# EE384A Programming Assignment #1
# IEEE 802.1 LAN Bridging

Assigned: Thursday January 18, 2001
Due: Friday, February 2, 2001*

- You may do this assignment in groups of two or individually.
- *Automatic extension: submissions before 11:59 pm on Sunday, February 4 will not be penalized. Any submissions later than this will receive NO credit!

## I. Introduction

The goal of this assignment is to provide you with a thorough understanding of LAN bridging, as needed for the implementation of the related standards. In the process, you will have the opportunity to be exposed to the actual implementation of an important industry standard.
In this assignment you are asked to implement the main components of the IEEE802.1D standard, namely:

1. The Spanning Tree Protocol
2. The Learning Process
3. The Forwarding Process
4. The Dynamic Multicast Filtering Capability (GARP/GMRP)

In this handout, we describe the requirements for your implementation of the first three of the above components. Requirements for GARP/GMRP will be provided in a separate handout.

## II. Scope

The entities to be implemented are specified in the IEEE 802.1D standard document. In order to reduce the complexity of the assignment, the following assumptions are made. It is your responsibility to extract and implement the reduced functionality implied by the assumptions below from the complete description given in the standard.

### II.1 General Assumptions

1. We will be limiting the LAN segment technology used to Ethernet. Therefore, bridges do not have to translate frames between different formats.
2. Bridges do not implement Traffic Class Expediting. You are not required to support the requirements related to this functionality.
3. You don't have to implement any network management related functionality.
4. We consider that there are no errors in transmission on the link. Therefore you don't have to deal with error detection.
5. You don't have to implement the MAC service at the ports.

### II.2 Spanning Tree, Forwarding and Learning Assumptions

1. We won't be using the bridge or port priority fields in the computation of the spanning tree.
2. All LAN segments are of the same speed (say 10 Mbps Ethernet). Therefore, the path cost is the same for all links. The path cost is assumed to be 1 for each link (hop count).
3. You don't have to deal with static entries in the filtering database.
4. You don't have to implement the *hold timer*.

### II.3 GARP/GMRP

Assumptions specific to GARP/GRMP are described in the GARP handout.

## III. Implementation Structure

The elements of the bridge functionality you are to implement are shown in Figure 1 below. Squares represent data structures while ovals represent processes. Namely:

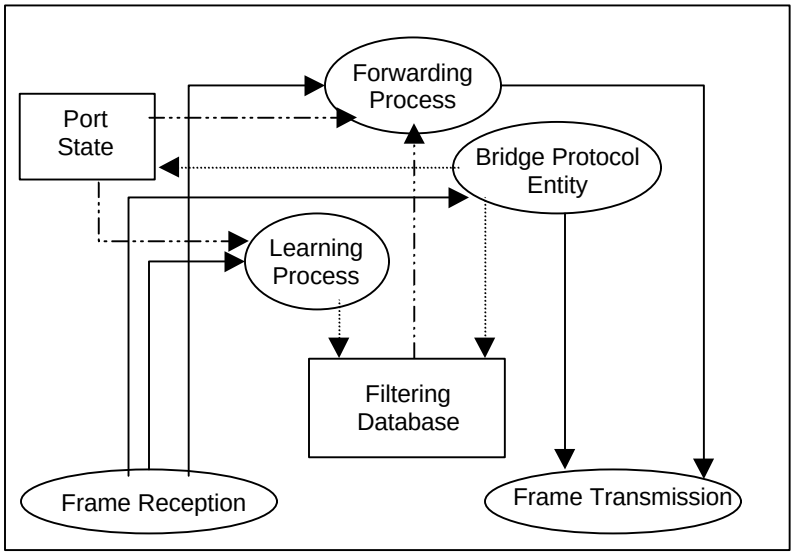


**Figure 1**

### III.1 Port State

You are to design and implement an appropriate data structure for this purpose, and support the interface function described in section V.

### III.2 Filtering Database

The information about MAC address locations is placed in this structure, and is used by other entities of the bridge architecture. You are to design and implement an appropriate data structure for this purpose, and support the interface function described in section V.

