

ICS 362 Distributed Systems

Distributed Systems: Part 13

Lecturer: Toby Daniel

Synchronisation Issues

- Protecting shared Resources
- Transaction Types

Mutual Exclusion in Distributed Systems

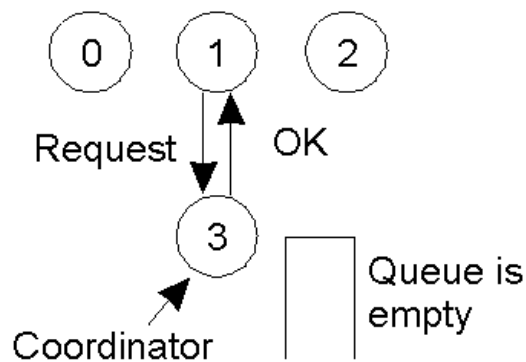
- Protecting shared Resources
- It is often necessary to protect a *shared resource* within a Distributed System using “mutual exclusion” – for example, it might be necessary to ensure that no other process changes a shared resource while another process is working with it.
- In non-distributed, uniprocessor systems, we can implement “critical regions” using techniques such as semaphores, monitors and similar constructs – thus achieving *mutual exclusion*.

Distributed Mutual Exclusion Techniques

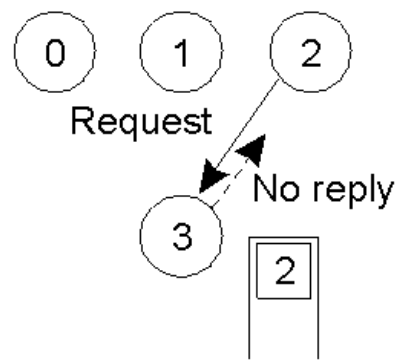
- **Centralised:** a single coordinator controls whether a process can enter a critical region.
- **Distributed:** the group *confers* to determine whether or not it is safe for a process to enter a critical region.
- **Token Ring:** All processes are given turns at entering the critical region.

Centralised Algorithm

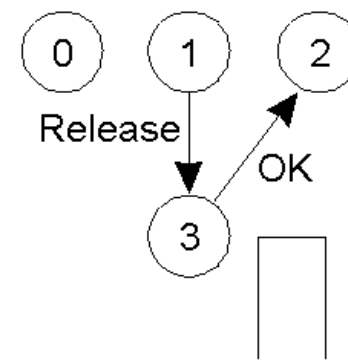
- Process 1 asks the coordinator for permission to enter a critical region. Permission is granted by an OK message (assuming it is, of course, OK).
- Process 2 then asks permission to enter the same critical region. The coordinator does not reply (but adds 2 to a queue of processes waiting to enter the critical region). No reply is interpreted as a “busy state” for the critical region.
- When process 1 exits the critical region, it tells the coordinator, which then replies to 2 with an OK message.



(a)



(b)



(c)

Centralised Algorithm

- ***Advantages:***
 - It works.
 - It is fair.
 - There's no process starvation.
 - Easy to implement.
- ***Disadvantages:***
 - There's a single point of failure!
 - The coordinator is a bottleneck on busy systems.
- ***Critical Question:*** When there is no reply, does this mean that the coordinator is “dead” or just busy?

Distributed Algorithm

1. When a process (the “requesting process”) decides to enter a critical region, a message is sent to all processes in the Distributed System (including itself).
2. What happens at each process depends on the “state” of the critical region.
3. If not in the critical region (and not waiting to enter it), a process sends back an OK to the requesting process.
4. If in the critical region, a process will queue the request and send back *no reply* to the requesting process.
5. If **waiting** to enter the critical region, a process will:
 - a) Compare the timestamp of the new message with that in its queue (note that the lowest timestamp wins).
 - b) If the received timestamp wins, an OK is sent back, otherwise the request is queued (and no reply is sent back).
6. When all the processes send OK, the requesting process can safely enter the critical region.
7. When the requesting process leaves the critical region, it sends an OK to all the process in its queue, then empties its queue.

