SurfNet Data Layer Design

Darrick J. Wong 23 February 2004

In this paper, I present the design specification for the data layer of SurfNet, a simple network simulator. The goal of SurfNet is to achieve verisimilitude with a real Ethernet network without requiring users to know too much about the minute details of a computer network. Ostensibly, SurfNet could be used as starter software for a course in computer networking, as it also aims to follow established networking standards as closely as possible.

Contents

1 Introduction

SurfNet is a simulator of an Ethernet-based computer network. It employs a variety of different technologies to construct a VLAN in software, though it adheres to well-known specifications whenever possible. We hope to build a network simulator sufficiently robust that it may operate flawlessly on a real Ethernet. This project was kicked off in early 2004 by Stefan Savage, Geoff Voelker, Yu-Chung Cheng and Darrick Wong as a replacement for the FishNet software used in the computer networking course at UCSD.

1.1 Copyright

1.2 Disclaimers

This design document strives to be a useful explanation of the ideas that went into the creation of SurfNet. It should not be a big surprise then, that the document provides a bird's eye view of the software components in SurfNet and how they are put together, but does not cover every detail of how the source code works. Indeed, one should read the SurfNet code itself and use this document as a tour guide.

All copyrights are owned by their owners, unless specifically noted otherwise. Use of a term in this document should not be regarded as affecting the validity of any trademark or service mark.

Naming of particular products or brands should not be seen as endorsements.

Do not plug a network card into a running computer. It will not like you.

1.3 Revision History

Date	Author	Comments
02/10/04	Darrick Wong	Initial revision.
02/17/04	Darrick Wong	Initial reviews by Yu-Chung Cheng

Date	Author	Comments
02/23/04	Darrick Wong	Review by Stefan Savage and Geoff Voelker

2 Terminology

SurfNet: At the lowest level, SurfNet is defined as a bidirectional graph of nodes and links. During operation, each link is associated with two nodes at each endpoint, and each node is associated with one link. A network interface is interposed between a node and a link. Several nodes and links may exist within a computer process; these processes are known as a simulator. Links may exist entirely as an in-memory abstraction, or they may exist as wrappers around networked I/O streams between different simulators.

Simulator: Software process that contains nodes, interfaces, and links, and is in charge of maintaining the software stack as well as the correct routing data between all components. In that respect, the simulator acts like kernel-level OS networking software.

Node: A node owns several network interfaces. Data can be routed between any of the interfaces or further up the network software stack.

Interface: This object represents the interface between a node and the network link. In a real computer, this would be the Network Interface Card (NIC). It plays the part of the data layer in the OSI model. A network, then, is defined as a graph of network nodes and links; interfaces are sandwiched between a node and a link.

Note: Nodes are always attached to links through an interface, even if that is not explicitly stated in this document.

Link: A network Link plays the part of the physical layer in a regular network: It is responsible for transmission of Ethernet frames between two interfaces in a network. Links are only to be established between two interfaces.

Link Between Simulators: A link between two interfaces in different simulators is a specialization of the generic Link. This kind of link uses some sort of I/O stream and communications protocol to ship data between SurfNet simulators. Henceforth, these shall be known as **NetLinks**.

Link Within a Simulator: A link between two interfaces in the same simulator does not use the host OS's I/O facilities at all. Rather, it utilizes an in-memory queue of frames as a pretend network link. Because pointers to frames are passed within a process, they are much more efficient than frames passed between simulators. These links will be known as **MemLinks**.

Address: The unique identifier of a network interface. The Ethernet equivalent is a MAC address.

Frame: Basic unit of data used for Ethernet transfers, consisting of a data payload and other metadata.

Hub: A hub is constructed out of a bunch of links and interfaces; its purpose is to connect several interfaces together so that the nodes on the other ends of the links see all traffic.

Hub NetLink Factory: The fact that network traffic is sent through a hub and not to it provides us with remarkable flexibility in this case: a Hub's interfaces do not really need addresses! They simply forward all incoming frames upwards to a hub object, which handles rebroadcasting. Thus, we can create a modified Stream Server NetLink that creates a new Link/Interface pair for each incoming connection, attaches the link to the interface and the interface to the hub.[1] Obviously, these NetLinks are not automatically reconnecting.

3 Interactions

Several use cases have been identified during the elaboration of the logical objects contained within SurfNet's data layer:

3.1 Creation of Network Nodes

1. A node is created, and a name is associated with it.
2. Packet handlers that handle incoming frames are attached to the node.