### III.3 Bridge Protocol Entity

This entity handles the spanning tree computation. You are to implement it as specified in the standard document. The timers related to this entity are to be updated from information contained in the *Execute()* function call as described in section V.
You are required here to implement the full protocol, i.e. the spanning tree must be maintained over time, and must react to changes in the topology using the mechanisms

specified in the standard. You also have to implement the interface functions that are described in section V.

### III.4 Learning Process

This process extracts MAC address location information from received frames and updates the filtering database. You are to provide an implementation that follows the specifications of the standard.

### III.5 Forwarding Process

The functionality that you are required to implement for this process is the forwarding of frames to the appropriate port(s), based on the filtering database information and the port state information. You are *not* to implement the requirements for the selection of queued frames for transmission and priority mapping.

### III.6 Frame Reception and Transmission

We model frame reception as follows. Each bridge has one receive queue, in which frames received on the different ports are placed as they are received. You are to implement the distribution of frames received in this queue to the different processes in the bridge. Refer to section V for the description of the implementation of the receive queue.

Frame transmission is modeled as follows. Frames that are to be sent from a bridge port on the appropriate segment are passed to *SendMACFrame()*, a function that we provide. More details on this function, including the arguments to provide in the function call, are provided in section V.

## IV Programming Specifications

The following programming specifications are also made in order to reduce the complexity of the implementation:

1. You will be dealing with one frame format only, which is fixed and specified later in this document. This frame is used to carry BPDUs, GARP messages and user data. You are free to use any format for BPDU encapsulation in the frame since BPDUs will only be parsed by your own code. However, you will need to provide the exact specified fields when asked in the interface function *GetConfigBPDUatPort()*.
2. To simplify the code further, we will consider MAC addresses to be 32 bit integers instead of 48 bit values (i.e., must be declared as type `int` in *C*). We follow the convention that unicast addresses are positive values while multicast addresses are negative (this relates to the fact that the first bit on the wire would be 1, which is the first bit in a negative integer). In particular, the destination address in the special broadcast frame is given a special value (BROADCAST).
3. We use a simplified architecture for the bridge, in particular concerning the reception and transmission of MAC frames (described in section III.6 above).

4. All timers used have to be updated when the bridge *Execute()* code is called (see section V). You will not need to work with the timer functions available in *C*. Timers should be declared as type `unsigned int` or, preferably, `unsigned long int`. The range for an int should suffice because timers are rolled back to zero upon expiry and the largest possible timer is the topology change timer in the root bridge (i.e., max age + forward delay = 35,000 ms).
   Although we are not going to check the transient states at such a fine granularity for it to matter, you should update the timers *before* you process the receive queue.

## V. Code

The skeleton files for this assignment can be found on the class web page, or from your leland acccount at `/usr/class/ee384a/WWW/progAssig1`. The files are:

`bridge.h` : Declaration and description of the functions in bridge.c that are used by the simulator program to interact with your code.
`bridge.c` : You are to implement these functions as described in this handout and bridge.h.
`network.o` : Object file for the simulator program
`Makefile` : For compilation of the bridge code.
`topology.in` : A sample topology file. This corresponds to the topology in homework assignment 1. But the port and segment numbering here starts with 0 and not 1 as in the homework assignment.
`scenario.in` : A sample scenario file for this topology.
`topology.out` : Output produced by the simulator program for this topology and scenario files.
`station2.log` : A sample station log file for station 2 in the above topology.in file. This corresponds to the traffic activity specified in the above scenario.in file.

Please refer to the `bridge.c` and `bridge.h` files for a detailed and authoritative description of the functions and data structures for this assignment. Below we only summarize that information.

### V.1 Functions

You are required to write the code for the LAN bridge in the `bridge.c` file. You are provided with object code of a bridged LAN simulator program, which is described in section VI. The interface to the simulator, which you must use, consists of the following functions:

```
1. void InitializeBridge(. . . .);
2. int Execute(. . . .);
3. PortStateType GetPortState(. . . .);
4. int GetRootPort(. . . .);
5. MACFrame *GetConfigBPDUatPort (. . . . .);
6. int *GetFDBEntry (. . . . .);
7. void* AllocateBridge();
```

You are provided with the following function that allows use to send MAC frames on a segment:

```
void SendMACFrame(. . . .);
```

This function takes as arguments the **global** port number of the sending port, and the frame to be sent. It delivers the frame to the appropriate ports of the bridges and stations on the segment connected to the sending port.

