Chapter 5

Synchronization

5.1. Introduction

- On this chapter, we are going to deal with
  - time and making sure that processes do the right thing at the right time i.e. allowing processes to synchronize and coordinate their actions.
- Coordination
  - refers to coordinating the actions of separate processes relative to each other and allowing them to agree on global state (such as values of a shared variable).
  - **Examples** of coordination include
    - ensuring that processes agree on what actions will be performed (e.g., money will be withdrawn from the account), who will be performing actions (e.g., which replica will process a request), and the state of the system (e.g., the elevator is stopped).
- Synchronization
  - is coordination with respect to time, and
  - refers to the ordering of events and execution of instructions in time.
  - **Examples** of synchronization include ordering distributed events in a log file and ensuring that a process performs an action at a particular time.

- Timing models for distributed system
  - Synchronous model
    - the time to perform all actions, communication delay, and clock drift on all nodes, are bounded.
  - Asynchronous distributed systems there are no such bounds.
    - Most real distributed systems are asynchronous, however, it is easier to design distributed algorithms for synchronous distributed systems.
    - Algorithms for asynchronous systems are always valid on synchronous systems; however, the converse is not true.

Time and clock

- In a distributed system, each computer has its own clock.
- Because no clock is perfect, each of these clocks has its own **skew** which causes clocks on different computers to **drift** and eventually become out of sync.
    - **Skew**: difference between two clocks at one point in time.
    - **Drift**: two clocks tick at different rates.
- There are several notions of time that are relevant in a distributed system.
- First of all, internally a computer clock simply keeps track of ticks that can be translated into physical time (hours, minutes, seconds, etc.). This physical time can be global or local.
  - Global time is a universal time that is the same for everyone and is generally based on some form of absolute time. Currently Coordinated Universal Time (UTC) is the most accurate global time.
  - Besides global time, processes can also consider local time.

5.2. Physical Clocks and clock synchronization algorithms

- Physical clocks keep track of physical time.
- In distributed systems that rely on actual time it is necessary to keep individual computer clocks synchronized.
- The clocks can be synchronized to
  - global time (external synchronization)

    E.g. Cristian's algorithm and Network Time Protocol (NTP) or to
  - each other (internal synchronization)

    E.g. The Berkeley Algorithm
- One of the problems encountered when synchronizing clocks in a distributed system is that unpredictable communication latencies can affect the synchronization.

- ***Cristian's algorithm***
  - requires clients to periodically synchronize with a central time server (typically a server with a UTC receiver).
  - attempts to calculate the communication delay based on the time elapsed between sending a request and receiving a reply.
  - Example:

Network Time Protocol (NTP)

- Synchronization is performed using time servers and an attempt is made to correct for communication latencies.
- Unlike Cristian's algorithm, however, NTP is not centralized and is designed to work on a wide area scale. As such, the calculation of delay is somewhat more complicated.
- NTP provides a hierarchy of time servers, with only the top layer containing UTC clocks.
- The NTP algorithm allows
  - client-server and peer-to-peer (mostly between time servers) synchronization.
  - clients and servers to determine the most reliable servers to synchronize with.
- NTP typically provides accuracies between 1 and 50 msec depending on whether communication is over a LAN or WAN.
- Example:

- ***Berkeley algorithm***
  - does not synchronize to a global time.
  - a time server/master/coordinator polls the clients to determine the average of everyone's time.
  - The server/master then instructs all clients to set their clocks to this new average time.
- Note that:
  - in all the above algorithms a clock should never be set backward.
  - If time needs to be adjusted backward, clocks are simply slowed down until time 'catches up'.
- Example:

5.3 Logical Clocks

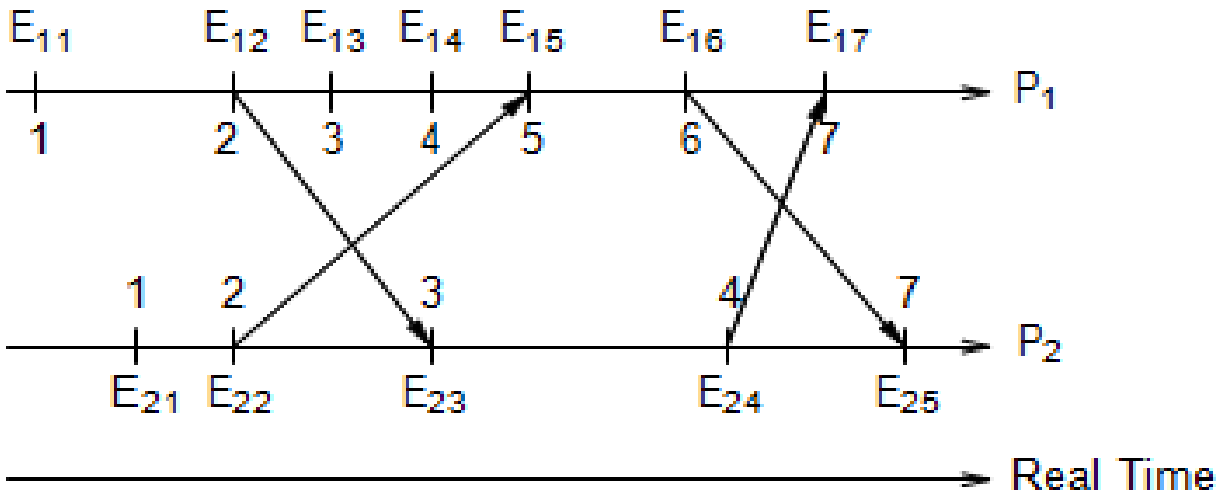
- For many applications, the relative ordering of events is more important than actual physical time.
- In a distributed system, all processes must agree on ordering of causally related events (e.g., sending and receiving of a single message).
- Given a system consisting of N processes P_i , $i \in \{1, \ldots, N\}$, we define the local event ordering $\rightarrow_i$ as a binary relation, such that, if P_i observes e before e′ , we have $e \rightarrow_i e'$
- Based on this local ordering, we define a global ordering as a happened before relation →, as proposed by Lamport.
- The relation → is the smallest relation, such that
    - $e \rightarrow_i e'$ implies $e \rightarrow e'$,
    - for every message m, send (m) → receive (m), and
    - $e \rightarrow e'$ and $e' \rightarrow e''$ implies $e \rightarrow e''$ (transitivity).

