Causal Multicast over a Reliable Point-to-Point Protocol

Authors:

- **Anne-Marie Bosneag**
- **Ioan Raicu**
- **Davis Ford**
- **Liqiang Zhang**

Table of Contents

Cover Page 1

Table of Contents 2

1.0 Causal Order Multicast Proposal 3
- 1.1 Causal Order Multicast Details 3
- 1.2 Issues concerning the system implementations 3

2.0 Programming Interface and Design Document 5
- 2.1 Application Layer - Design Interface 5
  - 2.1.1 Required arguments 5
  - 2.1.2 Optional arguments 5
- 2.2 Causal Order Multicast Layer 6
  - 2.2.1 Algorithms 7
    - 2.2.1.1 Logical and Vector Timestamps Algorithms 7
    - 2.2.1.2 Causal Order Algorithms 8
- 2.3 Reliable Point-to-Point Communication Layer 9

3.0 Final Report 10

4.0 Works Cited 15

1.0 Causal Order Multicast Proposal

Our project's objective is to implement a causally ordered multicast system on top of a reliable point-to-point communication layer. For a non-reliable multicast system, we only need to satisfy validity and integrity. Validity is: "if a correct process multicast a message m, then it eventually delivers m. Integrity is: "for any message, m, every correct process delivers m at most once and only if some process sent m. Our causal multicast implementation will permit unrelated messages to be delivered in any order.

1.1 Causal Order Multicast Details

On top of the point-to-point layer will be another layer which will implement causal order multicast. The point-to-point layer will take care of the integrity property. That is, any message passed up to the causal order layer will be ensured not to be a duplicate by the point-to-point layer. The validity property can be handled by the causal order layer. If we wish to multicast a message, we can immediately deliver it to ourselves without violating our algorithm.

We plan on implementing the causal order multicast using the Algorithm based on Birman et al. (1991), which is described in Figure 11.16 in the text [1]. Basically, the algorithm will exploit the use of vector timestamps. These will be implemented as integer arrays, with slots for each member of the group. Each time a message is sent, the process will increment the value for it's own timestamp in the vector and send the vector in the message header. When a process, p_i receives a message from p_j, it first places it in its holdback queue, and then checks that the following two conditions are met:

a) it has delivered any earlier messages sent by the p_j
b) it has delivered any message that process p_j had delivered at the time it multicast the message.

If these two conditions are met, the causal order layer will deliver this message to the application layer.

1.2 Issues concerning the system implementations

1. Without having fully defined all of our data structures, it is difficult to specifically decide exactly which will need to be protected in order to have thread safe code. As stated below, we will be having two queues (one for messages and one for acknowledgements) that will need to be protected. We believe it will be more safe if we define one function for each queue that is allowed to manipulate the data in the queue. We will then put a mutex lock on these functions. As far as our initial design goes, we foresee these two queues as being the only global data that will need to be protected. Therefore, threads can operate concurrently in a safe manner, but only one thread will be allowed to call these queue manipulation functions at any given time.
2. We will have a timer implemented on the sender, which will calculate the RTT based on when the ACK is received. Based on the size of the message, we can deduce an initial value for the timeout. If we don't receive an ACK within the time-out, we retransmit,

and increase the timeout value exponentially. After a certain threshold, which is also determined according to the message size, we can assume that the process has failed.

3. We have decided to initially limit our message sizes to no more than 1500 bytes to avoid fragmentation. If we see the need in the future for larger message sizes in the application, the lower layer protocols should take care of fragmentation and re-assembly for us. Message size also will affect our exponential backoff algorithm as indicated above.
4. We must first differentiate between the messages seen at the reliable point-to-point layer, and those seen at the ordering layer. For the point-to-point communication, sender id, sequence number, transmission number, and a 1 bit field differentiating between ACKs and data, will be included in the header, apart from the body of the message (the actual data to be transmitted). For the ordering layer, we will need to add the vector clock to the message. The point-to-point communication layer sees [message, vector clock] as the body of the message, to which it will append all other information needed at that layer.
5. Assuming that we use the 3-way hand-shake, each message should be deleted once an ACK is received from the client, however, if an ACK is not received, it should be deleted according to the exponential back-off algorithm. We will have a queue for the acknowledgements. The exponential backoff algorithm indicated above has a time window in which we will wait before discarding a message and considering a process has crashed. We can calculate the max value for this and then define a queue size that will safely hold all these messages. In the event that the queue becomes full, we will be able to safely discard messages in FIFO order because the exponential timeout will have expired for those messages.
6. Each ACK should be deleted once an ACK of the ACK is received from the client, however, if an ACK is not received, it should be deleted according to the exponential back-off algorithm. We will also have an ACK queue that is similar to the message queue, and will operate in the same manner (deleting messages in FIFO order).
7. We will be using FIFO ordering for the point-to-point communication layer.
8. The application will have an interface that permits the user to specify a list of recipients, thus not forcing every message to go to the entire group.
9. Each process will have unique ID defined by the socket (IP address and port number).
10. We will have 2 sockets for each process, one for sending and one for receiving.

