ConcurrentMentor: User Manual

Kishor Joshi

September 9, 2007

1 Overview

ConcurrentMentor consists of a message passing library which provides various communication and synchronization abstractions, a visualization system to depict the run time behavior of user programs and a run time system to set up the necessary addressing, synchronization, and remote process execution facilities. In a nutshell, users write programs using functions from the message passing library and use the run time system to execute their programs on specified hosts. A command line tool `cmrun` is provided which is responsible for setting up the run time system and executing specified processes. The visualization system, if enabled, comes up automatically after `cmrun` is invoked. There is also an option to save visualization related data for later playback in standalone mode. This document provides a reference to the command line tool `cmrun` and the message passing library.

2 Hello World

Like most programming tutorials, we start with a hello world program. Here we describe how to create a simple program and run it using our system. It is true that any any executable can be run using our system, but the purpose is not served unless it is linked with the message passing library and the program uses some form of message passing. A simple hello world program:

```
// hello.cpp
#include <iostream>
#include "cm.h"

using namespace std;
int main()
{
        int myId, numProcs;
        getMyId(&myId);
        getNumProcs(&numProcs);

        if(myId == 0)
```

```
                cout<<"Hello World from process 0"<<endl;
        else if(myId == 1)
                cout<<"Hello World from process 1"<<endl;
}
```

The include file "cm.h" is required for using the message passing library. It contains prototypes of all the objects supported by the library. Henceforth all our programs will include this header file.

getMyId() is a library function which returns the process id of the current process. This process identifier is not the one assigned by the operating system, but it is the one assigned by our run time system.

getNumProcs() is another library function which returns the total number of processes running in that particular computation. Total number of processes is specified by the user while running the programs.

These functions should be called at the beginning of the program. However, it is not mandatory. But there is no reason not to do so. It makes a lot of things easier and in fact parallel programming systems provide ways to access them via a function call or implicitly. Each process has a unique process id assigned by the control process. The process ids start from 0 and are contiguous. So if there are N processes in a particular computation, the process ids would be 0, 1, ..., N-1.

After saving the above code using your favorite editor, you will have to compile the program using a C++ compiler. You should specify the correct path to the include file "cm.h" and the library file "libcm.a"", but assuming that everything is installed in required locations, you can compile your program as follows

```
g++ -o hello hello.cpp -lcm
```

Now to run the program, you will have to use the cmrun command. For this example we will require two process running, which can be specified as an option to cmrun. At the shell type the following:

```
cmrun -np 2 hello
```

After this, the visualization window pops up. At this point you have to click the start button to proceed. The output at the terminal is as follows:

```
Hello World from process 0
Hello World from process 1
```

The main visualization window is shown in figure 1 with various sections labeled.

3 cmrun

As mentioned earlier, cmrun is the run time system of ConcurrentMentor. When it is fired up, it will perform the necessary initialization: setting up facilities for message passing, synchronization,

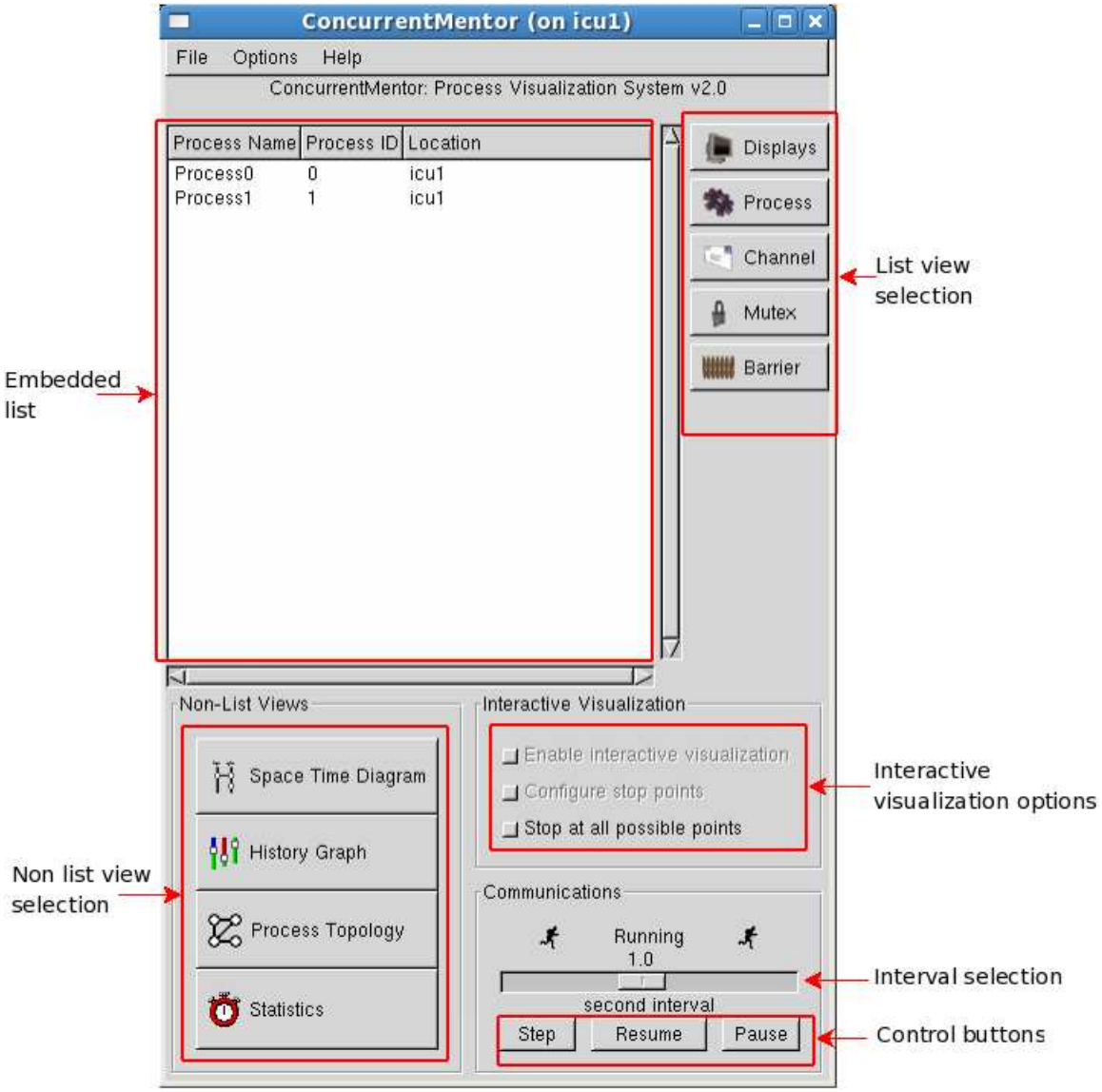

Figure 1: Main visual window

interfacing with the visual system, and then running the user programs on the specified machines. The internal working of `cmrun` is not discussed in this manual, we will focus only on the command line interface provided to the end users. The format of the command line is:

```
cmrun [-rsh|-ssh] [-nv] [-np nprocs] [-mf machinefile] [-help] userprog [arg1 ...]
cmrun [-rsh|-ssh] [-nv] [-mf machinefile] -pf programfile
```

`[-rsh|-ssh]:` specifies either rsh or ssh as remote login program. Default is `rsh`.

`[-nv]:` provides an option to disable the visualization system.

`[-np nprocs]:` -np is followed by a number which is the total number of processes to spawn.

`[-mf machinefile]:` machinefile contains the names of remote machines on which the programs are to be run.

`[-pf programfile]:` programfile contains a list of programs and their arguments.

`[-help]:` displays a summary of options available.

`[userprog arg1 arg2 ... arg n]:` Specifies a user program and its arguments. `userprog` is either an absolute or relative pathname, or a program name in the shell search path and `arg1 arg2 ... argn` are its arguments.

Notes:
- It is required to specify either a `userprog` in the command line or a program file via -pf option but not both.
- When a program file is specified, -np option cannot be used or else it will result in an error.
- If machine file is not specified, all the programs run on the local host (the host where `cmrun` was invoked from)
- When -help option is specified, a summary of available options is printed regardless of other options.
- But anything specified after userprog is not considered an argument to `cmrun`, not even the -help option.

`cmrun` supports two models of computation: Single Program Multiple Data (SPMD) and Multiple Program Multiple Data (MPMD). For SPMD model, program name can either be specified via the command line or a program file. For MPMD, a program file has to be used.

4 File format

4.1 Program File

Program file is a text file which contains a list of programs and their command line arguments. The program entries are separated by a newline. The arguments to a program should be specified in the same line as the program. Anything beginning with # up to the next newline is treated as a comment and ignored. Insignificant spaces, tabs or newline characters are also ignored.

You can specify absolute or relative paths of the programs in program file. You can also specify programs that are in the shell search path.

The contents of a sample program file is listed for your convenience

```
#Comments are for your convenience. They are ignored
/home/grad/myid/progs/one my args #This is another comment

   /home/grad/myid/progs/two other args
#And there can be more comments
```

Program file is specified via -pf option to `cmrun`. You have to specify either a program file or a program name, but not both. If both are specified, `cmrun` prints an error message and quits. If you specify the -pf option, you should not specify the -np option.

Process ids are assigned starting from the first entry. The first entry gets process id 0 and so on.

4.2 Machine File

Like program file it is also a text file, but it contains the machine names on which the user programs are to be run. A machine name entry can either be a host name, a fully qualified domain name (FQDN), or an IP address of the machine. Each entry is separated by one or more whitespace characters (space, newline, tab). Comments start with # character upto the next newline. Comments are for convenience and are ignored.

Machine file can contain duplicate entries. Duplicate entries are not ignored i.e. if a machine name appears twice in a machine file, it is highly likely that the particular machine runs two programs.

The machine names specified in this file must be resolvable. If any entry cannot be resolved, it is ignored and a warning message is displayed. If all entries cannot be resolved, the local host is used. It is upto you to make sure that the machines specified in this file are up during the computation. ConcurrentMentor does not deal with fault tolerance.

The entries in this file has one to one correspondence with the entries in the program file as far as possible. The first program specified in the program file is run on the first machine specified in the machine file. If there are more programs than available machines, the machines are used in round robin fashion i.e. repeating from the top of the list after reaching the end. If there are more machines than programs, the remaining machines are ignored.

