are valid, so they all remain in their current state. This leads to busy-waiting, because no actual work is being done, and processor cycles are wasted.

This problem is solved by checking the port's configuration to see exactly which variables trigger the transition from one state to another, and are not modified by any of the machines. These correspond to the timers and the user modified

configuration variables. These are the external inputs that can trigger a state change, when every state machine is continuously checking their transition conditions. For that, the above code is modified so the for loop ends when no state machine has transitioned in another state. After that, the thread running the above code blocks on an event descriptor, waiting for timers to expire or BPDUs to arrive, or for a user to modify some parameters. Once the event has been triggered, it goes back to executing the for loop.

There is one state machine that is not executing in the above code : the Port Timers state machine. Its job was to decrement the timers each second. Since we have no way of making the above loop execute each second, nor do we want it to do that, another method of implementation had to be found. An alarm mechanism was set up, that transmitted signals once every each second. These signals were caught by a routine that upon execution, decremented each timer.

What this means is that now, multiple threads can access the timer, so concurrent access is possible. It does not matter in what order the threads access the timers, and modifiy them, since they will be continuously checked by the for(;;) loop. What does matter is that the modifications are visible from all threads. This is a problem because most of the times the compiler will assume that variable values will not change asynchronously so it will make optimizations based on this. To override this behaviour the C type qualifier **`volatile`** is used in the definition of the timers, so the compiler does not make any suppositions concerning their values. It tells the compiler that the timers are subject to sudden changes for reasons which cannot be predicted from a study of the program itself, and forces every reference to them to be a genuine reference[13].

Chapter 5

Testing and results

Figure 5.1 shows the network that the protocol implementation has been tested on. It's main components are 3 Cisco switches, one SOHO router and a server running LiSA. The grayed out switches are used for accessing the server and the other switches through a web interface.

The SOHO router ensures that the network is completely isolated from the Internet. To provide access to the network, ports 22, 80 and 443 are forwarded. Port 22 is used by the SSH protocol to allow remote connections to the network, while ports 80 and 443 are used by the HTTP and HTTPS protocols to provide a web interface.

Having a real network to test the protocol on, proved to be a great advantage, because the implementation could be tested against an already functioning one.Thus, even if one flag was set incorrectly in a BPDU sent by the server running LiSA, this would have lead the other switches to signal this incorrect behaviour by sending BPDUs that didn't match the normal protocol conduct. For example, in the early stages of testing, the implementation erroneously set the *agree* flag in each sent BPDU. In the proposal/agreement mechanism, this means that the other switch had previously sent a BPDU with the proposal flag sent. Because that didn't happen, the mechanism failed, and the switch kept sending BPDUs that didn't acknowledge the ending of the proposal/agreement process. Considering that there were other errors in the implementation, in an environment where all machines were servers running LiSA, such an error might not have been caught. All servers would correctly choose the right roles and states for the ports, but they would have done so using an incorrectly implemented process. Once another switch that didn't run LiSA would have been connected to the network, the protocol would have failed because it did not have a correct implementation. That is why, it is preferable that the implementation of a layer 2 protocol that already is provided by other network components, is tested against these, so interoperability is ensured.