2.0 Programming Interface and Design Document

Our project will have three layers that will be implemented by our group, which will be utilizing four other layers of the OSI model: transport layer (UDP), network layer (IP), data link layer, and the physical layer. Our first layer will be the application layer, which will involve creating the program interface that the user/client will see when using the system. The application layer will talk directly to the multicast layer, which will be implemented using causal ordering. The multicast layer will then talk to the reliable point-to-point communication layer, which will be reliant on UDP/IP, which can be found in the lower layers.

2.1 Application Layer - Design Interface

Although the specified requirements were to separate the client and the server code, we came to the conclusion that if we implemented the first part of the project as a peer-to-peer design, the amount of work needed to integrate the multicast layer with the point-to-point communication layer will be minimized. In our multicast implementation, we will have every process as being a peer of another, and thus it makes more sense to make even our point-to-point communication layer based on a peer-to-peer approach.

We are implementing the interface through command line arguments. There will be required arguments and optional arguments, which if not specified, will have default values assigned to them at run time. Each argument will be preceded by a flag, which will denote what the argument will be.

2.1.1 Required arguments

- -H <destination hostname or IP address>
    - host name or IP address of other processes where message will be sent
- -P <destination port>
    - port number where other process will be listening when sending messages
- -p <receiving port>
    - port number where process will listen for incoming messages

2.1.2 Optional arguments

- -h <print this help message>
    - lists all the required and optional arguments and their flags
    - default: no message
- -n <number of retransmissions>
    - number of retransmissions in case of failure (NO ACK received)
    - default: 5
- -t <server time delay>
    - if 1 (enabled), will prompt the user for each message, if the acknowledgement should be delayed and for how many seconds
    - default: 0 (disabled)

- -s <server sequence mod rate>
  - if 1 (enabled), will prompt the user for each message, if the sequence number should be changed, and to what new sequence number
  - default: 0 (disabled)
- -D <provides debug>
  - if 1 (enabled), will do the following:
    - print messages received
    - print whether acknowledgement is sent, dropped, delayed, or modified
    - print message indicating acknowledgement is received
    - print message indicating retransmission attempts
    - print message indicating failure of retransmission
  - default: 0 (disabled)

The interface that is described above is implemented using the command line arguments option that c/c++ makes available (int main(int argc, char *argv[]), and using a switch that will break apart the arguments according to the flags that precede the argument.

Since each process will be both a client and a server, when the process initializes, it will automatically start listening, just like any server would, and it will prompt the user for an input from the command line, which will send when the user hits the "ENTER" key. Since we are using threads to implement our project, the receive will be running in its own thread, which means that it will be constantly running in the background. The send will also have its own thread, which will be used whenever a user wants to send a message.

For the first part of the project, the interface will talk directly to the reliable point-to-point communication layer, which is described in farther detail below. For the second part of the project, we will have to adapt the interface to communicate with the multicast layer, which is described below.

2.2 Causal Order Multicast Layer

On top of the point-to-point layer will be another layer which will implement causal order multicast. The point-to-point layer will take care of the integrity property. That is, any message passed up to the causal order layer will be ensured not to be a duplicate by the point-to-point layer. The validity property can be handled by the causal order layer. If we wish to multicast a message, we can immediately deliver it to ourselves without violating our algorithm.

We plan on implementing the causal order multicast using the Algorithm based on Birman et al. (1991), which is described in Figure 11.16 in the text [1]. Basically, the algorithm will exploit the use of vector timestamps. These will be implemented as integer arrays, with slots for each member of the group. Each time a message is sent, the process will increment the value for it's own timestamp in the vector and send the vector in the message header. When a process, p_i receives a message from p_j, it first places it in its holdback queue, and then checks that the following two conditions are met:

- it has delivered any earlier messages sent by the p_j
- it has delivered any message that process p_j had delivered at the time it multicast the message.

