# Message Passing Library

# CS 415

Programming Project 4 - Design Phase

**Due: 2 April 2008**

Remember to check the function comments in provided header files—these define more precisely the interfaces you will be implementing. As always, some behaviors may not be defined as well as they need to be, and so you will need clarify the behavior before completing your design.

The design document will be due at the date above. Please fully document the following interfaces:

**minimsg**

**directory**

Network.h contains the new virtual hardware interface that you may make use of. Also note that functions in a private header may be treated as utility functions.

Remember that problem categories should be the very level categories of things that can go WRONG. It is NOT a list of the specifications (though it is somewhat related to the inverse of these). The goal is to provide a few categories into which all bugs can be grouped—it is an abstraction to make reasoning and communicating about tests easier.

## Overview

Message passing is a form of interprocess communication. In this project we will be creating a message passing library to allow for communication between processes, and since each minisystem application has a single process, this will be inter-application communication as well.

Our message passing system will be in the spirit of Mach message passing, and as such mailboxes will be called *ports*. Also drawing from Mach, our message passing will be machine agnostic, such that interprocess communication will work the same even if the other process is on a different machine.

Like the alarm system, the minimsg system will started up when the minisystem starts and shut down when the minisystem exits. The minimsg library will allow messages to be sent from one port to another. Messages will be queued at the receiving port (mailbox) until receive is called on that port. If receive is called on a port with an empty queue, it will block until a new message comes in.

The third main operation provided by the minimsg interface is *rpc* or remote procedure call, when allows location independent access to services. It simply sends a request message to the destination port, and then immediately waits to receive a response from that same port. Note the semantics of the request and the reply are only defined at the application level—to minimsg, it is all just binary data.

To keep things simple, and in line with the rest of minisystem, we will not be enforcing any ownership or protection privileges on ports. However, applications are encouraged to use ports in a way such that the creating thread is considered to be the owner of the port and is thus the only thread that calls receive on the port.

## Directory

The minimsg system must keep track of all local ports, or ports which are created by it. Since ports are referred to by IDs (which are integers), we need a directory data structure that maps local ports IDs to the data structure which implements that local port, holding the receive queue and any other necessary data. Also, this lets us determine whether or not the port referenced by a particular ID is a local port—if we look up the ID in the directory and it is not found, then we know that it is not a local port (and thus we don't have to worry about receiving a message addressed to that port).

We will implement a directory library to use for this purpose. It will provide operations for adding a (key,value) pair, finding a value based on its key, and deleting a (key,value) pair based on the key.

Our directory will treat (key,value) pairs as representing a one-to-one mapping, so adding a new (key,value) pair for which the key already exists in the directory will cause the old entry to updated with the new value.

The network interrupt handler will need to frequently determine if port IDs on received messages are local ports and so the get operation must be as efficient as possible.

## Network

To enable our message passing system to work across machines, the implementation of the message passing system must make use of a virtual network interface. You will have to multiplex this single virtual network interface among the many local ports, each of which may try to send to a port on another machine.

The network interface must be started just like the clock interface in the last project, and at this time a network interrupt handler function is set. This function is called whenever a network interrupt is received. Network interrupts are generated when a new message is received by the network interface. While interrupts are disabled, network interrupts are queued, so you don't have to worry about missing network interrupts (though you should still always receive and deal with network interrupts as soon as possible).

The network interface also provides a broadcast_message function which sends the given message to all hosts on the network. It also provides various functions for dealing with network addresses and a send_message function which sends a message to a specific host, as specified by address.

When sending minimsg messages over the network, will need to prepend a header to the minimsg message data so that the receiving minimsg system will be able to determine what should be done with the message.

While it would be more efficient to use directed sends, it will be much easier to implement using network broadcasts. Since our systems will be used on small local networks, this will work fine for us. However if you have time and already have your system working, you can try to refine the use of the network interface so that when possible directed messages are sent.

# Details

Each port should have two associated queues—a receive queue and a send queue.

The send queue is necessary to implement reliable delivery as well as ordering for same sender port to same destination port sends. After a message is sent, it must be kept around until a received acknowledgement is sent by the receiver. If TIMEOUT amount of time elapses and an acknowledgement has not been received, then you will resend the message. Only after a message has been acknowledged can you send another message to the same port, so sometimes messages must be queued up for sending.

Also, you will be multiplexing a single piece of network hardware among many threads / ports, so sometimes a send must be queued because the hardware is currently unavailable. It would be possible to make send a blocking call, in which case you could have send block instead of maintaining an internal queue. However, to get some practice writing asynchronous calls, we will be writing send such that it never blocks (although receive does block).

In general, think about the power of the message passing paradigm, and try to understand which features of previous abstractions are subsumed by this abstraction. In the code portion we will get a chance to implement a program using message passing.

Also try to start working a design pattern for multiplexing an asynchronous piece of hardware (one for which you issue commands and then later hear the results via an interrupt. This important because a disk controller is a more complex piece of hardware, but works in this same way.