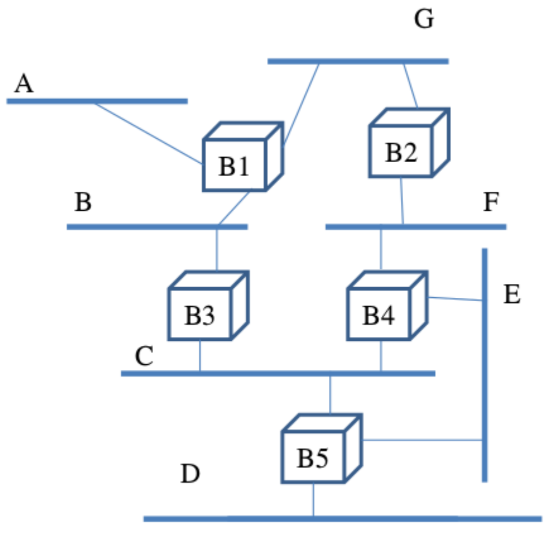
Lab 07 - Spanning Tree Protocol Implementation and Bridge Simulation

<u>Can be written in C++ or python</u>

In this lab you will implement the spanning tree protocol on a given LAN and bridge topology, and then simulate the functioning of the learning bridges for a sequence of given data transfers.

For example, consider the following LAN topology



Topology

This will be specified to you as follows:

1
5
B1: A G B
B2: G F
B3: B C
B4: C F E
B5: C D E

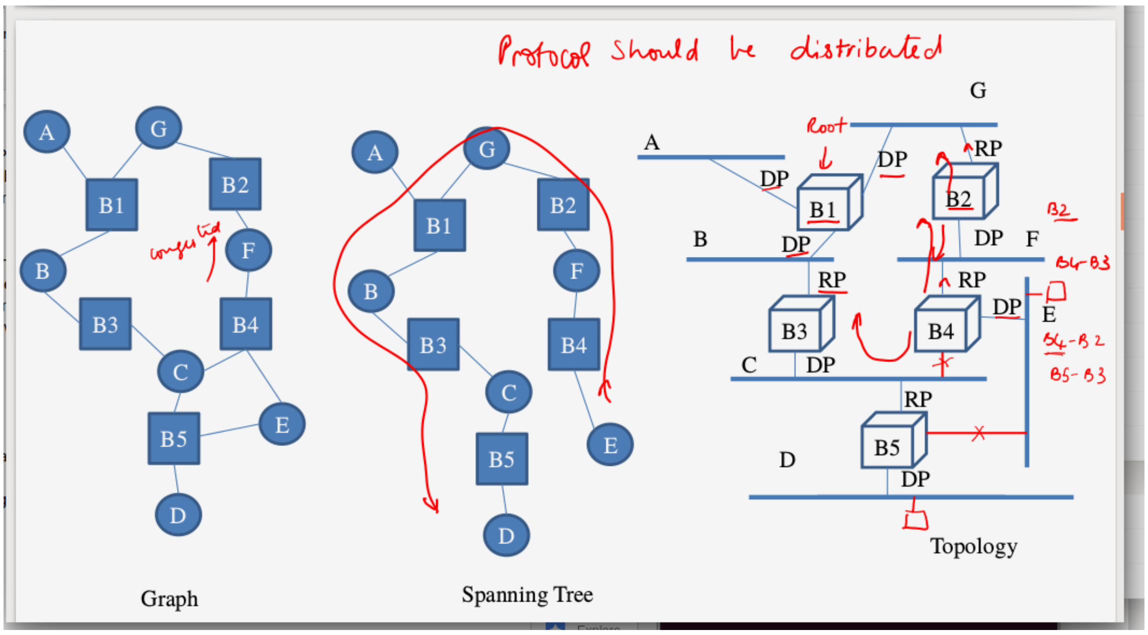
Here, 1 is a trace flag, which if set to 1 should write a detailed trace to stdout, and if set to 0 should produce no trace. 5 specifies the number of bridges whose details will be specified. Each Bridge is then listed in the given syntax showing the LANs to which it is connected directly. You may assume that bridges names will be B1, B2, B3… and LAN names will be single Characters. The bridge list will be specified in order of its ID (bridge ID).