Distributed Algorithm

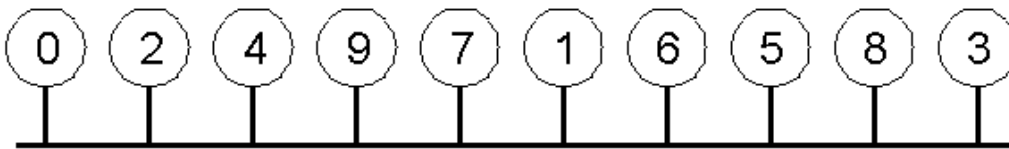
- The algorithm works because in the case of a conflict, the lowest timestamp wins as everyone agrees on the total ordering of the events in the distributed system.
- **Advantages:**
 - It works.
 - There is no single point of failure
- **Disadvantages:**
 - We now have multiple points of failure!!!
 - A “crash” is interpreted as a *denial of entry* to a critical region.
 - (A patch to the algorithm requires all messages to be ACKed).
 - Worse is that all processes must maintain a list of the current processes in the group (and this can be tricky)
 - Worse still is that one overworked process in the system can become a *bottleneck* to the entire system – so, everyone slows down.

Distributed Algorithm

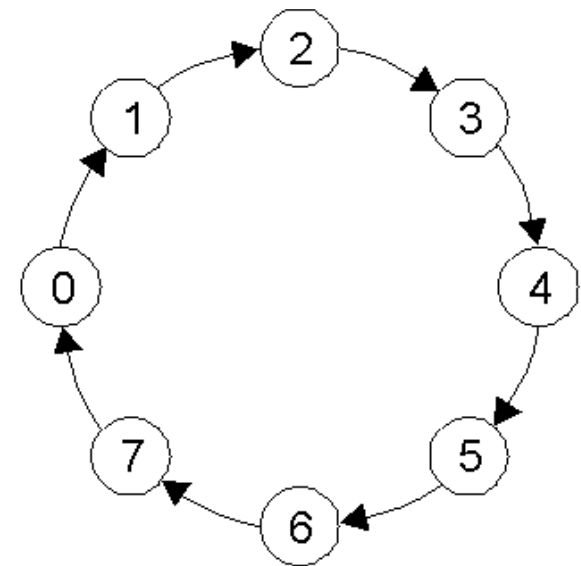
- It isn't always best to implement a distributed algorithm when a *reasonably good* centralised solution exists.
- What's good *in theory* (or on paper) may not be so good *in practice*.
- Think of all the message traffic this distributed algorithm is generating (especially with all those ACKs). **Remember:** every process is involved in the decision to enter the critical region, whether they have an interest in it or not.

Token Ring Algorithm

- a) An unordered group of processes on a network. Note that each process knows the process that is next in order on the ring after itself.
- b) A logical ring is constructed in software, around which a token can circulate – a critical region can only be entered when the token is held. When the critical region is exited, the token is released.



(a)



(b)

Token Ring Algorithm

- **Advantages:**
- It works (as there's only one token, so mutual exclusion is guaranteed).
- It's fair – everyone gets a shot at grabbing the token at some stage.
- **Disadvantages:**
- Lost token! How is the loss detected (is it in use or is it lost)? How is the token regenerated?
- Process failure can cause problems – a broken ring!
- Every process is required to maintain the current logical ring in memory – not easy.

Which is best?



Which is best?

- The “Centralized” algorithm is simple and efficient, but suffers from a single point-of-failure.
- The “Distributed” algorithm has nothing going for it – it is slow, complicated, inefficient of network bandwidth, and not very robust. It “sucks”!
- The “Token-Ring” algorithm suffers from the fact that it can sometimes take a long time to reenter a critical region having just exited it.
- All perform poorly when a process crashes, and they are all generally poorer technologies than their non-distributed counterparts. Only in situations where crashes are very infrequent should any of these techniques be considered.

Distributed Transactions

- Related to Mutual Exclusion, which protects a “shared resource”.
- Transactions protect “shared data”.
 - Often, a single transaction contains a collection of data accesses/modifications.
 - The collection is treated as an “atomic operation” – either all the collection complete, or none of them do.
 - Mechanisms exist for the system to revert to a previously “good state” whenever a transaction prematurely aborts.

Transaction Primitives

Primitive	Description
BEGIN_TRANSACTION	Make the start of a transaction
END_TRANSACTION	Terminate the transaction and try to commit
ABORT_TRANSACTION	Kill the transaction and restore the old values
READ	Read data from a file, a table, or otherwise
WRITE	Write data to a file, a table, or otherwise

Grouping Transactions

- Suppose a transaction has 2 phases;
 - Step 1: Withdraw amount a from account 1.
 - Step 2: Deposit amount a into account 2.
- What happens if the connection is broken after the first step, but before the second?
- We want either neither or both actions to be processed.

Example

```
BEGIN_TRANSACTION  
  reserve CNX -> BKK;  
  reserve BKK -> Nairobi;  
  reserve Nairobi -> Malindi;  
END_TRANSACTION
```

(a)

```
BEGIN_TRANSACTION  
  reserve CNX -> BKK;  
  reserve BKK -> Nairobi;  
  reserve Nairobi -> Malindi full =>  
ABORT_TRANSACTION
```

(b)

Transaction ACID



Four key transaction characteristics:

- **Atomic**
- **Consistent**
- **Isolated**
- **Durable**

ACID - atomic

- To be atomic, a transaction must execute completely or not at all. This means that every task within a unit-of-work must execute without error. If any of the tasks fails, the entire unit-of-work or transaction is aborted, meaning that changes to the data are undone. If all the tasks execute successfully, the transaction is committed, which means that the changes to the data are made permanent or durable.

ACID - consistency

- must be enforced by both the transactional system and the application developer. Consistency refers to the integrity of the underlying data store. The transactional system fulfills its obligation in consistency by ensuring that a transaction is ACID. The application developer must ensure that the database has appropriate constraints (primary keys, referential integrity, and so forth) and that the unit-of-work, the business logic, doesn't result in inconsistent data. For example, a debit to one account must equal the credit to the other account.

ACID - isolated

- A transaction must be allowed to execute without interference from other processes or transactions. In other words, the data that a transaction accesses cannot be affected by any other part of the system until the transaction or unit-of-work is completed.

ACID - durable

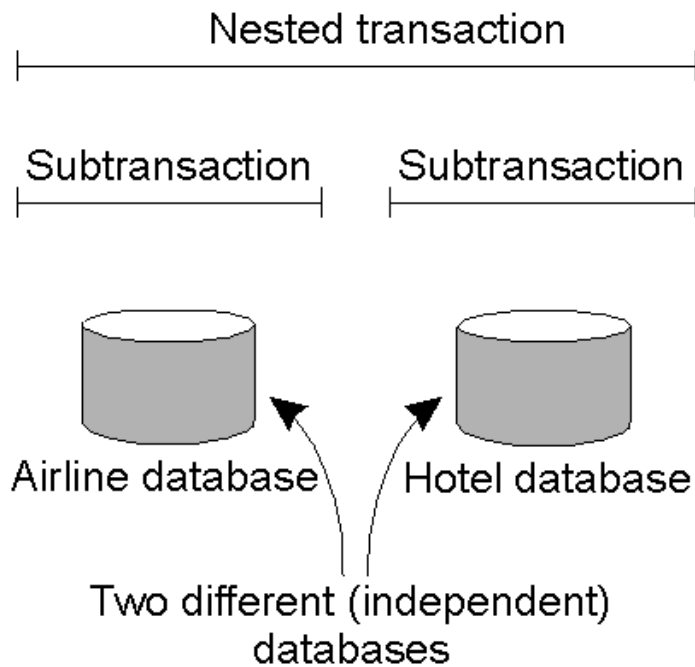
- Durability means that all the data changes made during the course of a transaction must be written to some type of physical storage before the transaction is successfully completed. This ensures that the changes are not lost if the system crashes.

