Distributed Systems HW1

Samuel Alaniz, Juan Neri

Fall 2022

1 Relations

1.1 Prove or disprove that every symmetric and transitive relation is reflexive

We **disprove** the above claim by providing an example where it does not hold.

Let $A = \{0, 1\}$ and $R \equiv xRy \iff x + y = 2$

The relation is **symmetric**:

$\forall x, y \in A :$
$xRy \implies x + y = 2 \implies y + x = 2 \implies yRx$

The relation is **transitive**:

$\forall x, y, z \in A :$
$xRy \land yRz \implies (x + y = 2) \land (y + z = 2) \implies x = y = z = 1 \implies x + z = 2 \implies xRz$

However, the relation is not **reflexive**:

$0 + 0 = 0 \neq 2 \implies \neg 0R0$

1.2 Prove or disprove that every irreflexive and transitive relation is asymmetric

We first show that every irreflexive and transitive relation cannot be symmetric.

Let R be irreflexive and transitive. Assume by contradiction that R is symmetric.

We therefore have that $\forall x, y \in A : xRy \implies yRx$. Since R is transitive this means that xRx. This is a contradiction because R is irreflexive by construction.

We now proceed to show that it must also be asymmetric.

Recall that a relation R is said to be asymmetric if $\forall x, y \in A, if xRy \land yRx \implies x = y$

However, as we have shown above it can never be true that in this relation $xRy \land yRx$, therefore it is trivially true that $\forall x, y \in A, if xRy \land yRx \implies x = y$

2 Lamport's Algorithm With Concurrent Reads

We present a modification to Lamport's algorithm that allows reads to happen concurrently. The main modification is to add a boolean field to the queue *isRead*. If at any point the queue contains multiple read entries at the top, all those processes are allowed to enter the critical section:

- To request the critical section, a process sends a timestamped message indicating if they wish to read or write to all other processes and adds the request to the queue.
- On receiving a request message, the request, its timestamp and *isRead* flag are stored in the queue and a timestamped acknowledgment is sent back.
- To release the critical section, a process removes the request from its queue and sends a release message to all other processes.
- On receiving a release message, the corresponding request is deleted from the queue.
- A process determines that it can access the critical section if its request is at the head of the queue or its request is a read requests and there are no write requests ahead of it in the queue. The process must also have received an acknowledgement message from every other process.

P_i ::
var
q : queue of (int, pid, isRead) initially $null$;
$numAcks$: integer initially 0;
// Also assumes logical clock algorithm running

$request$:
send $request$ with $(logicalClock, i, isRead)$ to all other processes;
insert $(logicalClock, i, isRead)$ in q;
$numAcks := 0$;

On $receive(request, (ts, j, isRead)))$ from P_j:
insert $(ts, j, isRead)$ in q;
send $(ack, logicalClock)$ to P_j;

On $receive(ack, ts)$:
$numAcks := numAcks + 1$
if $(numAcks = N - 1)$ and either P_i's request smallest in q or P_i's request and all requests ahead of it have $isRead = true$ **then** enter_critical_section;

On $receive(release)$ from P_j:
delete the request by P_j from q
if $(numAcks = N - 1)$ and either P_i's request smallest in q or P_i's request and all requests ahead of it have $isRead = true$ **then** enter_critical_section;

$release$:
delete the request by P_i from q
send $release$ to all processes;

3 K-mutual Exclusion

3.1 Lamport's Mutex for K-mutual Exclusion

- To request the critical section, a process sends a timestamped message to all other processes and adds the request to the queue.
- On receiving a request message, the request and its timestamp are stored in the queue and a timestamped acknowledgment is sent back.
- To release the critical section, a process removes the request from its queue and sends a release message to all other processes.
- On receiving a release message, the corresponding request is deleted from the queue.
- A process determines that it can access the critical section if its request is at one of the top k requests in the queue and it has received an acknowledgement message from every other process.

3.2 Ricart and Agrawala's Mutex for K-mutual Exclusion

- To request a resource, the process sends a timestamped message to all processes.
- On receiving a request from any other process, the process sends an okay message if either the process is not interested in the critical section or its own request has a higher timestamp value. Otherwise, that process is kept in a pending queue.
- To release a resource, the process sends okay to all the processes in the pending queue.
- The process is granted the resource when it has requested the resource and it has received the okay message from $n - k + 1$ other processes in response to its request message.