The contents of a sample machine file

```
#This is a comment
bear bova        anthony nonexistent.mtu.edu
asimov
herbert     #Your note: This host goes down often.
```

From the above machine file there are five resolvable hosts bear, bova, anthony, asimomv, herbert which are used, while the host nonexistent.mtu.edu is ignored and anything specified after # character upto the next newline is treated as a comment.

IMPORTANT NOTE on remote machines:
ConcurrentMentor uses rsh or ssh to spawn remote processes. It is upto you to make sure that you have proper access to the remote machines i.e. you do not require to enter password/passphrase while accessing the machines remotely. For rsh protocol, make sure that you specify required machine names in the `~/.rhosts` file. For ssh, make sure that you use key based authentication and can access remote machines without a passphrase.

5 System Information

From the point of view of user programs, two information about the runtime system are significant: process id and number of processes.

1. Process id: As mentioned earlier, process id is the identifier of a process assigned by the run time system. Process id begins from 0 and continue upto N-1, where N is the total number of processes in that computation. The library provides a function to get the id of the current process and most programs call it at the beginning.

   ```
   void getMyId(int *myId);
   ```

   `myId` is a pointer to the location where the information about the current process id is to be stored. It is upto the user to make sure that the specified memory location is allocated. The user program behavior is undefined in case the pointer points to an unallocated memory.

2. Number of processes: This is the information about total number of processes spawn by the control process in the current computation. The number of process is specified either by the -np option or the number of program entries in the program file. If none is specified, then the default is 1. The library provides a function to get this information and it is called alongside the `getMyId()` function.

   ```
   void getNumProcs(int *numProcs);
   ```

   `numProcs` is a pointer to the memory location where the information about number of processes is to be stored. It should be a valid pointer as above, otherwise the behavior is undefined.

3. Get last error: The library also provides a way to print a message that describes the last error encountered.

   ```
   void cmerror(char * msg);
   ```

   `msg` is the pointer to the message to print alongside the error message.

6 Message Passing

ConcurrentMentor provides support for message passing between processes. Message passing makes it feasible for multiple processes to communicate with each other, and synchronize. Exchanging messages using low level communication facilities is possible, but it is a cumbersome process and much of the effort gets wasted in learning details about them. So ConcurrentMentor provides an abstraction of communication facilities to the user programs, which is transparent to actual location of the machine on which they run. The communication primitives are called channels. ConcurrentMentor supports one to one channels only i.e they connect two processes and the connected processes can communicate with each other in either directions. You need to create a channel object in your program in order to use the facilities. There are two types of channels: synchronous and asynchronous.

6.1 Synchronous Channel

Synchronous channel is the one in which both send and receive blocks until it is complete in a lockstep. The message passing is considered complete only when the message sent by the sender is received by the receiver and the sender gets an acknowledgment of the receipt. Since corresponding send and receive complete in a lockstep, both processes will have the same vector time right after the completion.

Channel constructor

```
SynChannel(int destID, char *channelname = NULL);
```

Description:
Creates a synchronous channel between the calling process and the destination process as identified by `destID`. Both the processes involved need to create a synchronous channel object in between them in order for the channel creation to be successful.

Arguments:
`destID`: The destination process id.
`channelname`: Name of the channel. If not specified, the default value is NULL. In the default case, the channel name is assigned by the library. This name is used by the visualization system for display purpose.

Preconditions:
There should be no pre-existing synchronous channel between the calling process and the destination process. This implies that between two processes there can be at most one synchronous channel. In addition to that, a process cannot create a synchronous channel with itself.

Result:
A channel is created between the current process and the destination process if all the conditions are met. If a process already has an existent synchronous channel to the destination process or it tries to create a channel with itself, channel creation fails. When a channel cannot be created, all subsequent operations on such channel also fail.

A simple program to create synchronous channel is listed below:

```
// syncc.cpp
// Program to create a synchronous channel and send data
#include <iostream>
#include <string.h>
#include <unistd.h>
#include "cm.h"

using namespace std;

int myId, numProcs;

int main(int argc, char *argv[])
{
        int numbytes = 4;

        getMyId(&myId);
        getNumProcs(&numProcs);

        if (myId == 0) {
                char *sendbuf = new char[numbytes + 2];
                int retval;
                for (int i = 0; i < numbytes; i++) {
                        sendbuf[i] = 'A';
                }
                sendbuf[numbytes] = '\0';

                SynChannel *ch1 =
                     new SynChannel(1);
                retval = ch1->send(sendbuf, numbytes);
                cout << "send returned " << retval << endl;
                delete sendbuf;
        }

        else {
                char *recvbuf = new char[numbytes + 2];
                int retval;
                SynChannel ch1(0);
                retval = ch1.receive(recvbuf, numbytes);
                if (numbytes != -1) {
                        recvbuf[numbytes] = '\0';
                        cout << "Received data::"
                        << recvbuf << endl;
                }
                cout << "receive returned " << retval << endl;
```

```
                delete recvbuf;
        }
        return 0;
}
```

Compile this program and run it as follows:

```
g++ -o syncc syncc.cpp -lcm
```

```
cmrun -np 2 syncc
```

The following output is observed at the terminal:

```
Received data::AAAA
send returned 4
receive returned 4
```

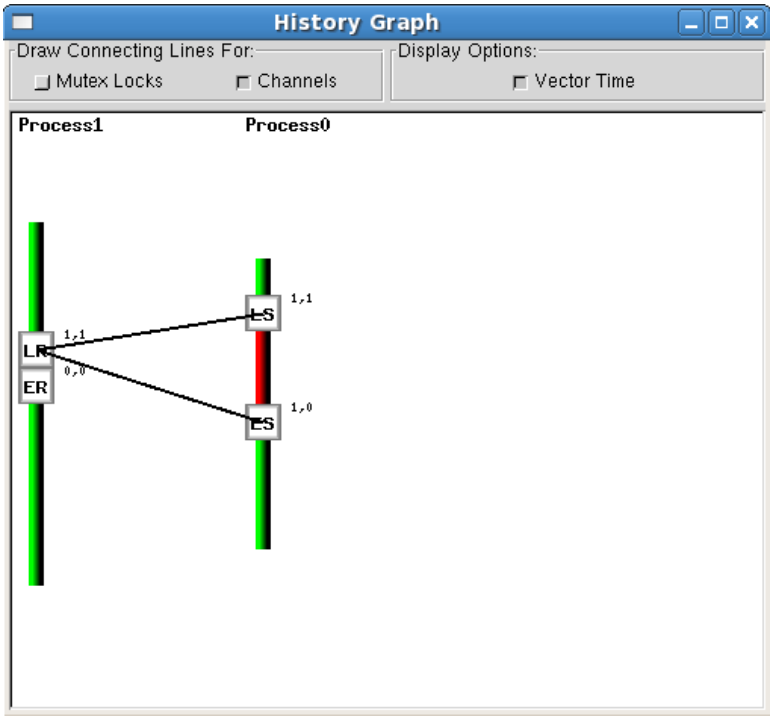

Figure 2: History diagram for synchronous send

Figure 2 depicts the history diagram for the above program. A history diagram portrays the states of all processes and various events of interest like send, receive etc. during the lifetime of the processes. Each process is represented by an event line which is green when the process is running normally and red when the process is blocked. The events are denoted by an event tag, with the name of the event, in the event line. There are options to display the vector time corresponding to all events, and to draw connections between events in different processes. A brief description about the tags can be obtained by clicking Help->Graph/Diagram Tags menu from the main window.

Clicking on each event brings up a window highlighting the line of source code of the user program which was responsible for that event. For this to work properly the source code should be in the current working directory, i.e the directory from where `cmrun` is invoked.

Synchronous send

```
int send(void *message, int message_size);
```

Description:
Sends message through the channel. This call blocks until the message is received and the receipt is acknowledged by the receiver.

Arguments:
`message`: The pointer to message buffer. This cannot be NULL.
`message_size`: The total number of bytes to send. Must be greater than or equal to 0.

Preconditions:
The channel should be existent. If the creation of channel had failed earlier, this operation too will fail. For successful completion, it requires that the receiver is up for the entire duration. If communication breaks between the two processes, this operation fails.

Return value:

If successful, the return value is the total number of bytes sent.

If communication breaks, return value is 0.

In case of nonexistent channel, NULL message buffer, or negative message size is specified, return value is -1.

Results:
Sender and the receiver will have the same vector clock values at the end.

Synchronous receive

```
int receive(void * message, int message_size);
```

Description:
Receives message through the channel. Like send, this call too blocks until the message is received. This call returns only after sending an acknowledgment to the sender.

Arguments:
`message`: The pointer to buffer where the receive message should be stored. This cannot be NULL.
`message_size`: The total number of bytes to receive. Must be greater or equal to 0.

Preconditions:
Preconditions are similar to the send operation.

Return value:

Upon successful completion, the return value is total number of bytes received.

In case of error, the return value is -1.

Note: If there are less bytes available from the sender, then that value is returned instead of the one requested. It is up to the user to pass a sufficiently allocated message buffer. If larger message is received from the sender, only the requested size is copied to the given buffer and the remaining message is discarded.

Results:

Sender and receiver will have the same vector clock values at the end.

The buffer pointed by message will have requested bytes or less.

```
// syncring.cpp
// Program to create a synchronous channel ring
// and pass data around the ring
#include <iostream>
#include <string.h>
#include <unistd.h>
#include "cm.h"

using namespace std;

int myId, numProcs;

int main(int argc, char *argv[])
{
        int counter = 0;
        getMyId(&myId);
        getNumProcs(&numProcs);

        //create a ring and pass message around
        SynChannel *ch_left =
           new SynChannel((numProcs + myId - 1) % numProcs);
        SynChannel *ch_right =
           new SynChannel((myId + 1) % numProcs);

        cout << "Hello from process " << myId << endl;

        if (myId == 0) {
                ch_right->send(&counter, 4);
                ch_left->receive(&counter, 4);
                cout << "Counter at P0 = " << counter << endl;
        } else {
                ch_left->receive(&counter, 4);
                counter++;
                ch_right->send(&counter, 4);
```

```
        }
}
```

Compile this program and run it as follows:

```
g++ -o syncring syncring.cpp -lcm
```

```
cmrun -np 3 syncring
```

Note: Number of processes must be greater than or equal to 3.

