MANCHESTER 1824

The University of Manchester

# Introduction to Basic Concepts: Synchronization

# Example 2: Stopping Events from Happening Simultaneously When Sharing Resources

- Assume that there are multiple distributed components collaborating to achieve one goal.
- For example: the selling of a game ticket to a client. The involved components include a database of available tickets, at least two bank accounts (one for the ticket vendor, one for the person buying the ticket), and possibly other systems for validating addresses, and arranging for the ticket to be delivered electronically.
- These systems need to choreographed to act as one for the purposes of selling a ticket, to avoid situations such as the following: (1) one ticket is selected by a client, which happens to be the last ticket. But, just before the client is able to pay for it, the ticket is purchased by another client who had also selected it; or (2) when paying for the ticket, the client confirms the payment transaction, and so the money leaves his account, but a system crash prevents the money from arriving at the seller's account.
- Possible solutions:
  - Centralised Lock Server (and Mutual Exclusion Locks (mutex))
  - Two Phase Commit Algorithm



# Centralised Lock Server and Mutual Exclusion Locks (Mutex)

Client:

1. Sends a request to the lock server for a mutex on a given resource.
2. When a reply comes back, it starts executing its critical process over the resource.
3. When its finished, it sends a message to release the mutex.

The Lock Server:

1. If the resource is available, it marks it as being used by the client and sends a reply to say that the lock has been granted. Otherwise, it puts the request in a queue.
2. When the mutex is released by the client, if another client wants the resource (i.e., is in the queue waiting), pass the mutex to them, otherwise mark it as being available.

**Note 1:** This solution presents a basic limitation: a single point of failure (i.e. the central lock server).

**Note 2:** Although the solution is able to protect sequences of events that need to be treated as 'atomic' (indivisible) operations, it cannot make sure that, if any part of the sequence of event fails, the state of the system remains unchanged.



# Two Phase Commit Algorithm

- The Two Phase Commit Algorithm ensures that a sequence of events either runs to successful completion, or, if the sequence fails, that the overall system is returned to its original state as though nothing had happened. In other words, it allows intermediate steps that have occurred to be undone (i.e., rolledback) to return things back to a safe, sensible state, if a fault is detected.

Potential point of failure or bottleneck!
Which node should be the coordinator?

- A coordinator node requests a transaction, and sends a request to all participants nodes
  - e.g., to node C1, it sends 'request to remove X pounds from account', and to C2 sends 'request to add X pounds to account'.
- All participants respond as to whether they are willing and able to execute the request, and send VOTE_COMMIT or VOTE_ABORT.
  - They log their current state, and then perform the transaction.
  - All participants log their vote
- The coordinator looks at the votes. If everyone has voted to commit, then the co-ordinator sends a GLOBAL_COMMIT to everyone; otherwise it sends a GLOBAL_ABORT.
- On receiving the decision from the coordinator, all participants record the decision locally. If it was an ABORT, participants ROLL BACK to their previous safe state.



# Electing a Coordinator Node

- The **Bully Algorithm** is a mechanism for choosing a coordinator from a set of candidate nodes.
  - The algorithm gets its name from the fact that higher numbered nodes 'bully' lower numbered nodes into submission.
- **Note 1:** the algorithm relies on the use of timeouts to decide when to 'give up' waiting for responses from nodes that have potentially died (so the usual problem of 'how long is it reasonable to wait' applies here).
- **Note 2:** the algorithm assumes that the participating nodes are ordered.

- P sends an ELECTION message to all nodes with higher numbers.
- If no one responds, P wins the election and becomes the co-ordinator. It informs all the other nodes that it is now the coordinator.
- If one of the higher-numbered nodes Q answers, P concedes that it is not the winner, and Q begins the election process again until one node eventually wins.

# Clock Synchronization

- In distributed systems, clock synchronisation is not completely possible by the fact that messages are not sent instantaneously over real networks, and that there usually is some degree of variation in the time messages take to arrive at their destination.
- As long some amount of error is acceptable, there are at least two widely acceptable ways for getting clocks in different parts of a system into near-synchronisation.
  - Cristian's Algorithm
  - The Berkeley Algorithm



# Cristian's Algorithm

- Cristian's algorithm works between a process P, and a time server S connected to a source of UTC (Coordinated Universal Time).
- It relies on the accuracy of an estimate based on the Round Trip Time (RTT), which is the elapsed time between the time when a request is sent from P to S, and the time when a response is received by P from S.
- But what happens if the RTT estimate is not accurate, i.e., it is different when the update is sent to that which was measured?
  - Clocks tend to drift.
- And what happens if send and receive do not take the same amount of time, which affects RTT/2?
- These issues are dealt with by a similar but more complex algorithm, The Berkeley Algorithm.

- P requests the time from S.
- After receiving the request from P, S prepares a response and appends the time T from its own clock. The response is sent to P.
- P then sets its time to be T + RTT/2, where RTT is Round Trip Time of the request P made to S.

This method assumes that the RTT is split equally between both request and response, which may not always be the case but is a reasonable assumption on a LAN connection.



# The Berkeley Algorithm

- Unlike Cristian's algorithm, the server process in the Berkeley algorithm, called the master, periodically polls other slave processes.
- The clock synchronisation method used in this algorithm, the average, cancels out individual clock's tendencies to drift.

1. A master is chosen via an election process such as the Bully Algorithm.
2. The master polls the slaves who reply with their time in a similar way to Cristian's algorithm.
3. The master observes the round-trip time (RTT) of the messages and estimates the time of each slave and its own.
4. The master then averages the clock times, ignoring any values it receives far outside the values of the others.
5. Instead of sending the updated current time back to the other process, the master then sends out the amount (positive or negative) that each slave must adjust its clock. This avoids further uncertainty due to RTT at the slave processes.