# CSci 4061 Assignment 3: Message Passing

**Due: April 11th 2016 at 11:55pm**, you may work in a group of 2 or 3.

## 1. Purpose:

In this assignment, you will become more familiar with the another type of IPC: message-passing via message queue/mailbox and the use signals. In addition, you will also learn about interrupt-driven programming (via alarm and I/O signals).

## 2. Background:

In this assignment, you will simulate the basic mechanism of TCP (Transmission Control Protocol) protocol that is commonly used for sending data through the Internet. TCP enables two processes running on any Internet-connected machines to exchange data. In this assignment we will only simulate the way the protocol is used to exchange data between different processes on a single machine. We will also only implement a very simple version of TCP-style message-passing without many of its core features.

### 2.1. Packets

In general, the data that is sent through a packet-switched network is broken down into one or more ***packet****(s)* depending on the data's size and the packet's size (the packet size is fixed). Each packet consists of some header fields and data. The header fields may consist of information such as the sender's identity, packet number, length of data encapsulated in the packet, etc., while the data stored is the actual content of the message. Figure 1 shows an example of a single packet.

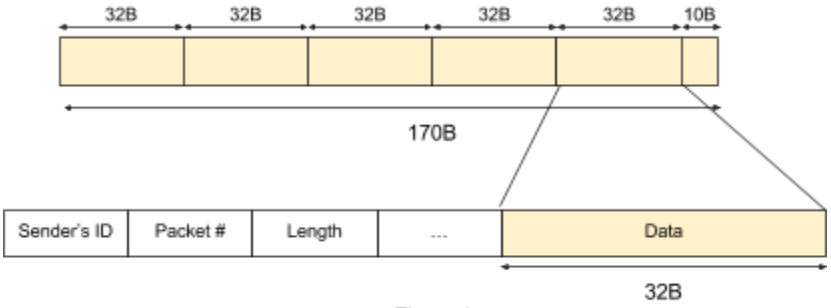



Figure 1

In this specific example, the message is the top figure. Below this depicts a single packet (#5) of the message. The maximum size of data that can be stored in a single packet is 32B. Thus, if the total message is 170B, the sender needs to send 6 packets in total (5 containing 32B of the message and 1 containing the last 10B of the message). On the receiving side, the receiver will also receive the message as a series of packets and deliver the original message by combining each individual packet's data in the correct order (#1, #2, .. #6). In the Internet (and in our lab), the packets may arrive to the receiver in an arbitrary order! While receiving each individual packet, the receiver needs to notify the sender that the packet has been successfully received, which is called an ***acknowledgement*** (**ACK**) that is also enclosed in a packet. If the sender does not receive an ACK for any of the transmitted packet after a certain amount of time, the sender will **TIMEOUT** and resend every packet whose ACK has not yet been received (i.e. acknowledged) by the receiver. This mechanism is used to handle various issues in computer networks such as packet loss (Figure 2a), corrupted data, etc.