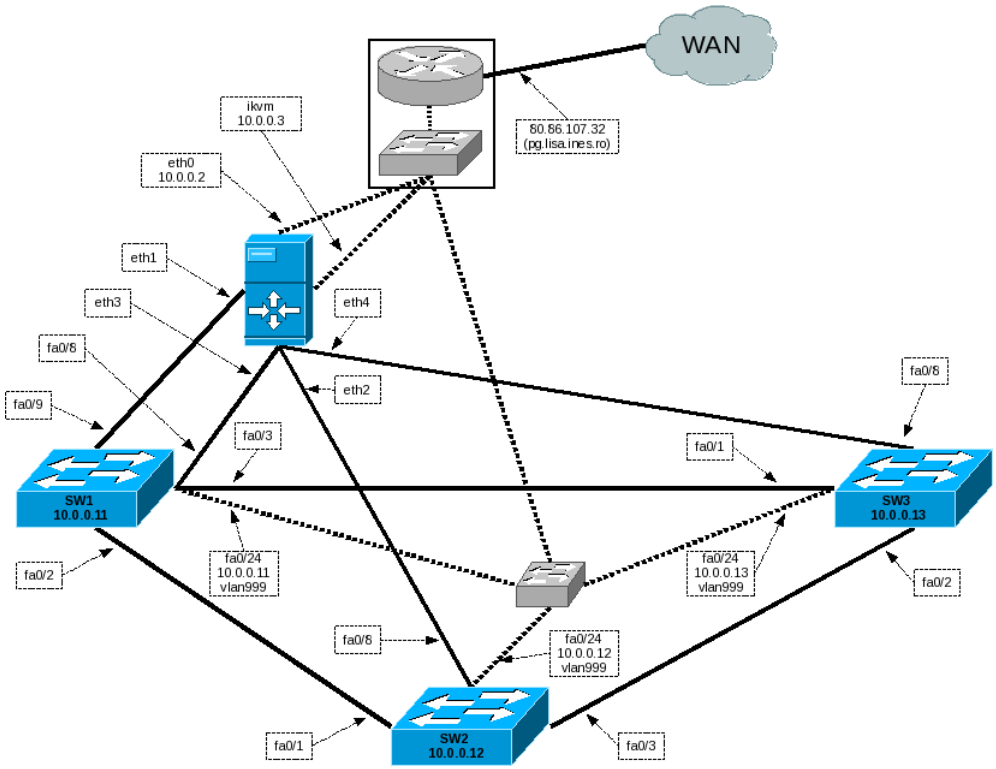

Figure 5.1: Network used for testing RSTP

The tests aimed to confirm the following:

- A server running LiSA can be inserted in a network where the Root Bridge had better priority than the server
- A server running LiSA can be inserted in a network where the Root Bridge had worse priority than the server
- Backwards compatibility with STP works
- Loops are eliminated according to the protocol

For the first test, the *eth1 - fa0/9* connection was activated as Bridge SW1 was already running the RSTP protocol. The SW1 Bridge had a priority value of 32768, while the server had a greater one of 32769. In this configuration, the SW1 Bridge is the Root Bridge and it remains so after the link is activated. The implementation on the server had to assign the eth1 port the Root Port role. After the proposal/agreement mechanism, the implementation managed to create a stable configuration, with SW1 as Root Bridge.

The second test is similar to the previous one, only this time the server running LiSA has a better priority so it should become the Root Bridge when the *eth1 - fa0/9* link is activated. This means that after the server announces that it is the Root Bridge (all switches do that when they are first connected), SW1

will actually confirm that after comparing their priorities. The implementation handled this case too, making the eth1 port a Designated Port, and getting the SW1 bridge to assign the fa0/9 port the Root Port role.

Making sure that backwards compatibility with the Spanning Tree Protocol works is very important since not all bridges offer support for RSTP. In networks where the topology does not change very often, STPs high convergence time is not an important issue. Bridges SW2 and SW3 from the above network, can be enabled to run STP. Thus, to test compatibility, the previous two tests were repeated with SW2 to see if the communication between the server and the switches was functional even when using TCN BPDUs and Configuration BPDUs. The implementation yielded good results, managing to create a stable configuration in a few seconds after the link was enabled. These extra seconds have appeared due to the timer-based root election process that the Spanning Tree Protocol defines, and in a larger network would have a significant impact on the protocol's re-convergence time.

Finally, the test that actually verifies the core functionality of the protocol checks if loops are properly handled. Given the above network, there are several ways to combine the four switches (the 3 Cisco switches and the server with LiSA) to create a loop. For simplicity, let's take the scenario where bridges SW1 and SW3 plus the server are connected to each other in a physical ring topology. That means, that links *eth1 - fa0/9* , *eth4 - fa0/8* and *fa0/1 - fa0/3* are all activated. There are two sub-scenarios to test here, each corresponding to whether the server is Root Bridge or not. In case it is a Root Bridge then all ports are Designated Ports and it is the job of the other two switches to block one of the ports. The other scenario, where the server is not a Root Bridge is more relevant since it tests if our implementation is able to block the ports. Each of these tests passes, the implementation managing to block one of the ports so there is no loop.

