HY486: Principles of Distributed Computing

Spring Semester 2025
Second Programming Assignment
Deadline: 14/6/2025

1. General Description

In the second programming assignment, you are asked to implement a distributed library management system that handles loaning and returning events from borrowers. The programming assignment must be implemented in the C language using the Message Passing Interface (MPI), which is installed on the department's machines.

2. Implementation

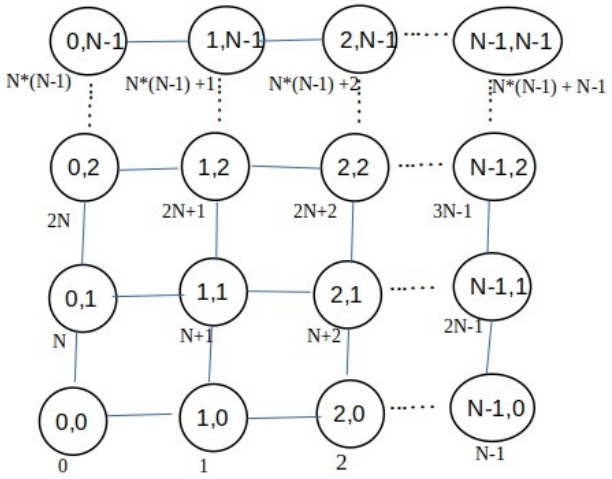
In this assignment, you must implement a library management system. The system consists of library processes (servers) and borrower processes (clients). Each borrower process may request to register or monitor events in a specific library process to which it is assigned. Library processes are responsible for executing these operations. For simplicity, we assume that each borrower process is assigned to exactly one library.

Based on the above, the system must be structured as follows:

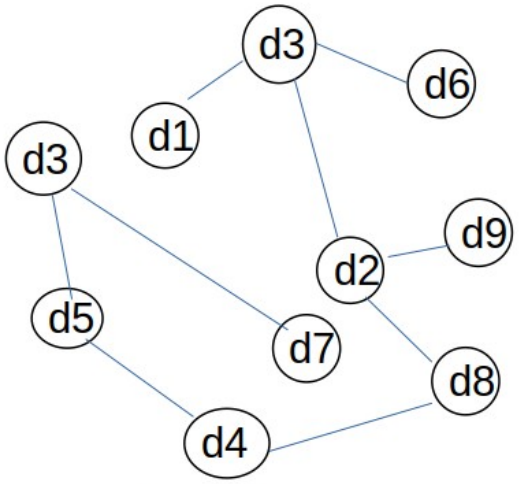
- There will be borrower processes requesting to perform actions and library processes serving these requests.
- The process with identifier (MPI rank) 0 will act as the coordinator, whose main purpose is to read a testfile containing the description of the events to be simulated by the other processes and inform them accordingly. Thus, the coordinator process is neither a library process nor a borrower process. The structure of the testfile and the events are described in detail later.

The library nodes create a logical network in the form of an NxN grid, as shown in Figure 1. Thus, there are N^2 library processes in the system. The node at coordinates <x,y> in the grid has one or two neighbors in its row and one or two neighbors in its column. The node at <0,y> has no left neighbor, while its right neighbor is at <1,y>. Similarly, the node at <N-1,y> has no right neighbor, and its left neighbor is at <N-2,y>. Additionally, nodes at <x,0> have no lower neighbors, and nodes at <x,N-1> have no upper neighbors. If a node has two neighbors in its row, the left neighbor is at <x-1,y>, and the right neighbor at <x+1,y>. Likewise, if a node has two neighbors in its column, the upper neighbor is at <x,y+1>, and the lower neighbor at <x,y-1>.

Moreover, for every i,j where $0 \leq i,j \leq N-1$, the library at position (i,j) corresponds to the library process with identifier l_id = N*j + *i. There are $N^3/2$ borrower (Rounded down to the nearest whole number) processes with identifiers c_id = N^2+j, for each j, $0 \leq j \leq (N^3/2)-1$. Each library has N/2 borrowers assigned to it. Specifically, the library with identifier k, $0 \leq k \leq N^2-1$ has the following borrowers: N^2 + k*N/2,… , N^2 + (k+1)*N/2. (The definition of N is described in Chapter 4.)