## V.2 Data Structures

The following data structures are specified for the interface between your code and the simulator.

### *V.2.1 MACFrame*

This structure defines the encoding of a MAC frame. It is an abstraction of the real MAC frame format to simplify the programming. This structure is used to send BPDUs, GARP messages and user data.
The `length` field is set by the bridges for BPDUs and the GARP messages sent by the bridges, while the sending stations set it for a data frame. This field has to be a valid number in an Ethernet LAN, and indicates the size *in bytes* of the frame, as it would be in reality.
The `data` field contains the appropriate information, depending on the frame type.

```
/* MAC frame format */
typedef struct {
  int dstAddress;
  int srcAddress;
  int length;
  int DSAP;
  int SSAP;
  int data[FIELDS];
} MACFrame;
```

### *V.2.2 MACQueueElement*

This structure defines the encoding of an element in the receive queue of a bridge. It contains a pointer to a received frame, the global port ID of the port where the frame was received and a pointer to the next element in the queue. The frames in the queue are in chronological order of their receipt, where the frame at the head of the queue is the first one received. When traversing the queue, follow the pointer to the next element. The last element in the queue points to a *NULL* next. An empty queue has the value *NULL*. *The simulator frees the memory for this structure – you should not free it after you are done processing it – since the same pointer might be in the receive queues of other bridges.*

```
typedef struct MACqueue_element{
  MACFrame *frame;
  struct MACqueue_element *next;
  int globalPortID;
} MACQueueElement;
```

## V.3 Constants

The following constants are defined. You are to use them where needed.

### *V.3.1 Addresses*

| | |
|---|---|
| *BROADCAST* | destination address to be placed in a MAC broadcast frame. |
| *ALLBRIDGES* | destination address to be placed in a BPDUs. |

*V.3.2 Protocol Numbers*

| | |
|---|---|
| *IP* | DSAP and SSAP value for the IP protocol. |
| *BRIDGEPROTOCOL* | DSAP and SSAP value for the BPDU |

*V.3.3 Port States*

| | |
|---|---|
| *Blocking* | port state for blocked port. |
| *Listening* | port state for DP or RP. |
| *Learning* | port state for DP or RP. |
| *Forwarding* | port state for DP or RP. |

*V.3.4 Other*

| | |
|---|---|
| *FIELDS* | Number of fields in a data frame. This is set to 250 as the simulator uses a static array for the data fields. |

## VI. The Network Simulator

You are provided with an object code of a network simulator that you can link to your code to test it. The simulator uses input text files that specify the network topology, as well as the scenario of events (e.g., a bridge going down, a station sending a frame etc…) that occur over time.

### VI.1 Brief Simulator Description

The network simulator performs the following functions:

1. It allows you to define the network topology, in terms of bridges and stations, and the segments that interconnect them.
2. It is responsible for the transfer of frames between from a network entity (bridge or station) to the other entities that are connected to the same segment. When a bridge needs to send a BPDU or forward a data frame on one of its ports, it calls the *SendMACFrame()* function provided by the network simulator. This function takes care of delivering the frame to the appropriate ports of the recipients on the segment connected to the sending port specified in the function call.
3. It allows you to specify the occurrence of events like a bridge going down, a station sending a data frame, display of a bridge's port states or filtering database, etc... in a scenario file. It is responsible for the processing of these events at the specified times.
4. At startup, it calls the *Initialize()* function in each bridge to set its ID, number of ports, timer parameters etc. Although the standard specifies that configuration BPDUs have to be sent at initialize time, this is not possible in the simulator (*SendMACFrame()* cannot be called before all bridges are initialized). Instead, you should send these BPDUs at the first *Execute()* call.

5. It performs a round robin call of each bridge *Execute()* functions, providing it with a queue of frames that it received since last executed. When all bridges have been called, the time is advanced one STEP (a constant defined internally within the simulator). The process is repeated until the time for the end of the simulation that is indicated in the scenario is reached.
6. It writes log files to a directory called *logdata* in your working directory (the directory where you run the simulator program). Each log file, named stationx.log (x = an integer) – contains the headers of the packets received by the station with ID x. Please note that you have to create the *logdata* directory *before* running the simulator.

After you create a directory called *logdata* in your working directory, to run the simulator, type `make` in your working directory (you are provided with a Makefile). This will create an executable file called *bridge*. You must compile from the leland `saga` or `elaine` machines, as they are compatible with the network.o object file. After compilation, the simulator is invoked by typing:

```
bridge topology.in scenario.in
```

If this causes a "*command not found*" error then you need to add the current directory "." to your path in the `.cshrc` file or use `./bridge` instead of `bridge` above.

At the end of a simulation, the output of the simulator is:

1. The state of each port in every *active* bridge at the time the simulation ended.
2. The contents of the filtering database in every bridge.
3. In the directory *logdata*, a text log file per station (`station#.log`) that contains the header info of all the frames received by that station.
4. A file called *SignalFile* will be present in the directory indicating that the program exited normally. It is used for automated grading purposes, so you may safely ignore it / delete it.

### VI.2 Network Topology File

The network topology is described in a text file (e.g., `topology.in`), which is passed as a command line argument to the simulator. The contents of the input file are as follows, note that commands are shown in normal face, while numerical values are shown in *italic*:

```
no_of_bridges          number
no_of_segments         number
bridge      0          bridgeID    numPorts
bridge      1          bridgeID    numPorts
...
port        globalPortID portID    bridge#     segment#
port        globalPortID portID    bridge#     segment#
...
port        globalPortID portID    bridge#     segment#
station     globalPortID stationID segment#
...
station     globalPortID stationID segment#
```

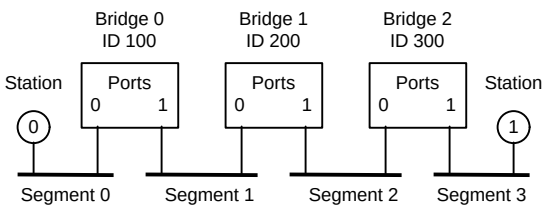


**Figure 2**

The following rules must be satisfied:

1. `no_of_bridges` and `no_of_segments` are positive integers.
2. *`bridge#`* lies between 0 and (`no_of_bridges`–1), both inclusive. Therefore, it is a unique number.
3. `globalPortID` is a integer value that uniquely identifies a bridge port or a station, somewhat like a MAC address. The range is 0 to total number of ports on all bridges plus total number of stations minus 1. It is the value used as source/destination address in data MAC frames, and as source address for BPDUs.
4. *`portID`* lies between 0 and (`numPorts`–1), -both inclusive- where *`numPorts`* is the number of ports on that bridge. PortID's are thus unique within a bridge.
5. *`numPorts`* is an integer greater than 1.
6. *`segment#`* lies between 0 and (`no_of_segments`–1), both inclusive. Therefore it is a unique number.
7. *`bridgeID`* is a unique non-negative integer that identifies the bridge.
8. *`stationID`* lies between 0 and the number of stations you want to specify minus 1. *`StationID`* is a unique number, and is used for internal bookkeeping purposes.

```
no_of_bridges 3
no_of_segments 4
bridge 0 100 2
bridge 1 200 2
bridge 2 300 2
port 0 0 0 0
port 1 1 0 1
port 2 0 1 1
port 3 1 1 2
port 4 0 2 2
port 5 1 2 3
station 6 0 0
station 7 1 3
```

**Figure 3**

The simulator performs checks for violation of these rules. However, it is always advisable to check the input files for mistakes when you encounter problems.

An example input file describing the topology in Figure 2, is shown in Figure 3.

### VI.3 Scenario File

In this file, which is also passed as a command line argument, you can set up a scenario of events which would happen over time, as well as the length of time the simulator should run. The events that are supported are:

| | | | | | |
|---|---|---|---|---|---|
| end_time | *time* | | | | |
| bridge_up | *bridge#* | *time* | | | |
| bridge_down | *bridge#* | *time* | | | |
| bridge_state | *bridge#* | *time* | | | |
| bridge_fdb | *bridge#* | *time* | | | |
| station_send | sendingGlobalPortID | *sourceGlobalPortID* | *destinationGlobalPortID* | *length* | *time* |
| station_reg | *sourceGlobalPortID* | *groupAddress* | *time* | | |
| station_dereg | *sourceGlobalPortID* | *groupAddress* | *time* | | |

The last two messages are related to Part B of this assignment (GARP/GMRP), which is described in another document.

**NOTE:** As mentioned earlier, the events in the scenario file must be specified in chronological order, *except the* end_time *statement*, which is placed at the beginning of the file.

*VI.3.1 end_time*

This statement specifies the time at which the simulation stops. The simulator works in small steps (finite time increments), and stops when the last increment resulted in a time value greater than `end_time`.

*VI.3.2 bridge_down*

This statement results in the specified bridge being brought down. From the time specified and on, the bridge does not receive any frames and its Execute() function is not called. A bridge in this state can be brought back to life using a `bridge_up` statement.

*VI.3.3 bridge_up*

This statement results in the specified bridge being brought back to work. The bridge is then initialized and starts functioning normally, as specified in the standard.

*VI.3.4 bridge_state*

This statement results in the specified bridge displaying its port states at the time specified.

*VI.3.5 bridge_fdb*

This statement results in the specified bridge displaying its forwarding database contents at the time specified.

*VI.3.6 station_send*

The source station in this statement sends a data MAC frame, of the length specified, destined to the destination station, originating at the sending station at the time indicated, and having as source address the *sourceGlobalPortID* specified. The *sourceGlobalPortID* refers to the MAC address that will be carried in the frame, while the *sendingGlobalPortID* identifies the actual station which will send the frame on the segment it is attached to. While it may be confusing at first, the distinction between “source” station and “sending” station is there in order to mimic a change of location for a station. Thus, when a station is sending from its actual location, the *sendingGlobalPortID* will be the same as the *sourceGlobalPortID*. However, to mimic a change of location for a station, specify as *sendingGlobalPortID* the globalPortID of a station lying on the segment to which the station has moved. Then, the station_send command causes a frame with source MAC address = *sourceGlobalPortID* to be sent on the segment where the station with global port ID *sendingGlobalPortID* lies.

*VI.3.7 station_reg*

The source station in this statement sends a GARP Join message for the MAC Group Address specified, at the time indicated.

*VI.3.8 station_dereg*

The source station in this statement sends a GARP Leave message for the MAC Group Address specified, at the time indicated.
An example scenario file is shown in figure 4 above.

```
end_time 20000
bridge_down 1 5000
station_send 6 6 7 1000 6000
bridge_up 1 10000
```

**Figure 4**

*VI.3.7 NOTES*

1. Given that the processing is performed in finite time steps, you cannot specify arbitrary small times between events that depend on each other. A safe value for inter-event time is 1 second.
2. *All* `time` values are given in *milliseconds*.

## VII. Deliverables

You are required to submit your *C* source files, a README file that contains your name and email address (and that of your partner if done in a group) as well as a short report that describes your implementation, and its status. This file is important for the evaluation of your work so please make sure you submit it.

### VII.1 Submission

You have to turn in your `bridge.h`, `bridge.c` and README files. To submit them, go to the directory where they are located and type:

```
/usr/class/ee384a/bin/submit
```

You should get a "submission successful" message indicating the success of the operation.

### VII.2 Comments from the TAs

Please make sure your submission follows the additional guidelines below.

1. Make sure you remove any debugging *printf* statements in your code. Your assignment will be graded by an automated script, which compares your output to the correct output, and therefore may interpret any unrecognizable output as errors. You may be penalized for this extraneous output.
2. Make sure your code finishes in a reasonable time. A correct implementation should terminate in less than a minute on the leland system's *sagas* and *elaines*. Very long run times are mostly due to incorrect behavior, rather than to "inefficient" programming techniques, such as the use of statically allocated arrays and simple search algorithms and so on.
3. Create new scenarios and topologies, both simple and complex, to test your code. The fact that it works with the example topology provided doesn't guarantee its correctness.