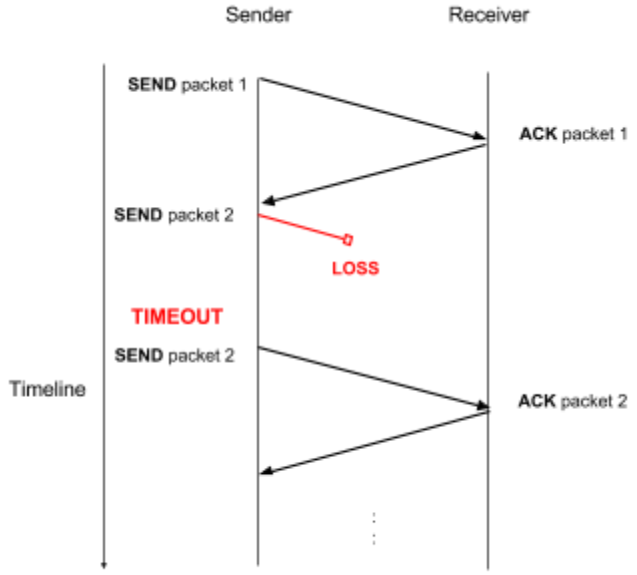



Figure 2a

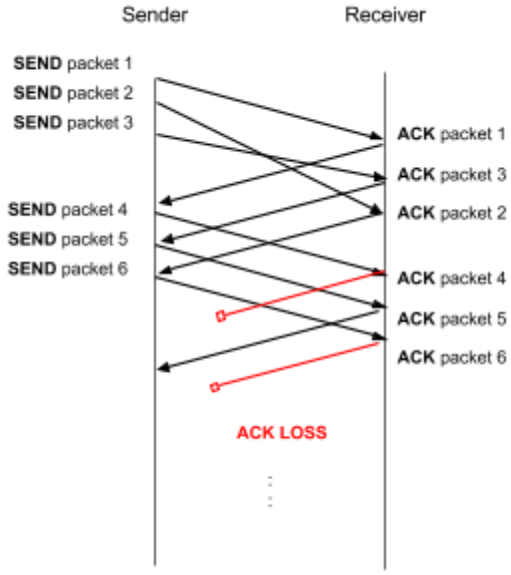



Figure 2b

### 2.2. Sliding Window

Sending a single packet at a time may result in a long total time to send the whole message if one packet must be acknowledged before the next is sent. Thus, it is desirable to send multiple (*n)* packets simultaneously and wait for multiple ACKs at a time. A group of packets that is sent together is called a ***window*** (of size n). Figure 2a has a window size = 1 and Figure 2b has a window size = 3.

While sending multiple packets at a time is desirable, this may lead to an issue where the receiver may receive the packets unordered. This may happen since any of the packets may get lost in the middle of the network or transferred via slower network path. Thus, the receiver has to be able to construct the message even if the packets come in a random order. In a ***sliding window*** protocol, the sender can send another packet whenever it receives an ACK for a sent packet instead of waiting for all ACKs to arrive. (In Figure 2b, the sender can send packet 4 right after it receives an ACK for an earlier packet, e.g. packet 1). Thus, the sliding window protocol allows the sender to have at most *n* packets in transit (i.e. not acknowledged) at a time. Initially when the window is empty, *n* packets will be sent together. After this, packets are sent when ACKs arrive to maintain the window *n*.

## 3. Description:

In this assignment, you will implement several APIs (in `process.c` and `process.h`) that will be called by an application program to exchange messages using the protocol discussed in the previous section. This program can act as a receiver, sender process, or both. We have provided an application program, `application.c`, that will use the APIs defined in `process.c`. *When we refer to sender or receiver process this is the application program operating as either a sender or receiver at that point in time.* There are 3 APIs/functions that will be called by the application program:

1. `int init(char *process_name, key_t key, int wsize, int delay, int to, int drop);`

   The init API is the initialization function that needs to be called before calling `send_message` or `receive_message`. API There are two functions performed by `init`. First, it will create a message

queue for the given key and save the application process' information (pid, process' name, and mailbox id) to a file that is shared to the other processes that may wish to communicate with it. The pid will be used to send a signal to another process, the process name will be used to name the file, and the key will be used to get the process' mailbox. The mailbox is the place where a sender process sends packets and a receiver process retrieves packets. There is a mailbox created for **each** process that calls `init`. Second, it will save the message configuration (`wsize`: window size, `delay`: maximum delay, `to`: timeout, and `drop`: drop rate) and set signal handlers (SIGIO and SIGALRM) that will be used throughout the program.

A SIGIO signal will be sent by the sender process using `kill` to indicate that a new packet has been placed into the receiver's mailbox. This will be handled by the `receive_packet(int sig)` function.

A SIGALRM signal is used to indicate a TIMEOUT. Whenever the sender process sends a packet to other process' mailbox, the sender will set a TIMEOUT and wait for an ACK from the receiver. If the ACK is not received before the TIMEOUT expires, the sender will resend any packets whose ACKs have not yet been received (this is handled by the `timeout_handler(int sig)` function).

2. `int send_message(char *receiver, char* content);`
   This API is used to send a message to another process. To send a message to another process: the sender process first has to obtain the receiver's information (pid, and mailbox key) from the receiver's file (named with the receiver's name, `receiver`), break down the message (`content`) into one or more packets, send each packet to the receiver's mailbox, and send a SIGIO signal to the receiver. The sender needs to set a TIMEOUT and wait for ACKs from the receiver to indicate whether the packets have been successfully sent.
3. `int receive_message(char *data);`
   This API is used to indicate that the program is ready to receive a message from any sender. If there is more than one sender that tries to send a message, only the message from the first sender will be read. Any other senders will TIMEOUT and cancel sending the message. In this function, your program should `pause`/block until a SIGIO message is received, indicating that there is a new packet in the mailbox. The program should save the content of the packet and send an ACK to the sender. Once all packets are received, the message will be saved to the `data` and returned.