Libraries
Figure 1a



Borrowers
Figure 1b

In order to implement this grid, each library node must store information about its neighboring nodes in the grid. An important part of the exercise is the implementation of a message routing algorithm within the grid of library nodes. Additionally, each library process stores information about the books it possesses, which books have been loaned, and how many times each book has been loaned.
For this purpose, each library process l_id implements a collection of books (a serial data structure) containing all the books that have ever been in the library (even temporarily), e.g., a simple linked list. Each book has 2 attributes: the book's identifier (id) and its cost.

It is assumed that there are $M = N^4$ books in total and that each of the N^2 libraries has N different books, each available in N copies. Thus, each library has N^2 books, but only N distinct books among them.
The library with identifier i, $0 \leq i \leq N^2-1$, has the books with identifiers b_id = i * N , …, (i+1)*N – 1
Each library must be initialized with the corresponding books and the correct number of copies.
For the cost, use a random number between 5 and 100.

Each borrower must store information about which books they have borrowed and how many times they have borrowed each one.
To implement this, each borrower process must implement a collection in a similar manner as the library processes.
The graph of borrowers (Figure 1b) is a tree and is formed based on the CONNECT-type events described in section 2.2.

The system will support the following events (as specified in the testfile):

1. Creation of borrower process tree connections (CONNECT).
2. Loaning a book by a borrower (takeBook).
3. Donating books by a borrower to its library (donateBook).
4. Finding the most popular book (getMostPopularBook).
5. Checking that no books have been lost (checkNumberOfBooksLoaned).

2.1 Initialization

At system startup, a routine called `MPI_Init()` is executed, which initializes all the processes that will run in the system (see section 4 for system initialization details).

2.2 Creating the Borrower Process Tree

The borrower processes connect with each other to form a tree (Figure 1b). The tree is defined by the CONNECT-type events described in the testfile.
The coordinator process begins by reading the testfile, which initially describes the connections between the processes in order to form the borrower tree (without specifying any node as the root).

Specifically, these events have the following format:

CONNECT <client_id1> <client_id2>

Upon this event, the coordinator first sends a CONNECT-type message to the process with identifier <client_id1> to inform it that its neighbor in the tree is the process with identifier <client_id2>. The process with identifier <client_id1> records the process with <client_id2> as its neighbor and then sends it a NEIGHBOR-type message to inform it that it is its neighbor.
The process with identifier <client_id2> records it as one of its neighboring nodes and sends back an ACK message. When process <client_id1> receives the ACK, it sends an ACK message to the coordinator to inform it that the event has been completed.

For each of these CONNECT-type messages, the coordinator must wait for an **ACK** message from the process to which it sent the message, before processing the next event in the testfile. This ensures that all borrower processes know their role and connectivity in the borrower tree, so the system simulation can start smoothly, as described in the events following the CONNECT-type events in the testfile.

2.3 Leader Election Among Library Processes

Next, the testfile contains an event of the type `<START_LE_LIBR>`. When the coordinator reads such an event in the testfile, it sends a `<START_LEADER_ELECTION>` message to every library process in the system and then waits to receive a `<LE_LIBR_DONE>` message from the process elected as leader.
The leader election algorithm among library processes is as follows:
Constructing a DFS Spanning Tree without a Specified Root (page 14, section 7 of the slides).

Summary of the algorithm:

- Each process attempts to create a DFS spanning tree with itself as the root, using a different copy of the tree.
- If two trees attempt to connect to the same node, the node will connect to the tree whose root has the highest identifier (rank).
- Thus, eventually, a tree will be created with the process with the highest identifier as the root, which will be the leader.

The library process elected as leader must inform all other library processes that it has been elected leader and then send a `<LE_LIBR_DONE>` message to the coordinator to inform it of the end of the leader election phase and who the leader is.
The coordinator stores this leader.

2.4 Leader Election Among Borrower Processes

Immediately after the <START_LE_LIBR> event, the testfile contains an event of the type <START_LE_LOANERS>.
To process such an event, the coordinator sends a <START_LE_LOANERS> message to every borrower process in the system and then waits to receive a <LE_LOANERS_DONE> message from the process elected as leader.

The leader election algorithm among borrower processes is the **STtoLeader** (page 6, section 9 of the slides).
When a borrower process with identifier client_id receives a <START_LE_LOANERS> message from the coordinator, it starts the leader election algorithm by sending an <ELECT> message to each of its neighboring nodes.

