Leopold-Franzens-Universität Innsbruck

Institute of Computer Science
Information Security

Visualizing network topologies with the help of the Spanning Tree Protocol

Bachelor Thesis

Alexander Schlögl

advised by
Dr. Pascal Schöttle

Innsbruck, September 20, 2016

Contents

Chapter 1

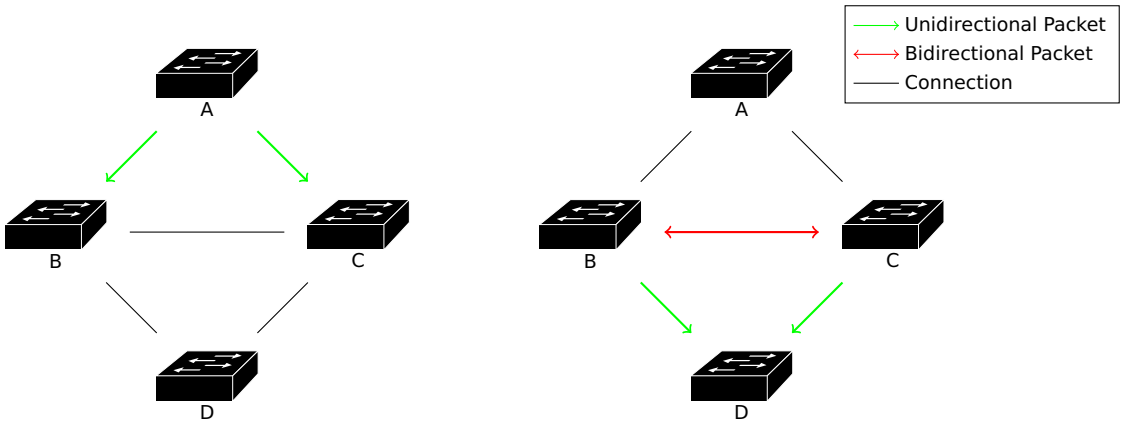
Introduction & Motivation

As networks grow larger and more complex, redundancy becomes desirable and necessary. Adding redundant links always introduces loops to the network. Switches and hubs respond to broadcast messages by sending them over every port (except for the one the packet was received on). This is done in order to ensure every node in the network receives the message. In conjunction with loops, this behaviour leads to broadcast messages being repeated indefinitely. With the right network setup, these messages can also multiply exponentially. Conditions like this are called broadcast storms[19]. Figure 1.1 shows how a broadcast storm forms. In Figure 1.1a you can see how Bridge *A* starts the broadcast. Bridges *B* and *C* then propagate this broadcast over every other port. This leads to the broadcast being sent in both directions between bridges *B* and *C*, as well as twice along to bridge *D*, as seen in Figure 1.1b. As bridges do not keep messages in memory after they have been sent, they do not recognize broadcast messages they received before. Thus, they simply propagate the messages in compliance with their policy, leading to the packets multiplying (Figure 1.1c). After the next iteration of propagations the same broadcast message will be sent in both directions over every bridge connection, shown in Figure 1.1d.

To prevent this the Spannning Tree Protocol (STP)[15] was created. It creates an overlay network, in the shape of a tree, to remove loops from the network. In this overlay, broadcast storms are not possible anymore. An example for broadcast propagation in an STP overlay network is shown in Figure 1.2. As shown in Figure 1.2, the overlay ignores the connection between bridges *C* and *D*. This leads to the broadcast being propagated to every node in just two steps, without any extra packets being generated.

For this thesis, we will only talk about switches. Hubs, while still existent, are hardly ever used and not STP capable. We will also refer to switches as bridges, in order to be compliant with STP nomenclature[15].

Networks with STP utilization are usually quite large. Additionally, changes and outages are hidden by STP and redundant links in the network. This can make it hard to keep track of the current network layout. Debugging STP configurations is also a difficult and time consuming task. It is possible to check STP configurations via the Simple Network Management Protocol (SNMP). However, this requires an administrator to connect to every single bridge and collect the STP information manually. While there are tools to automate this, they rely primarily on SNMP, which may not be supported by all bridges in the network.



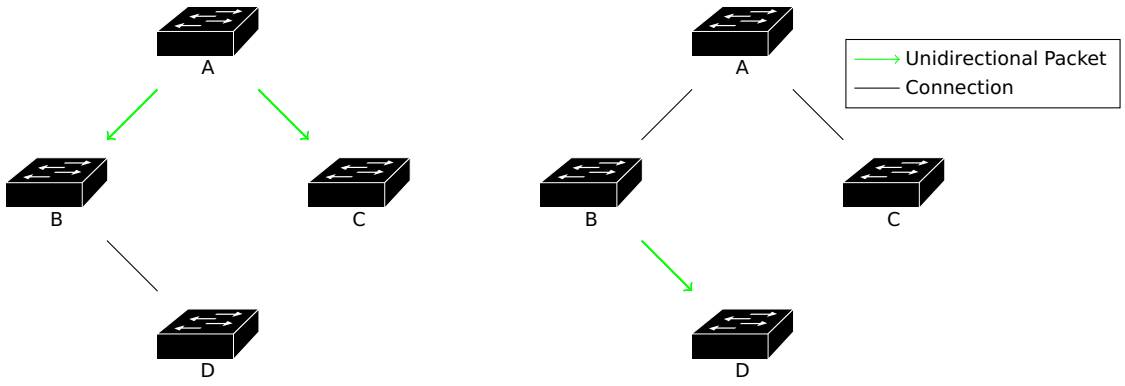
(a) Broadcast is sent

(b) Broadcast is propagated

(c) Broadcasts start multiplying

(d) Broadcast storm

Figure 1.1: An example of a broadcast storm



(a) Broadcast is sent

(b) Broadcast propagation done

Figure 1.2: Broadcast propagation in an STP network

The aim of this thesis is to create a versatile tool for STP visualization. We chose to name this tool *STPViz*. STPViz has the following requirements:

- **STP only**: STPViz should only rely on STP packets
- **Low traffic**: STPViz should create as low a traffic as possible
- **Passive**: STPViz should not provoke any changes in the network and only use packets obtained passively
- **Distributed**: STPViz is to be run on multiple nodes to send information to a central server where everything is pieced together
- **Low hardware load**: STPViz should run on simplistic hardware (e.g. a Raspberry Pi) or on a regular PC without slowing it down noticeably
- **No maintenance**: once set up, STPViz should not require any maintenance (except when updating to a new version)
- **Multiple trees**: if the clients send data from multiple different spanning trees (e.g. one tree per VLAN), these need to be kept seperate.

The rest of this thesis is structured as follows: Chapter 2 will cover the background knowledge needed to fully understand STPViz we developed. In chapter 3 we will discuss the implementation and our approach to gathering information from STP packets. Chapter 4 shows the details of our *software-switch* tool, which we had to develop due to problems we encountered with the STP implementations in free router OSs[14][5]. Chapter 5 explains how and to what extent we tested STPViz, as well as the results. Finally, Chapter 6 will summarize this thesis and give an outlook on possible future improvements to STPViz.

Chapter 2

Background

2.1 Existing alternatives

As of writing this thesis, only very few tools support monitoring of STP topologies. The ones we found are LoriotPro[11], LiveAction[10] and L2Discover[9]. Of these three network monitoring tools, only L2Discover has its source code openly available. The company SolarWinds has an open vote on their website on whether or not to include this feature in their monitoring software since 2014[3]. LoriotPro and L2Discover use the SNMP for their topology discovery, with STP only being used to discover duplicate and unused links. STPViz uses only STP, thus not needing the networking hardware being SNMP capable.

While commercial tools mostly use active methods for network discovery, several research projects have also attempted to develop purely passive or hybrid tools. One of the first of these tools was developed by Becker et al.[1] in 1995. In more recent years, Blue et al.[2] and Wongsuphasawat et al.[20] published papers on passive network visualization tools in 2008 and 2009. However, not much work has been done on purely passive STP visualization.

2.2 Spanning Tree Protocol (STP)

Port States

The most important part of STP is its introduction of port states. These port states ensure that the spanning tree does not contain loops. Ports can be in one of three states (see Figure 2.1):

- **Root Port**: the port leading to the root or "upwards" in the tree. Every non-root bridge has **exactly one** root port.
- **Dedicated Port**: dedicated ports are ports where packets are sent. These are the ports leading "downward" in the tree. The root has only dedicated ports.
- **Blocking Port**: no packets are sent via a blocking Port. This state is used to disable alternate paths to the root or "sideways" in the tree.

In the original STP paper[15], the term Local Area Network (LAN) referred to any connection to a bridge, as well as between bridges. Because the STP information must be propagated

to every LAN, there needs to be exactly one dedicated port in a LAN. In a LAN between two non-root bridges, the bridge with the smaller root path cost (see Figure 2.2) will make its connected port dedicated. Should both bridges have the same path cost to the root, the one with the smaller bridge identifier becomes the dedicated bridge (see Section 2.2: STP Packets). This can be seen in the connection between the bottom bridges in Figure 2.1. If no superior STP packets are received on a blocking or root port for a set amount of time (called the forward delay) it transitions to the dedicated state.

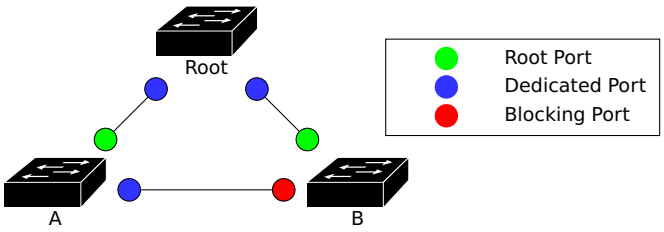


Figure 2.1: An example of STP port states

STP Packets

In this section we will explain the individual fields in an STP Bridge Protocol Data Unit (BPDU), as seen in Figure 2.2.

<table>
<tr><th colspan="32">0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31</th></tr>
<tr><td colspan="16">Protocol Identifier</td><td colspan="8">Protocol Version Id</td><td colspan="8">BPDU Type</td></tr>
<tr><td colspan="8">Flags</td><td colspan="24">Root Identifier</td></tr>
<tr><td colspan="32">Root Identifier</td></tr>
<tr><td colspan="8">Root Identifier</td><td colspan="24">Root Path Cost</td></tr>
<tr><td colspan="8">Root Path Cost</td><td colspan="24">Bridge Identifier</td></tr>
<tr><td colspan="32">Bridge Identifier</td></tr>
<tr><td colspan="8">Bridge Identifier</td><td colspan="16">Port Identifier</td><td colspan="8">Message Age</td></tr>
<tr><td colspan="8">Message Age</td><td colspan="16">Max Age</td><td colspan="8">Hello Time</td></tr>
<tr><td colspan="8">Hello Time</td><td colspan="16">Forward Delay</td></tr>
</table>

Figure 2.2: An STP BPDU

The fields we used for STPViz are:

- **Flags**: The flags byte is used for the *topology change (TC)* and *topology change acknowledgement (TCA)* flags.
- **Root/Bridge Identifier**: The Identifier consists of three parts and has the same layout for the root and regular bridges:

  - Priority (4 Bits): A value between 0 and 61440 configurable in increments of 4096
  - System ID Extension (12 Bits): Used for keeping the bridge ID unique if multiple VLANs are configured for a bridge
  - Bridge MAC (6 Byte): The MAC address of the bridge

  The conjunction of these three parts is used in comparisons as one large 8 byte number.

- **Root Path Cost**: The root path cost is the sum of all port costs (which can be configured in the bridge) along the current path. The root path cost in packets sent by the root is 0.
- **Message Age**: The message age is the number of bridges that have been passed (in addition to the root) along the current path.
- **Max Age**: If the message age surpasses this maximum, packages are not forwarded any more.
- **Hello Time**: The delay in seconds between sent *Hello* BPDUs. *Hello* BPDUs have the form shown in Figure 2.2.
- **Forward Delay**: After this time, if no superior packet was received, a root, or blocking port will transition to dedicated. Note that if a root port transitions to dedicated state that means the bridge now assumes it is the root.

A BPDU only contains information about the root and the bridge that sent the package. This means that if there is a bridge between these two, we will not know about its ID. We will however still know that it is there, because of the message age. During the buildup of the tree it is possible to obtain information on intermediate nodes. This possibility is explained more in-depth in the section on packet handling (Section 3.2).

Default Parameters

The parameters discussed default to the following values[4]:

- **Priority**: 32768
- **System Id Extension**: 0
- **Max Age**: 20
- **Forward Delay**: 15
- **Hello Time**: 2

Spanning Tree Algorithm

The original paper on STP does not describe an algorithm itself, it merely lists constraints that have to be fulfilled. Algorithm 1 shows the algorithm we use in our own *software-switch* testing tool. We use a reduced number of port states, ignoring the transition states, by utilizing the timestamps of received BPDUs. The algorithm's purpose is to keep loops out of the network, as well as keeping the tree topology consistent.

Data:
$received$ = received packet
$current$ = data of the current bridge

if $received.rootId < current.rootId$ **then**
    /* There is a new root in the network */
    $current.rootId = other.rootId$
    set receiving port as root-port
    set other ports to dedicated
end
if $received.rootPathCost < current.rootPathCost$ **then**
    /* This means that the path via the other bridge is shorter, so this should be our new root-path */
    set receiving port as root-port
    set other ports to dedicated
else if $received.rootPathCost == current.rootPathCost \wedge received.bridgeId < current.bridgeId$ **then**
    /* Both bridges are equidistant from the root, but the other bridge has a lower bridge Identifier and should be the dedicated bridge on the connection */
    set receiving port as blocking port
end
/* Check if the transitions described in the section on **STP Packets** are necessary */
doPacketTimeOutTransitions()

Algorithm 1: Spanning Tree Algorithm (STA)

Extensions

Since the inclusion of STP in the IEEE 802.1D[7] standard, multiple additions have been made to the STP. Rapid STP (RSTP) has been developed to decrease the time needed for tree convergence. Per VLAN STP (PVSTP) has been developed by Cisco and is a proprietary variant which allows for multiple spanning trees to exist, one for each VLAN. The Multiple STP (MSTP) is an extension of RSTP which, like PVSTP, allows for multiple spanning trees to exist in their own VLAN. In addition to that, it also allows for VLAN specific trees to be combined into one large spanning tree.

As an extension to STP the Shortest Path Bridging (SPB)[6] protocol was developed. It allows for multiple paths to be active at the same time, and uses global information to allow for the calculation of shortest paths. While it would be a lot easier to obtain network information using SPB, we decided to use to the simpler and still widely used STP, as we were unsure whether SPB had the same prevalence as STP.

2.3 Technologies used

PCAP

PCAP[18] is a shortening of packet capture. It used to be a part of the *tcpdump* tool before it was pulled into it's own library. We used the UNIX version *libpcap* for this thesis. *Libpcap* is a C library and can be directly linked with C and C++ without using a wrapper. It provides functions for opening live network devices and *.pcapng* files. After starting a live capture on a device, *libpcap* will call a callback function for every packet received on the opened interface. The prototype of the callback function looks as follows:

Listing 2.1: Pcap Callback Prototype

```
1 typedef void (*pcap_handler)(u_char *user, const struct pcap_pkthdr *h,
      const u_char *bytes);
```

Where the parameters are:

- **u_char *user** is used to pass user defined parameters.
- **pcap_pkthdr *h** contains useful information about the packet, like source and destination addresses, as well as size.
- **u_char *bytes** contains the actual (binary) data of the packet.

The **typedef void** (*pcap_handler) part of the prototype just means that the function returns a **void ***.

Usage of this function will be covered more in-depth in the chapter on STPVizitself (Chapter 3).

JSON

JSON stands for Java Script Object Notation and was used for two reasons:

1. It keeps the network communication independent from any programming languages used.
2. Utility libraries for JSON are easily available for most programming and scripting languages, saving us the work of inventing and implementing our own notation.

JSON has a simple notation for declaring objects and arrays, as well as primitive data types. An example is shown in Listing 2.2.

Listing 2.2: JSON Example

```
1 { "users":[
2     {
3         "firstName":"Ray",
4         "lastName":"Villalobos",
5         "joined": {
6             "month":"January",
```

```
            "day":12,
            "year":2012
        }
    },
    {
        "firstName":"John",
        "lastName":"Doe",
        "joined":{
            "month":"April",
            "day":28,
            "year":2010
        }
    }
]}
```

The JSON library used in this thesis is *jsoncpp*[13].

TikZ

We chose TikZ ist kein Zeichenprogramm (TikZ)[17] as the file format for our output. Using TikZ allows us to generate *.tex* files rather than images. These are easier to generate, as well as to modify after creation. It is also very easy to integrate in LaTeX papers. While TikZ is very powerful and therefore complex, the parts we use are fairly simple. Listing 2.3 shows the TikZ code for Figure 1.1b.

Listing 2.3: A TikZ Example

```
\begin{subfigure}[b]{0.4\textwidth}
    \begin{tikzpicture}
        \node (root) at (4,8) {\switch{0.8}{A}};
        \node (B) at (2,6) {\switch{0.8}{B}};
        \node (C) at (6,6) {\switch{0.8}{C}};
        \node (D) at (4,4) {\switch{0.8}{D}};

        \draw
        (root) edge (B)
        (root) edge (C);

        \draw[green, thick, ->]
        (B) edge (D)
        (C) edge (D);

        \draw[red, thick, <->]
        (B) edge (C);
    \end{tikzpicture}
    \caption{Broadcast is propagated}
\end{subfigure}
```

The \node keyword draws a node with the id given in parenthesis. It is centered around the coordinates given with the *at* keyword and has the text written in braces. The (0,0) coordinate is in the top left corner of an image. For our graphics we used a self defined \switch macro, but regular text or LaTeX commands work as well. Edges are drawn using the \draw command. This command can be used with raw coordinates or identifiers. Options for the edges can be passed in brackets. Note that *edge* can be substituded with - -. All TikZ commands are ended with a semicolon. TikZ graphics are automatically cropped on rendering.

Chapter 3

STP Tree Generator

We split the application in three parts:

- **Client**: collects STP information and sends it to the server. The client also handles piecing together the packets into paths.
- **Server**: saves the data from the clients and combines them into one tree.
- **Parser**: contacts the server to receive the tree and converts it into output format.

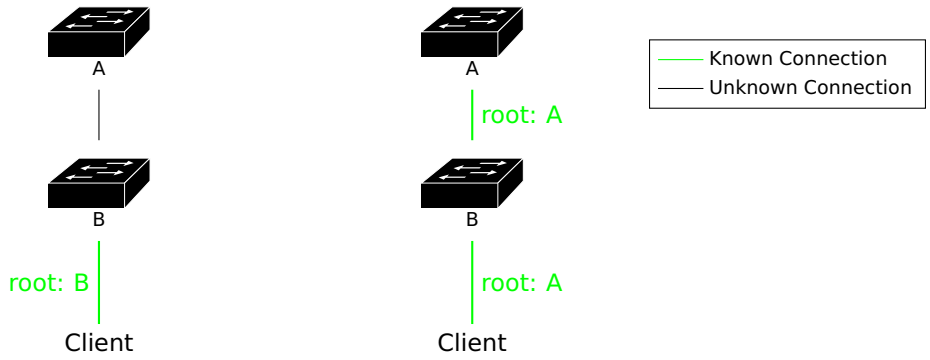
The intended form of usage is to have multiple clients in the network connecting to one server. We combine information received from multiple clients to reach a better understanding of the network.

STP uses only local data, which means that bridges have no knowledge of the network, except for their own port states. Unfortunately, this makes it hard to find connections between bridges. The only way to obtain this information is by capturing packets during the tree build up.

When the client witnesses the message age increasing, it assumes the new root to be prepended to the previous one. This assumption is uncertain, as this connection cannot be guaranteed, but the risk of making mistakes can be reduced. Details on this are discussed in the section on uncertain assumptions (Section 3.4)

Figure 3.1 shows a rudimentary example of information gained during tree build up. As bridge *B* has not yet received an STP packet from bridge *A* yet, it thinks itself root, so it sends STP packets with its ID as bridge and root identifier (Figure 3.1a). The client therefore only knows about bridge *B*. When bridge *B* receives an STP packet from *A*, it updates its local informtion and broadcasts it, as shown in Figure 3.1b. This leads to the client adding bridge *A* to its vector of known bridges and increments the saved message age for all other bridges. Both bridges are contained in an STP packet from *B* as root and bridge respectively, but the process stays the same for larger topologies.

We only assume network growth for the cases where the message age increases at the same time that the root changes. For other cases (message age decreasing or staying constant) we did not find ways to gain knowledge about the network. Because we cannot guarantee that the topology change does not affect the path we assume, STPViz clients currently clear their data unless the message age increases. With the limited time and networking hardware at our disposal, we were not able to find a way to reduce the amount of informtion



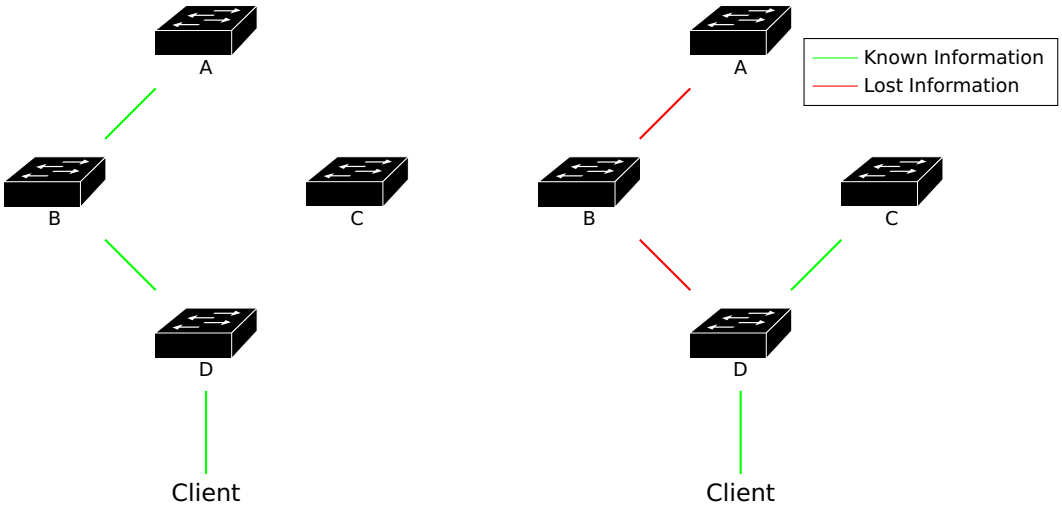
(a) B thinks it is root

(b) B updates its root information

Figure 3.1: Information gained on STP build up

that has to be discarded. We believe that by increasing the communication between clients, STPViz would be able to only discard information that is known to be false.

Figure 3.2 shows an example case where the client has to reset its data. Due to the message age decreasing when the root changes, STPViz discards all data and adds the two bridges contained in the STP packet again. This leaves it with knowledge of bridges *D* and *C*. A detailed explanation of how STPViz handles incoming STP packets can be found in the section on packet handling (Section 3.2).



(a) Some information known

(b) Previous information lost

Figure 3.2: Information lost during tree build up

3.1 Class Structure

The classes we created made saving the STP data easier and reduced the effort needed to generate the *JSON* and *TikZ* output. We created classes to represent MAC addresses, bridges, and complete spanning trees. All these classes have conversion functions from and to *JSON* format, as well as a function to generate *TikZ* output. Altogether the following classes were created:

- **Mac**: A container class for MAC addresses. It is used to store the address in cleaner format.
- **Bridge**: Stores a **MAC** object in conjunction with the priority and message age.
- **SpanningTree**: This class represents an entire tree. It has functions for creating the *TikZ* export as well as combining and manipulating subtrees. The **SpanningTree** class is a recursive data structure, storing the root as a *Bridge* object and its children as a vector of **SpanningTree** objects. As it is the most important and also the most complex class, the header file is shown in detail in Listing 3.1.
- **Sniffer**: Does the actual packet sniffing.
- **Client**: Handles the client side communication.
- **Server**: Handles the server side communication, as well as combining the trees and removing incorrect information.

Listing 3.1: The SpanningTree header file

```
1  class SpanningTree{
2      private:
3          Bridge root;
4          std::vector<SpanningTree> children;
5          std::pair<std::string, int> toTikzHelper(double lowerX, double
               upperX, int y, int yStep, int oldMessageAge, int index) const;
6
7      public:
8          SpanningTree();
9          SpanningTree(const Bridge& r);
10         SpanningTree(const Bridge& r, std::vector<SpanningTree> c);
11         SpanningTree(const SpanningTree& other);
12         ~SpanningTree();
13         void addChild(const SpanningTree& child);
14         int addSubTree(const SpanningTree& other);
15         int containsRoot(const SpanningTree& tree) const;
16         Json::Value toJson() const;
17         std::string toTikz(double lowerX, double upperX, int Y, int yStep,
                int oldMessageAge, int index) const;
18         int maxWidth() const;
19         int widthAtLevel(int levelsRemaining) const;
```

```
20          int size() const;
21          Bridge getRoot() const;
22
23          static SpanningTree fromJson(const Json::Value buildFrom);
24
25          friend SpanningTree operator+(const SpanningTree& lhs, const
                SpanningTree& rhs);
26          friend int operator==(const SpanningTree& lhs, const SpanningTree&
                 rhs);
27          friend int operator!=(const SpanningTree& lhs, const SpanningTree&
                 rhs);
28          SpanningTree operator=(const SpanningTree& other);
29  };
```

The STP data is saved in the **Sniffer** as a vector. This is easier than storing them in a fixed size array, and already combining them into a **SpanningTree** object would keep the server from performing the steps described in Section 3.4.

3.2 Packet Handling

Packets are handled by the sniffer class. While the function mostly skips unused fields in the STP packet, here we will take a close look at the more important parts. In order to check if the packet is actually an STP packet, we use the Ethernet destination address. Listing 3.2 shows how that is done. The *bytes* variable is provided by the required *pcap* callback prototype (see Section 2.3: PCAP). This way is computationally more expensive than saving the destination in binary format and comparing memory. It is however also more readable and a lot easier to change should the need arise. The address we compare the destination to is the broadcast target for STP.

Listing 3.2: Filtering for STP packets

```
1 struct ethhdr *ethh = (struct ethhdr*) bytes;
2
3 char buffer[17];
4 sprintf(buffer,"%.2X:%.2X:%.2X:%.2X:%.2X:%.2X",
5         ethh->h_dest[0], ethh->h_dest[1],
6         ethh->h_dest[2], ethh->h_dest[3],
7         ethh->h_dest[4], ethh->h_dest[5]);
8 if(std::string(buffer, 17) != "01:80:C2:00:00:00")
9     return;
```

Pcap provides us with a pointer to the binary packet data. We can go through the packet by incrementing this pointer by set amounts (after skipping to the STP data). This repeats for the bridge identifier, as well as other fields. Listing 3.3 shows the beginning of the payload handling.

Listing 3.3: Going Through the Payload

```
//handle payload
const u_char *payload = bytes+sizeof(struct ethhdr);
int psize = header->len - sizeof(struct ethhdr);
//payload starts at logical link control level
//llc has 3 bytes
payload+=3;
psize-=3;
//payload is now at the start of the stp payload
//protocol identifier stp (2 bytes)
payload+=2;
psize-=2;

//one byte for version and bpdu type
payload+=2;
psize-=2;

//tc flag
bool tc = *(payload++);
psize--;

//root identifier
//bridge priority is the next 2 bytes
unsigned short rPriority;
//actual priority is 4 bits
rPriority = *((short*)payload);
payload+=2;
psize-=2;
//next 6 bytes is the root bridge id
Mac rootMac(payload);
payload+=6;
psize-=6;
```

After the data is extracted from the packet, it is constructed into our custom classes. We then check if the two bridges (root and first hop) were previously known, as seen in Listing 3.4. Concerning the root, knowing whether or not it was previously known is enough. For the first hop, we also require knowledge about the old message age.

Listing 3.4: Checking for Previously Known Information

```
//generate Bridge objects
Bridge root(rootMac, rPriority, 0);
Bridge firstHop(bridgeMac, bPriority, messageAge);

//check if the two nodes are contained
int rootContained = 0, oldHopMa = -1;
for(Bridge b : bridges){
    if(b == root)
        rootContained = 1;
```

```
10      if(b == firstHop)
11          oldHopMa = b.getMessageAge();
12  }
```

Listing 3.5 shows how we update the bridge information in the sniffer. The *clearAndAdd* function is just a shorthand to clear the bridge vector before adding the two bridges to it.

Listing 3.5: Bridge Data Update

```
1  //handle network changes
2  if(rootContained){
3      if(oldHopMa >= 0){
4          //both contained
5          if(oldHopMa != firstHop.getMessageAge()){
6              //upstream changes
7              clearAndAdd(firstHop, root);
8          }else{
9              //everything is as it was
10             //if something changed upstream a tc flag will be sent
11             //->reset
12             if(tc && !hadTC)
13                 clearAndAdd(firstHop, root);
14         }
15     }else{
16         //first hop not contained
17         //this means that the node was plugged in on a different bridge
18         //(most likely)
19         //we don't know how long we were disconnected, so reregister the
               client
20         if(!noConnect)
21             client->regServer();
22         clearAndAdd(firstHop, root);
23     }
24 }else{
25     //root not contained
26     if(oldHopMa >= 0){
27         //if the first hop was contained this means there were some
               changes upstream
28         //the root moving away means we assume network growth
29         if(oldHopMa < firstHop.getMessageAge()){
30             std::cout << "root moved away\n";
31             int maDiff = oldHopMa - firstHop.getMessageAge();
32             //this means we have to increase every message age and add the
                   new root
33             for(Bridge &b : bridges)
34                 b.setMessageAge(b.getMessageAge() + maDiff);
35             bridges.push_back(root);
36         }else{
```

```
37              clearAndAdd(firstHop, root);
38          }
39      }else{
40          //entirely new setup
41          clearAndAdd(firstHop, root);
42      }
43  }
```

3.3 Communication

Our distributed architecture is hierarchical, as we have a distinct *client-server* separation. We also used a *push* style communication, meaning the clients will "push" their messages to the server, who simply waits for incoming connections. The alternative would be to have the server poll the registered clients for their data. This would have doubled the communication effort needed and made the handling for disconnected clients more complicated. For these reasons that we chose not to do this. As discussed in the background section (Section 2.3) wer are using JSON for the client server communication. To inform all involved components about the purpose of a JSON message, we use a *messagetype* field. The possibilities for *messagetypes* are as follows:

- **Register**: before they can send data to the server, the clients need to register to receive an identifier. These unique identifiers are used to keep the data from different clients separated.
- **Push**: clients send *push* messages when they transmit their data to the server. These messages contain lists of bridges. They are transmitted the same way that they are stored in the sniffer, without modifications.
- **Report**: the parser sends a message to the server containing this *messagetype* and nothing else. The server then combines the client data and transmits it to the parser.

Bridge data is transmitted from the client to the server in standard JSON array notation. When the data is transmitted from the server to the parser, it is transmitted as a full tree. We added conversion functions to and from JSON to all our custom data classes, to keep this transmission simple. Listing 3.6 shows an example transmission from client to server, and Listing 3.7 shows a transmission from server to parser. The JSON tree shown in Listing 3.7 the tree resulting from our test runs. It shows how STPViz can use changing packets to gather information about bridge connections.

Listing 3.6: Client-Server Transmission

```
1 {
2   "bridges" :
3   [
4     {
5       "mac" :
6       {
```

```
        "address" : "F4:F2:6D:7D:BF:BD"
      },
      "messageAge" : 0,
      "priority" : 144
    }
  ],
  "messagetype" : "push"
}
```

Listing 3.7: Server-Parser Transmission

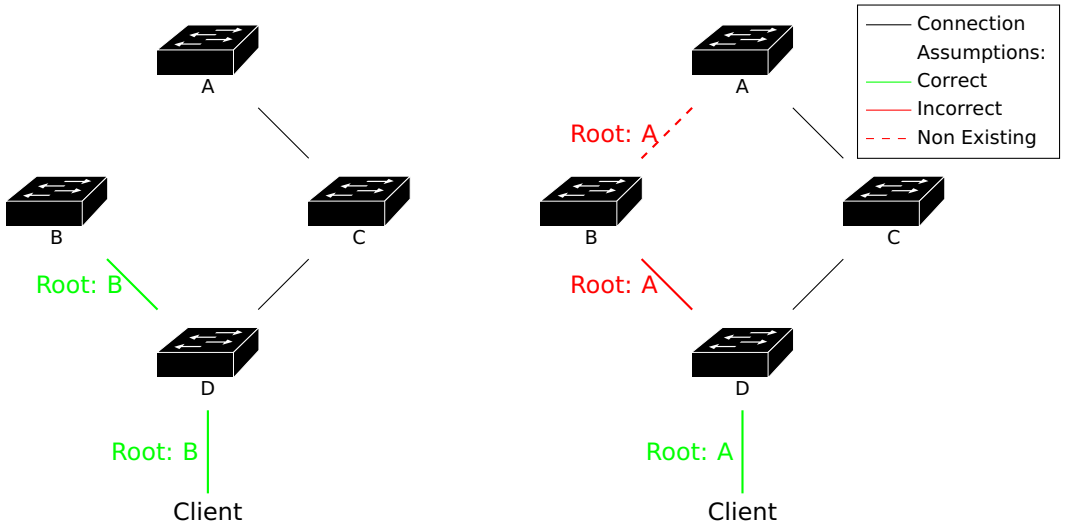
```
{
    "children" :
    [
        {
            "children" :
            [
                {
                    "children" : null,
                    "root" :
                    {
                        "mac" :
                        {
                            "address" : "F4:F2:6D:7D:BF:BD"
                        },
                        "messageAge" : 2,
                        "priority" : 240
                    }
                }
            ],
            "root" :
            {
                "mac" :
                {
                    "address" : "00:09:6B:93:28:2E"
                },
                "messageAge" : 1,
                "priority" : 384
            }
        },
        {
            "children" : null,
            "root" :
            {
                "mac" :
                {
```

```
                    "address" : "50:7B:9D:0C:9F:CD"
                },
                "messageAge" : 1,
                "priority" : 384
            }
        }
    ],
    "root" :
    {
        "mac" :
        {
            "address" : "BC:AE:C5:EB:7D:B6"
        },
        "messageAge" : 0,
        "priority" : 128
    }
},
{
    "children" : null,
    "root" :
    {
        "mac" :
        {
            "address" : "BC:AE:C5:EB:7D:B6"
        },
        "messageAge" : 0,
        "priority" : 128
    }
}
```


3.4 Uncertain Assumptions

As previously stated, the assumptions we make about bridge connections are not necessarily true. They can be wrong, for certain cases, one of which is shown in Figure 3.3. The client knows about the path to (at the time) root *B* in Figure 3.3a. When the global root information gets updated, the message age will increase causing the client to prepend *A* to its known path, even though there is no connection between *A* and *B*. This leads the client to thinking the path looks like in Figure 3.3b.

We did not find an alternate way to gather information about connections in the network, so we tried to reduce errors created by our method. To this end we combine information gathered by multiple clients to identify and remove false information. If a client makes a false assumption about a bridge, the message age for that bridge will be lower than its actual messge age. An example can be seen in Figure 3.4. It uses the same topology as Figure 3.3 The assumed (nonexistent) connection between bridges *A* and *B* is shown in red. Figure 3.4a shows the message ages assumed by STPViz. With the assumed connection the message age



(a) Correct information

(b) False assumption

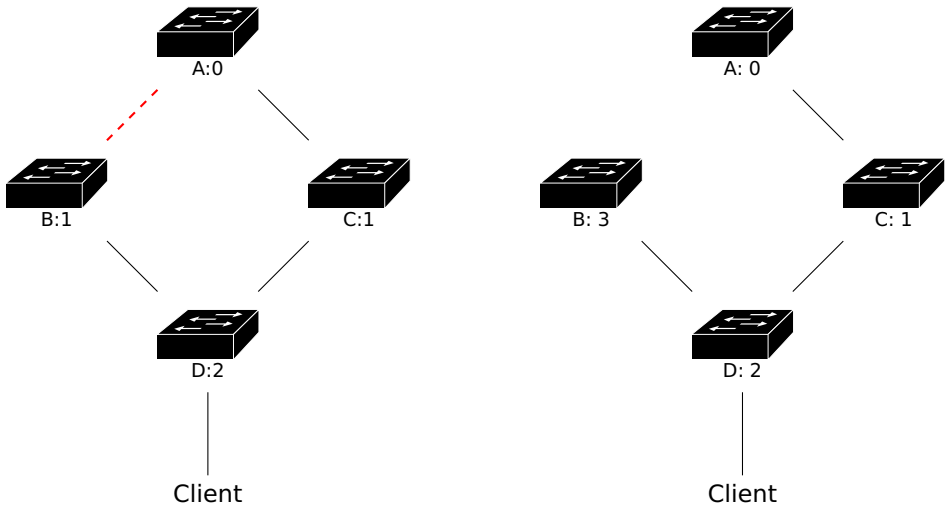
Figure 3.3: An example case of false assumptions about network structure

of bridge *B* would be 1. However, as Figure 3.4b shows, the actual message age is 3. If a client connected to bridge *B* sends its actual message age to the server (or any message age larger than 1), the server will remove *B* from the path sent by the client.

By checking all bridges for inconsistencies we can remove any false assumptions from the data we use for building the visualization. The code for doing this is shown in Listing 3.8.

Listing 3.8: Removing Incorrect Bridge Data

```
1  //remove invalid entries before creating SpanningTree
2  //check every received vector against every other
3  for(auto mapIt1 = clientData.begin(); mapIt1 != clientData.end(); ++mapIt1
       ){
4      std::vector<Bridge> vec1 = mapIt1->second;
5      for(auto mapIt2 = clientData.begin(); mapIt2 != clientData.end(); ++
           mapIt2){
6          if(mapIt1 == mapIt2)
7              continue;
8          std::vector<Bridge> vec2 = mapIt2->second;
9          for(auto vecIt1 = vec1.begin(); vecIt1 != vec1.end(); ++vecIt1){
10             for(auto vecIt2 = vec2.begin(); vecIt2 != vec2.end(); ++vecIt2
                   ){
11                 //check every bridge against every other
12                 //if one has a smaller message age, it is incorrect and
                       needs to be removed
13                 //this might overlook some errors if the number of clients
                        is insufficient
14                 if(vecIt1->getMessageAge() < vecIt2->getMessageAge()){
```



(a) Assumed message ages

(b) Actual message ages

Figure 3.4: Assumed and actual message ages

```
15                      //remove bridge
16                      vecIt1=vec1.erase(vecIt1);
17                      //decrement vecIt1 to have it at the correct position
                            in the next loop iteration
18                      --vecIt1;
19                      //continue with outer loop (vec1 iteration)
20                      break;
21                  }else if(vecIt2->getMessageAge() < vecIt1->getMessageAge()
                        ){
22                      vecIt2=vec2.erase(vecIt2);
23                      --vecIt2;
24                      //no break needed -> next step is inner loop (vec2
                            iteration)
25                  }
26              }
27          }
28      }
29  }
```

The bridges are removed from the vector they each are in. This way any connection information that is not proven wrong is retained, which means that if previous assumptions were correct, they will not be lost.

3.5 Combining the data

After false information is removed, the bridge data is combined into one SpanningTree object. Creating large SpanningTree objects is done in multiple steps. First, we combine the bridge vectors to individual trees. These trees each represent a path from a client to the root. After the individual trees are obtained, we combine the ones with the same root. The code for this can be seen in Listing 3.9.

Listing 3.9: Combining the Bridge Data

```
1  std::vector<SpanningTree> indivTrees = std::vector<SpanningTree>();
2  for(auto mapIt = clientData.begin(); mapIt != clientData.end(); ++mapIt){
3      auto vec = mapIt->second;
4      std::sort(vec.begin(), vec.end(), [](Bridge a, Bridge b) {return a.
           getMessageAge() < b.getMessageAge();});
5      indivTrees.push_back(treeHelper(vec.begin(), vec.end()));
6  }
7
8  //now combine these trees
9  std::vector<SpanningTree> ret;
10 for(auto indivIt = indivTrees.begin(); indivIt != indivTrees.end(); ++
       indivIt){
11     bool contained = false;
12     for(auto treeIt = ret.begin(); treeIt != ret.end(); ++treeIt){
13         if(treeIt->containsRoot(*indivIt) && *treeIt != *indivIt){
14             *treeIt = *treeIt + *indivIt;
15             contained = true;
16             break;
17         }
18     }
19     if(!contained)
20         ret.push_back(*indivIt);
21 }
```

Combining the individual trees into one is done using the operator we overloaded. This operator executes a recursive tree combine, which is shown in Figure 3.10.

Listing 3.10: Recursive Tree Combination Algorithm

```
1  SpanningTree operator+(const SpanningTree& lhs, const SpanningTree& rhs){
2      if(lhs == rhs){
3          return SpanningTree(lhs);
4      }
5
6      SpanningTree ret = SpanningTree(lhs.root);
7
8      //add all similar children
9      for(SpanningTree leftChild : lhs.children){
```

```
10          for(SpanningTree rightChild : rhs.children){
11              //if they have the same root, add the combination
12              if(leftChild.root == rightChild.root){
13                  //should we make an incorrect assumption, set the message
                        age to the higher one.
14                  //this should not be needed, as this should be filtered in
                         the server.
15                  SpanningTree newTree = leftChild + rightChild;
16                  newTree.root.setMessageAge(std::max(leftChild.root.
                        getMessageAge(), rightChild.root.getMessageAge()));
17                  ret.addChild(newTree);
18                  break;
19              }
20
21              //if rightChild is contained in left (additions "downstream"),
                     add it there
22              if(leftChild.containsRoot(rightChild)){
23                  SpanningTree combined = SpanningTree(leftChild);
24                  leftChild.addSubTree(rightChild);
25                  ret.addChild(combined);
26              }
27
28              //vice versa
29              if(rightChild.containsRoot(leftChild)){
30                  SpanningTree combined = SpanningTree(rightChild);
31                  leftChild.addSubTree(leftChild);
32                  ret.addChild(combined);
33              }
34          }
35      }
36
37      //add all remaining children of lhs
38      for(SpanningTree leftChild : lhs.children){
39          int contained = 0;
40          for(SpanningTree added : ret.children){
41              if(added.root == leftChild.root){
42                  contained = 1;
43                  break;
44              }
45          }
46          if(!contained){
47              ret.addChild(leftChild);
48          }
49      }
50      //add all remaining children of rhs
51      for(SpanningTree rightChild : rhs.children){
52          int contained = 0;
```

```
53          for(SpanningTree added : ret.children){
54              if(added.root == rightChild.root){
55                  contained = 1;
56                  break;
57              }
58          }
59          if(!contained){
60              ret.addChild(rightChild);
61          }
62      }
63
64      return ret;
65  }
```

3.6 Visualization

STPViz generates *.tex* files containing *TikZ* graphs. Nodes in these graphs contain the bridge data. Edges show the (assumed) connections in the topology. Node information is displayed in the format *priority, system id extension - MAC address, message age*. For cases where STPViz has no knowledge of the path to the root, empty nodes are used. These can be seen as gaps in the final output. Figure 3.5 shows an example.