You have to write a program that first reads the above input, creates some internal representation of the LAN topology, and then starts with states of a Bridge's ports as active on all ports. It should then simulate the running of the spanning tree protocol - thus at t=0 all bridges will send their advertisements and then as time progresses will behave according to the protocol. After the protocol converges, messages will stop, and then your program should output the state of each port as follows:

```
B1: A-DP  B-DP  G-DP
B2: F-DP  G-RP
B3: B-RP  C-DP
B4: C-NP  E-DP   F-RP
B5: C-RP  D-DP  E-NP
```

(Print bridges and port IDs in increasing order)If there are multiple bridges, then for all i,j such that i<j, bridge Bi should be printed before Bj.
Similarly if there are multiple ports for a bridge, print them in lexicographic order. Note that there can be at most 26 distinct ports.

Here DP = Designated Port, RP = Root Port, and NP = Null Port (deactivated port). Note here that the port of a bridge is simply referred to by the name of the LAN which it is connected to. This implements the spanning tree as shown below.



In the simulation, you should assume that the time required for data transfer across any LAN segment is **one** time unit. E.g., in this topology, a message from B1 sent at time *t* will reach B2 and B3 at time *t+1*. When the trace flag is on, you should produce a trace with the following format while the simulation is going on.

t   s|r Bk   (Bi, d, Bj)

where
t is the time of the event
Bk is the ID of the node at which the event has happened
s or r represents send or receive event
(Bi, d, Bj)   is the message indicating that Bridge Bj thinks Bridge Bi is the root and it is at a distance *d* from the root.
Trace outputs should be first sorted by time and for the same time, lexicographically sorted. Note that Bi comes before Bj if i<j. So B9 will come before B11 and so on. *Note that traces will NOT be graded per se. They are for aiding manual grading of the actual output in case auto-grading of that output has some genuine problem for your submission and we need to understand how your code is working - this is the ONLY PURPOSE of the trace (debugging for you and your TA, if required). You can enhance/modify the trace in any way you wish.*

After this, the program should read a list of host IDs per LAN and a set of data transfer end points which will be specified as follows

```
A: H1 H2 H3
B: H4 H5
C: H6 H7 H8
D: H9 H10
E: H11
F: H12 H13
G: H14
3
H9 H2
H4 H12
H3 H9
```

Explanation of above input:
3: This means that 3 transfers will be specified.
H9 H2 : means Host H9 is sending to Host H2
And so on

Your program should print out the forwarding tables at each bridge after each data transfer, in the following syntax. E.g. after reading H9 H2, it should print out:

```
B1:
HOST ID | FORWARDING PORT
H9 | B
```

//Above means that a packet arriving with destination address H9, will be forwarded on the port connected to LAN B. The rest of the output will be:

```
B2:
```

HOST ID | FORWARDING PORT
H9 | G
B3:
HOST ID | FORWARDING PORT
H9 | C
B4:
HOST ID | FORWARDING PORT
H9 | F
B5:
HOST ID | FORWARDING PORT
H9 | D
Furthermore, the table for bridge Bi should be printed before bridge Bj if i<j.
Also, in each table, if there are multiple entries, then for all i,j such that i<j, entry for Hi should be printed before Hj.
Note: There will be a space between HOST ID, | and Forwarding port

If trace flag is set to one, produce the following trace for this simulation (assume again that "crossing" each LAN segment will incur one time unit delay.

```
t   s|r  Bk     X --> Y
```

This means at time t, at Bridge Bk, a packet arrived (r) or was sent (s), where the packet source address was on LAN X and packet destination address was on LAN Y.
The trace output should be lexicographically sorted.
Note that Bi comes before Bj if i<j. Similarly for Hi and Hj.
For example, B9 comes before B11 and H8 comes before H12.

Do not simulate other delays (e.g. processing delays at bridge), behaviours (e.g. bridge failures), MAC protocol (e.g. backoff, collisions), etc etc etc etc complications that are not mentioned in this problem statement. The **unit transfer delay is constant.** The bridges do not fail so after convergence of spanning tree your simulation should stop (the root bridge does not need to keep sending configurations). These assumptions are so that the assignment remains simple and doable in 7-8 hours.

This is an individual lab. You may code in any language of your choice among C++, Java, python as long as the input and output syntax is STRICTLY followed. Please note we will use auto-grading for functionality check and you WILL lose marks if output syntax is not strictly followed. You may discuss concepts and approaches with friends but every single line of the program should be yours and yours alone. We will run sophisticated plagiarism detectors and even the slightest hint of copying will result in a report to DDAC..