3.2 Creation of Network Interfaces

Interfaces must be created as intermediaries between nodes and links. For data frames to be routed correctly, each interface must have a unique address on the SurfNet. SurfNet itself does not enforce address uniqueness; it is up to the programmer to do this himself. Node creation works as follows:

1. A link is created.
2. An address is created.
3. An interface is created with the address created in step 1.
4. The node is connected to the interface.
5. The interface is connected to a node, and callbacks are attached to the interface.

3.3 Automated Network Topology Setup

When programming, debugging, or evaluating network code, it is useful to have a quick method to save and reload particular network topologies. This is quite useful for the creation of test harnesses, as well as making automated network construction far less costly than it is in FishNet. This mechanism works as follows:

1. The programmer creates a configuration file listing information about network nodes and links between the nodes.
2. The programmer passes this configuration file to the simulator at startup.
3. The simulator interprets the configuration and assembles various software components.
4. Network simulation begins.

3.4 Network Tear Down

The natural complement of an automated network setup utility is a facility to save the network configuration to a file on disk.

1 There are practical limits; don't complain to me that you can't create 500,000 links to a hub!

1. The user will send the simulator a signal to dump the network configuration.
2. The simulator writes a configuration file to disk.

3.5 Sending Ethernet Frames

1. An interested user (probably the network layer, not discussed here) tells a node to send a packet with some type code to a certain address.
2. The node selects (or is given) the interface that will be used to send the frame, and passes the data to send to it.
3. The interface constructs the missing parts of the Ethernet frame (sender and CRC) around the packet and passes it to the associated link.
4. The link performs any necessary processing and sends out the frame.

3.6 Receiving Ethernet Frames

1. A frame comes in on a link.
2. The link passes the frame to the interface.
3. The interface passes the frame to the node.
4. The node uses the header information to validate the frame's contents, decide which handler to execute, and then dispatches the packet to the associated handler.

3.7 Frame Forwarding with a Hub

A frame forwarder does exactly what its name implies--forwards frames from one link to another. More specifically, a hub does not discriminate frames--it merely forwards incoming frames to all outgoing links. Hubs should not be used in networks with loops—frames will be forwarded around and around until a network crash. This describes a hub's operation:

1. A hub is created; various interfaces are attached to it.
2. The hub waits for incoming frames on any link.
3. Upon receipt of a valid frame, the hub broadcasts the frame on all of the other links.

4 Data Structures and Objects

4.1 Address

Addresses on a SurfNet are, in fact, standard Ethernet MAC addresses. This means that they are 48 bits, expressed as 12 hexadecimal digits (0-9, plus A-F, capitalized). The first 6 digits indicate the vendor of the Ethernet node and the last 6 digits specify a serial number for that node. Note that these requirements can be relaxed for a SurfNet that is not bridged with a real Ethernet.

Note: the second hex digit from the left must be even for multicast and broadcast addresses, and odd for all others.

Interested parties are advised to read *http://map-ne.com/Ethernet/* for a list of MAC address vendor codes and other information about broadcast/multicast MAC addresses.

4.2 Frame

In keeping with the practice of mimicking a regular network as much as possible, SurfNet employs the frame format outlined in the Digital-Intel-Xerox Ethernet standard of 1978. There is another Ethernet standard, IEEE 802.3[2]; fortunately, the two are fairly interchangeable. In any case, the frame format is presented here:

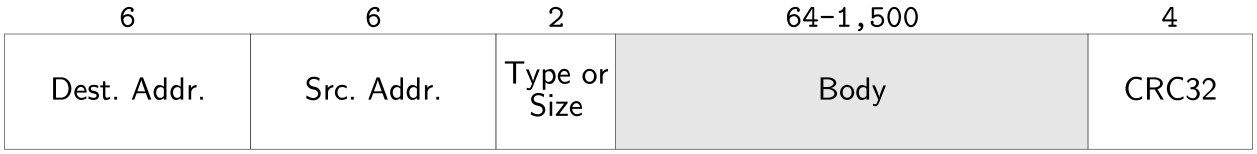


Figure 1: A D-I-X Ethernet Frame

The first six bytes are Ethernet address to which this frame is destined. The next six bytes are the address from which this frame originated. The address format is described in the previous section.