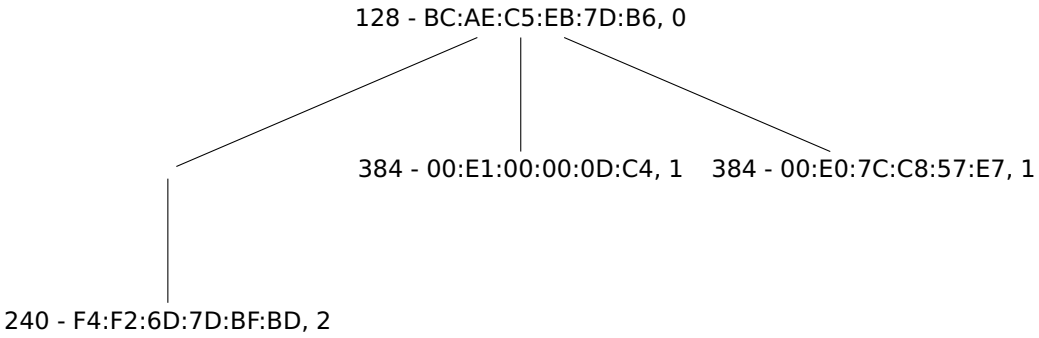


Figure 3.5: An example for STPViz output

The output is generated using a recursive function in the **SpanningTree** class. Our code for this is rather confusing when first read, so we decided to split it into multiple listings, allowing for clearer explanations. No code is skipped between the listings, so combining Listings 3.12-3.15 would yield the original function. To make it easier to follow we also did not modify any indentations. Viewing the code together with the raw LaTeX output makes it much clearer, so Listing 3.11 shows the output for Figure 3.5

Listing 3.11: *.tex* code for Figure 3.5

```
1 \begin{tikzpicture}
2 \node (0) at (7.000000,20) {128 - BC:AE:C5:EB:7D:B6, 0};
```

```
\node (1) at (2.333333,18) {};
\node (2) at (2.333333,16) {240 - F4:F2:6D:7D:BF:BD, 2};
\draw (1) -- (2);
\node (3) at (7.000000,18) {384 - 00:E0:7C:C8:57:E7, 1};
\node (4) at (11.666667,18) {384 - 00:E1:00:00:0D:C4, 1};
\draw
(0) -- (1)
(0) -- (3)
(0) -- (4);
\end{tikzpicture}
```

For an explanation on TikZ keywords please refer to the section on TikZ (Section 2.3).

Listing 3.12: TikZ conversion function, part 1

```
std::string SpanningTree::toTikz(double lowerX, double upperX, int y, int
    yStep, int oldMessageAge, int index) const {
    return toTikzHelper(lowerX, upperX, y, yStep, oldMessageAge, index).
        first;
}

std::pair<std::string, int> SpanningTree::toTikzHelper(double lowerX,
    double upperX, int y, int yStep, int oldMessageAge, int index) const {
    //this helper is needed because the previously used child index
        calculation was faulty if the algorithm was adding empty nodes
    std::string retString;
    int retInt = index;

    double dist = upperX - lowerX;
    double x = lowerX + dist/2;
```

The *lowerX*, *upperX*, *y* and *yStep* parameters are used to calculate the position for the current root, as well as the subtrees. *OldMessageAge* is used to discern whether or not STPViz needs to draw empty nodes. As we use incrementing IDs to draw connections between the nodes (see Listing 3.11), we need to *index* to pass the current index. This function is executed recursively until the current **SpanningTree** has no children, so the first step is always to calculate the position of the current root (see Listing 3.1). The previous child index calculation will be discussed in the explanation for Listing 3.15.

Listing 3.13: TikZ conversion function, part 2

```
    if(root.getMessageAge() > oldMessageAge+1){
        //if there is an unknown node between the previous node and this
            one create an empty node
        retString += "\\node (" + std::to_string(index) + ") at (" + std::
            to_string(x) + "," + std::to_string(y) + ") {};\n";
        std::pair<std::string, int> subRet = toTikzHelper(lowerX, upperX,
            y-yStep, yStep, oldMessageAge+1, index+1);
```

```
5           retString += subRet.first;
6           retInt = subRet.second;
7           retString += "\\draw (" + std::to_string(index) + ") -- (" + std::
                to_string(index+1) + ");\n";
```

If there is an unknown bridge between the current root and the previous one, we draw an empty node to visualize this. After the empty node is drawn, the same function will be called with increased y coordinate, because the current root has not been handled yet. We also know that the ID will be the current index plus 1. With these informations we can write the command for drawing the edge beforehand. Note that we include newlines to keep the output readable. As the returned string is a C++ **std::string** we use the **\n** escape sequence.

Listing 3.14: Tikz conversion function, part 3

```
1       }else{
2           //draw this node
3           retString += "\\node (" + std::to_string(index) + ") at (" + std::
                to_string(x) + "," + std::to_string(y) + ") {" + root.toTikz()
                 +"};\n";
4
5           //draw children
6           //we need the return values of the children later, for the index
                calculation
7           double xPerWidth = dist / maxWidth();
8           std::vector<std::pair<std::string, int>> childRets;
9           for(unsigned int i=0; i<children.size(); i++){
10              upperX = lowerX + xPerWidth * children[i].maxWidth();
11              if(i==0)
12                  childRets.push_back(children[i].toTikzHelper(lowerX,
                        upperX, y-yStep, yStep, root.getMessageAge(), index+1)
                        );
13              else
14                  childRets.push_back(children[i].toTikzHelper(lowerX,
                        upperX, y-yStep, yStep, root.getMessageAge(),
                        childRets[i-1].second+1));
15
16              lowerX = upperX;
17          }
18
19          //add the strings in an additional loop, to have nicer formatting
                of the output
20          for(auto ret : childRets)
21              retString += ret.first;
```

Here we draw the node for the current root. To calculate the amounts of horizontal space the subtrees can use, we use their respective maximum width. This width is calculated using the functions shown in Listing 3.16. For subtree spacing we simply use the relation between

the width of the full tree and the width of all children. In Listing 3.14 we see the reason why we have to return a pair of **string** and **int**. As it would be rather expensive to check for the number of empty nodes that need to be drawn beforehand (this would double the required computational effort), we return the last used index as well. This eliminates ID conflicts. To keep drawing the nodes and edges in blocks, we use multiple loops for drawing them. This leads to edges being drawn in "layers", as edges from a node to all its children are drawn at once. Only "deep" topologies (with a spanning tree of depth > 3) are affected.

Listing 3.15: Tikz conversion function, part 4

```
        //draw edges to children
        if(childRets.size() >0){
            retString += "\\draw ";
            for(unsigned int i=0; i<childRets.size(); i++){
                if(i==0)
                    retString += "\n(" + std::to_string(index) + ") -- ("
                        + std::to_string(index+1) + ")";
                else
                    retString += "\n(" + std::to_string(index) + ") -- ("
                        + std::to_string(childRets[i-1].second+1) + ")";

            }
            retString += ";\n";
            //set return index to largest children index
            retInt = childRets[childRets.size()-1].second;
        }
    }

    return std::pair<std::string, int>(retString, retInt);
}
```

Listing 3.15 shows how we draw edges between nodes and their children. We use the same method of accessing the indices as in Listing 3.14. Before returning we set the the returned *int* half of the pair to the largest index used by a recursive call.

Listing 3.16: Calculating the maximum width of a tree

```
int SpanningTree::maxWidth() const{
    int max = 1;
    for(int level = 0, currentWidth; (currentWidth = widthAtLevel(level))
        > 0; level++)
        max = max > currentWidth ? max : currentWidth;

    return max;
}

int SpanningTree::widthAtLevel(int levelsRemaining) const{
    if(levelsRemaining <= 0)
```

```
11          return 1;
12      if(levelsRemaining == 1)
13          return children.size();
14
15      int ret = 0;
16      for(SpanningTree s : children)
17          ret += s.widthAtLevel(levelsRemaining-1);
18      return ret;
19  }
```

3.7 Installation & Usage

STPViz was written in C++ and requires a compiler capable of C++11. Usage of the included Makefile is strongly advised. We included the version of *jsoncpp* we used, so it does not need to be downloaded. The client requires an installation of the *pcap* development library to compile, and the regular library to run. Available build targets are **client**, **server**, **parser** for single components and **all** for all components. An **install** target for global installation is not provided.

Client Launch Parameters

-if *inputFile* is used to specify the name of an input file which is written in *.pcapng* format. If specified packets will not be captured live, but instead taken from the input file. For easier usage with this parameter, launching the server with a higher timeout duration is recommended. The server will remove data from this client, as it is not programmed to resend data taken from input files.

-of *outputFile* is used to specify a different output file. Output files are used for logging data. Logging directly to **STDOUT** is currently not possible.

–no-connect / -nc will stop the client from connecting to a server. This was used for debugging the client, but remains in the current version.

-h *hostname* will tell the client which hostname to connect to. IPv4 addresses in dot notation, as well as hostnames (including URLs) are accepted. For more information on accepted formats please refer to the manual page for **gethostbyname**.

-p *port* specifies the port on which to connect.

-dn *deviceName* tells the client the device name of the interface to use. The names can be obtained using commands such as **ifconfig** or **ip addr show**.

Server Launch Parameters

-p *port* specifies the port to listen on.

-of *outputFile* specifies the filename for the output file, which is used for logging incoming data, as well as the current state.

-np disables the creation of a *.pid* file containing the process id. In cases the server is meant to be launched automatically in the background this file provides an easy means of accessing the process id.

-t *timeout* tells the server how large the timeout time should be before removing a client's data.

Parser Launch Parameters

-p *port* specifies the port to connect on.

-h *hostname* specifies the hostname to connect to.