Overall Input for the above example

Your program should read input such as above from an input file (by just redirecting stdin). The overall file for the above example will contain:

```
0
5
B1: A G B
B2: G F
B3: B C
B4: C F E
B5: C D E
A: H1 H2 H3
B: H4 H5
C: H6 H7 H8
D: H9 H10
E: H11
F: H12 H13
G: H14
3
H9 H2
H4 H12
H3 H9
```

The overall output can also be written to a file (by redirecting stdout). For the above example (since trace flag is 0) it will be:

```
B1: A-DP B-DP G-DP
B2: F-DP G-RP
B3: B-RP C-DP
B4: C-NP E-DP F-RP
B5: C-RP D-DP E-NP
B1:
HOST ID | FORWARDING PORT
H9 | B
B2:
HOST ID | FORWARDING PORT
H9 | G
B3:
HOST ID | FORWARDING PORT
H9 | C
B4:
HOST ID | FORWARDING PORT
H9 | F
B5:
```

HOST ID	FORWARDING PORT
H9	D

B1:

HOST ID	FORWARDING PORT
H4	B
H9	B

B2:

HOST ID	FORWARDING PORT
H4	G
H9	G

B3:

HOST ID	FORWARDING PORT
H4	B
H9	C

B4:

HOST ID	FORWARDING PORT
H4	F
H9	F

B5:

HOST ID	FORWARDING PORT
H4	C
H9	D

B1:

HOST ID	FORWARDING PORT
H3	A
H4	B
H9	B

B2:

HOST ID	FORWARDING PORT
H4	G
H9	G

B3:

HOST ID	FORWARDING PORT
H3	B
H4	B
H9	C

B4:

HOST ID	FORWARDING PORT
H4	F
H9	F

B5:

HOST ID	FORWARDING PORT
H3	C
H4	C

Original Submission Instructions

Create a directory named rollno where rollno is your roll number. The directory must contain a script titled run.sh, which must take in exactly one command line argument corresponding to a input file. For testing your code, the following command will be used from your submission directory.

```
./run.sh inputFileName
```

wherein inputFileName will include the full path.

Include a file called readme.txt in the submission directory, that describes details regarding class and method design, and algorithm sketch. In summary: you will place the following files in submission.

run.sh and all the code that it needs to run.
readme.txt

Compress the directory into rollno.tar.gz and upload on Bodhitree.

Updated Submission Instructions (autogradeable on Bodhitree itself)

Note: Submit the following files: main.cpp bridgesim.cpp, bridgesim.h, bridge.h, bridge.cpp. The code is expected to have the following architecture:

1. bridge.cpp and bridge.h has the declarations and definitions of the bridge class, which you will most definitely need.
2. bridgesim.cpp and bridgesim.h will code the simulation
3. main.cpp will just declare the required objects and call all the required methods

Put all these code files in a directory, tar and submit it using a <rollno.>.tar.gz

We'll soon be adding auto-graded testcases on bodhitree.

Note that the best way to run your code repeatedly for debugging is to put the inputs in a file (don't type at the terminal), and remember to use input and output redirection commands for reading and writing input. E.g. :

First compile your code:

```
g++ -o bsim main.cpp bridgesim.cpp bridge.cpp.
```

Then run it as

```
./bsim < inp1 > out1
```

where inp1 is where you have written all your testcase input and out1 will capture the output. Check your output by the following command:

```
more out1
```

Or

```
less out1
```

Be efficient in your coding. Have separate edit windows to edit your code, and a separate terminal to keep compiling/debugging, or of course, use a good IDE like eclipse or netbeans.

Ensure you use DEBUG MODE (of your IDEs) for quick debugging. “Print” is not the best way to debug.

Hand-executed Examples

See peterson & Davie relevant extract

Example 1 (Submitted by Saurav, worked out by Varsha)

```
1
2
B1: A
B2: A
```

//B1, B2 have one port each both connected to A
t   s|r Bk   (Bi, d, Bj)

Initially, all ports are DPs.

Config messages ||| Why ||| Local State
0 s B1 (B1,0,B1) ||| Initialization ||| B1 is root, Port to A is DP
0 s B2 (B2,0,B2) ||| Initialization ||| B2 is root, Port to A is DP
1 r B1 (B2,0,B2) ||| B1 is one hop away ||| No change at B1
1 r B2 (B1,0,B1) ||| B2 is one hop away ||| B1 is root, Port to A is RP
1 s B1 (B1,0,B1) ||| B1 still doesn’t have a ‘better message’ ||| B1 is root
~~1 s B2 (B1,1,B2) ||| B1 has lower ID, one hop away ||| B1 is root, Port to A is RP~~
2 r B1 (B1,1,B2) |||messages reaching ||| No change
2 r B2 (B1,0,B1) |||messages reaching ||| No change
2 s B1 (B1,0,B1) ||| B1 doesn’t have a ‘better message’ ||| B1 is root, Port to A is DP