The following output is observed at the terminal:

```
Hello from process 0
Hello from process 1
Hello from process 2
Counter at P0 = 2
```

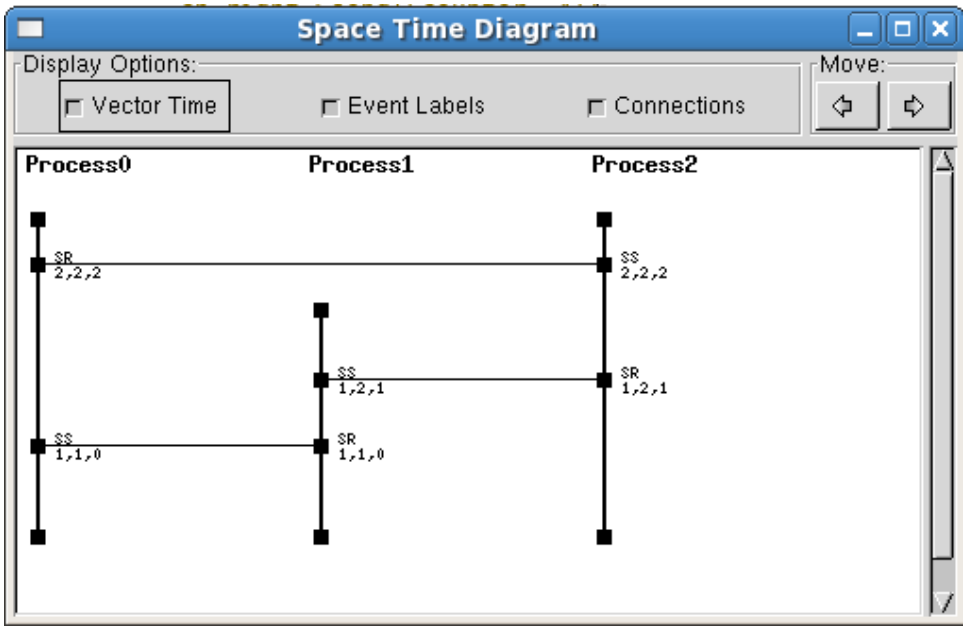

Figure 3: Space Time diagram for synchronous ring

Figure 3 depicts the space time diagram for the above program. A space time diagram shows the causal relationship between the events of different processes. Each process is represented by an event line which in turn consists of the events of interest. Unlike history diagram, the events are limited to Synchronous/Asynchronous Send/Receive only. There are options to connect the corresponding send and receive events, display the event labels, and display the vector time. A brief description about the event labels can be obtained by clicking Help->Graph/Diagram Tags menu from the main window. The events can be slided up and down in the event line as long as the causal relations with other events are not violated.

Synchronous poll

```
int poll(int num);
```

Description:
Checks whether the channel is available for reading num bytes of data at that time. This call returns immediately.

Arguments:
num: The size of message to check the readability.

Return value:

If successful, the number of bytes available for reading is returned.

If channel is unavailable for reading return value is -1.

Return value is -2 for permanent failures i.e in case of nonexistent channel, or invalid message size.

```
//syncp.cpp
// Synchronous channel poll
#include <iostream>
#include <string.h>
#include <unistd.h>
#include "cm.h"

using namespace std;

int myId, numProcs;

int main(int argc, char *argv[])
{
        int numbytes = 0;
        getMyId(&myId);
        getNumProcs(&numProcs);

        numbytes = 1;

        if (myId == 0) {
                char *sendbuf = new char[numbytes + 2];
                int retval;
                for (int i = 0; i < numbytes; i++) {
                        sendbuf[i] = 'A';
                }
                sendbuf[numbytes] = '\0';
                SynChannel *ch1 = new SynChannel(1);
                sleep(1);
                retval = ch1->send(sendbuf, numbytes);
                cout << "send returned: " << retval << endl;
                cout << "Data sent: " << sendbuf << endl;
```

```
                delete sendbuf;
        }

        else {
                char *recvbuf = new char[numbytes + 2];
                int retval, count = 0;
                SynChannel ch1(0);
                while ( (retval = ch1.poll(numbytes)) < 0) {
                        cout << "Iteration " << count++ << " in poll\n";
                        if (retval == -2)
                                break;
                }

                retval = ch1.receive(recvbuf, numbytes);
                cout << "receive returned " << retval << endl;
                cout << "Data received: " << recvbuf << endl;
                delete recvbuf;
        }
        return 0;
}
```

Compile this program and run it as follows:

```
g++ -o syncp syncp.cpp -lcm
```

```
cmrun -np 2 syncp
```

The following output is observed at the terminal:

```
Iteration 0 in poll
receive returned 1
Data received: A
send returned: 1
Data sent: A
```

Checking channel validity

```
bool isValid();
```

Description:
To check whether the channel was created successfully.

Return value:
true if channel is existent.
false if channel is nonexistent.

6.2 Asynchronous Channel

Asynchronous channel has a nonblocking send and both blocking and nonblocking receive. The sender does not wait for receiver to receive message, it rather returns immediately after the data is copied to the communication buffer. This type of channel also supports optional message loss. Message loss is specified by reliability factor, which is a value between 0.0 and 1.0, ranging from totally unreliable to fully reliable. This factor can be specified at the time of channel creation by either supplying a number or a user defined reliability function.

6.2.1 Channel constructor

This type of channel has two types of constructor depending upon how the user wants to specify reliability. If a value is to be specified for reliability factor, then the first one can be used. In some cases where nonuniform distribution is required, the second one can be used. The reliability function should return a float value between 0.0 and 1.0.

```
AsynChannel(int dest, float rel, char * chname);
AsynChannel(int dest, float (*relfunction) (void), char *chname);
```

Description:
Creates an asynchronous channel between the calling process and the destination process as identified by dest. As in synchronous channel both the process involved need to create an asynchronous channel between them in order for the channel creation to be successful. If channel is created by only one process, the creation blocks until it hears from the destination.

Arguments:
`dest`: The destination process id.
`chname`: The name of the channel. This value is NULL by default, and in that case a default name is assigned to the channel.
`rel`: The reliability factor. The default value is 1.0
`relfunction`: The reliability function.

Preconditions:
There should be no pre-existing asynchronous channel between the calling process and the destination process and a process cannot create channel with itself.

Result:
A channel is created between the calling process and the destination process if all conditions are satisfied. In case of error, all subsequent operations on such channels fail.

6.2.2 Asynchronous Send

```
int send(void * msg, int msgsz);
```

Description:
Sends message through the channel. This call does not wait for an acknowledgment from the

receiver.

Arguments:
msg: The pointer to message buffer to send. This cannot be NULL or unallocated.
msgsz: The total number of bytes to send. Must be greater than or equal to 0.

Preconditions:
The channel should be existent. If the creation had failed earlier, the operation will fail. It does not matter whether the receiver is reading the data send, however, it requires that the connection is still intact.

Return value:
Number of bytes sent in case of successful operation
0 in case communication between the destination breaks.
-1 in case of nonexistent channel, NULL message buffer, or negative message size.

```
// asyncc.cpp
// To demonstrate the use of reliability function in asynchronous channel
#include <iostream>
#include <string.h>
#include <unistd.h>
#include <time.h>
#include "cm.h"

using namespace std;

int myId, numProcs;
int start, now;
int msgcount;

float reliabilityfunc()
{
        float reliab = 0.0;
        now = time(NULL);
        if (msgcount > 5) {
                reliab = 0.0;
        } else {
                reliab = 1.0;
        }
        msgcount++;
        return reliab;
}

int main(int argc, char *argv[])
{
        float rel;
        start = time(NULL);
```

```
        msgcount = 0;

        getMyId(&myId);
        getNumProcs(&numProcs);

        if (myId == 0) {
                int i = 0;
                char *sendbuf = "abcd";
                AsynChannel *ch1 = new AsynChannel(1, reliabilityfunc);
                cout << "Reliability :" << reliabilityfunc() << endl;
                for (i = 0; i < 10; i++) {
                        ch1->send(sendbuf, sizeof(sendbuf));
                }
                cout << "Send count : " << i << endl;;
        }

        else {
                char recvbuf[10];
                AsynChannel *ch1 = new AsynChannel(0);
                while (ch1->receive(recvbuf, 4) >= 0) {
                        msgcount++;
                }
                cout << "Reveive count: " << msgcount << endl;
        }
        return 0;
}
```

Compile this program and run it as follows:

```
g++ -o asyncc asyncc.cpp -lcm
```

```
cmrun -np 2 asyncc
```

The following output is observed at the terminal:

```
Reliability :1
Send count : 10
Reveive count: 5
```

Figures 4 5 depicts the history diagram and space time diagram for the above program respectively. Some send events in those figures do not have a corresponding receive events because of message loss.