In this assignment, you are given the application program (`application.c`) where it will call the `init` API, and perform as a sender/receiver depending on the user's input. A sender process will call the `send_message` API whereas a receiver process will call the `receive_message` API. All The tasks you need to implement are as follow:

## Part A (60 points)

### 1. Initialization

The application program ( `application.c` ) takes 6 inputs from the user:

1. Name: the process name (you can assume there is no other process with the same name)
2. Mailbox key: the key of the mailbox
3. Window size: the size of the window (you can set it to 1 for this part)
4. Max delay: maximum delay in sending an ACK (in seconds)
5. Timeout: maximum waiting time to resend a packet (in seconds)
6. Drop rate: the probability of a packet to be lost (in %)

When the program gets these parameters, it will pass these inputs to the `init` API which will save the name, mailbox key and pid to a file and set the configuration of the messaging mechanism.

The program provided in `application.c` waits for the user's input via `scanf()` to perform whether as a sender or a receiver. A sender process requires two other inputs from the user: 1. the receiver's name, 2. the message. On the other hand, a receiver process will wait/block until it receives a new message in its mailbox.

### 2. Sending a Message

A message that is provided by a user may vary in size (< MAX_SIZE). While getting a message, the sender will:

1. Break the message into one or more packets.
2. Send each packet to the receiver's mailbox using a sliding window.
3. Send a SIGIO signal, which will notify the receiver that a new packet has arrived to the mailbox.
4. Set an alarm as a TIMEOUT. If there is a TIMEOUT, resend all sent packets whose ACKs have not yet been received.
5. Wait for an ACK for each sent packet.

The basic structure of the packet is given in file `process.h`, which contains the type of packet (DATA/ACK) and the sender's name. Once all ACKs have been received, the process will wait for another user's input to determine whether to send a new message or wait for a message from another process.

### 3. Receiving a Message

When the receiver gets a SIGIO, the receiver will retrieve **all** packets sent to its mailbox and reply to each received packet with an ACK. In constructing a message, the receiver needs to know whether the packet it receives is a new packet or a duplicate (duplicates may be sent by the sender if an ACK for the packet is delayed and causes the sender to retransmit after a TIMEOUT). In addition, the receiver needs to determine whether the packet belongs to a new message, or an uncompleted message, or a duplicate for an old message. Once the message has been successfully constructed, the process will print the message to the standard output and wait for another user's input to determine whether to send a new message or wait for a message from another process.

**NOTE**: To simulate packet loss in the network, we provided a `drop_packet` function (a probability function that may drop a packet) that you **must** use when receiving a packet. In addition, we also provided a delay in responding to a packet. These two conditions are used to trigger a TIMEOUT at the sender side. There are no TIMEOUTs within the receiver process.

## Part B (30 points)

### 1. Sliding Window

In this part, you will implement a sliding window mechanism, i.e., the sender needs to ensure that *n* packets are in flight at a time, where *n* is the size of the window. Once an ACK is received, the next packet (if any) will be sent right away. As mentioned in Section 2, sending *n* packets at a time may result in an random arrival order of packets. Thus, your receiver has to be able to put the data content of each packet in the correct order in assembling the message.

**NOTE**: To simulate the random arrival order of the packets, your sender **must** use the `get_next` function that we provide. This function will return a packet number that you need to send next.

## Part C (10 points)

### 1. Error Handling & Robustness

You should handle any possible errors for:

- System call failure.
- Memory leak (you have to free any allocated memory when it is no longer needed). You may want to use Valgrind to check if your program has a memory leak problem.
- When a sender is trying to send a message to an unavailable receiver, the sender should retry on TIMEOUT until a maximum number of TIMEOUTs (`MAX_TIMEOUT` defined in `process.h`). Once the number of TIMEOUTs reaches the `MAX_TIMEOUT` threshold, the `send_message` API will return and the application program will wait for new user input.
- If there are multiple senders sending messages to the same receiver concurrently, the receiver may simply ignore the message from the later sender. This should cause the later sender to timeout.

### 2. Documentation

You must include a README containing the following information:

- Your group ID and each member's name and X500.
- How to compile and run your program.
- A brief explanation on how your program works.

In addition you should also address these questions:

- What is the effect of having a very short TIMEOUT period and a very long TIMEOUT period?
- What is the effect of having a very small window size (e.g., n=1) and a very large window size?