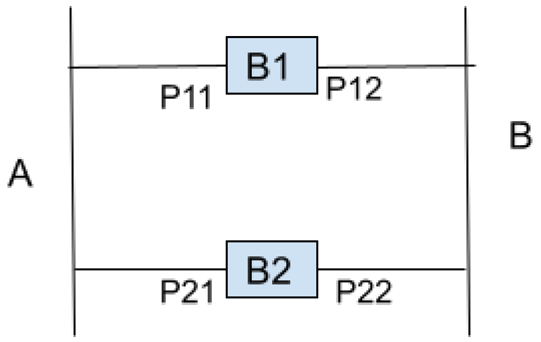
~~2 s B2 (B1,1,B2) ||| B2 doesn't have a 'better message' ||| B1 is root, 1 hop away, Port to A is RP~~

Once no bridge is changing any message generated/forwarded, simulation can stop.

Example 2 (Submitted by Saurav, Worked out by Varsha)

1
2
B1: A B
B2: A B

//B1, B2 have two ports each, connected to A and B respectively



t s|r Bk (Bi, d, Bj)

Initially, all ports are DPs.

Config messages ||| Why ||| Local State
0 s B1 (B1,0,B1) ||| Initialization, send on p11 ||| B1 is root,both ports DP
0 s B1 (B1,0,B1) ||| Initialization, send on p12 ||| B1 is root, both ports DP
0 s B2 (B2,0,B2) ||| Initialization, send on p21 ||| B2 is root, both ports DP
0 s B2 (B2,0,B2) ||| Initialization, send on p22 ||| B2 is root, both ports DP
1 r B1 (B2,0,B2) ||| message reaching port p11 ||| B1 is root,both ports DP
1 r B1 (B2,0,B2) ||| message reaching port p12 ||| B1 is root, both ports DP
1 r B2 (B1,0,B1) ||| message reaching port p21 ||| B1 is root, this port RP
1 r B2 (B1,0,B1) ||| message reaching port p22 ||| B1 is root, this port NP
//p22 could have been RP too, this tie is broken by ID
1 s B1 (B1,0,B1) ||| message send on port p11 ||| B1 is root,both ports DP
1 s B1 (B1,0,B1) ||| message send on port p12 ||| B1 is root, both ports DP

From Peterson & Davie for reference

Specifically, the configuration messages contain three pieces of information:
1. The ID for the bridge that is sending the message
2. The ID for what the sending bridge believes to be the root bridge
3. The distance, measured in hops, from the sending bridge to the
root bridge

Each bridge records the current best configuration message it has seen on each of its ports ("best" is defined below), including both messages it has received from other bridges and messages that it has itself transmitted.

Initially, each bridge thinks it is the root, and so it sends a configuration message out on each of its ports identifying itself as the root and giving a distance to the root of 0. Upon receiving a configuration message over a particular port, the bridge checks to see if that new message is better than the current best configuration message recorded for that port. The new configuration message is considered better than the currently recorded information if any of the following is true:

Rule1. It identifies a root with a smaller ID.
Rule2. It identifies a root with an equal ID but with a shorter distance.
Rule3. The root ID and distance are equal, but the sending bridge has a smaller ID

If the new message is better than the currently recorded information, the bridge discards the old information and saves the new information. However, it first adds 1 to the distance-to-root field since the bridge is one hop farther away from the root than the bridge that sent the message.

(Rule 4) When a bridge receives a configuration message indicating that it is not the root bridge—that is, a message from a bridge with a smaller ID—the bridge stops generating configuration messages on its own and instead only forwards configuration messages from other bridges, after first adding 1 to the distance field. Likewise, (Rule 5) when a bridge receives a configuration message that indicates it is not the designated bridge for that port—that is, a message from a bridge that is closer to the root or equally far from the root but with a smaller ID (here the message should be identifying the same bridge as the root, as the receiving bridge currently identifies) —the bridge stops sending configuration messages over that port. Thus, when the system stabilizes, only the root bridge is still generating configuration messages, and the other bridges are forwarding these messages only over ports for which they are the designated bridge. At this point, a spanning tree has been built, and all the bridges are in agreement on which ports are in use for the spanning tree. Only those ports may be used for forwarding data packets in the extended LAN.