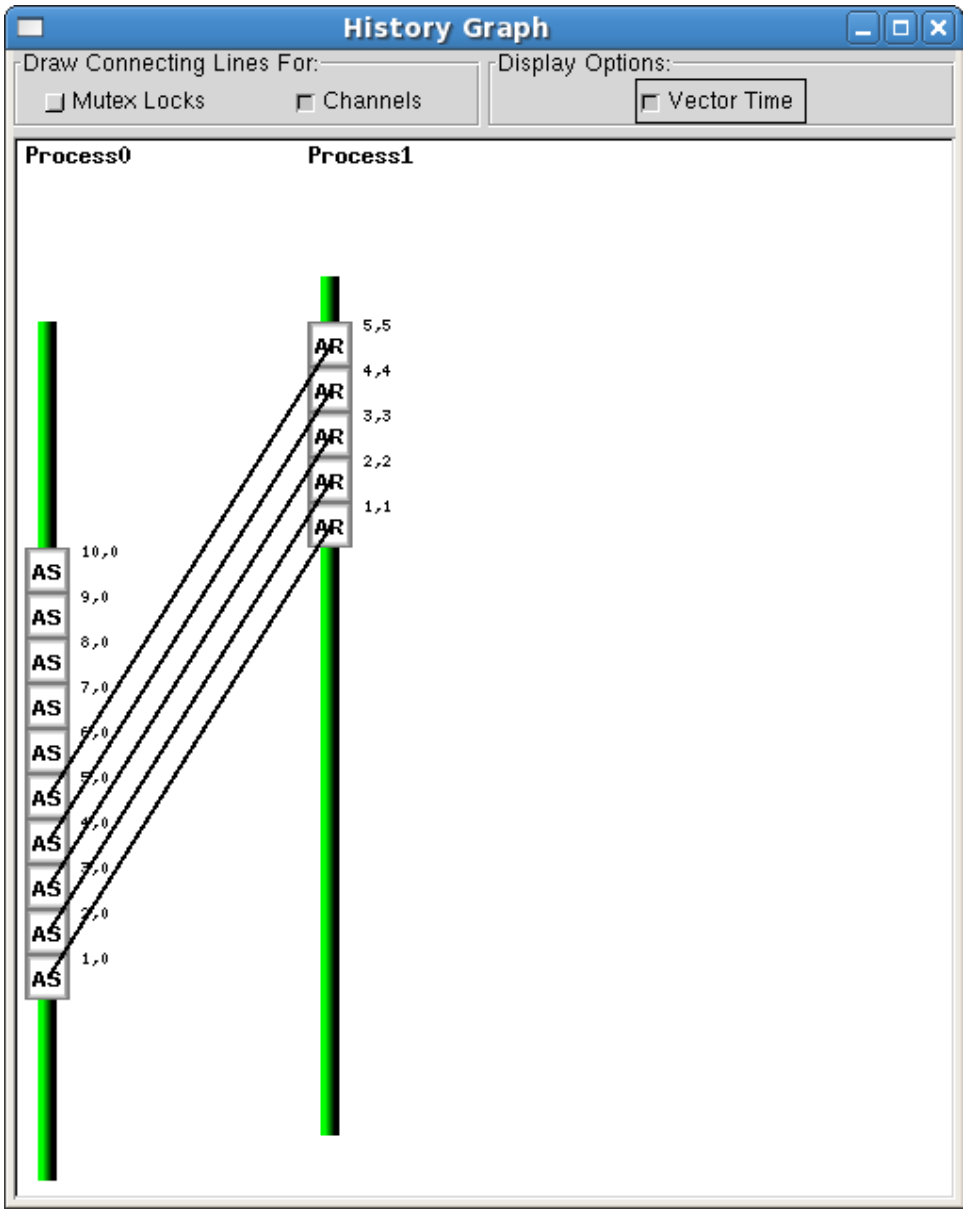

Figure 4: History diagram for asynchronous send

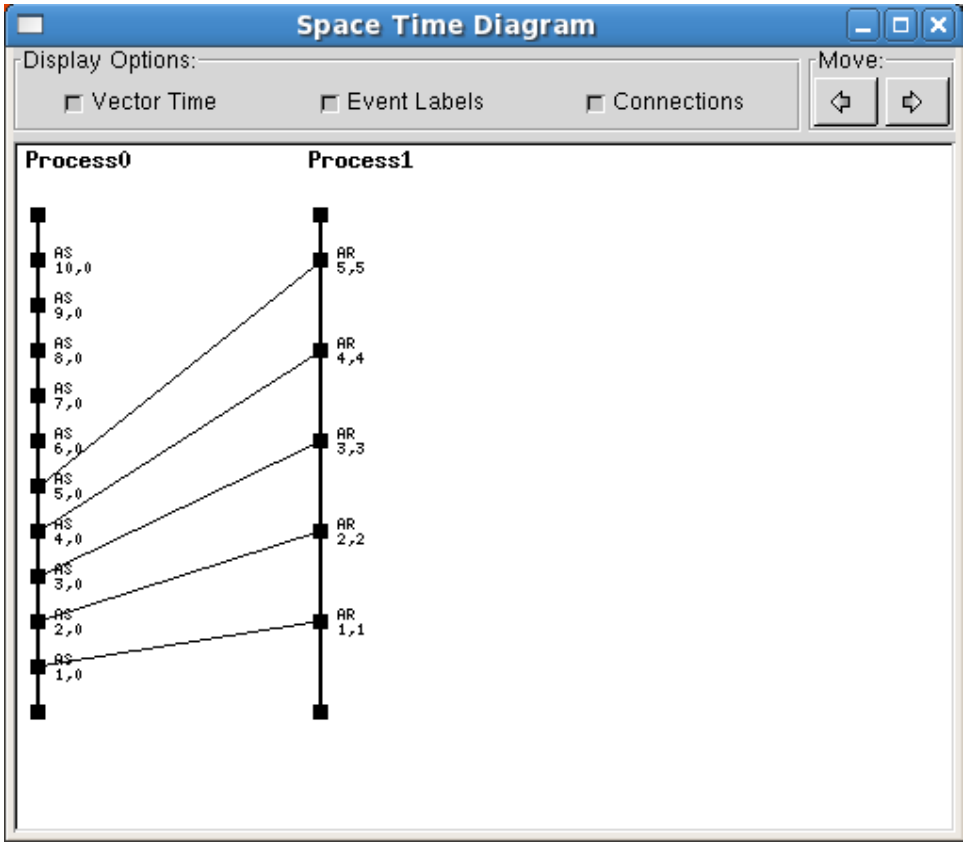

Figure 5: Space Time diagram for asynchronous send

6.2.3 Asynchronous Receive

```
int receive(void * msg, int msgsz);
int nbreceive(void * msg, int msgsz);
```

Description:
`receive` is the blocking version and `nbreceive` is the nonblocking version. This call receives data from the channel. The blocking version waits until data is received, the nonblocking version does not wait for data if not available at the time of call.

Arguments:
`msg`: The pointer to the buffer where the received message is to be stored.
`msgsz`: Number of bytes to receive from the channel.

Preconditions:
Channel should be existent. Receive on a create failed channel returns error.

Return value:
Number of bytes sent in case of success.
-1 in case of error. Error may be caused due to nonexistent channel, NULL message buffer, or negative message size. Error also occur due to problem with connection.
Note: Nonblocking receive returns number of bytes if it can read that size of data from the channel. Otherwise, the return value is -1 and it returns immediately without waiting for the data to arrive.

```
// asyncnbr.cpp
// To demonstrate the nonblocking receive
#include <iostream>
#include <string.h>
#include <unistd.h>
#include "cm.h"

using namespace std;

int myId, numProcs;

int main(int argc, char *argv[])
{
        int numbytes = 10;
        getMyId(&myId);
        getNumProcs(&numProcs);

        if (myId == 0) {
                char *sendbuf = new char[numbytes + 2];
                int retval;
                for (int i = 0; i < numbytes; i++) {
                        sendbuf[i] = 'M';
                }
                sendbuf[numbytes] = '\0';
```

```
                AsynChannel *ch1 = new AsynChannel(1);
                sleep(5);
                retval = ch1->send(sendbuf, numbytes);
                cout << myId << ": This time I snooze " << retval << endl;
                sleep(15);
                delete sendbuf;
        }

        else {
                char *recvbuf = new char[numbytes + 2];
                int retval;
                AsynChannel ch1(0);
                retval = ch1.nbreceive(recvbuf, numbytes);
                if (retval == -1)
                        cout << myId << ": No message in channel" << endl;

                //Let the sender send peacefully!!
                sleep(6);

                if (retval == -1) {
                        retval = ch1.nbreceive(recvbuf, numbytes);
                        if (retval == -1)
                                cout << myId << ": OOPS! no message again"
                                    << endl;
                        else
                                cout << myId <<
                                    ": YAHOO!! I finally received data" <<
                                    endl;
                }

                delete recvbuf;
        }
        return 0;
}
```

Compile this program and run it as follows:

```
g++ -o asyncnbr asyncnbr.cpp -lcm
```

```
cmrun -np 2 asyncnbr
```

The following output is observed at the terminal:

```
1: No message in channel
0: This time I snooze 10
1: YAHOO!! I finally received data
```

6.2.4 Asynchronous poll

This is similar to synchronous poll.

6.2.5 Checking channel availability

This is similar to synchronous counterpart.

6.3 Vector Clock

Vector clock for a system of N processes is an array having N elements where each element is the corresponding logical clock of each process. Each process increments its clock value every time an event of interest occurs and timestamps every outgoing message using the vector clock. Each process maintains vector clock and the clock is updated every time a message is received from other processes or the control process. This class provides an interface to the vector time maintained by the library. Since each process maintains one vector time, creating multiple objects of this type in a single process refer to the same vector time.

6.3.1 Constructor

`VectorClock()`
Creates a vector clock object which points to the vector time maintained by the channel library.

6.3.2 +, +=, and ++ Operators

These operators increment the vector clock value corresponding to the current process by the specified operand which must be greater than or equal to 0. The ++ operator increments the clock value by one.

6.3.3 [] Operator

This operator returns the clock value corresponding to the process specified by the operand. If the operand is out of range, the clock value corresponding to the current process is returned. This operator provides a read-only access.

6.3.4 Print vector clock

`void Print(void);` This function prints the current vector clock value in string form, which is a comma separated string containing clock values of each process starting from process 0.

```
// vectorc.cpp
// To demonstrate the usage VectorClock class
#include <iostream>
#include <string.h>
#include <unistd.h>
#include "cm.h"

using namespace std;

int myId, numProcs;

int main(int argc, char *argv[])
{
        getMyId(&myId);
        getNumProcs(&numProcs);

        VectorClock c1;
        ++c1;

        if(myId == 0){
                cout << "Printing vector clock id = " << myId << endl;
                c1.Print();
        }

        CMBarrier();
        if (myId == 0) {
                cout << "Entering round 2" << endl;
                c1.Print();
        }

        c1 += 10 + myId;
        if(myId == 0){
                cout << "Printing vector clock id = " << myId << endl;
                c1.Print();
        }

        CMBarrier();

        c1 + 20;
        if(myId == 0){
                cout << "Printing vector clock id = " << myId << endl;
                c1.Print();
        }
        CMBarrier();

        if (myId == 0) {
                cout << "Entering round 3" << endl;
                c1.Print();
```

```
        }

        c1 += -10;
        if(myId == 0)
                c1.Print();

        CMBarrier();
        c1 += -1000;

        CMBarrier();
        if(myId == 0)
                c1.Print();
        return 0;
}
```

Compile this program and run it as follows:

```
g++ -o vectorc vectorc.cpp -lcm
```

```
cmrun -np 2 vectorc
```

The following output is observed at the terminal:

```
Printing vector clock id = 0
<1 0>
Entering round 2
<1 1>
Printing vector clock id = 0
<11 1>
Printing vector clock id = 0
<31 12>
Entering round 3
<31 32>
<31 32>
<31 32>
```

6.4 Channel Monitor

This is an interface provided by the library to monitor availability of both synchronous and asynchronous channels for receiving messages. This is similar to individual poling mechanisms in channels, but it provides a way to monitor multiple channels simultaneously.