If these two conditions are met, the causal order layer will deliver this message to the application layer. To be able to assure a causal ordering, we will be using vector clock, and piggy-backing them on each message as a trailer.

2.2.1 Algorithms

2.2.1.1 Logical and Vector Timestamps Algorithms

The formal definition of causal ordered multicast as given by [1] is if *multicast*(*g*,m) happens-before *multicast*(*g,m'*), then every correct process that delivers *m'* will deliver *m* before *m'* .

Causal ordering of events in a distributed system is based on the well-known happens-before relation. The happens-before relation defined by Lamport is defined by the following three rules:

- If a and b are events in the same process and a comes before b, then a happens-before b.
- If a is the sending of a message by one process and b is the receipt of the same message by another process, then a happens-before b.
- If a happens-before b and b "happened before" c, then a happens-before c.

Note that the happens-before relation is irreflexive, asymmetric, and transitive. The relation is also referred to as the causality relation in and potential causality in.

Since there is no global clock in a distributed system, Lamport describes a mechanism for causal ordering of events by using vector time stamps, which are based on logical clocks.

A logical clock is a way of assigning a number (such as a counter) to an event where the number is the time at which the event occurred; since the clocks have no relation to physical clocks, they are called logical clocks. A logical clock is correct if it observes the following clock condition: if an event a occurs before another event b, then a should happen at an earlier time than b. To guarantee that the system will satisfy this clock condition, the following rules must be followed:

- Each process P_i increments counter C_i between two successive events.
- If event a is the sending of a message m by process $P_{i,}$ then the message contains a timestamp $TS=C_i$(event a).
- Upon receiving of a message m, process P_j sets C_j greater than or equal to its present value and greater than TS.

One method of determining the causal relationship between events is to assign a complete causal history (all events that precede a particular event) to each event. This is obviously a very inefficient and impractical approach because of its size of causal history sets is of the order of the total number of events; therefore, Lamport defines Vector Time Stamps that should solve the problem that the causal history approach had to some extent.

By observing the causal history approach, it can be deduced that the causal history can be sufficiently characterized by the largest index among its members; thus the causal history can be uniquely represented by an N-dimensional vector. The vector time V_i of a process P_i is maintained according to the following rules:

- $V_i[k]:=0$, for k=1,…,N processes.
- On each internal event e, process P_i increments V_i as follows: $V_i[i]:= V_i[i]+1$.
- On sending message m, P_i updates V_i as in (2), and attaches the new vector to m.
- On receiving a message m with attached vector time V(m), P_i increments V_i as in (2). Next P_i updates its current V_i as follows: $V_i:=\max\{V_i, V(m)\}$.

The major drawback of vector time stamps is the size of the entire time vector, in the event that the number of processes participating in the multicast is very large, and thus the overhead becomes more and more with more processes participating. Singhal and Kshemkalyani made some important observations and came up with compressed vector time stamps in order to alleviate the problem of unwanted overhead when large number of processes participated in a multicast. Their proposed system, compressed vector timestamps, offers drastic improvement in the overhead, but has its drawbacks as well. Due to the way it works (very much like MPEG compression, as it only sends the changed timestamps), some information about causal relationship between different messages sent to the same receiver is lost. Specifically, it is no longer possible to decide whether two such messages are causally dependent solely by comparing their compressed timestamps. The compressed vector time stamps only work well for FIFO ordering multicast, and thus cannot be considered for our implementation.

2.2.1.2 Causal Order Algorithms

As stated in [1], Birman et al. came up with a causal order multicasting algorithm using vector timestamps. As we were exploring other algorithms, such as Meldal extensions, Schiper-Eggli-Sandoz Protocol, and Raynal-Schiper-Toueg Protocol, we also came across the FM (Fidge-Mattern) Protocol. The FM protocol on the surface seems identical to the algorithm presented in[1], which gave credit to Birman. To keep things consistent, we will be referring to our algorithm as the FM algorithm.