STtoLeader algorithm

- The leaves of the tree (i.e., nodes with only one neighbor) start a convergecast using <ELECT> messages.
  1. Each leaf can initially send an <ELECT> message to its only neighbor.
  2. Each node that has received <ELECT> messages from all its neighbors except one can send an <ELECT> message to the remaining neighbor.
- Eventually, one of the following conditions will occur:
  1. A process receives <ELECT> messages from all its neighbors before sending any <ELECT> message itself. This process is elected as leader.
  2. <ELECT> messages have been sent in both directions of a specific edge. The process with the highest rank among those connected to this edge is elected as leader.

The borrower process elected as leader must inform all other borrower processes that it has been elected leader by sending them a <LE_LOANERS> message containing its rank(leader rank), and then send a <LE_LOANERS_DONE> message to the coordinator to inform it of the end of the leader election phase and who the borrower leader is.
The coordinator stores this leader.

3. Events

The library leader is responsible for locating books when a library does not have a specific book, but one of its borrowers requests it. To do this, it maintains an array of N^4 elements, mapping each book to the library that holds it. When it is requested to locate a book with identifier b_id, it accesses the corresponding element in the array to find the library where the book resides.

1. TakeBook <c_id> <b_id>

This event signifies that the borrower process with identifier c_id wants to borrow a book.

When the coordinator reads such an event in the testfile, it notifies the borrower process c_id. The borrower process c_id requests to borrow book b_id from the library process l_id to which it is assigned (l_id must be computed algorithmically). To achieve this, the borrower process c_id sends a LEND_BOOK message to the library process with identifier l_id.

When the library process l_id receives the message, it checks if the book is in its collection.

- If yes, it sends a `<GET_BOOK>` message to borrower c_id, simulating the book's delivery.
- If not, it sends a message of the type:

FIND_BOOK <b_id>

to the library leader process, to locate the identifier l_id` of the library that has the book. When the library leader receives the message, it returns the identifier l_id` of the library process that has the book b_id, calculating l_id` using appropriate operations.

Upon receiving the leader's response, the borrower process sends a message to the library process with identifier l_id`:

BOOK_REQUEST <b_id>, <c_id>, <l_id>

The library with identifier l_id` responds to the requesting library l_id with an **ACK_TB <b_id>** message, indicating it found the book. Then, the library l_id replies to borrower c_id with an **ACK_TB <b_id>** message. After receiving the **ACK_TB <b_id>** message, the borrower updates its collection statistics, indicating it has borrowed the book b_id and increments the counter showing how many times it has borrowed this book. The same update is done by the library l_id process.

Finally, the borrower process sends the coordinator a `<DONE_FIND_BOOK>` message to indicate the completion of the FIND_BOOK event.

2. DonateBook <c_id>, <b_id>, <n_copies>

This event signifies the donation of a book with identifier b_id by borrower process c_id, in n_copies number of copies.
When the coordinator reads such an event in the testfile, it notifies borrower process c_id. The borrower c_id donates the book b_id. To do this, it sends a `<DONATE_BOOK>` message to the borrower leader process, which then distributes all the copies (n_copies) among the libraries in a round-robin manner, maintaining a variable donate_next, initially set to 0.

Every time it receives a `donateBook` request, it distributes the books to libraries with identifiers:

donate_next % N^2 , (donate_next + 1) % N^2 ,…, (donate_next + n_copies) % N^2

Each library with these identifiers receives one copy of the book. If there are more copies than libraries, the round-robin repeats, again updating donate_next accordingly.

Then, each library responds with an `ACK_DB` message to the borrower leader to confirm receipt of the book b_id and updates its statistics, indicating the new book is available for loan.

3. GetMostPopuralBook

This event signals the retrieval of the most popular book per borrower group in each library l_id, i.e., the book that has been borrowed the most times by a single borrower, not cumulatively across all borrowers. Each library l_id corresponds to a specific group of borrowers as described earlier.

When the coordinator reads such an event in the testfile, it informs the borrower leader with a <GET_MOST_POPULAR_BOOK> message. Upon receiving this message, the borrower leader must determine the most popular book at the borrower level.
To achieve this, it performs a broadcast through the borrower tree, followed by a convergecast to gather from each borrower the most popular book they have borrowed for their library.

Each borrower identifies the book b_id they have borrowed the most times and sends this information to the borrower leader via a message:

<GET_POPULAR_BK_INFO> <b_id> <times_loaned> <l_id>

This message contains the book's id, how many times it has been borrowed, and to which library l_id it belongs (l_id must be computed algorithmically).