The two-byte field after the addresses is where the two standards diverge. Under the D-I-X standard, this field identifies the type of the data being carried in the frame. This is quite an interesting property of Ethernet frames; it means that they are self-identifying. When this frame reaches its destination, the network stack uses this type field to route the data to the appropriate protocol. Under the IEEE 802.3 standard, however, this field is the size of the body; there is usually other protocol information encapsulated in the frame body. Note that the D-I-X standard is a bit more widespread than 802.3. Because type information is encapsulated in the payload, 802.3 frames will be dispatched to an 802.3-specific layer, or be dropped if such a layer is not implemented.

Note: Frame type codes will always be greater than 1,500, and payload sizes will always be less than or equal to 1,500. This is how IEEE 802.3 frames are detected.

See *http://www.iana.org/assignments/ethernet-numbers* for a list of Ethernet frame types.

After that comes the frame's data payload. The payload may contain between 64 and 1,500 bytes of data.

Finally, a CRC32 ends the frame and provides a convenient method to verify data integrity.

4.3 Node

Nodes own one or many network interfaces. To write a frame to a particular interface, one must acquire a handle for the interface and then call its methods directly.

4.4 Interface

Interfaces contain an Address, some state flags, and a pointer to a Link that takes care of the details of sending and receiving Frames across a SurfNet. When there is a connected Link attached to this interface, frames can flow freely; in all other situations, attempts to send frames will fail.

2 Actually, there are more; they are neither widely used nor discussed here.

However, some elaboration about those data are required. First, the Address is attached to the header of all outgoing frames. Moreover, this Address is also used to filter out frames that are not attached to this SurfNet. Second, there is also a “promiscuous mode” flag that one can set to have the interface pass all received packets to the upper layers of the network stack; it is left up to the upper layers to handle those extraneous packets correctly.

4.5 Link

Links are very interesting creatures, primarily because there are so many different variations of them. Chapters 6 and 7 of this design specification will provide specific details of how various Links are implemented; in this section, the general plan for links will be laid out.

Despite the variation in actual transport mechanics (TCP socket, UDP socket, memory queue, etc.), all SurfNet links have two states and two operations. The two states are connected, in which the link can transport data, and disconnected, where it cannot. Not surprisingly, the two operations are write and read, which send and receive frames, respectively.

The decision to use a stateful link design was not easy; originally, attaching a link to an interface meant that data could be sent, and detaching it mean that data can no longer be sent. However, the design was changed to accommodate the use of a link over network sockets. Consider the following case: We want to link with some other SurfNet via a socket that is listening on `tiger:9999`. Furthermore, we want this link to be reliable; should `tiger` drop the connection, we would like the link to attempt to reconnect to `tiger` when it becomes available. Allowing links to remain attached to an interface during disconnected operation gives us the flexibility to provide such a service—the link will remain disconnected but attached until the underlying TCP connection can be recreated. This can even be done automatically.

5 Utilities

Several extra functional areas have been identified that have the potential to make a SurfNet programmer’s life easier.

5.1 CRC32

Sometimes, errors are introduced in to frames. This can result from electrical interference, thermal problems, hostile network equipment, or gamma rays from outer space. To guarantee the integrity of frames transferred across the Ethernet, a CRC32 checksum is attached to the frame to detect such errors and arrange for corrective action.

However, SurfNet, by running inside a computer program or over a reliable transportation mechanism such as TCP/IP, does not inherently suffer from such problems[3]. While it may be tempting to simply ignore the checksums on these grounds, both D-I-X and 802.3 Ethernet standards dictate the inclusion of a CRC32 checksum for verification purposes. Therefore, we must provide this support; to focus SurfNet programmers’ development efforts on the network, however, a facility will be provided to perform the calculation on a given chunk of memory.

5.2 Frame Manufacturing

For the novice programmer, it may be useful to have a built-in function that fills out a frame header automatically. Because beginning programmers are not expected to know the minute details of the Ethernet frame format, functionality will be built into the SurfNet interface to facilitate the automatic creation of headers.

3 It *may* suffer from data corruption, however, if used over a datagram-based NetLink.

6 SurfNet: Single Threaded

The following is a much more detailed elaboration of how the data layer of the SurfNet simulator works in a single-threaded simulator.

6.1 Establishing a NetLink

Generically speaking, NetLinks are links whose communications are based entirely on Unix file handles. Sending packets translates into write() system calls, and receiving packets results in read() system calls. This design leverages the Unix design philosophy that "everything is a file"; indeed, future SurfNets may employ NetLinks that use the ethertap or tun/tap devices to bridge a SurfNet with a real Ethernet.