Transaction Types

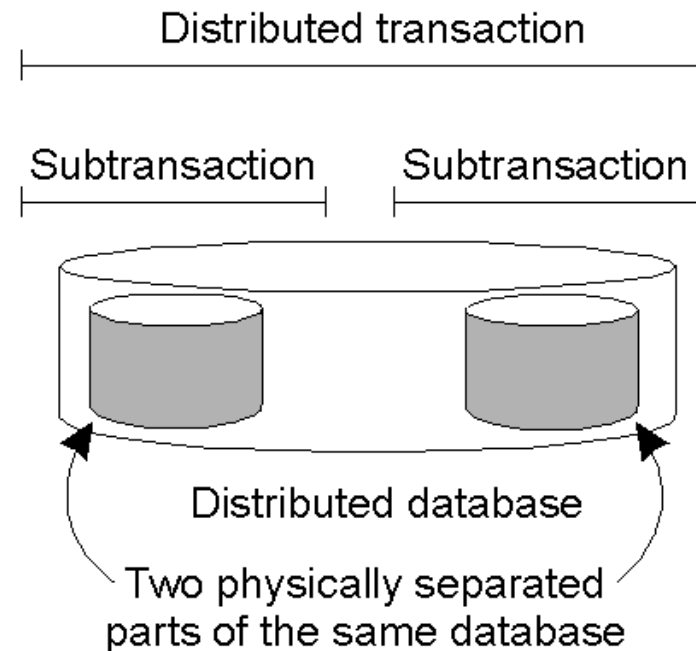
- **Flat Transaction:** this is the model that we have looked at so far. Disadvantage is it's too rigid, partial results cannot be committed. i.e., the “atomic” nature of Flat Transactions can be a downside.
- **Nested Transaction:** a main, parent transaction spawns child sub-transactions to do the real work. Disadvantage: problems result when a sub-transaction commits and then the parent aborts the main transaction.
- **Distributed Transaction:** this is sub-transactions operating on distributed data stores. Disadvantage: complex mechanisms required to lock the distributed data, as well as commit the entire transaction.

Nested / Distributed Transactions

- a) A nested transaction – logically decomposed into a hierarchy of sub-transactions.
- b) A distributed transaction – logically a flat, indivisible transaction that operates on distributed data.



(a)



(b)

ATM Transaction Example

- Let's say that a bank has 100 ATMs in a metropolitan area
- each ATM processes 300 transactions (deposits, withdrawals, or transfers) a day
- If each transaction, on average, involves the deposit, withdrawal, or transfer of about B100, **How many Baht would move through the ATM system per day?**
- In the course of a year

$$(365 \text{ days}) \times (100 \text{ ATMs}) \times (300 \text{ transactions}) \times (\text{B}100.00) = \text{B}1,095,000,000.00$$

ATM Transaction Example

- How well do the ATMs have to perform in order for them to be considered reliable?
- For the sake of argument, let's say that ATMs execute transactions correctly 99.99% of the time.
- This seems to be more than adequate: after all, only one out of every ten thousand transactions executes incorrectly.
- But over the course of a year how much money is lost or misplaced in errors?

ATM Transaction Example

$$B1,095,000,000.00 \times .01\% = B109,500.00$$

- Obviously, this is an oversimplification of the problem, but it illustrates that even a small percentage of errors is unacceptable in high-volume or mission-critical systems.
- That's why we need to use ACID

Medical System Example

- In a medical system, important data--some of it critical--is recorded about patients every day, including information about clinical visits, medical procedures, prescriptions, and drug allergies. The doctor prescribes the drug, then the system checks for allergies, contraindications, and appropriate dosages. If all tests pass, then the drug can be administered.
- The tasks just described make up a unit-of-work in a medical system. A unit-of-work in a medical system may not be financial, but it's just as important. A failure to identify a drug allergy in a patient could be fatal.

Fault tolerance for distributed systems

- Fault-tolerant system
- RAID
- CDP - continuous data protection