Within your code you should use one or two sentences to describe each function that you write. You do not need to comment every line of your code. However, you might want to comment portions of your code to increase readability. At the top of your README file, please include the following comment:

```
/* CSci4061 S2016 Assignment 3
* Name: <full name1>, <full name2>
* X500: <X500 for first name>, <X500 for second name> */
```

## 4. Simplification:

- You may assume that the process name for every process at a given time is unique.
- You may assume that the message is a **null terminated string** without any spaces at the sender side. Therefore, the message must be **null terminated** when returned to the receiver.

## 5. Miscellaneous:

- As in lab #2, we have given some complete functions, and some partial functions to help you. All data types have been provided. You are free to ignore or change of the provided code if you wish. In particular feel free to optimize.
- Useful functions: `memcpy, sigprocmask, alarm, kill, msgrcv, msgsnd, msgget, signal/sigaction, pause, getpid`
- `alarm(0)` turns off any pending alarms
- Be aware of race conditions: block signals to handle these cases as needed
- Remember to reset any global state once message-passing is done to enable new messages as needed

## 5. Test Cases:

All tests will be done in a CSE labs UNIX machines. So, you need to make sure that your program can be compiled and run using one of those machines. You may consider these following cases for testing:

- Invalid process information.
- Short/long message with small/large packet size.
- Multiple processes that exchange data between them in a sequential order (e.g.: A → B, C → A, B → A, B → C). In this scenario that application instance at runs A, B, C respectively can keep running and change whether it is a sender or receiver based on user input.
- Handle packet loss (both in sending DATA and ACK) and delay (only on ACK packet).
- Handle unordered packets.
- Handle duplicate packets (you should also consider the case where a duplicate packet is received when the message has already been completed).

## 6. Deliverables:

1. Files containing your code
2. A README file (readme and c code should indicate this is assignment 3).

All files should be compressed into a single tar file, named `assignment3_group<your group>.tar` and submitted through Moodle. This is your official submission that we will grade. We will only grade the most recent and on-time submission. Failing to follow the submission instruction may result in a 5% deduction.

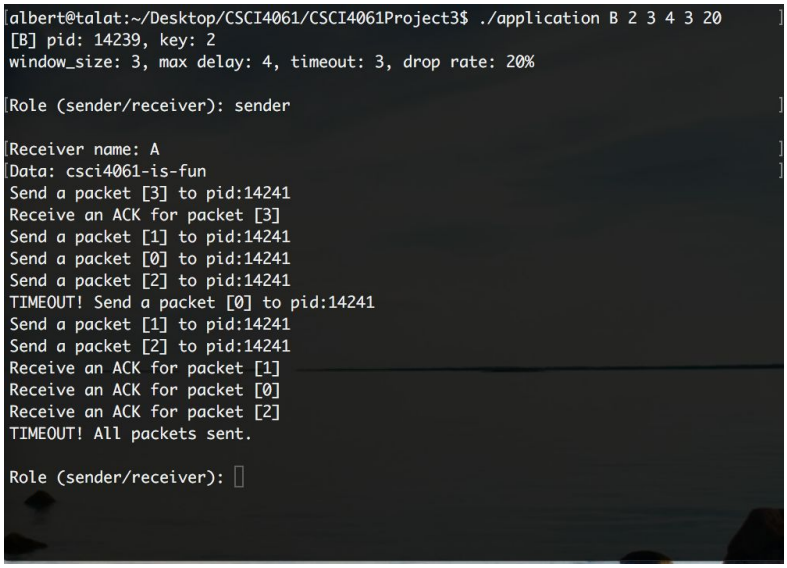
## 7. Limitations and Clarifications:

1. In this assignment, we will not simulate the case where there is a delay in sending a DATA packet to a receiver. However, either DATA or ACK packets may be lost if the drop rate > 0 (and require retransmission). NOTE: You do **not** need to change the code as the loss code been provided.
2. Regardless of the WINDOW SIZE, the first packet must be send and ACK'ed before any new packets are sent. This is called stop-and-wait. The reason is that the ACK from the receiver will contain a message_id field that determines the id of the message. All other subsequent packets for the same message have to include this message_id to tell the receiver which message the packets belongs to. The number of subsequent packets that are sent together is determined by the WINDOW SIZE.
3. Corner case: One-packet-message and ACK loss. In this specific case, unless the receiver is listening for a new message, the duplicate packet will be ignored, which may cause the sender to reach the number of TIMEOUTs and stop sending the packet.

## 8. Sample Snapshot:

The sample program below was run with a packet size = 4.

Sender:




Receiver: