Project 3: Networks and Distributed Systems

The final assignment is to experiment with networking. We will provide you with some low-level network communications facilities; you will build a nicer abstraction on top of those, and then use that abstraction in building a distributed chat program.

Each node in the network will be implemented as a separate instance of Nachos. Therefore you will have one JVM running for each network node; these JVMs will be running on the same actual machine, even though we are pretending as though they are distributed on a network.

Each Nachos "node" has a single connection to the network, which is implemented using UDP sockets. The class `nachos.machine.NetworkLink` provides this functionality for you. Each node has a unique **link address** which represents the physical network address of that node. You can access the network link using the method `Machine.networkLink()`. Calling `getLinkAddress()` on the `NetworkLink` object will return the link address of this node; the type of the address is just an `int`. Since multiple Nachos nodes will be running on the same actual machine, you will need to use **different swap files for each node** to avoid interference. An easy way to do this is to append the network link address to the swap filename.

You can test out the basic network functionality by building in the `proj3` directory and simultaneously running `nachos` in two different windows *on the same machine*. As Nachos initializes, the first line of text contains in it the address of its network link. For example, if you see:

```
nachos 5.0j initializing... config interrupt timer processor console
network(1) user-check grader
```

this indicates that the network link address is `1` (the value in parenthesis after `network` in the above output).

The new machine files for this assignment include:

- `machine/NetworkLink.java` - emulation of the physical network hardware. The network interface is similar to that of the console, except that the transmission unit is a packet rather than a character. The network provides ordered, unreliable transmission of limited size packets between nodes. This class provides `send()` and `receive()` messages to send and receive packets on the network. Since this class emulates a network device driver, you must provide mechanisms which guard access to the network link and ensure that it is used properly. The network link generates interrupts when packets are received and after a packet is transmitted; you will use these interrupts to implement higher-level communication mechanisms over the low-level network link interface.
- `machine/Packet.java` - a network transmission unit. The maximum size for a packet is given by the constant `machine.Packet.maxPacketLength`, which happens to be 32 bytes. Each packet contains a 4-byte header (see `Packet.java` for details) and a 28-byte payload. Use the constant `machine.Packet.maxContentsLength` to refer to the size of the payload in your code (that is, do not hard-code in a value of 28 bytes).

The new kernel files for this assignment, found in the `network` directory, include:

- `network/NetKernel.java` - a kernel that supports networking; extends `VMKernel`.
- `network/NetProcess.java` - a process that supports networking syscalls; extends `VMProcess`.
- `network/PostOffice.java` - a simple messaging abstraction modeled after a "post office", implemented on top of the network link. This provides synchronized delivery and receipt of messages to/from specific ports (mailboxes). There are multiple mailboxes per machine.

`PostOffice` provides a more convenient communications abstraction than the raw network link. It provides user-to-user communication on top of the network link's node-to-node communication, but does not provide reliable message delivery. In this project you will have to implement other layers on top of the `PostOffice` and network link abstractions. For example, you will need to implement reliable delivery on top of the underlying unreliable communications link. Since the `PostOffice` is not entirely suitable for this project, you are free to use it as a starting point, but you are also welcome to start from scratch if you wish.

Note that `NetKernel` extends `VMKernel` and `NetProcess` extends `VMProcess`, so modify the classes to extend `UserKernel` and `UserProcess` instead, so that your code depends upon project 2. In order for this to work, ensure that the total number of physical memory pages is large enough to hold the entire code for your test programs. This is controlled by the line

```
Processor.numPhysPages = 16
```

in the `nachos.conf` file.

I. (75%) Implement the two networking syscalls (`connect()` and `accept()`, documented in `syscall.h`), which provide reliable, connection-oriented, byte-stream communication between connection endpoints. A connection endpoint is a link address combined with a port number. A user program connects to a remote node by calling `connect()` with a remote network link ID (also called a host ID) and a port number. The remote node must call the `accept()` system call to accept the connection; the system call takes a local port number on which connections can be made.

   When the connection is established, each system call returns a new file descriptor representing the connection (in UNIX terminology, this file descriptor is called a "socket"). The application can then send messages to the network by calling `write()` on the socket, and can read messages from the network by calling `read()` on the socket. The connection is closed by one end of the connection performing a `close()` operation on the socket file descriptor. After a connection is closed, any `read()` or `write()` calls on the socket (on **either end** of the connection!) should return -1, indicating an error.

   There is one important detail to consider here. If a program does something like:

```
write(socket, buf, 100);
close(socket);
```

then although the connection has been closed, the 100 bytes of data have probably not been acknowledged yet by the remote host. Your implementation **must ensure** that all data written to a socket is delivered to the remote host, even if the socket is closed before all outgoing data is acknowledged. This also means that any `read` calls on the socket by the remote host must succeed until all of pending data has been read, even though the socket has been closed. This means that you cannot simply "drop" the state for a socket when it is closed; you have to keep the socket "alive" until all of the data is delivered!

Note that the `close()` call given above **should not block waiting for all outgoing data to be transmitted**; the `close()` should return immediately and the network layer should handle transmitting the remaining data.

Your implementation of `read()` and `write()` on sockets must provide **full-duplex, reliable, byte-stream communication with no message size limits**. This means that packets can flow in both directions on the connection simultaneously. Also, if one node calls `write()` with a buffer of size N, then the remote host will receive exactly the same data by calling `read()` (although multiple calls to `read()` might be required). The `read()` call may not return all of the data at once. For example, if node A calls

```
write(socket, buffer, 100);  /* Sender */
```

and node B calls

```
read(socket, buffer, 10); /* Receiver */
```

then B will have to call `read()` ten times before receiving all of the data. Also, `read()` may return less data than the user requested. For example, if B calls

```
read(socket, buffer, 1000); /* Receiver */
```

then if only 100 bytes are available, `read()` will return only those 100 bytes. In general the amount of data returned by each call to `read` need not be equal to the corresponding `write()` call on the remote host. Also, `read()` does **not** block waiting for data to be received on the network; it returns immediately (with a return value of 0).