-pw *pictureWidth* specifies the width of the resulting TikZ picture. Note that if a width of 20cm is specified but only 10 are used, the picture will be snipped automatically

-ph *pictureHeigth* specifies the height of the resulting TikZ picture. The same snipping occurs as for the width.

-s *yStep* defines the vertical distance between layers.

Default Parameters

Client parameters default to the following values:

- **-if**: defaults to nothing, live capture is used
- **-of**: client.log in the current working directory
- **-nc**: not set
- **-h**: *localhost*
- **-p**: 80
- **-dn**: first named device

Server parameters default as follows:

- **-p**: 80
- **-of**: server.log in the current working directory
- **-np**: not set
- **-t**: 10

Parser parameters default to the following values:

- **-p**: 80
- **-h**: *localhost*
- **-s**: 2
- **-ph**: 20
- **-pw**: 14

Configuration Files

The binaries all create configuration files with their current configuration if none exist. On subsequent launches they will take their configuration from these files. Command line parameters take precedence over configuration files, which in turn take precedence over default parameters.

Chapter 4

Software Switch Testing Utility

The original plan for testing STPViz was to use cheap routers running *OpenWrt*[14] with STP enabled. This did not work for multiple reasons.

- *OpenWrt* routers handle the internal switch as one interface, which stops it from disabling single switch ports.
- Through testing we found that the packets sent by the *OpenWrt* implementation are not recognized by our other devices. The other devices were a *TP-Link TL-SG2008 Gigabit Smart Switch* and an *ASUS RT-N56U Wireless Router*. These two devices would just forward the *OpenWrt* STP packets while still sending their own.
- We also found that *OpenWrt* just forwards received STP packets, without appending its own data. This alone would make it unusable for our purposes.

After making these discoveries, we tried *dd-wrt*[5] as an alternative. Unfortunately, it shares the same behaviour with *OpenWrt* and we did not know whether or not we could make either of these operating systems work for this thesis.

In order to be able to stay within the time limit while still testing STPViz we decided to implement a bridge capable of STP. It uses *pcap* to handle incoming packets and react to STP packets. Our *software-switch* can be run on any number of interfaces, however, it has not been tested on more than two. The source code for the *software-switch* can be found in its git[16]. It was written in C, in contrast to the *stp-tree-generator* which was written in C++.

4.1 Saving Switching Data

Listing 4.1 shows the variables and arrays we use for saving the switching data.

Listing 4.1: The Containers for Switching Data

```
1 pthread_mutex_t ifaceMutex;
2 int n;
3 char **names;
4 unsigned char **neighbours;
5 unsigned char **interfaces;
```

```
6 unsigned char *bridgeId;
7 unsigned char ***macTable;
8 int *macIndices;
```

The *names*, *neighbours* and *interfaces* arrays are arrays of "strings". They store the local interface names (as returned by the *ifconfig* command), the connected MAC addresses and the local MAC addresses respectively. As basic C does not have strings, these are arrays of **char** arrays. In C, arrays are saved as pointers to the first element, with the other elements following sequentially after that. This makes *names*, *neighbours* and *interfaces* pointers to pointers. Because the *software-switch* can be run on multiple interfaces we need to keep track of the number of interfaces it is running on, which we do in *n*. To know which interface to forward a packet on, switches keep a so-called MAC table. We keep a list of MAC addresses for every interface, making *macTable* an array of arrays of **char** arrays. This is due to us saving MAC adrresses as **char** arrays as well.

The *ifaceMutex* variable (mutex stands for mutual exclusion) is used to synchronize our different threads. We use two threads per interface. One to handle and forward incoming packets, as well as update our STP port states. The other one sends the STP packets every $helloTime$ seconds. Sharing memory between *pthreads* (the C thread implementation we used) does not require any setup, as all heap memory is shared[12]. We need to synchronize these accesses to keep the different threads from corrupting data due to concurrent writes.

4.2 Handling STP Packets and Port States

The code that extracts data from the STP packets is identical to STPViz, due to the packets having the same format. Handling of this data is a lot different however. The code for this can be found in Listing 4.2. Parameters prefixed with *r* or *b* are gained from the incoming packet and stand for root and bridge (first hop) respectively. Values prefixed with *root* are for the root information currently saved in the bridge. *CurrentIndex* is the index of the interface the packet was received on.

Listing 4.2: Updating Port States

```
1 void updatePortStates(int currentIndex, unsigned char rPriority, unsigned
      char rExtension, unsigned char *rMac, unsigned int pathCost, unsigned
      char age, unsigned char bPriority, unsigned char bExtension){
2     pthread_mutex_lock(&ifaceMutex);
3
4     //check for a root change
5     if(compareBridges(rPriority, rExtension, rMac, rootPriority,
          rootExtension, root) < 0 ||
6             (compareBridges(rPriority, rExtension, rMac, rootPriority,
                  rootExtension, root) == 0 && pathCost + portCost <
                  rootPathCost)){
7         memcpy(root, rMac, 6);
8         rootPriority = rPriority;
9         rootExtension = rExtension;
10        rootPathCost = pathCost + portCost;
```

```
11          messageAge = age+1;
12
13          for(int i=0; i<n; i++)
14              if(states[i] == ROOT)
15                  states[i] = DEDICATED;
16
17          states[currentIndex] = ROOT;
18          sendTCN(currentIndex);
19      }
20
21      //check if we would be the correct root
22      if(compareBridges(priority, extension, bridgeId, rootPriority,
            rootExtension, root) < 0){
23          memcpy(root, bridgeId, 6);
24          rootPriority = priority;
25          rootExtension = extension;
26          rootPathCost = 0;
27
28          for(int i=0; i<n; i++)
29              states[i] = DEDICATED;
30      }
31
32      //if a port is in the BLOCKING state but shouldn't be, change it
33      if(states[currentIndex] == BLOCKING){
34          //if our root is the correct one, set the port to dedicated
35          if(compareBridges(rootPriority, rootExtension, root, rPriority,
                rExtension, rMac) < 0)
36              states[currentIndex] = DEDICATED;
37
38          //if we have the same root, but have should be preferred, change
                to DEDICATED
39          if(compareBridges(rootPriority, rootExtension, root, rPriority,
                rExtension, rMac) == 0 &&
40              (rootPathCost < pathCost || (rootPathCost == pathCost &&
                    compareBridges(priority, extension, bridgeId, bPriority,
                    bExtension, neighbours[currentIndex]) < 0)))
41                  states[currentIndex] = DEDICATED;
42      }
43
44      //if a port is DEDICATED but shouldn't be, change it
45      //only possibility should be same root different path cost
46      if(states[currentIndex] == DEDICATED){
47          //only change if the neighbour has the same root (smaller root is
                handled by root change, larger root is ignored -> stay
                DEDICATED)
48          //note that here we have a rootPathCost == pathCost (without
                adding the portcost). This is because here we are interested
```

```
            in whether or not they have the same distance to the root.
49          if(compareBridges(rPriority, rExtension, rMac, rootPriority,
                rootExtension, root) == 0)
50              //even then only change to BLOCKING if they have a shorter
                    path or should be preferred
51              if(rootPathCost > pathCost || (rootPathCost == pathCost &&
                    compareBridges(priority, extension, bridgeId, bPriority,
                    bExtension, neighbours[currentIndex]) > 0))
52                  states[currentIndex] = BLOCKING;
53      }
54
55      pthread_mutex_unlock(&ifaceMutex);
56  }
```

4.3 Handling Non-STP Packets

Non-STP packets are forwarded based on port states and knowledge of the target MAC address. We did not implement any special handling for broadcast messages. None of the packets that the bridge receives should ever come from a broadcast address, which means they should never be contained in the MAC table. As packets to unknown MAC addresses are broadcast by default, we do not need special behaviour.

Listing 4.3: Handling Non-STP Packets

```
1  //handle non-stp packets here
2  //skip packets originating from this interface
3  if(memcmp(ethh->h_source, interfaces[currentIndex], 6) == 0){
4      return;
5  }
6
7  //skip packets sent by another interface listener
8  for(int i=0; i<numPackages; i++)
9      if(sentPackages[i] != 0 && memcmp(sentPackages[i], bytes, header->len)
           == 0)
10         return;
11
12 memcpy(sentPackages[packageIndex], bytes, header->len);
13 packageIndex = (packageIndex+1)%numPackages;
14
15 //add src mac to mac table
16 int found = 0;
17 for(int i=0; i<macTableSize; i++){
18     if(macTable[currentIndex][i] == NULL)
19         continue;
20     if(memcmp(macTable[currentIndex][i], ethh->h_source, 6) == 0){
21         found = 1;
```

```
22          break;
23      }
24  }
25  if(!found){
26      memcpy(macTable[currentIndex][macIndices[currentIndex]], ethh->
            h_source, 6);
27      macIndices[currentIndex] = (macIndices[currentIndex] + 1)%macTableSize
            ;
28  }
29
30  //check if the receiving interface is the intended target
31  if(memcmp(ethh->h_dest, interfaces[currentIndex], 6) == 0)
32      return;
33
34  //find right interface to send it on
35  int targetIndex = -1;
36  for(int i=0; i<n && targetIndex < 0; i++){
37      if(i==currentIndex)
38          continue;
39      for(int j=0; j<macTableSize; j++){
40          if(memcmp(macTable[i][j], ethh->h_dest, 6) == 0){
41              targetIndex = i;
42              break;
43          }
44      }
45  }
46
47  if(targetIndex < 0 || ethh->h_dest[0] & 1){
48      //if the packet was received on a blocking interface it is travelling
            sideways
49      //and needs to be stopped
50      if(states[currentIndex] == BLOCKING)
51          return;
52
53      //we don't know where to send it
54      //this sends the packet via all dedicated ports and the root if it is
            going up (coming from a dedicated port)
55      //and all dedicated ports if it is going down (coming from the root
            port)
56      for(int i=0; i<n; i++)
57          if(i!=currentIndex && states[i] != BLOCKING)
58              write(socks[i], bytes, header->len);
59  }else{
60      write(socks[targetIndex], bytes, header->len);
61  }
```

4.4 Installation and usage

This utility tool was written in C99[8]. While only one build target exists, we included a Makefile for convenience. The *pcap* development library is required to compile, and the regular *pcap* library is needed on launch. Following parameters can be set on launch:

- **-p** sets the STP priority (default: 0x80).
- **-e** sets the system id extension (default: 0).
- **-ms** sets the number of MAC addresses to store per interface (default: 30).

After these optional parameters the names of the interfaces to bridge are expected. The parameters used are printed to **stdout** for debugging purposes. Information on STP packets sent, including the respective port states is printed to **stdout** as well.