6.4.1 Constructor

```
Monitor(Channel *channels[], int num);
```

Description:
Creates a monitor object.

Arguments:
channels: An array of synchronous, asynchronous, or both type of channels. Channel is the base class of both types of channels, so the array could contain mixed type of channels.
num: The number of channels from the array to monitor.

Preconditions:
The channels array should be valid and populated. `num` must be greater than or equal to zero and should contain the actual number of channels to monitor. It is the responsibility of user to make sure that it does not lead to access outside the boundary of the array.

Result:
Creation of this type of object should always be successful unless there is some problem with accessing the array in which case the behavior is undefined.

6.4.2 Select ready channels

```
int selectReady(int bytes, int ms);
```

Description:
Selects channels from which given size of data can be read or return within a timeout.

Return value:
The return value is the total number of channels out of the monitored channels that are readable. If no channels are available then the return value is 0.

6.4.3 Get ready channels

```
int *readyChannels();
```

Description:
Returns the indices of channels that are ready for reading.

Return value:
The return value is an array containing the indices of channels that are ready for reading. The number of valid elements in the given array is the return value of the call `selectReady()`. There is also a separate function `numReady()` which returns the number of ready channels after the last call to the `selectReady()` function.

6.4.4 Get number of ready channels

```
int numReady();
```

Description:
Returns the number of channels ready for reading after the last call to the `selectReady()` function.

Return value:
The number of channels ready for reading or 0 if no channels are ready.

7 Mutex

7.1 Constructor

```
DistMutex(char * name);
```

Description:
Creates a mutex object in the calling process.

Arguments:
`name`: Name of the mutex to be created. Mutexes across different processes are identified by names. Two or more process have to create a mutex with the same name if they want to use the same shared resource.

Preconditions:
There should no pre-existing mutex with the same name in the calling process. The `name` parameter should not be NULL and the calling process should not exceed `MAX_SYNC_OBJ` number of mutexes.

Result:
A mutex object is created in the given process if all conditions are met. If any of the conditions is not met, mutex creation fails and any subsequent operation on such mutex also fails.

7.2 Mutex Lock

```
int lock();
```

Description:
Locks the given mutex. If the mutex had been locked earlier by another process, then the calling process waits until it is unlocked by the holding process.

Preconditions:
The mutex should be existent. If creation of mutex had failed earlier, this operation too will fail. If a process tries to lock the same mutex in succession, without unlocking, then the behavior is undefined.

Return value:
If successful, the return value is 1
On error, the return value is -1.

7.3 Mutex Unlock

```
int unlock();
```

Description:
Unlocks the given mutex.

Preconditions:
The mutex should be existent. If creation had failed earlier, this operation too will fail. If a process tries to unlock a mutex without locking it first, or tries to unlock the same mutex in succession, then the behavior is undefined.

Return value:
If successful, the return value is 1
On error, the return value is -1.

```
// mutex.cpp
// To demonstrate the use of distributed mutex
#include <iostream>
#include <string.h>
#include <unistd.h>
#include "cm.h"

using namespace std;

int myId, numProcs;

int main(int argc, char *argv[])
{
        getMyId(&myId);
        getNumProcs(&numProcs);

        if (numProcs < 2)
                cout << "Only one process!" << endl;

        DistMutex *m1 = new DistMutex("Mutex1");

        m1->lock();

        cout << "Process " << myId << ", locking the mutex" << endl;
        cout << "Other processes, if any, will follow after one second" << endl;
        sleep(1);
```

```
        m1->unlock();

        return 0;
}
```

Compile this program and run it as follows:

```
g++ -o mutexl mutexl.cpp -lcm
```

```
cmrun -np 4 mutexl
```

The following output is observed at the terminal:

```
Process 2, locking the mutex
Other processes, if any, will follow after one second
Process 0, locking the mutex
Other processes, if any, will follow after one second
Process 3, locking the mutex
Other processes, if any, will follow after one second
Process 1, locking the mutex
Other processes, if any, will follow after one second
```

The history diagram for mutex lock and unlock operations is depicted in figure 6.

8 Barrier

8.1 Constructor

```
DistBarrier(char * name, int size);
```

Description:
Creates a barrier object in the calling process.

Arguments:
`name` : Name of the barrier to be created. Barriers across different processes are identified by names. If two or more processes create a barrier with the same name, then the processes refer to the same barrier.
`size` : The size of barrier.

Preconditions:
There should be no pre-existing barrier with the same name in the calling process. The `name` parameter should not be NULL and the calling process should not exceed `MAX_SYNC_OBJ` number of barriers. The size parameter should be between 1 and total number of processes. At least `size` number of processes must create a barrier before it can be used.

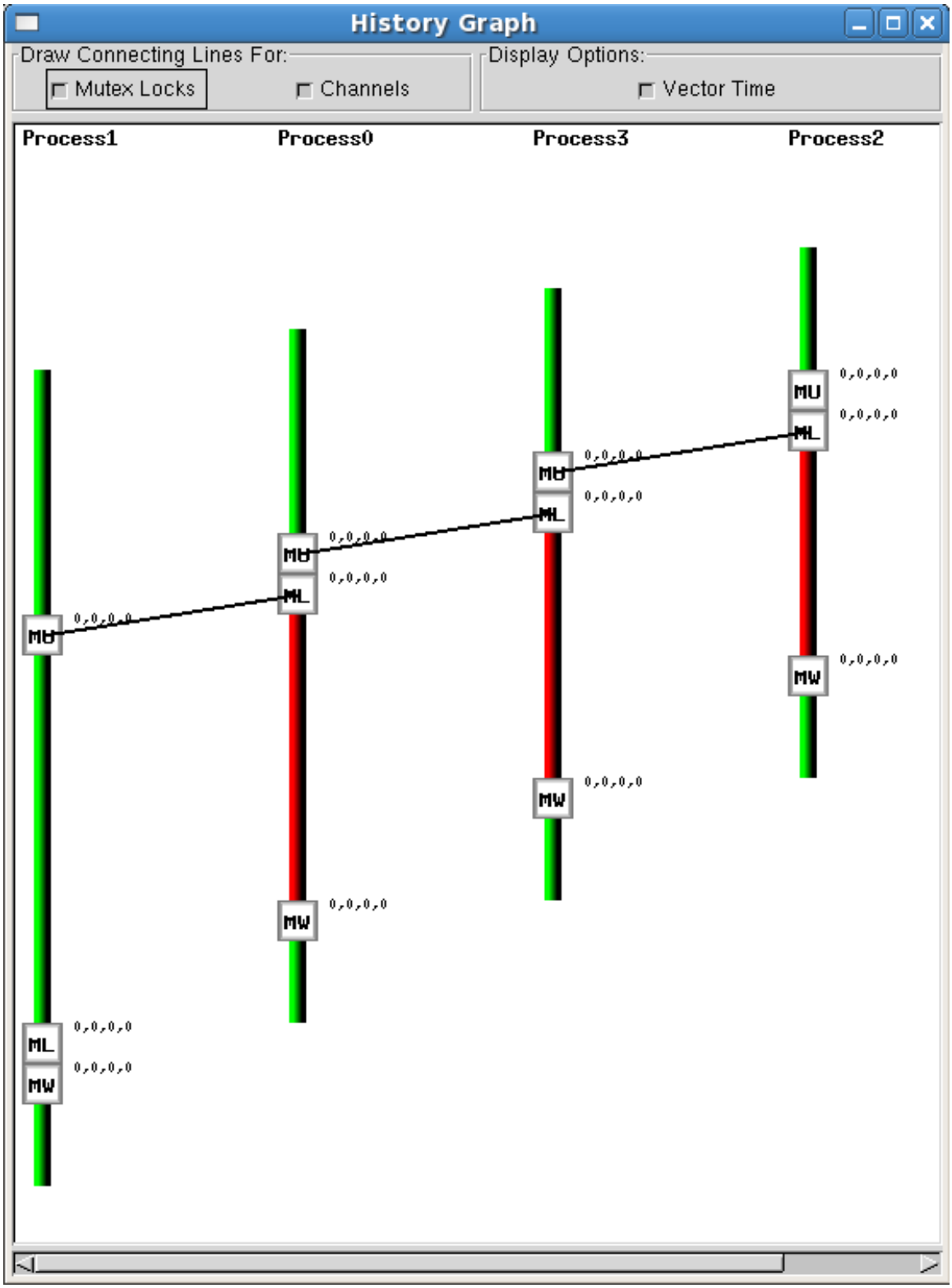

Figure 6: History diagram for mutex lock and unlock

Result:
A barrier object is created in the calling process if all the conditions are met. If any of the conditions is not met, barrier creation fails and any subsequent operation on such barrier also fails

The channel library also provides an implicit barrier named `barrierALL` with the size same as the total number of processes. The implicit barrier need not be created by the user and can be used via call to the function `CMBarrier()`. This barrier is also used by the collective operations in topologies.

8.2 Barrier Wait

```
void wait();
```

Description:
Wait on the given barrier. All processes calling wait block until a number of processes, equal to the size of the barrier, call wait on the given barrier. A barrier completes a generation when all the processes creating it go through a wait operation. Barrier generation is handled transparently by the library. If a barrier's size is less than number of processes, only the first `size` number of processes are blocked in each generation.

Preconditions:
The barrier should be existent. If creation had failed earlier, this operation too will fail.