The FM algorithm is based on a vector of logical clocks in order to implement causal ordering. Both vector timestamps and logical clocks have been thoroughly discussed earlier, so we will not be covering any of those details. In this algorithm, each process maintains a natural number to represent their local clocks. Each process initializes its clock to 0 and increments it at least once before performing each event. Whenever processes send or receive messages, they pass the local clock information to each other along with the messages.

The logical time is defined by a vector of length N, where N are the number of processes in the multicast group. The logical time vector is denoted as VT; the logical time on process P is denoted as VT(P); and the timestamp for message M is VT(M). The logical time of the system will evolve accordingly:

- When a local event occurs at process P_i, the ith entry in the vector $VT(P_i)$ is incremented by one: $VT(P_i)[i]:=VT(P_i)[i]+1$;
- When P_i receives a message M, timestamped VT(M), the rule states:
  for j=I, $VT(P_i)[j]:=VT(P_i)[j]+1$;
  for j^1 I, $VT(P_i)[j]:=\max(VT(P_i)[j], VT(M)[j])$

As stated earlier, the major drawback with this system is the scalability issue, since the more processes are members of the multicast group, the larger the overhead becomes compared to the payload size.

2.3 Reliable Point-to-Point Communication Layer

This implementation will utilize a multi-threaded design in order to make both the senders and receivers are non-blocking. The system will utilize a queue for messages that will be protected by a mutex lock. We will use a mutex lock on the function that has the capability to manipulate the queue. At this point, our only global data will be this queue, and thus is the only thing that needs to be protected. This queue will be implemented in a FIFO (first in first out) ordering, and as described later, it will discard of old messages in this manner. Therefore, threads can operate concurrently in a safe manner, but only one thread will be allowed to call these queue manipulation functions at any given time. There will be 2 sockets open, one for sending and one for receiving. Both of these sockets will be managed by 3 different threads: the first will handle the first transmission; the seconds will handle any retransmission if necessary; and the third will handle any incoming messages.

Each message will have a set structure to it, such as a header and the data. For the point-to-point communication, sender id defined by the socket (IP address and port number), sequence number, acknowledgement number, and a 1 bit field differentiating between ACKs and data, will all be included in the header, apart from the body of the message (the actual data to be transmitted).

There will be a timer implemented on the sender, which will calculate the RTT based on when the ACK is received. Based on the size of the message, we can deduce an initial value for the timeout. If the system doesn't receive an ACK within the time-out, it retransmits, and increases the timeout value exponentially. After a certain threshold, which is also determined according to the message size, the system will give up assuming that the process has failed.

From a simplicity point of view, we have decided to initially limit our message sizes to no more than 1500 bytes to avoid fragmentation. If we see the need in the future for larger message sizes in the application, the lower layer protocols should take care of fragmentation and re-assembly for us. Message size also will affect our exponential back off algorithm as indicated above.

Assuming that we use the 3-way handshake, each message should be deleted once an ACK is received from the client (piggy-backed on messages), however, if an ACK is not received, it should be deleted according to the exponential back-off algorithm. The exponential back off algorithm indicated above has a time window in which we will wait before discarding a message and considering a process has crashed. We can calculate the max value for this and then define a queue size that will safely hold all these messages. In the event that the queue becomes full, we will be able to safely discard messages in FIFO order because the exponential timeout will have expired for those messages.

3.0 Final Report

We implemented causal ordering multicast, following the guidelines presented in the "Programming interface and design" section. Our model was designed as peer-to-peer from the very beginning, which helped us in the second part.

When the program begins running, we first read all members of our communication group from the input file ("members.dat"). With this information stored, we can begin sending messages. When a user types in a message, this is sent by the application layer to the causal layer, which timestamps it and sends it further to the point-to-point layer. The causal layer puts n number of messages on the send queue, where n represents the number of active processes. The point-to-point layer takes care of transmitting the message (and the vector timestamp attached to the message) over the network. After the successful sending of each message, the message is deleted from the send queue of the causal layer. If one or more hosts appear to be down, these are eliminated from the group and the process continues, and never tries to retransmit to them ever again, unless the process is restarted. When receiving a message over the network, the point-to-point receive thread sends it higher in the hierarchy, to the causal receive function. This function has to take care of the ordering requirement. If the message can be delivered, it is sent to the application layer (printed out on the screen). Otherwise, it's kept into the holdback queue, until it can be delivered.

We'll further describe the architecture of our model, give some implementation details and explain how the information flows between the layers.