Chapter 5

Testing

5.1 Setup

In order to ensure STPViz fulfills the requirements stated in Chapter 1 we performed several tests. To be able to check the results for correctness, we kept the testing network small and simple. The layout of the test setup can be seen in Figure 5.1b. While a very basic network, it is sufficient for the tests described below, which will guarantee the required capabilities of the STPViz. The root in figure 5.1b was made root by manually assigning it a higher priority than the other bridges (note that *higher* in this case means *smaller*). Bridges *A*, *B* and *C* had their priorities left to the default value. Nodes *A*, *B* and *C* were running *Wireshark* and STPViz. Additionally using *Wireshark* allowed us to check the actual packages involved in the testing process to monitor progress and note situations not handled correctly by STPViz. The server for STPViz was run on Node *C*. Test results were checked for their correctness by checking the report provided by STPViz against the layout the network actually has. This layout was deduced using the debug output from our running *software-switch* instances and the configuration of the hardware switches.

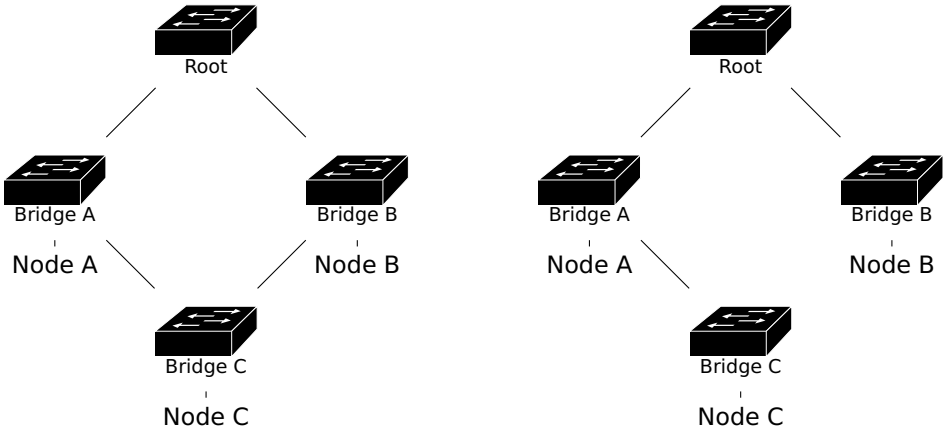
5.2 Tests

"Plug and Play" Test

This test is meant to ensure the general bridge discovery ability of STPViz. It is designed to simulate the connection of nodes to an established network. Using a simple setup like shown in Figure 5.1b allows us to collect information about the whole network. If we were using less nodes than we had non-root bridges in the network, this would not be possible.

Performing the test

1. The layout shown in Figure 5.1b was established and all devices were started.
2. No STPViz instance was started yet, identification during the tree establishment is covered by Section 5.2: Tree Establishment Test.



(a) Test setup

(b) Expected STP overlay

Figure 5.1: The expected physical and logical network topologies

3. We waited for the bridges to establish a stable spanning tree, checking the progress by observing the STP packets for the TC flag.
4. After the tree had stabilized we started the server on node *A*, as well as STPViz on all the nodes.
5. We waited for the STPViz instances to send their data to the server, which takes one $helloTime$ plus the latency between the nodes and the server.
6. When all nodes had sent their data to the server we created a report to check it against the expected result.

Expected Result

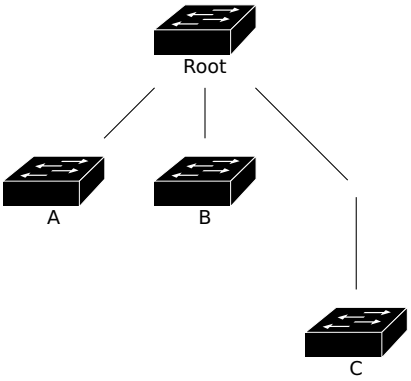
With the limited size of the test network all bridges can and must be identified correctly for this test to be counted as successful. If the bridges in the network behave as they should, STPViz should not be able to gather information about connections between the bridges. Figure 5.2a shows the output we expect.

STPViz Output

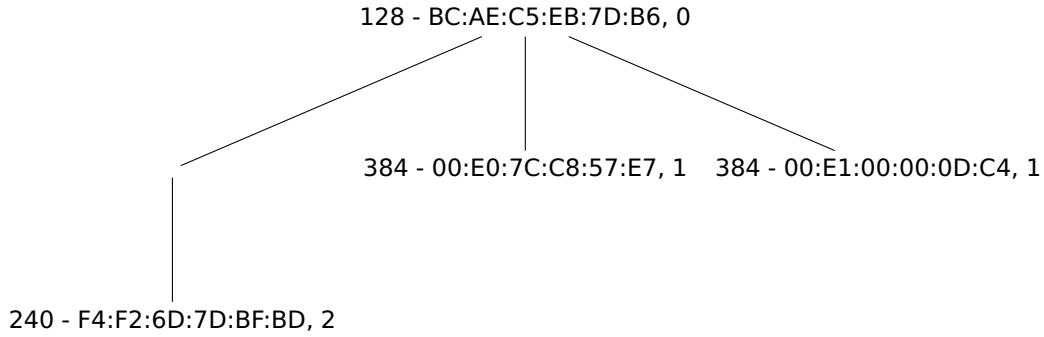
All nodes are correctly identified. As the tree is already established when STPViz is started on all nodes, no information about connections between the bridges can be gathered. Figure 5.2 shows a comparison between expected output ant STPViz output.

Tree Establishment Test

To test whether STPViz can handle bridges being added to the network during runtime, we start STPViz on all nodes before the establishment of the spanning tree, and check for correct identification afterwards.



(a) Expected output



(b) Visualization output

Figure 5.2: A comparison between expected and STPViz output for the "Plug and Play" test

Performing the test

1. The layout shown in Figure 5.1b was established and all devices were started.
2. Before enabling STP on all the bridges we started the server and STPViz instances.
3. After enabling STP on the bridges we waited for the tree to be established, again checking the progress via the TC flag.
4. Finally we checked the result for correctness.

Expected Result

Again all the bridges can and must be correctly identified. Additionally, this time the identification of bridge connections is possible, and the topology should therefore be fully identified.

STPViz Output

If bridge *D* is connected to bridges *B* and *C* before they are connected to bridge *A*, the tree can build up slowly and the clients will know about the connections. The resulting visualization

can be seen in Figure 5.3

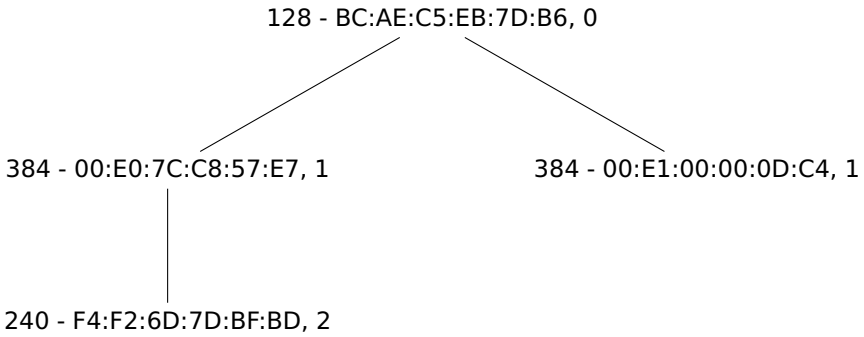


Figure 5.3: The test setup with fully recognized connections

Bridge Removal Test

STPViz must also be able to react to bridges or clients dropping from a network. This test makes sure that capability is given.

Performing the test

1. We started all the nodes STPViz instances and waited for the spanning tree to be established and correctly identified (see the sections on the Usage Test 5.2 and the Tree Establishment Test 5.2).
2. After the tree was constructed we unplugged Node *A* and waited for the tree to stabilize.
3. Finally we checked the output of the report for correct identification of the smaller tree.

Expected Result

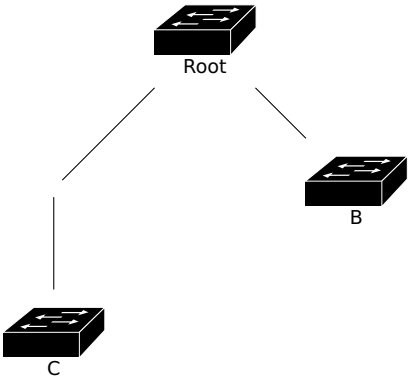
Bridge *A* must correctly be removed from the topology visualization. The expected topology is shown in Figure 5.4a.

Actual Result

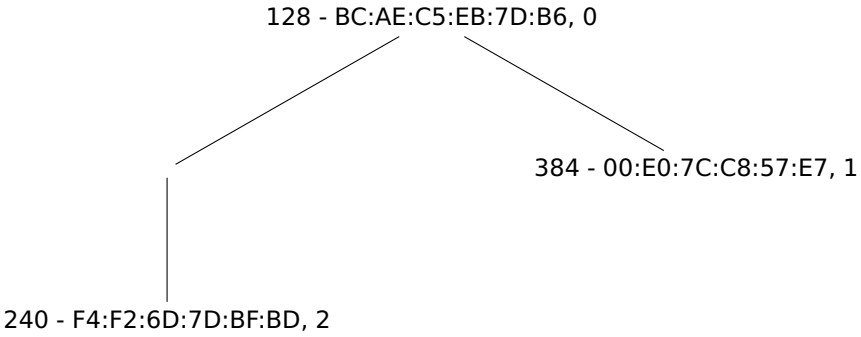
As visible in Figure 5.4, the bridge is correctly removed.

Slow Dynamic Change Test

Changes in the network topology must not be a problem for STPViz. By successfully testing for additions and removals from the network (Section 5.2: "Plug and Play" test and Section 5.2: Removal Test) one could assume that changes can be handled as well, but caution demands that we test specifically for changes in the topology. Due to the rather complex nature of this test, Figure 5.5 shows the physical network topology after every step.



