# Message Passing Library CS 415

Programming Project 4 - Code Phase **Due: 21 April 2008**

Make sure to download the new provided project base code. In particular, you should use the data structures and utility functions defined in `minimsg.c`. There were also a few interface changes for Directory and Minimsg (details in a later section of this document).

Required deliverables:

- working **minimsg** subsystem
- working **directory** library
- **test_directory.c**, conforming to our library test requirements, which are clearly stated in the comments of this file
- **app_mp_elevator.c**, a version of your elevator application in which all communication is via message passing rather than semaphores.

To clarify some issues of terminology: throughout this project, a port is merely a mailbox identifier, as it is used in Mach message passing systems (for those with some networking experience, this term port has absolutely nothing to do with TCP or UDP ports).

For minisystem, each running application runs on a unique virtual machine, so each terminal window open represents a unique system. The slightly weird aspect of minisystem is that each system always has a single process, so all inter-process communication is also inter-system communication. Thus communicating with other processes and thus other virtual machines, even when on the same physical machine, requires the use of the virtual network interface.

## Getting Started

Backup your working project 3. Copy in the new base files to your current code directory. A few base files, such as `interrupts.c` and `Makefile`, were updated, so make sure that these new files overwrite their older versions.

A new `minithread_private.h`, should be included in minithread.c instead of including `minithread.h` there. `alarm.c` and `minimsg.c` should also include `minithread_private.h`. This gives us a standard way for tying together minisystem subcomponents.

An updated `alarm_private.h` is also included. It mirrors the pattern found in `minimsg_private.h`, and completes the previous goal of having a uniform subcomponent interface. It also allows the minithread system to know when there are registered alarms remaining, so that it does not exit in the case that there are no threads but there are alarms.

You will first want to get your Directory working, if you have not already. Fill in `test_directory.c` and be confident that your directory works correctly before moving on.

## Message Passing

Now that you know all libraries relied on by minimsg are correct, start implementing this library. First implement this library such that it works correctly in a single system. It should correctly run app_mp_buffer, which is the same bounded buffer example as before, with the exception that it uses message passing instead of a memory buffer and semaphores. If your message passing is correct, there should be no noticeable difference in execution speed from the normal app_buffer.

Next, implement the network layer of minimsg so that messages can be passed between different minisystems. To test this, set the `INTER_PROCESS` define at the top of `app_mp_buffer.c` to `1`. When this is set, the app will again first run the single process bounded buffer example, but will then wait for a second process. Once found, it will run the same bounded buffer example, but between 2 separate systems. This inter-process bounded buffer will run more slowly than before. The virtual network interface is unfortunately quite slow.

## MP Elevator

Now that you have message passing working, you will adapt your elevator application to use message passing rather than semaphores and shared memory for all inter-thread communication. Copy your old `app_elevator.c` into a new file called `app_mp_elevator.c` for this application. (**Important**! make sure your project directory still contains app_elevator.c, as this semaphore version will be compared against the new message passing version when graded).

First get `app_mp_elevator.c` working in a single process, which will be much easier as threads can use global variables to easily find the correct port ID to use as the destination for sends. It will be helpful to first determine a pattern you can use to replace certain semaphore and / or shared memory operations with certain message passing operations, and then merely apply this pattern to all inter-thread communication. As with the first version of the elevator application, this will be easiest if you map some concrete aspect onto each port object.

Finally, once app_mp_elevator is working within a single process, back up this version and begin work on representing each person and the elevator as a unique process. (**Important**! Only start this after getting app_mp_elevator working in a single process). The only change from the previous message passing version of the elevator will be determining which process represents which person or the elevator, and then sharing port ID's for all of the used ports.

It is acceptable to have the elevator process coordinate this determination process. For example, the process with the lowest system port ID could become the elevator process, and then it would send messages to all the other processes, telling them which person they were representing.

# Interface Updates

- `directory_iterate` – this function now takes a key as an argument in addition to the value, so as to have all the necessary data to correctly call `PFmap` functions
- `minimsg_system_port` – this new interface function provides a means of retrieving the port id for a system's default port. This port is created when the system is created and cannot be destroyed until the system itself is. This port is special in that it receives messages sent to its unique port ID as well as messages sent to `MINIMSG_SYSTEM_PORT_BCAST_ID`.
- `minimsg_send` – this function now takes a message ID `response` parameter which is used to specify which RPC query is being responded to. For normal sends, this is merely set to `MINIMSG_UNDEFINED`.
- `minimsg_receive` – this function now takes a pointer to a port ID, `from_p`, and and pointer to a message ID, `id_p`. Both are allowed to be NULL and if so are simply ignored. If not, they are used to return out additional information about the received message.

# Base Code Behaviors

- `network_interrupt_arg_t` – a structure of this type is passed to the network interrupt handler. Remember that it is the network interrupt handler's responsibility to free this memory when done with it.
- `network_reserve_next_token` – use this function to get a new globally unique port ID
- `MINIMSG_GROUP_ID` – this define must be set to a unique value for each group. Use it to fill in the `system_id` field of `minimsg_net_header`, and then check all received packets to make sure they belong to your group before further processing. Otherwise your system would be confused by broadcasts from other groups.
- `MINIMSG_IN_CSUG` – defining this to be 1 configures the virtual network interface to broadcast to all computers in CSUG. When set to 0, the virtual network interface only broadcasts to other virtual machines on the same physical machine. If your code works in one of these modes, then it will definitely work in the other mode, so leave this set to 0 until you have your system working (no sense in overwhelming the real network while debugging). However, do try running your software on multiple computers after you get it working.

# Minimsg Design

### Minimsg Layer

This layer contains the external interface and any message passing system level state. Once inside this layer, we generally package up a message into a `minimsg_msg` object and then pass this object down through other layers to accomplish the needed operation.

### Msg Object

This object has fields for all the needed message attributes. For simplicity, we also use the same structure to format data we send over the network. All groups use the same definition for this structure, so if 2 groups correctly implement the rest of the message passing library, they will be able to set their `MINIMSG_GROUP_ID`s to the same value and then pass messages between

between their systems. Try this with app_mp_buffer in INTERPROCESS mode, with one process being your code and the other process being another group's code.

### Mbox Layer

This layer holds the details of individual mailboxes. It uses objects from the corresp layer as proxies for the ports it corresponds with. Other objects use its deliver function to deliver a message to this mbox object.

### Corresp Layer

This layer holds objects which represent the correspondents of a mailbox, and as such these objects are unique to a particular mailbox. A corresp can represent a port that is in the same system or a port in a different system, so the above layers need not know about the network. A mailbox asks corresp's it owns to send messages. The corresp then asks the destination corresp to deliver the message in the case that destination is local, or asks the net layer to deliver the message over the network to the destination corresp in the case that it is in another system. Get the local case working first before worrying with the network case.

When delivering normal messages, the *corresp* object just asks its parent *mbox* object to deliver the message. RPC response messages are accessed by sender, so *corresp* objects are the final holding place for these messages until they are received.

One detail worth noting – the default system port should maintain a corresp object representing the system port broadcast ID. This corresp object will naturally be used for messages sent to the broadcast system port ID, but you should also take care to make sure this corresp object also receives messages addressed to this special ID (rather than the corresp object representing the true sender of the message). Otherwise, sending a broadcast message and then sending a message from the same port to directly to a system port will look like a duplicate message when it actually is a new message.

### Net Layer

The net layer uses the virtual network interface to provide a reliable, ordered tunnel for communicating a *msg* object between 2 *corresp* objects on different machines. It makes use of state stored in corresp objects.