To keep things simple for now, however, a NetLink can be formed only on top of sockets. This means that we can use stream or datagram semantics with either Internet or Unix sockets; these three options alone give us quite a bit of flexibility.

Because stream sockets in Unix are modeled on a client-server environment, the Link in one of the SurfNets must be designated as a server that listens for connections from other simulators. The Link in the other SurfNet simulator must be configured to connect to another machine.

6.1.1 Establishing Stream Server Links

Server-side stream links generally follow this pattern:

1. Wait for a connection.
2. When one is received, attach its file descriptors to this Link.
3. If the connection drops, go to (1).

Unfortunately, step 1 is a tad complicated in a single-threaded program where I/O operations cannot block for long periods of time. Certainly, we don't want the entire simulator to grind to a halt because an I/O operation is pending; this requires the use of non-blocking I/O. Thus, step 1 is a bit more involved:

1. Establish a non-blocking socket on the port that we're using for this server-side stream link.
2. Call listen(); if there is a connection immediately available, skip to step 2 above.
3. Otherwise, there is nothing out there yet. Set up an I/O Poller read handler for this listening socket; as soon as something tries to connect, our handler will be called and we can skip to step 2 above.

6.1.2 Establishing Stream Client Links

Client-side stream links operate thusly:

1. Try to connect to the Stream Server Link.
2. When a connection is made, assign the socket's file descriptors to this Link.
3. If the connection is broken, return to step (1).

Just as before, non-blocking I/O must be used here. Step 1 is composed of:

1. Establish a non-blocking socket.
2. Try to connect to the other endpoint that we were given; if a connection is made, skip to step 2 above.
3. Otherwise, there is nothing out there. Set up an I/O Poller read handler for this listening socket; as soon we can connect, our handler will be called and we can skip to step 2 above.

6.1.3 Establishing Datagram Links

Datagram sockets, unlike stream sockets, have no notion of a client and a server. Instead, they operate more in a peer-to-peer manner—data packets are sent from one end of a socket to the other. Such a configuration is quite beneficial to NetLinks, because very little setup work has to be done. Unfortunately, there is a downside when using Internet sockets—datagrams are unreliable. (They are reliable when going through Unix domain sockets.) This makes them a good choice when linking SurfNet simulators by domain socket on a Unix machine.

Datagram links are set up in this fashion:

1. The datagram socket is created and bound to some port.
2. The socket's file descriptors are attached to the Link.
3. The Link is now ready for frame transport; outgoing frames are sent to the machine on the other end. Hopefully, there is a machine waiting for datagrams.

6.2 Sending a Frame to a Node Outside of the Simulator

1. An interested party instructs an interface to send some data, along with the type of the data, to a particular address.
2. The interface splits up the payload into 1,500-byte chunks, grows the appropriate headers around the chunk(s), and then feeds them all to a connected NetLink object.
3. The NetLink enqueues the frames for later processing, and registers its frame writing handler with the I/O Poller so that outgoing frames can be written when the socket is free. This handler is unregistered once there are no more frames to write.

6.3 Receiving a Frame from a Node Outside of the Simulator

1. The I/O Poller will call our link's frame handler whenever there is data to be read from the socket.
2. The NetLink attempts to read in a frame off the network socket. For each frame read, the frame handler passes the frame up to the attached interface's frame handler.
3. The interface's frame handler verifies the CRC32 of the frame. If the frame is invalid, it is dropped. If the frame is not destined for this interface (and the interface is not in promiscuous mode), the frame is dropped. Otherwise, the frame type is read, and the frame is dispatched to higher level protocol handlers.

6.4 Establishing a MemLink

The current MemLink design runs entirely in memory. Thus, a special hack has to be made to the I/O Poller to recognize and process these queues. Should an even cleaner (but less efficient) mechanism be desired, MemLinks could be replaced with a piped NetLink, thus leveraging the existing Unix file I/O system. However, zero-copy networking seemed to be a tantalizing goal. Here is a description of the MemLink creation process:

1. Two interfaces are created.
2. A link is created
3. The link is attached to both interfaces.
4. The link registers itself with the I/O Poller so that incoming frames may be dispatched appropriately.

6.5 Sending a Frame to a Node Within the Simulator

1. An interested party instructs an interface to send some data, along with the type of the data, to a particular address.
2. The interface splits up the payload into 1,500-byte chunks[4], grows the appropriate headers around the chunk(s), and then feeds them all to a connected MemLink object.
3. The MemLink enqueues the frames for later processing.