Chapter 6

Conclusions

The end result of the project was an RSTP implementation integrated in the LiSA project. The implementation provides backwards compatibility with the Spanning Tree Protocol and manages to create a loop-free topology in the network described earlier. Since it is now a part of the LiSA project, the code is open-source, so further refinements and fixes can be made by anyone. This provides an advantage since the implementation will be tested against different configurations and this will trigger any bugs and errors that our network wasn't able to catch.

LiSA provides an excellent framework for developing and implementing layer 2 protocols. By abstracting all the kernel implementation details from the user, and making frame handling as easy as reading and writing from a socket, it manages to simplify protocol implementation and allows the user to focus on the protocol's details instead of lower-level[1] issues. More than that, it provides a CLI similar to that offered by Cisco's switches, which makes it easier for network administrators to get accustomed to. The fact that LiSA is based on the Linux kernel and takes advantage of its networking stack, means that it can run on any platform supported by Linux without having to modify the source code. This is a great advantage since once a protocol is supported, its implementation can easily be ported on different architectures without changes in the code.

The biggest challenge of the project was to get the finite state machines to work. The solution to this problem was found after going through most of the synchronization primitives that the GNU/Linux operating system has to offer. Taking each of these, analyzing their strengths and weaknesses, lead to an implementation using event file descriptors that was slow because of the high number of threads. Because debugging multi-threaded programs is difficult and the source code can be hard to maintain, and because race conditions are hard to track, this implementation was scrapped in favour a single threaded solution that provides the functionality of multiple threads working together. All synchronization problems have disappeared because only one thread accesses the data at a single

[1]By lower-level we mean the lower level of the Linux kernel networking stack implementation

time, so testing the transitions' conditions is a simple matter of checking a few variables for equality.

The underlying problem of the implementation is the simulation of asynchronous finite state machines. These are usually implemented in hardware because the change of one line[1] can trigger the immediate jump in another state using fast combinational circuits. A hardware-based implementation of the RSTP protocol would be faster than anything software can produce since it relies heavily on state machines to define its logic.

Another idea to keep in mind is that RSTP really shows its strength in large networks, where STP can reach a re-convergence time between 15 and 30 seconds. The network used for testing the protocol, only had 4 switches, so the advantage of RSTP over STP might not be that obvious.

The current implementation of RSTP is still in its early stages. There are numerous scenarios and corner-case situations that arise in practice that are hard to test. Making sure that the protocol functions no matter what happens in the network is of top priority. This includes providing support for edge ports and allowing user-controlled port transitions.

A major improvement would be the implementation of MSTP. Originally defined in 802.1s and later merged into 802.1q-2003, MSTP is an extension to RSTP that allows separate spanning trees to be computed for each defined VLAN. It shares a lot of RSTPs functionality and since LiSA already offers VLAN support, this improvement would boost LiSA's functionality.

[1]For example, from *HIGH* to *LOW*

Bibliography

[1] Radia Perlman. An Algorithm for Distributed Computation of a Spanning Tree in an Extended LAN, 1985.

[2] http://en.wikipedia.org/wiki/Packet_switching. Packet Switching.

[3] http://www.livinginternet.com/i/iw_packet_inv.htm. Packet Switching History.

[4] http://en.wikipedia.org/wiki/Circuit_switching. Circuit Switching.

[5] http://en.wikipedia.org/wiki/LAN_switching. Lan Switching.

[6] http://www.cisco.com/en/US/tech/tk389/tk621/technologies_white_paper09186a0080094cfa.shtml. Understanding Rapid Spanning Tree Protocol (802.1w).

[7] http://en.wikipedia.org/wiki/Spanning_tree_protocol. Spanning Tree Protocol.

[8] Rich Seifert and Jim Edwards. *The All-New Switch Book*. Wiley Publishing Inc., second edition, 2008.

[9] Wald Wojdak. Rapid spanning tree protocol: A new solution from an old technology, 2008.

[10] Nicu Ioan Petru. LiSA - Sistem de comutare a pachetelor, 2005.

[11] Radu Rendec, Ioan Nicu, and Octavian Purdila. Linux Multilayer Switching with LiSA, 2006.

[12] Radu Rendec. Sistem de rutare intre VLAN-uri bazat pe LiSA, 2005.

[13] Mike Banahan, Declan Brady, and Mark Doran. The C Book, 1991.