(a) The expected output

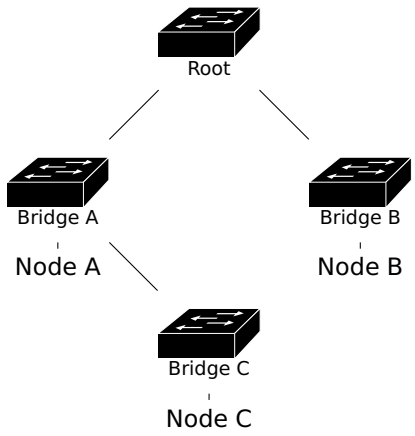


(b) The STPViz output

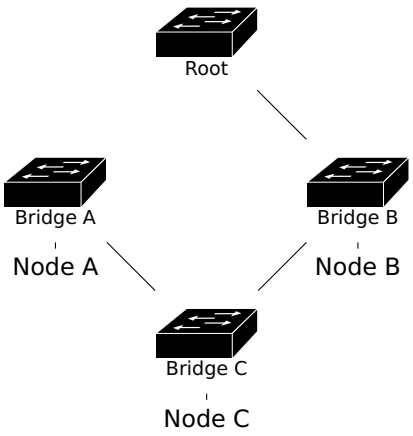
Figure 5.4: Expected output compared to STPViz output

Performing the test

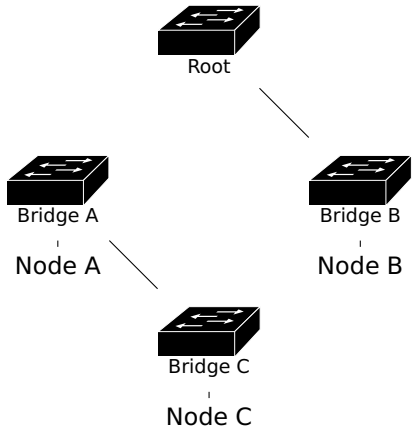
1. We set up all the nodes and waited for a stable tree to be generated by the bridges like before.
2. We disconnected bridge *A* from the root, resulting in a topolgy like in Figure 5.5b.
3. The connection between bridges *B* and *C* was also severed, leaving bridges *A* and *C* in a disconnected subtree, as shown in Figure 5.5c.
4. We waited for the spanning tree to stabilize after the removal.
5. Bridges *A* and *C* were connected to bridge *B* yielding the physical topology shown in Figure 5.5d.
6. After the tree had stabilized we checked the report output for correctness.



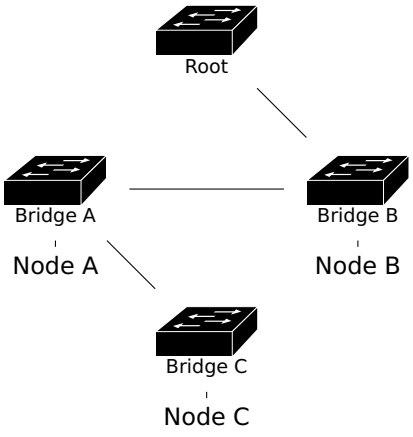
(a) The topology before the test



(b) The topology after step 2



(c) The topology after subtree removal



(d) The topology after the test

Figure 5.5: The physical topology during the Slow Dynamic Test

Expected Result

Because we wait for the small subtree to stabilize, establishing bridge *A* as root, the subtree should be correctly identified.

Actual Result

The visualization acquired before executing the test is the same as in Figure 5.3, as we used that test as preparation. For an unknown reason, the message ages for the subtree that we disconnect *00:E0:7C:C8:57:E7* and *F4:F2:6D:7D:BF:BD* does not get updated and increased. However, this is not an error in STPViz, but one in the *software-switch* tool. Unfortunately, it has not undergone much testing, and was developed only to the point where it is usable in testing STPViz. As we neither wanted to lose too much time on its development, nor draw the focus of this thesis away from its actual purpose, we decided to keep the effort expended into it minimal. Additional testing, fixing bugs like this and increasing general stability should be done in a future revision of the *software-switch*. The visualization we did obtain from STPVizcan be seen in Figure 5.6.

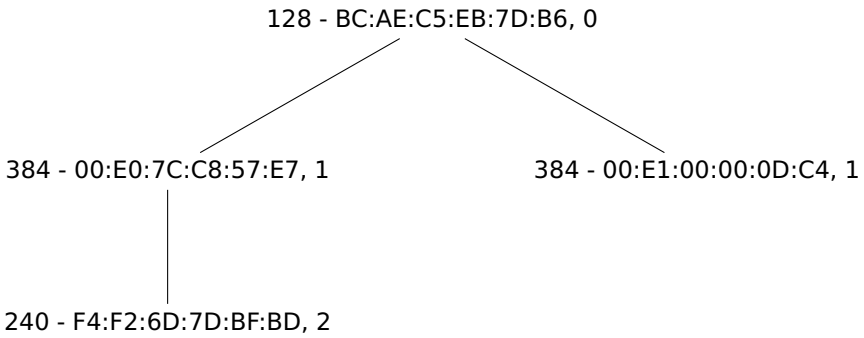


Figure 5.6: The obtained (incorrect) visualization for the slow dynamic change test

Fast Dynamic Change Test

Robustness to changes in the topology are a must for STPViz. It would however also be nice if STPViz were robust enough to handle topology changes without a spanning tree stabilization in between. The logical topology is the same as for the Slow Dynamic Change Test 5.2

Performing the test

1. We set up all the nodes and waited for a stable spanning tree.
2. After we removed node *A* from the network we immediately plugged nodes *A* and *C* into B.
3. We waited $2 * helloTime$ to make sure the changes were propagated to the server before plugging node *C* into the root.
4. Finally we waited for the tree to stabilize and checked the output for correctness.

Expected Result

The outcome of this test depends on the time passing between disconnecting and reconnecting the subtree. Bridge configurations also play a role in the outcome due to affecting the configuration. All bridges should be correctly identified after the tree has stabilized again. However, the identification of the connections is quite volatile, depending on the inner workings of the STP implemenetiations and configurations.

Actual Results

Due to the Slow Dynamic Test failing, and this test being highly dependent on STP implementations we decided not to run this test. With the hardware we had at our disposal, and especially with the low stability of our own STP implementation, we thought it not to be beneficial to this thesis.

Chapter 6

Conclusion

The purpose of this thesis was to create a distributed, passive, low impact network visualization tool that relies solely on STP packets. With STPViz, we created a network visualization tool that does just that. It can extract information about bridges from STP packets, and under the right circumstances even make assumptions about bridge connections. When the established spanning tree is expanded, and a new root is prepended, STPViz expands its known topology with it. The inherent uncertainty of these assumptions is reduced by combining data obtained from distributed nodes. This means that the usefulness of STPViz increases greatly with the number of clients running on a network. We tested STPViz to the greatest extend possible with the limited hardware we had at our disposal. In order to perform these tests we had to develop *software-switch*, a utiliy that mimics STP capable bridges. It is a very basic and minimal tool, developed just far enough to make it usable for our tests. STPViz passed the tests done in a static network environment (testing bridge discovery, connection discovery and bridge removal). When we tested STPViz in a dynamically changing network, disconnecting subtrees and reconnecting them in different places, we found faulty behaviour in our *software-switch* utility. Therefore, we were unable to accurately test STPViz in our last two scenarios.

The tests we could perform, however, showed that STPViz is fully capable of bridge discovery and removal, as well as making assumptions about bridge connections on topology updates. In our tests, STPViz correctly filtered out incorrect assumptions in the server by comparing information from different clients. Unfortunately we did not find any other general method of checking assumed topologies. The lack of mechanics like a hop count on the Data Link layer keeps us from implementing something like a *traceroute*, which we could use to check our assumptions manually.

Future extensions of STPViz could include features to increase usability, like a GUI, as well as other output formats. Extending the functionality of STPViz to STP extension, as well as alternatives to STP (discussed in Section 2.2) would also be very helpful. Additionally we should extend the parser to be capable of generating output from one or more *.pcapng* files without needing to launch a server and multiple clients beforehand. With more extensive testing and statistical analysis of the results, it is our opininon that the assumptions we make could be improved as well.

The *software-switch* utility could also be extended and tested more thoroughly, to make it usable as a general purpose switching utility. To this end, the update of root information

could be improved, as currently the bridges take a long time to react to a downed connection via their root port.

Other extensions of this thesis include work on the *OpenWrt* and *dd-wrt* projects in order to find a viable method to keep the STP implementation consistent with other hardware.

Bibliography

[1] Becker, R. A., Eick, S. G., and Wilks, A. R. Visualizing network data. *IEEE Transactions on Visualization and Computer Graphics 1*, 1 (Mar. 1995), 16–28.

[2] Blue, R., Dunne, C., Fuchs, A., King, K., and Schulman, A. *Visualizing Real-Time Network Resource Usage*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 119–135.

[3] brianwhelton. Solarwinds thwack forum. https://thwack.solarwinds.com/ideas/3034. Accessed: 2016-09-06.

[4] Brocade. STP Parameters and Defaults. http://www.brocade.com/content/html/en/configuration-guide/FI_08030_L2/GUID-256DE2C6-C2DC-4DE0-A0C2-A96DDF0C3406.html. Accessed: 2016-09-23.

[5] embeDD GmbH. dd-wrt. https://dd-wrt.com. Accessed: 2016-09-29.

[6] IEEE Standards Associaton. IEEE 802.1aq Standard. https://standards.ieee.org/findstds/standard/802.1aq-2012.html. Accessed: 2016-09-28.

[7] IEEE Standards Associaton. IEEE 802.1D standard. http://standards.ieee.org/getieee802/download/802.17a-2004.pdf. Accessed: 2016-09-28.

[8] International Organization for Standardization. ISO/IEC 9899:1999. http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=29237. Accessed: 2016-09-29.

[9] L2Discover. L2discover. http://www.inf.u-szeged.hu/~bilickiv/l2discover/. Accessed: 2016-09-06.

[10] LiveAction. LiveNX Network Performance Management. http://www.liveaction.com/. Accessed: 2016-09-06.

[11] LoriotPro. LoriotPro Network Management Software. http://www.loriotpro.com/. Accessed: 2016-09-06.

[12] Olukotun, K. Shared Memory Parallel Programming, Pthread/OpenMP Examples, 2011. Stanford University, http://web.stanford.edu/~tayo/temp/aust_parallel/lect.02.ShMemPrg.pdf, Accessed: 2016-09-21.

[13] open-source-parsers Github Community. A C++ library for interacting with JSON. https://github.com/open-source-parsers/jsoncpp. Accessed: 2016-09-06.

[14] OpenWrt Community. OpenWrt. https://openwrt.org/. Accessed: 2016-09-29.

[15] Perlman, R. An algorithm for distributed computation of a spanningtree in an extended lan. *SIGCOMM Comput. Commun. Rev. 15*, 4 (Sept. 1985), 44–53.

[16] Schlögl, A. software-switch. https://github.com/alxshine/software-switch. Accessed: 2016-09-29.

[17] Tantau, T. *The TikZ and PGF Packages*, 1.18 ed. Institut für Theoretische Informatik, Universität zu Lübeck, June 2007.

[18] tcpdump & libpcap Community. tcpdump & libpcap. http://www.tcpdump.org/. Accessed: 2016-09-29.

[19] The Linux Information Project. Broadcast Storm Definition. http://www.linfo.org/broadcast_storm.html. Accessed: 2016-09-09.

[20] Wongsuphasawat, K., Artornsombudh, P., Nguyen, B., and McCann, J. Network stack diagnosis and visualization tool. In *Proceedings of the Symposium on Computer Human Interaction for the Management of Information Technology* (New York, NY, USA, 2009), CHiMiT '09, ACM, pp. 4:29–4:37.