Return value:
NONE
In case of error, the desired synchronization is not achieved.

```
// barrier.cpp
// To demonstrate the use of distributed barrier
#include <iostream>
#include <string.h>
#include <unistd.h>
#include "cm.h"

using namespace std;

int myId, numProcs;

int main(int argc, char *argv[])
{
        int counter = 0;
        getMyId(&myId);
        getNumProcs(&numProcs);

        DistBarrier *gb = new DistBarrier("GoodBarrier", numProcs);

        AsynChannel ch_left((numProcs + myId - 1) % numProcs);
```

```
        AsynChannel ch_right((myId + 1) % numProcs);

        cout << "Hello from process " << myId << endl;
        gb->wait();

        if (myId == 0) {
                ch_right.send(&counter, 4);
                ch_left.receive(&counter, 4);
                cout << "Counter at P0 = " << counter << endl;
        } else {
                ch_left.receive(&counter, 4);
                counter++;
                ch_right.send(&counter, 4);
                cout << "Counter at P" << myId << " " << counter << endl;
        }

        gb->wait();

        delete gb;
}
```

Compile this program and run it as follows:

```
g++ -o barrier barrier.cpp -lcm
```

```
cmrun -np 4 barrier
```

The following output is observed at the terminal:

```
Hello from process 0
Hello from process 1
Hello from process 2
Counter at P1 1
Counter at P2 2
Counter at P0 = 2
```

The history diagram for barrier wait operation is depicted in figure 7. It can be noted that the vector time after passing a barrier is the same in all processes.

9 Topolgy

Topology is a higher level abstraction based on channels. It provides a convenient way to build communication infrastructure for a group of processes. Channel library provides supports for various types of topology classes which includes constructor, send/receive functions, and collective communication functions. Topology uses asynchronous channels for communication and since any

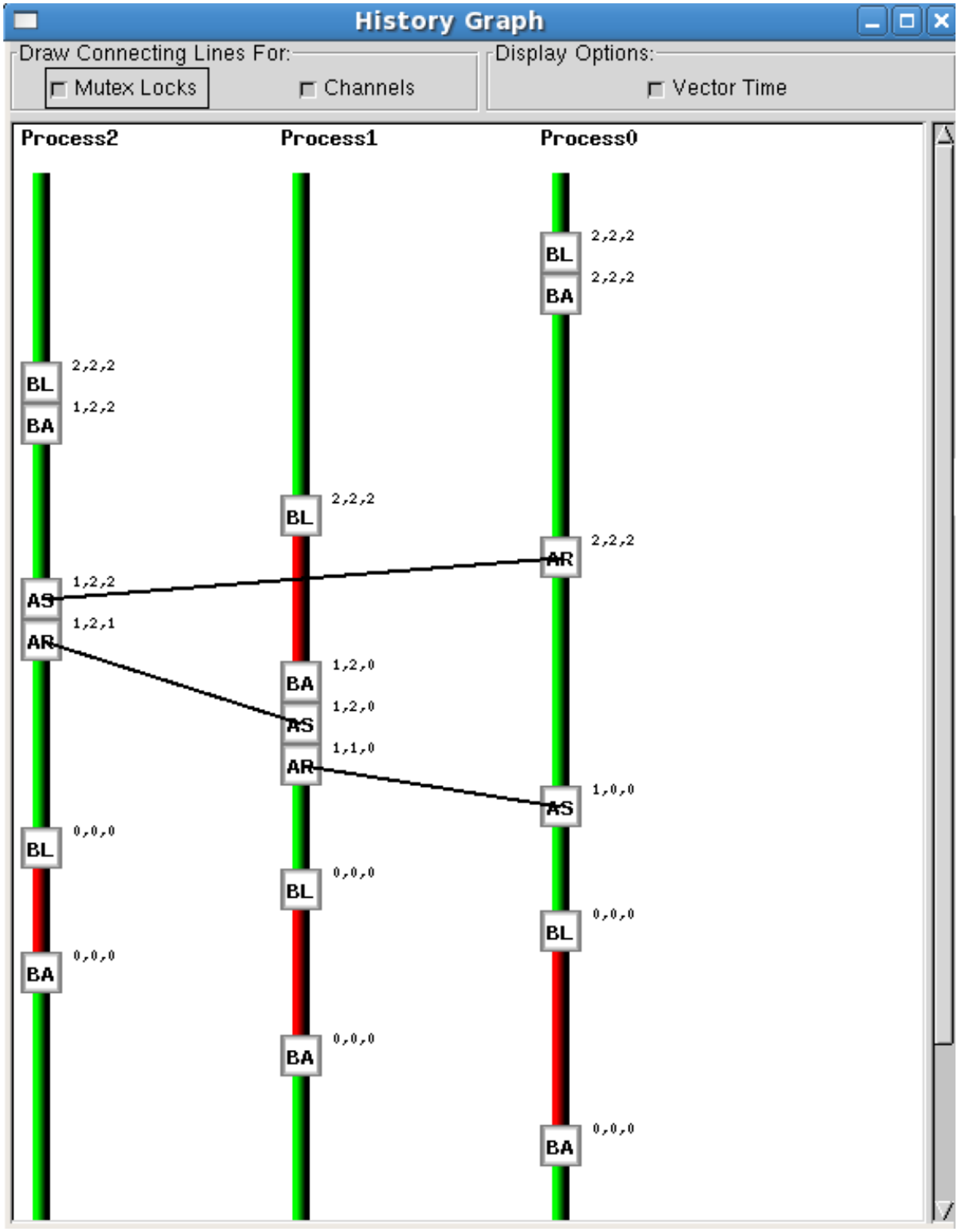

Figure 7: History diagram for barrier wait

two processes cannot have more than one such channel between them, processes cannot have common topologies in between them. For the processes that are in a topology, directed connected ones have an implicit asynchronous channel between them and have to use the send/receive functions provided by topology. Various types of topologies are supported.

General Topology: A topology is created based on user supplied topology map.
Constructor:

```
Topology(int **topmap);
```

`topmap` is a pointer to dynamically allocated 2D array which contains a map of the topology. A link between process i and j exists if `topomap[i][j]` is greater than 0. The map must correspond to a connected graph.

Fully Connected Topology: Each process is connected to every other process.
Constructor:

```
FullyConnectedTopology();
```

Linear Topolopgy
Constructor:

```
LinearArrayTopology();
```

Ring Topology
Constructor:

```
RingTopology();
```

Star Topology
Constructor:

`StarTopology(int center = 0);` `center` : The id of center process. Default value is 0.

Grid Topology
Constructor:

```
GridTopology(int row, int column);
```

`row` : The total number of rows in the topology.
`column` : The total number of columns in the topology.
`rows * columns` must be equal to the total number of processes

Torus Topology
Constructor:

```
TorusTopology(int row, int column);
```

`row` : The total number of rows in the topology.
`column` : The total number of columns in the topology.
`rows * columns` must be equal to the total number of processes

The following macros are defined in the message passing library:

`CM_UP`: The north neighbor of a process if exists.

`CM_DOWN`: The south neighbor of a process if exists.

`CM_LEFT`: The left neighbor of a process if exists.

`CM_RIGHT`: The right neighbor of a process if exists.

`CM_ANY`: Any directly connected neighbor of a process.

9.1 Communication methods in a topology

A topology class provides various communication methods.

9.1.1 Topology send

```
int tsend(int destId, void *sendbuf, int size);
```

Description:
This function provides a way to send message to a directly connected process. The behavior is same as asynchronous channel send.

Arguments:
`destId` : The process id of the destination process.
`sendbuf` : The pointer to the message buffer.
`size` : The total number of bytes to send. Must be greater than or equal to 0.

Preconditions:
Same as asynchronous channel.

Return value:
Same as asynchronous channel.

```
// tsend.cpp
// To demonstrate topology send and receive operations
#include <iostream>
#include "cm.h"

using namespace std;

int myId, numProcs;
int main()
{
        int **map;
        getMyId(&myId);
        getNumProcs(&numProcs);


        map = new int *[numProcs];
        for (int i = 0; i < numProcs; i++)
```

```
                map[i] = new int[numProcs];

        for (int i = 0; i < numProcs; i++)
                for (int j = 0; j < numProcs; j++)
                        map[i][j] = 1;

        Topology * t2 = new Topology(map);

        if(myId == 0){
                char * sndbuf = "mydata";
                int length = strlen(sndbuf);
                t2->tsend(1, sndbuf, length+1);
        } else if(myId == 1){
                char recvbuf[10];
                int length = 7;
                t2->treceive(0, recvbuf, length);
                cout<<myId<<": Data received: "<<recvbuf<<endl;
        }

        return 0;

}
```

Compile this program and run it as follows:

```
g++ -o tsend tsend.cpp -lcm

cmrun -np 5 tsend
```

The following output is observed at the terminal:

```
1: Data received: mydata
```

9.1.2 Topology receive

```
int treceive(int sourceId, void *sendbuf, int size)
```

Description:
This function provides a way to receive message from a directly connected process. The behavior is same as asynchronous channel receive.

Arguments:
`sourceId` : The process id of the destination process. If source is equal to CM_ANY, then message from any of the directly connected channel is received.
`sendbuf` : The pointer to the message buffer.
`size` : The total number of bytes to send. Must be greater than or equal to 0.

Preconditions:
Same as asynchronous channel.

Return value:
Same as asynchronous channel.

9.1.3 Topology broadcast

`int broadcast(int rootID, void *sendbuf, int ssize, void *recvbuf, int rsize)`

Description:
Message is broadcast from the originator to all the processes in the topology. All the processes in the topology need to call this function

Arguments:
`rootID` : The process id of the process broadcasting.
`sendbuf` : The pointer to the message buffer being broadcast. This applies to root process only.
`ssize` : The total number of bytes being sent. This applies to root only.
`recvbuf` : The pointer to the buffer to store the received message. This applies to all processes.
`rsize` : The number of bytes to receive. This applies to all processes.

Preconditions:
Topology must be existent. The value of rootID must be same in all processes. The required buffers should not be NULL and should point to a valid block of memory.

Return value:
On success, the number of bytes broadcast is returned for the root process and number of bytes received for their processes. On error the return value is -1.

```
// topob.cpp
// To demonstrate the topology broadcast operation
#include <iostream>
#include "cm.h"

using namespace std;

int myId, numProcs;
int main()
{
        int recvbuf;
        int sndbuf;
        getMyId(&myId);
        getNumProcs(&numProcs);

        LinearArrayTopology t;

        recvbuf = -1;
```

```
        if(myId == 0){
                sndbuf = 55;
        }
        t.broadcast(0, &sndbuf, 4, &recvbuf, 4);
        cout<<myId<<": Data received: "<<recvbuf<<endl;

        return 0;

}
```

Compile this program and run it as follows:

```
g++ -o topob topob.cpp -lcm
```

```
cmrun -np 5 topob
```

The following output is observed at the terminal:

```
3: Data received: 55
2: Data received: 55
0: Data received: 55
1: Data received: 55
```

9.1.4 Topology gather

```
int gather(int rootID, void *sendbuf, int ssize, void *recvbuf, int rsize)
```

Description:
The root process receives message segments from all processes, including itself, and constructs the final message. The arrangement of final message is based on ascending order of processes ids.

