Algorithms and protocols for distributed systems

We have defined process groups as having peer or hierarchical structure and have seen that a coordinator may be needed to run a protocol such as 2PC.

With peer structure, an external process may send an update request to any group member, which then functions as coordinator. We have seen that deadlock may occur.

If the group has hierarchical structure, one member is elected as coordinator. That member must manage group protocols, and external requests must be sent on to it. Note that this solves the potential deadlock problem of concurrent updates. But a single point of failure is created, and a potential bottleneck, so this is only suitable for small groups.

If the coordinator fails, a new one must be elected. For the election algorithm:

- each process has a unique ID known to all members
- the process with highest ID is coordinator
Election algorithm - Bully

P notices no reply from coordinator
P sends **ELECT** message to all processes with higher IDs
If any reply, P exits. Any process that replies must itself run an election.
If none reply, P wins – gets any required state from persistent storage
  – P sends **COORD** message to the group

\[ (i) \]

\[ (ii) \]

Distributed algorithms
Election algorithm - Ring

Processes are ordered into a ring structure that is known to all. A failed process can be bypassed, provided that the ordering around the ring is known and that messages are acknowledged, so that a no-ACK can be detected.

The election algorithm:
P notices that the coordinator is not functioning.

P sends an ELECT message to the next process in the ring, tagged with its own (P’s) ID

On receipt of ELECT by any process:
- without receiver’s (its own) ID:
  append receiver’s ID to message and send to next process in ring
- with receiver’s ID (it’s been all round):
  look for highest ID in list of appended IDs and send COORD (highest ID) message

Many election algorithms can run concurrently
All should agree on the same highest ID
Distributed mutual exclusion

Suppose $N$ processes hold an object replica and it is required that only one-at-a-time may access its replica.

Examples: - ensuring coherence of distributed shared memory
- distributed games
- distributed whiteboard
i.e. for use by *simultaneously running, tightly coupled components* managing object replicas in main memory. We have already seen the approach of LOCKING *persistent object replicas* for consistent, transactional update.

Assume that - processes update in place
- then the update is propagated (not shown as part of the algorithm)

Each process executes code of the form:

*entry protocol*

access object under exclusion, in a critical region

*exit protocol*
Distributed mutex 1. centralised algorithm

One process is the elected coordinator

**entry protocol**
- send *request* message to coordinator
- wait for *reply (OK-to-enter)* from coordinator

**access object under exclusion**

**exit protocol**
- send *finished* message to coordinator

+ fair FCFS or priority if coordinator re-orders
+ economical (3 messages in basic protocol, but need more ....)
– coordinator is single point of failure (need to elect a new one if it fails)
– what does no reply mean? waiting for exclusive access – OK
  but coordinator could have failed

Improve/solve by using extra messages:
  - coordinator ACKs request and sends again when process can proceed?
  - heartbeat protocol between coordinator and processes awaiting object access
Distributed mutex 2 – Token Ring

A token giving permission to access the object circulates indefinitely

**entry protocol**
wait for token to arrive

**access object under exclusion**

**exit protocol**
pass on token

- Not controllable – ring order, not request order or priority order
- Quite efficient, but token circulates when no process wants object access
- Must handle loss of token and regeneration, ensuring one token only
- crashes? Use ACKs, reconfigure, bypass failed processes
3. Distributed peer-to-peer algorithm

**entry protocol**
- send a timestamped *request* to all processes including oneself
  (there is a convention for ordering timestamps: earliest timestamp wins + tiebreaker)
  only when ALL processes have sent *reply* can the object be accessed
- Any process executing the protocol, so wanting access, sends a *reply* to all processes
  whose request messages have earlier timestamps than its own message

**access object under exclusion**

**exit protocol**
- reply to any deferred requests

+ fair FCFS
- not economical, 2N messages + any ACKs
- N points of failure
- N bottlenecks
- no reply? Failure or deferring?

A bad idea – requiring ALL processes to act before any one can proceed.