The borrower leader responds with an ACK_BK_INFO message to each borrower. After receiving all messages from the borrowers, it identifies the most popular book, i.e., the one borrowed the most times among the borrowers assigned to library l_id. In case of a tie, the book with the highest cost (based on the cost field) prevails.

Finally, the borrower leader sends the coordinator a <GET_MOST_POPULAR_BOOK_DONE> message to indicate the completion of the FIND_BOOK event.

4. CheckNumBooksLoaned

This event verifies the correctness between the number of books loaned by the libraries and the number of books borrowed by the borrowers. When the coordinator reads such an event in the testfile, it sends a

<CHECK_NUM_BOOKS_LOAN>

message to both the library leader and the borrower leader. The library leader seeks to find how many books each library has loaned. To do this, it sends a <CHECK_NUM_BOOKS_LOAN> message to library 0, which forwards the same message to the next library, and so on, until all libraries have received the message.

Specifically, library 0 initiates the message routing through the library grid in the following sequence:

<0,0>, <1,0>, <2,0>, …, <N-1,0>, <N-1,1>,…,<2,1> , <1,1> ,<0,1>, <0,2>, <1,2>, <2,2>,..,<N-1,2>,…

(i.e., the message traverses the grid row by row).
Each node must know to which node to forward the message.

Each library iterates through its collection and counts the total number of loaned books. Then, it sends a message to the library leader:

<NUM_BOOKS_LOANED> <num_books_loan>

containing the number of books loaned by that specific library l_id. Upon receiving the message, the library leader sends an ACK_NBL message to the library.

After receiving all <NUM_BOOKS_LOANED> messages, the library leader sums the loan numbers for each library and sends a <CHECK_NUM_BOOKS_LOAN_DONE> message to the coordinator.

Similarly, the borrower leader performs the corresponding actions on the borrower side: It executes a broadcast through the borrower tree to find the total number of book loans from all borrowers, followed by a convergecast to inform the borrower leader.

At the end, the number of loans recorded by the borrowers must match the number of loans recorded by the libraries.

Additionally, the library and borrower leaders print the number of loans as:

`Loaner Books <b1>`

`Library Books <b2>`

Where `b1` is the number of loans from the borrowers' side and `b2` from the libraries' side. If the numbers do not match, it prints:

`CheckNumBooksLoaned FAILED`

`CheckNumBooksLoaned SUCCESS`

4. Message Passing Interface (MPI)

For the implementation of this assignment, you must use the Message Passing Interface (MPI) library. This library provides the capability for message exchange between processes (which may run on the same or on different machines) via a standardized Application Programming Interface (API), regardless of language or implementation.

For the required simulation, each system node will be simulated by an MPI process. It is noteworthy that more than one MPI process can run on the same machine (although they simulate different system nodes).

The MPI system is installed on the Department's machines.
The two basic commands required for its operation are `mpicc` and `mpirun`.

- The `mpicc` command is used to compile programs that use the MPI API.
- With the `mpirun` command, you can run the executable simultaneously on multiple machines.

An example of usage is provided below:

`$ mpirun -np <count> --hostfile <file containing hostnames> <executable> <NUM_LIBS>`

Where the <np> option is equal to the sum:

`N^2 + N^3/2 + 1`

and defines the total number of MPI processes that will execute the executable given as the last argument in the above command line. In the `<hostfile>` option, you provide the name of the file containing the hostnames of the machines where the MPI processes will be executed. Each process started with the `mpirun` command has a unique identifier in the MPI system, called **MPI rank**. It can also know how many other MPI processes are running in the system and send them messages using their rank as an identifier. It is important to note that a process can only send messages to processes that were started from the same `mpirun` command and are thus members of the same **MPI world**.

For more information, you can read the material available at the following link:

https://mpitutorial.com/tutorials

Additionally, you will need to use the function `MPI_Type_create_struct`, information about which can be found at the following link:

https://www.mpich.org/static/docs/v3.1/www3/MPI_Type_create_struct.html

5. Assignment Submission

The assignment must be built and function correctly on the Department's machines.

Your submission must include:

- The collection of C source/header files where you implement your code.
- Your own Makefile for building the project.

It is also recommended to organize your implementation into multiple files.

The assignment must be submitted via the **elearn** system.