Arguments:
`rootID` : The process id of the process gathering.
`sendbuf` : The pointer to the message buffer being sent. This applies to all processes.
`ssize` : The total number of bytes being sent. This applies to all processes.
`recvbuf` : The pointer to the buffer to store the received message. This applies root process only.
`rsize` : The number of bytes to receive. This applies to root process only.

Preconditions:
Topology must be existent. The value of rootID must be same in all processes. The required buffers should not be NULL and should point to a valid block of memory.

Return value:
On success, the number of bytes sent to root is returned for the non root processes. The total number of bytes received is returned for the root process. On error the return value is -1.

```
// topog.cpp
```

```
// To demonstrate the topology gather operation
#include <iostream>
#include "cm.h"

using namespace std;

int myId, numProcs;
int main()
{
        int sndbuf;
        int * source = NULL;
        getMyId(&myId);
        getNumProcs(&numProcs);

        if(numProcs != 4){
                if(myId == 0)
                        cout<<"Error: Number of processes must be equal to 4"<<endl;
                exit(-1);
        }

        GridTopology t(2, 2);

        sndbuf = myId * 100;
        if(myId == 0){
                source = new int[numProcs];
                for(int i = 0; i < numProcs; i++)
                        source[i] = 0;
        }

        t.gather(0, &sndbuf, 4, source, 4*numProcs);

        if(myId == 0){
                cout<<myId<<": Data gathered: "<<endl;
                for(int i = 0; i < numProcs; i++)
                        cout<<source[i]<<endl;
        }

        delete [] source;

        return 0;

}
```

Compile this program and run it as follows:

```
g++ -o topog topog.cpp -lcm

cmrun -np 4 topog
```

The following output is observed at the terminal:

```
0: Data gathered:
0
100
200
300
```

9.1.5 Topology scatter

```
int scatter(int rootID, void *sendbuf, int ssize, void *recvbuf, int rsize)
```

Description:
The root process divides the given message into N (N: number of processes in the topology) unique chunks and sends the chunks to respective processes, including itself.

Arguments:
`rootID` : The process id of the process scattering.
`sendbuf` : The pointer to the message buffer being sent. This applies to root process only.
`ssize` : The total number of bytes being sent. This applies to root process only.
`recvbuf` : The pointer to the buffer to store the received message. This applies to all processes.
`rsize` : The number of bytes to receive. This applies to all processes.

Preconditions:
Topology must be existent. The value of rootID must be same in all processes. The required buffers should not be NULL and should point to a valid block of memory.

Return value:
On success, the return value is total bytes sent in root process and number of bytes received in other processes. The return value is -1 in case of errors.

```
// toposc.cpp
// To demonstrate the topology scatter operation
#include <iostream>
#include "cm.h"

using namespace std;

int myId, numProcs;
int main()
{
        int recvbuf;
        int * source = NULL;
        getMyId(&myId);
        getNumProcs(&numProcs);
```

```
        if(numProcs < 2){
                if(myId == 0)
                        cout<<"Number of processes must be greater than 2"<<endl;
                exit(-1);
        }

        StarTopology t(1);

        recvbuf = -1;
        if(myId == 0){
                source = new int[numProcs];
                for(int i = 0; i < numProcs; i++)
                        source[i] = 100*i;
        }

        t.scatter(0, source, 4, &recvbuf, 4);

        cout<<myId<<": Data received: "<<recvbuf<<endl;

        return 0;

}
```

Compile this program and run it as follows:

```
g++ -o toposc toposc.cpp -lcm
```

```
cmrun -np 5 toposc
```

The following output is observed at the terminal:

```
2: Data received: 200
0: Data received: 0
3: Data received: 300
4: Data received: 400
1: Data received: 100
```

9.1.6 Topology reduce

```
int reduce(int rootID, void *sendbuf, void *recvbuf, int size, void *(*reduce_function)
(void *, void *)
```

Description:
This function applies user defined global reduction function to the messages received from all processes.

Arguments:
`rootID` : The process Id responsible for doing the reduce operation. `sendbuf` : The pointer to the send buffer. Applies to all processes. `recvbuf` : The pointer to the receive buffer. Applies to root only. `size` : The number of bytes to send. `reduce_function` : The user defined reduction function.

Preconditions:
Topology must be existent. The value of rootID must be same in all processes. the required buffers should not be NULL and should point to a valid block of memory.

Return value:
On success, total number of bytes sent to the root process is returned. On error , the return value is -1.

```
// topore.cpp
// To demonstrate the topology reduce operation
#include <iostream>
#include "cm.h"

using namespace std;

void *reducefunc(void *a1, void *a2)
{
        int arg1 = *(int *) a1;
        int arg2 = *(int *) a2;

        int *result = new int;

        *result = arg1 + arg2;
        return result;

}

int myId, numProcs;
int main()
{
        int sndbuf;
        int recvbuf;
        getMyId(&myId);
        getNumProcs(&numProcs);

        FullyConnectedTopology t;

        sndbuf = myId * 100;
        if (myId == 0) {
                int recvbuf = -1;
        }

        t.reduce(0, &sndbuf, &recvbuf, 4, reducefunc);
```

```
        if (myId == 0) {
                cout << myId << ": Data reduced: " << recvbuf<<endl;
        }
        return 0;
}
```

Compile this program and run it as follows:

```
g++ -o topore topore.cpp -lcm
```

```
cmrun -np 6 topore
```

The following output is observed at the terminal:

```
0: Data reduced: 1500
```

10 Visualization System

The function of this system is to visualize the runtime behavior of user programs. There are two modes: realtime visualization invoked automatically by the control process or standalone playback invoked from the command line. Realtime mode generates displays as it receives information from the control process while the playback mode does so by reading previously saved visualization data from a file. Realtime visualization can further be divided into two types: Interactive and Non-interactive. The execution mode of visual system is specified during startup and it cannot be changed later. The interactive feature is not available in standalone playback mode. Visualization system is disabled if `cmrun` is started with a `-nv` option.

The main window of visualization system can be divided into following different portions:

Menus: Provides options to save the visualization related data to a file, or load visualization data from a file if in standalone mode, display the online help from ConcurrentMentor website, and display information about Graph/Diagram tags.

List view selection buttons: Selects the appropriate information to display in embedded list.

Non list view selection buttons: Opens a new window for each option.

Embedded list: A display area where the user can view various information depending upon selection made.

Interactive visualization selection: To enable or disable interactive visualization and to configure stop points.

Interval selection bar: Provides the user to specify the interval between reading two consecutive visual message.

Control buttons: Decide the state of visualization system. There are three buttons: Step, Start/Resume, Pause. Start button appears during startup and user has to click it in order to start the computation. After clicking, Start button changes to Resume. Step button is used to read a single visual message each time it is clicked. Pause stops reading visual messages. Clicking Resume after Step or Pause restores the default behavior of visualization system, i.e to poll for visual message within the given timeout.

Figure 1 depicts various portions of the main visualization window.

10.1 History Diagram

As discussed earlier, a history diagram shows the the states of all processes and various events of interest like send, receive etc. for every processes. Each process is represented by an event line which is green when the process is running normally and red when the process is blocked. The events are depicted by an event tag, with the name of the event, in the event line. There are options to display the vector time corresponding to all events, and to draw connections between events in different processes. The event tags are listed below:

MW - Mutex wait

ML - Mutex lock

MU - Mutex unlock

BW - Barrier wait

BL - Barrier leave

ES - Enter synchronous channel send

LS - Leave synchronous channel send

ER - Enter synchronous channel receive

LS - Leave synchronous channel receive

AS - Asynchronous channel send

AR - Asynchronous channel receive

Clicking on each event brings up a window highlighting the line of source code of the user program which generated that event. For this the source code should be in the current working directory, i.e the directory from where `cmrun` is invoked.

10.2 Space Time Diagram

As discussed earlier, a Space Time diagram shows the causal relationship between the events of different processes. Each process is represented by an event line which in turn consists of the events of interest. There are options to connect the corresponding send and receive events, display the event labels, and display the vector time. The following are the events of interest in this diagram:

AS - Asynchronous channel send event

AR - Asynchronous channel receive event

SS - Synchronous channel send event

SR - Synchronous channel receive event

The events can be slided up and down in the event line as long as the causal relations with other events are not violated.

11 Interactive Visualization

Interactive Visualization mode allows a user to specify stop points at which the user programs block for user input. Stop points are like break points where the user programs wait for user input. Those lines in the program source file where synchronous send/receive and asynchronous send/receive operations are used can be configured as stop points. This mode is selected by clicking in the **Enable interactive visualization** check box before clicking the start button. This feature is not available in the standalone mode.

Users have an option to configure stop points. To configure stop points, click on **Configure stop points** check box after which a window containing a list stop points pops up. The list may not be empty if configure had been used earlier. Each entry contains filename, line number, and comma separated process ids. An entry of the form `file.cpp 12 0,1,2` would imply that line number 12 in file.cpp should be treated as a stop point for processes 0, 1, 2. The values specified here can be arbitrary, and will be ignored if there is no match. If -1 is specified as a line number, every possible points are stop points for those processes.

There is also an option make the user programs stop at every possible stop points. For this **Stop at all possible points** check box should be selected. If this option is selected along with configure stop points option, then this selection has no effect on the behavior of visual system.

11.1 Interactive diagram

Interactive diagram is the history diagram in the interactive visualization mode. It is similar to the non-interactive mode history diagram, but has more features. Like history diagram, it displays the state of all processes and various events of interest. In addition to that, visualization waits for user input if the appropriate line in source file, causing synchronous send/receive and asynchronous

send/receive events, had been configured as a stop point. Those events are displayed in red color and the processes are blocked at that point. To resume execution of a blocked process, right click on the stopped event.

This diagram has a button **Release All** which releases all the stop points and causes the visualization system to continue without interactive functionality.