You are free to use the `PostOffice` class as a starting point for your implementation, or you can start from scratch on top of the network link. It's best that you not modify `PostOffice` itself, but rather you should implement your own communications interface which makes use of `PostOffice` (if you wish).

As you design your implementation, think about all of the possible cases of failure in the underlying network layer. The underlying network link may **drop** arbitrary packets, but it may not reorder packets or corrupt their contents. (Hint: This will make your job a lot easier!)

To test your implementation, the Nachos network link can be made to randomly drop packets by modifying the value of the `NetworkLink.reliability` key in `nachos.conf`. This is a floating-point number, between 0 and 1, representing the likelihood that a packet will be successfully delivered.

- Your implementation should continue to operate well with a drop rate as high as 10% (i.e. a reliability of 0.9). By "operate well", we mean that the actual data rate delivered to an application should not degrade catastrophically if the drop rate is 10% or less.
- Valid port numbers are between 0 and 127 inclusive. You may support a greater range of ports if you wish, but negative port numbers should not be allowed. Ports are specified as the C type `int` in `syscall.h`.
- Note that it is possible for more than one program on a given node to accept a connection on the same port. For example, program A might do:

  ```
  accept(100);
  ```

  and accept a connection from a remote node, and then program B (on the same node as program A) can do:

  ```
  accept(100);
  ```

  to accept another connection from a remote node. Both A and B are then receiving data on port 100. Your implementation must deal with this case.

- You will need to assign a local port number to connections established using `connect()` -- this port number is not necessarily the same as the port on the remote node. It is possible for the local port number for an outgoing connection (established using `connect()`) to be the same as that of another existing incoming connection (established using `accept()`). (If you do not understand why this is the case, you need to think through the protocol specification in more detail!)
- `syscall.h` contains a small **typo**. The `read()` system call should return the number of bytes read, 0 if no bytes are available, or -1 if there is an error or if the connection has been closed. If the socket has been closed by the remote host, `read()` should return any previously-received data, and return -1 after all received data has been read by the user.
- You should use a sliding window protocol, with a fixed window size of 16 packets. This means that each connection maintains a "credit count", which is the number of **data** packets that can be sent before an acknowledgment is received. (By "packet", here we are referring to the underlying network link packets, which have a maximum length as described above.) When sending a **data** packet, a node first waits for the credit count to become nonzero, and then decrements the credit count. An acknowledgment from the remote host increments the credit count.

  Since connections are full duplex you need a separate sliding window for each direction of communication. Also, each connection should maintain its own sliding windows. This means that if there are multiple connections to the same

port, up to 16 unacknowledged packets may be sent on each connection (in each direction) at a time.

- Each unidirectional flow of data on a connection requires a distinct send buffer on the sender side and a distinct receive buffer on the receiver side. Each send buffer can grow arbitrarily large, but each receive buffer may hold no more than 16 packets.
- You should not implement the protocol using one (or more) threads per socket. Rather, you can use a small set of threads shared by all sockets -- for example, one thread for sending data, another for receiving data, and another to implement timeouts.
- If a packet arrives at a receiver and its receive buffer for the connection is already full, the receiver should drop the packet without acknowledging it.
- Implement byte-stream communication. This means that sending 2 bytes with one call to write is functionally equivalent to sending the same 2 bytes with two separate calls to write (one call for each byte).
- The read/write syscalls should not wait for packets from a remote host. If there is no data available to be read on a port, then `read` returns immediately. `write` queues up the data on the port and then returns without waiting for acknowledgment. (However, `connect` will block until a connection can be established.)

II. (25%) Demonstrate your message passing primitives work by using them to implement an IRC-like "network chat" application among multiple (at least 3) users. Network chat is like the UNIX talk utility, except it can be used for N-way conversations (not just 2-way); each user sits at a different machine, and anything anyone types is broadcast to everyone else connected to the chat room.

You should write two programs: a chat server (called `chatserver.c`) and a chat client (called `chat.c`). The server runs on one node and accepts connections from chat clients. The chat client connects to the chat server and accepts user input from the console. After the user types a line of text, the line is transmitted to the chat server. The server then broadcasts the message to all of the clients connected to it. Each message received by the chat client should then be displayed on the console. Users can dynamically connect and disconnect from the chat room.

- The chat client program `chat.c` should take one command-line argument: the network link address of the machine running the chat server.
- Your chatserver must use port 15; all chat clients connect to the same port on the server.
- Chat clients can connect to the server at any time and disconnect at any time. Your server must handle clients entering and leaving the chat room at any point during the conversation.
- All users should have a consistent view of the messages being sent and received. This means that each individual message must be identical on each client. It is acceptable for messages to appear on different clients in different orders as long

as the sequence of messages **from a particular client** appears in the order it was typed. The interleaving of messages from different clients may vary, however.

- The chat client must only display **complete messages** as typed on other clients. It should not display a partial message (i.e. less than one line of text). Also, the chat client must avoid user input from the console being interleaved with messages received from the chat server. A simple way to do this is to have the chat client not display data from the chat server while the user is typing a line.
- The chat client must continue to display messages typed by other chat room users even if no input is read from the console. This means that you may not block waiting for data to be read from the console before displaying output from other clients. (To keep the display clear it is OK if the chat client blocks waiting for a line to be read after the user has started typing the line.)
- The chat client must exit gracefully upon receiving user input of a single period (".") at the beginning of a line. The chat server should exit upon receiving any data on standard input.