Assignment # 1

There are two parts to this assignment: theory and programming. The theory assignment should be done individually. The programming assignment may be done in teams of two or individually.

Deadline:
Theory – in class February 7. Programming – February 4, 10 PM [Submission through Canvas]

1. **(10 points)** (a) Prove or disprove that every symmetric and transitive relation is reflexive. (b) Prove or disprove that every irreflexive and transitive relation is asymmetric.

2. **(10 points)** Prove the following for vector clocks: $s \rightarrow t$ iff

$$(s.v[s.p] \leq t.v[s.p]) \wedge (s.v[t.p] < t.v[t.p])$$

3. **(10 points)** Some applications require two types of accesses to the critical section—*read* access and *write* access. For these applications, it is reasonable for multiple *read* accesses to happen concurrently. However, a *write* access cannot happen concurrently with either a *read* access or a *write* access. Modify Lamport's mutex algorithm for such applications.

4. **(10 points)** (a) Extend Lamport's mutex algorithm to solve k-mutual exclusion problem which allows at most k processes to be in the critical section concurrently. (b) Extend Ricart and Agrawala's mutex algorithm to solve the k-mutual exclusion problem.

5. **(60 points)** The goal of this assignment is to learn client server programming with TCP and UDP sockets. You are required to implement a server and a client for a ticket reservation system for a movie. The system should function with both TCP as well as UDP connections. Assume that the movie theater has c total seats. There is a single server, but multiple clients may access the server concurrently. Assume that a person can reserve only one seat at any given time.

   The server accepts only the following calls from a client:

   1. `reserve <name>` – inputs the name of a person and reserves a seat against this `name`. If the theater does not have enough seats(completely booked), no seat is assigned and the command responds with message: `'Sold out - No seat available'`. If a reservation has already been made under that name, then the command responds with message: `'Seat already booked against the name provided'`.
   Otherwise, a seat is reserved against the name provided and the client is relayed a message: `'Seat assigned to you is <seat-number>'`.

   2. `bookSeat <name> <seatNum>` – behaves similar to `reserve` command, but imposes an additional constraint that a seat is reserved if and only if there is no existing reservation against `name` and the seat having the number `seatNum` is available. If there is no existing reservation but `seatNum` is not available, the response is: `'<seatNum> is not available'`.

   3. `search <name>` – returns the seat number reserved for `name`. If no reservation is found for `name` the system responds with a message: `'No reservation found for <name>'`.

   4. `delete <name>` – frees up the seat allocated to that person. The command returns the seat number that was released. If no existing reservation was found, responds with: `'No reservation found for <name>'`.

   Here `<text>` is used to indicate the value of the parameter denoted by `text` in the given context. Note that you also have to write the client program that takes input from the user and then communicates with a server using sockets. Your program should behave correctly in presence of multiple concurrent clients.

   Important: Do not 'hard-code' configuration parameters such as theater capacity c , host server's ip-address/servername, and the connection port number. Use a configuration file for storing these values, so that your code does not require re-compilation if any of these configuration values are changed. Points would be deducted for hard-coding.