3.1 Architecture

The model is layered, according to the following structure:

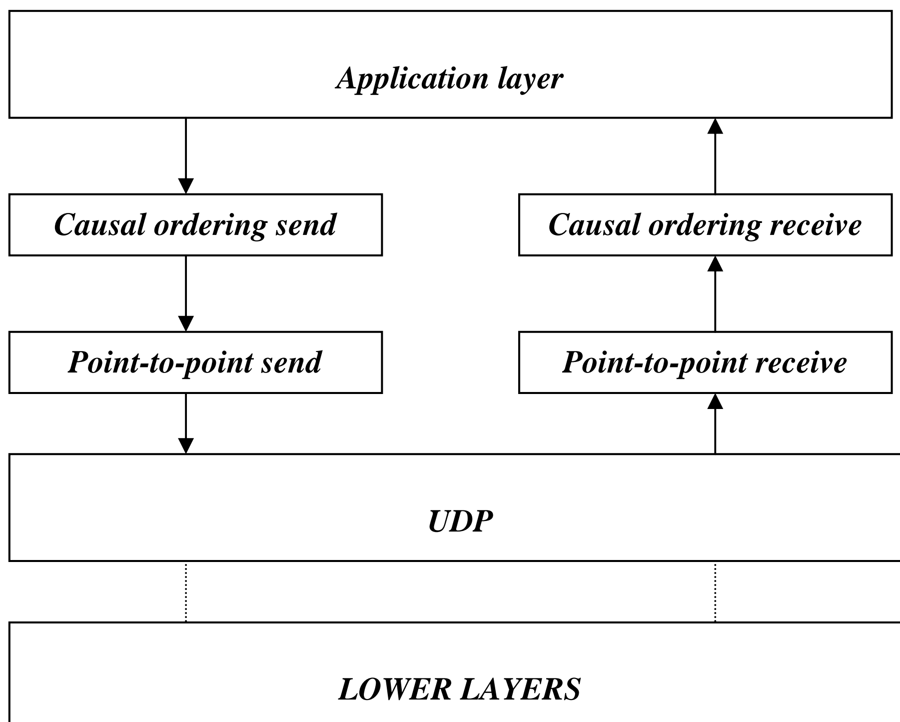

Figure 2. System Architecture

The application layer has two functions: gets input from the keyboard and passes it along to the causal layer, and receives and displays messages given by the causal layer.

Once the application layer starts working, it starts a worker thread that gets in an infinite loop that will send "KEEP ALIVE" messages every 10 seconds. The thread is implemented using usleep(), which has microsecond granularity, and does not block. That means that everybody (all other threads) can do their jobs uninterupted except for once every 10 seconds. The "KEEP ALIVE" messages are treated like ordinary message, except that the ack in struct message_info is changed to a 2 to represent that it is not a message, nor an ack, but a keep alive message. These keep alive messages were used in order to solve the issue of liveliness, as discussed in the lecture. These messages are not delivered to the application layer, however, they do help us make sure there is progress even if a process is not sending messages.

The causal ordering layer is comprised of 2 functions, one dealing with the sending of messages, and the other one for receiving messages. On the other hand, the point-to-point layer uses 2 separate threads to operate both the sender and the receiver. The causal layer invokes the sender thread call of the point-to-point layer, passing only the message to be sent reliably over the network. The point-to-point layer then does its magic like described in section 2.0. The point-to-point receiver thread invokes the causal receive whenever it receives a message over the network, who then has to do some computation in deciding whether or not to deliver the message to the application layer. The causal ordering layer is the one who deals with ordering the messages and delivering them to the application layer (printing them on the screen).

3.2 Implementation details

I have already briefly discussed the threads, now I'll concentrate on the data structures used in the project, together with the synchronization mechanisms associated with these data structures.