6.6 Receiving a Frame from a Node Within the Simulator

1. The I/O Poller asks each registered MemLink to examine its queue. If there are any frames to process, then the MemLink's frame handler is dispatched to deal with the frame.
2. For each frame in the MemLink's queue, the frame handler passes the frame up to the attached interface's frame handler.
3. The interface's frame handler verifies the CRC32 of the frame. If the frame is invalid, it is dropped. If the frame is not destined for this interface (and the interface is not in promiscuous mode), the frame is dropped. Otherwise, the frame type is read, and the frame is dispatched to higher level protocol handlers.

6.7 Tearing Down a Link

This is what happens when one wants to detach a link from an interface. Note that this is different from an interface being disconnected in that link tear down is used only when one wants to have a node communicate over a completely different communications medium.

- For a MemLink, the other end is told to detach.
- For a NetLink, the associated file handle is closed.

6.8 Handling Link Tear Down

When a link is being torn down, the associated interface shall dissociate itself from the link. Subsequent requests to send a frame will return an error; unsent frames will be dropped.

4 Note that the Linux lo(oopback) device has an MTU of 16,436 bytes, not 1,500. Solaris uses a 8,232 byte MTU for its lo0 device.

6.9 I/O Poller

The I/O Poller acts as a central nexus of network activity inside SurfNet. It is a tight loop that waits for read availability on incoming file handles, write availability on outgoing file handles, or incoming frames on a MemLink.

To be notified of read availability on a file handle, register the file handle in question and a callback that will read data from the file handle with the I/O Poller.

To be notified of write availability on a file handle, register the file handle in question and a callback that will write data to the file handle with the I/O Poller.

MemLinks need only register a pointer to themselves with the I/O Poller.

This is how the I/O Poller's main loop is supposed to work:

1. The set of file handles that we want to read from is assembled using the macros outlined in the select(2) man page.
2. The set of file handles that we want to write to is assembled using the macros outlined in the select(2) man page.
3. select() is called with the two sets of file handles and a timeout of user-configurable duration.
4. For all file handles that are ready, the associated handlers are called.
5. Each MemLink registered with the I/O Poller is polled for waiting frames. If there are any, the appropriate handlers are dispatched.

6.10 Setting up a Hub

As stated in the terminology section, hubs can use special Server Stream Links to handle a large number of incoming connections from other network-based links. Interfaces to memory links should be directly attached to the hub; details for the NetLinks follow.

6.10.1 Stream Links

Server-side stream links for hubs generally follow this pattern:

1. Wait for a connection.
2. When one is received, create a new link and interface.
3. Attach the socket's file descriptors to the link, attach the link to the interface, and the interface to the hub.
4. Go to step 1.

When a link is disconnected, it should terminate itself and the associated node. The technical details of step 1 mirror the second list of second part of Chapter 6.1.1.

6.10.2 Datagram Links

Datagram links for hubs are set up in this fashion:

1. The datagram socket is created and bound to some port.
2. A node is created, attached to this link, and attached to the hub.
3. When a frame comes in, pass it up to the interface's frame handler, no matter where the frame came from. The frame is then passed up to the hub for redistribution.

6.11 Simple Frame Forwarding

A hub is an example of a simple frame forwarder—given a frame coming in on a link, the hub simply blasts the frame out on all other links connected to the hub. For that reason, one does not want to use simple hubs on any network that has loops—messages will flood the network and eventually cause it to crash.

A simple hub operates rather differently from a normal network stack. First, the hub itself does not have a name—frames go through it, not to it. Furthermore, the interfaces that are attached to the hub do not need addresses either. Note that the hub does not discriminate among any of its attached interfaces.

The basics of hub operation:

1. One of the interfaces attached to this hub calls the hub's incoming frame handler.
2. The frame is written out via all other interfaces attached to the hub.

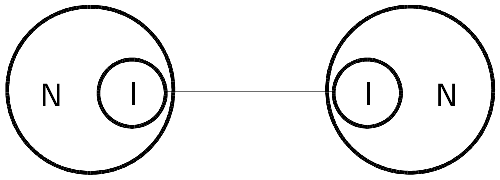
7 SurfNet: Multithreaded

This section hasn't been written yet. It has been fleshed out in the author's mind and notebook; he says "Just imagine the single-threaded SurfNet, only with no I/O Poller, as all I/O is done in separate threads which are allowed to block. Add a lot more synchronization primitives around everything." Personally, he thinks a massively multithreaded simulator could be a lot of fun to build. Or something to give him prematurely gray hair and indigestion. Then he will be able to audition for Pepcid commercials. Hoo-ray.