The interactive functionality can be used to force the computation along a path. As depicted in the example that follows, two processes create two mutexes and try to lock them in opposite order. The programs could deadlock, but there are some paths that could avoid a potential deadlock.

```
// interactive.cpp
// Program to test interactive system
#include <iostream>
#include <string.h>
#include <unistd.h>
#include "cm.h"

using namespace std;

int main(int argc, char *argv[])
{
        int myId, numProcs;
        int neighbor;
        int val = 0xdabe;
        int rcvbuf = 0;

        getMyId(&myId);
        getNumProcs(&numProcs);
        if(numProcs != 2){
                cout<<"Number of processes must be 2 for this case"<<endl;
                exit(0);
        }

        neighbor = (myId + 1) % numProcs;
        AsynChannel *ch1 = new AsynChannel(neighbor);
        if(!ch1->isValid()){
                cout<<"Couldn't create channel"<<endl;
                exit(0);
        }

        DistMutex *m1 = new DistMutex("goodmutex1");
        DistMutex *m2 = new DistMutex("goodmutex2");

        ch1->send(&val, sizeof(int));
        if(myId == 0)
                m1->lock();
        else
```

```
                m2->lock();

        val = 0xdead;
        ch1->send(&val, sizeof(int));
        if(myId == 0)
                m2->lock();
        else
                m1->lock();

        if(myId == 0){
                m2->unlock();
                m1->unlock();
        }
        else{
                m1->unlock();
                m2->unlock();
        }

        ch1->receive(&rcvbuf, sizeof(int));
        ch1->receive(&rcvbuf, sizeof(int));

        return 0;
}
```

Compile this program and run it as follows:

```
g++ -o interactive interactive.cpp -lcm
```

```
cmrun -np 2 interactive
```

There is no output observed at the terminal. For the interactive part, as soon as the visual window pops up, first select "Enable interactive visualization" and "Stop at all possible points" and then click start. The interactive view of first run is depicted in figure 8 in which the two processes are allowed to deadlock. For the second run, the interactive view of which is shown in figure 9, process 1 is allowed to get both the mutex locks while process 0 blocks for input from the user. After process 1 releases both the mutexes, process 0 is released by right clicking the blocked event. It can be observed that a different path could be forced using the interactive system.

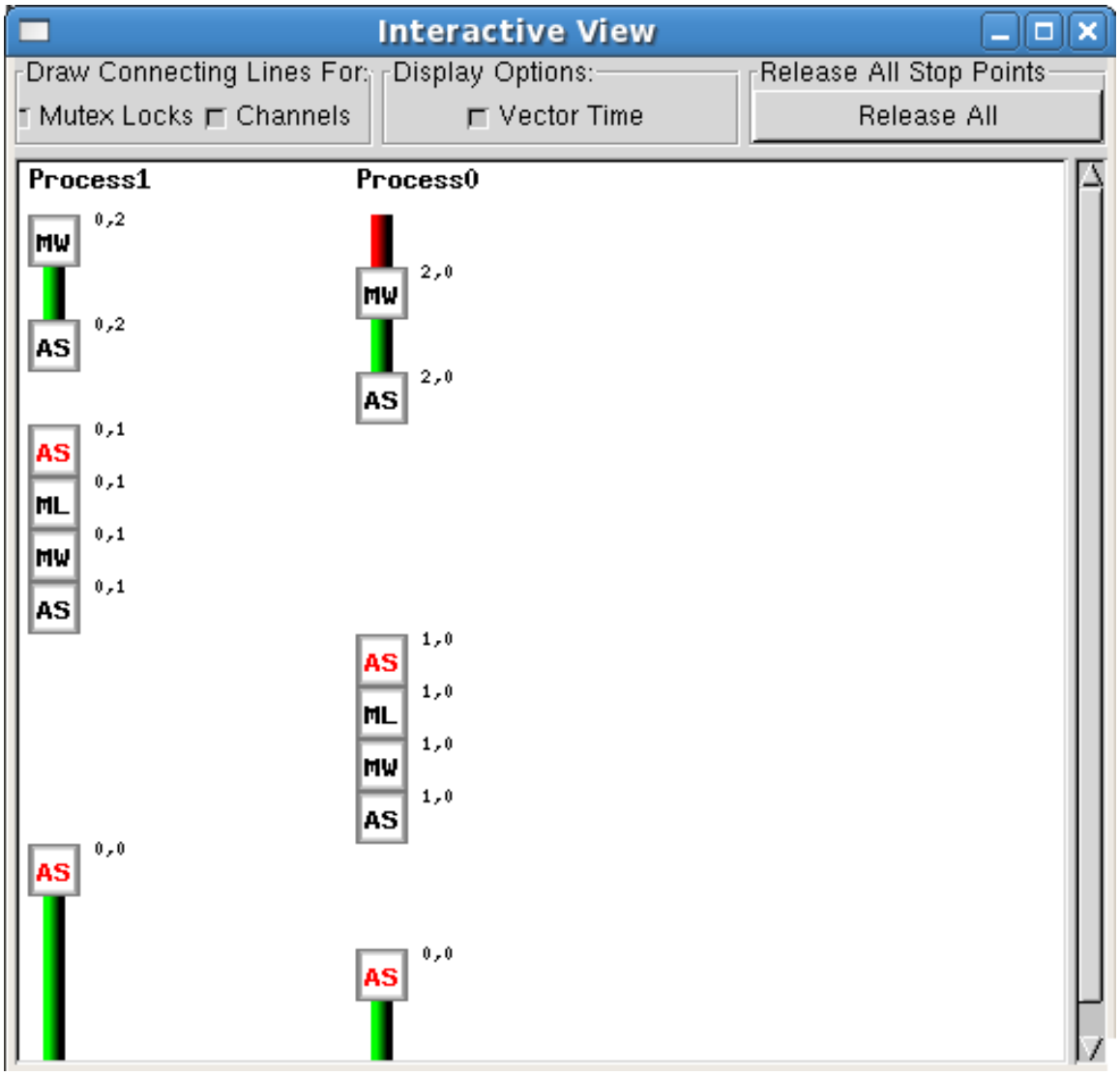

Figure 8: Interactive view: with two deadlocked processes

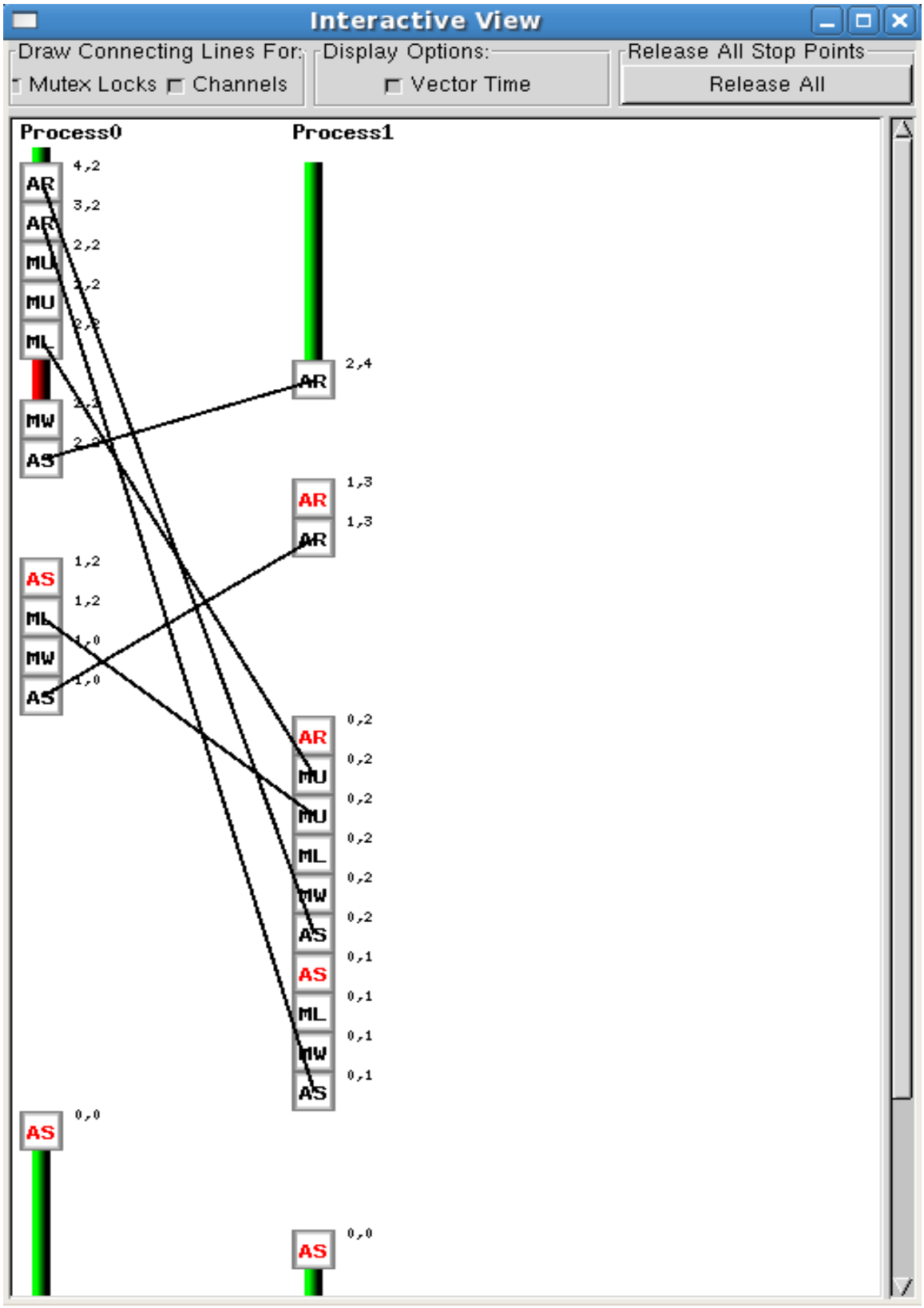

Figure 9: Interactive view: avoiding potential deadlock

12 Miscellaneous

1. If control process crashes in some unavoidable circumstances, the lock file `$HOME/.concurrentmentor` will not be deleted. This will prevent running further instances of ConcurrentMentor. If any error is displayed mentioning the above mentioned lock file, that file has to be deleted. Before deleting a lock file manually, the message queue has to be cleaned up. The id of the message queue is stored in the lock file.

2. When using remote hosts for running user programs, they must be checked for runaway processes if control process crashes.