The point-to-point communication layer defined one queue used by the sending thread, and another one for the receiving thread. The ptp send queue contains the message to be sent, and the information needed by the exponential backoff algorithm. The message to be sent is further divided by the causal ordering layer (in message + vector timestamp), but the ptp layer sees everything as one stream:

```
typedef struct ptp_send_queue{       /* ptp send queue */
ptp_info_t *info;                    /* node information */
int send_times;                      /* transmision times*/
long timeout_value;                  /* timeout vaslue for this packet, depends
                                        on the transmision times */
struct ptp_send_queue *next;         /* point to next node */
}ptp_send_Qnode;
```

Figure 2. struct ptp_send_queue

The receiving queue contains only the information received. We'll append the id of the sender, when we send it further to the causal layer.

```
typedef struct ptp_receive_queue{  /* ptp receive queue */
ptp_info_t *info;                  /* node information */
struct ptp_receive_queue *next;    /* point to next node */
}ptp_receive_Qnode;
```

Figure 3. struct ptp_receive_queue

The causal layer defined its messages to be of the type:

```
typedef struct causal_msg_info {
char message_buf[128];             /*message*/
int Vector_TS[MAX_MEMBERS];   /*vector timestamp*/
}causal_msg_info_t;
```

Figure 4. struct causal_msg_info

where message_buff contains the actual message, and Vector_TS contains the vector timestamp attached by the sender to the message. The point-to-point layer sees all this structure as one message – is unaware of the vector timestamp.

All the layers are totally independent of each other, as each one envelopes their current message into the next layers message as it is being passed along the different layers.

The causal layer makes use of a holdback queue, that keeps the messages that have been received, until they can be delivered to the application layer:

```
typedef struct causal_send_queue {           /* causal receive queue - holdback queue*/
  struct causal_msg_info_t message_buf;      /*the message received = msg + vector_TS*/
  int index;                                 /*index of the sender*/
  struct causal_send_queue *next;
}causal_send_Qnode;
```

Figure 5. struct causal_send_queue

Index in the structure above represents the id of the sender, which we need for verifying the causal ordering.

Each process is assigned an id when the program begins to run. All hosts and ports are read from the input file "members.dat". This information is stored into a vector, or better yet an array. We

also use a parallel array, called active_members, where we store 0 for inactive processes, and 1 for active processes. At the very beginning, all processes are active, that means innocent until proven guilty. When we are sending messages, if we realize that a host is down, we mark that host as inactive and we don't consider it a member of our group from that moment on. Thus, the group is shrinked. We can not add new hosts, without restarting the whole program, since new members may only be added at the beginning before any messages are sent. The size of the group is static, and although the group shrinks in size, the amount of data sent (the vector size) remains constant, the maximum allowed number of members.

We have used mutexes for synchronizing the access to all the queues, and to the global variables that we used for the hosts (the parallel arrays). We also used a mutex for protecting the global variable My_Vector_TS, which keeps the current vector timestamp, as seen by the process.

3.3 Changed Items since Project 1

Our interface changed a little bit, thus we will describe the interface again.

Required arguments

- -S <local hostname>
    - host name for the local machine as it appears in "members.dat"

Optional arguments

- -n <number of retransmissions for PTP layer>
    - number of retransmissions in case of failure (NO ACK received)
    - default: 5
- -t <server time delay for PTP layer>
    - if enabled, will delay the acknowledgement the specified number of microseconds
    - default: 0 (disabled)
- -T <server time delay for Causal layer>
    - if enabled, will delay the message that is sent to the last active process for the specified number of microseconds (only for testing the Causal layer behavior)
    - default: 0 (disabled)
- -s <server sequence mod rate for PTP layer>
    - if 1 (enabled), will prompt the user for each message, if the sequence number should be changed, and to what new sequence number
    - default: 0 (disabled)
- -D <provides debug for PTP layer >
    - if 1 (enabled), will do the following:
        - print messages received
        - print whether acknowledgement is sent, dropped, delayed, or modified
        - print message indicating acknowledgement is received
        - print message indicating retransmission attempts
        - print message indicating failure of retransmission
    - default: 0 (disabled)
- -C <provides debug for Causal layer >

- if 1 (enabled), will do the following:
  - print messages delivered
  - print information regarding the receive holdback queue
  - print vector time stamps for each message
- default: 0 (disabled)

3.4 Conclusion

We learned a lot about multicasts in general in attempting to resolve all the issues that came up in this project. The system works great, without any bugs as far as we can tell, however, it is not very efficient. Efficiency was not one of our main concerns, and thus it is very inefficient in terms of the passing of messages between the different layers. We all worked very hard, as we dedicated countless weekend to just this project. We included a readme file as well in helping to understand the interface. Comments should be very abundant throughout the program, that should make it easy enough to understand.

4.0 Works Cited

[1] Coulouris, George and Dollimore, Jean and Kindberg, Tim. Distributed Systems, Concepts and Design. Pages 448 ~ 450.