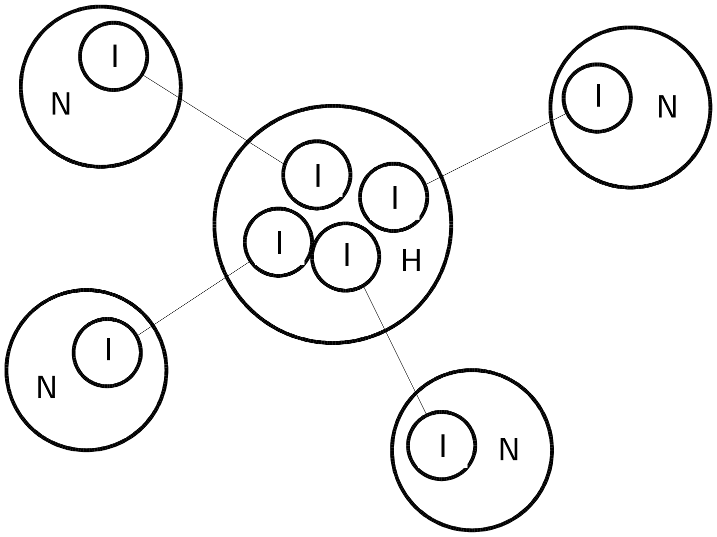
8 Sample Network Topologies

Here are some sample network topologies, just in case anyone is interested:

8.1 Peer to Peer Network

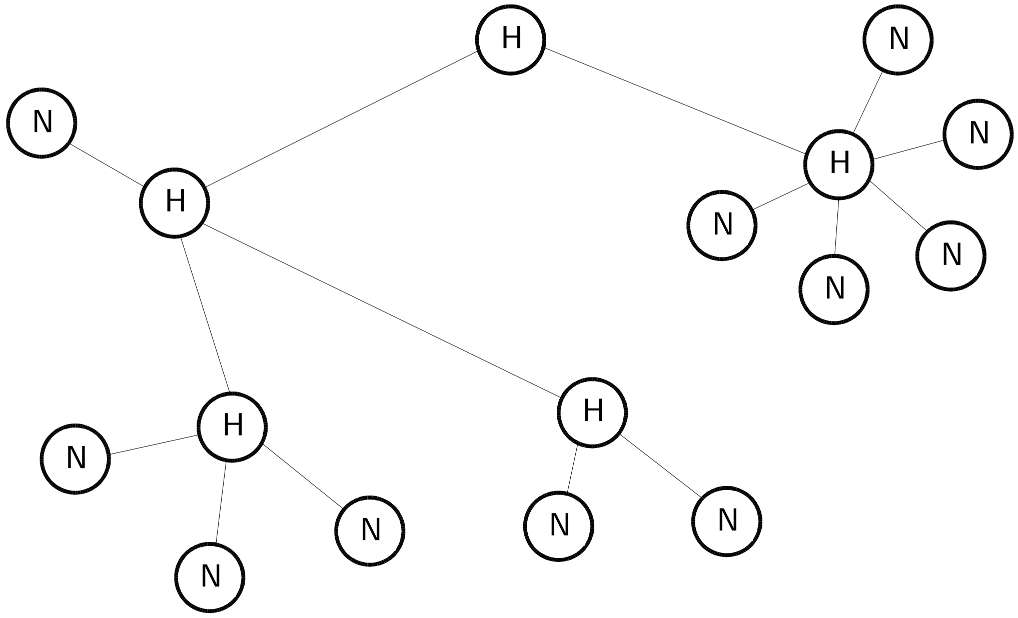


8.2 Simple Workgroup



8.3 Big Monster

(Interfaces are omitted to avoid clutter.)



9 Future Directions

The parts of SurfNet described within this document represent roughly what goes on inside the kernel-mode networking stack of a real operating system. However, it would be useful for outside programs to be able to access the SurfNet simulator, both for automated testing purposes and also so that SurfNet's client programs do not need to be integrated with the rest of the machine. Therefore, a future direction of the SurfNet project will be to construct some sort of RPC mechanism corresponding to the system call mechanism of a regular OS.

NetLinks between simulators will not be implemented until a suitable synchronization method for distributed systems is found.