5.3.1 Lamport clocks

- each process P_i maintains a logical clock L_i.
- Given such a clock, $L_i(e)$ denotes a Lamport timestamp of event **e** at P_i and L(e) denotes a timestamp of event **e** at the process it occurred at.
- Processes now proceed as follows:
  - Before time stamping a local event, a process P_i executes $L_i := L_i + 1$.
  - Whenever a message m is sent from P_i to P_j :
    - Process P_i executes $L_i := L_i + 1$ and sends the new L_i with m.
    - Process P_j receives L_i with m and executes $L_j := \max(L_j , L_i) + 1$. receive(m) is annotated with the new L_j.

Cont.…

- In this scheme, a → b implies L(a) < L(b), but L(a) < L(b) does not necessarily imply a → b.



- In some situations (e.g., to implement distributed locks), a partial ordering on events is not sufficient and a total ordering is required.
- In these cases, the partial ordering can be completed to total ordering by including process identifiers.
- Given local time stamps Li(e) and Lj (e'), we define global time stamps ‹Li(e), i› and ‹Lj (e), j›. We, then, use standard lexicographical ordering, where ‹Li(e), i› < ‹Lj(e), j› iff Li(e) < Lj(e), or Li(e) = Lj (e) and i < j.

5.4 Distributed Mutual Exclusion

- When concurrent access to distributed resources is required, we need to have mechanisms to prevent race.
- These mechanisms must fulfill the following three requirements:
  - **Safety**: At most one process may execute the critical section at a time
  - **Liveness**: Requests to enter and exit the critical section eventually succeed
  - **Ordering**: Requests are processed in happened-before ordering

1. Central Server

- a central server controls the entering and exiting of critical sections.
- Processes must send requests to enter and exit a critical section to a lock server (or coordinator)
- Upon leaving the critical section, the token is returned to the server.
- Processes that wish to enter a critical section while another process is holding the token are put in a queue.
- When the token is returned the process at the head of the queue is given the token and allowed to enter the critical section.
- does not scale well due to the central authority and it is vulnerable to failure of the central server.

2. Token Ring

- organizes all processes in a logical ring structure, along which a token message is continuously forwarded.
- Before entering the critical section, a process has to wait until the token comes by and then retain the token until it exits the critical section.
- Disadvantage
  - the ring imposes an average delay of N/2 hops, which again limits scalability.
  - the token messages consume bandwidth and failing nodes or channels can break the ring.
  - failures may cause the token to be lost.
  - if new processes join the network or wish to leave, further management logic is needed.

3. Using Multicast and Logical Clocks

- Each participating process P_i maintains a Lamport clock and all processes must be able to communicate pairwise.
- At any moment, each process is in one of three states:
  - Released, Wanted or Held
- If a process wants to enter a critical section, it multicasts a message (Li, Pi) and waits until it has received a reply from every other process.
- The processes operate as follows:
  - If a process is in Released state, it immediately replies to any request to enter the critical section.
  - If a process is in Held state, it delays replying until it is finished with the critical section.
  - If a process is in Wanted state, it replies to a request immediately only if the requesting timestamp is smaller than the one in its own request.

5.5 Elections

- Coordinator:
  - Some algorithms rely on a distinguished coordinator process
  - Coordinator needs to be determined
  - May also need to change coordinator at runtime
- Election:
  - **Goal**: when algorithm finished all processes agree who new coordinator is.

- Determining a coordinator:
  - Assume all nodes have unique id
  - **possible assumption**: processes know all other process's ids but don't know if they are up or down
  - **Election**: agree on which non-crashed process has largest id number
- Requirements:

  ① Safety: A process either doesn't know the coordinator or it knows the id of the process with largest id number

  ② Liveness: Eventually, a process crashes or knows the coordinator

5.5.1 BULLY ALGORITHM

- Three types of messages:
  - Election: announce election
  - Answer: response to election
  - Coordinator: announce elected coordinator
- A process begins an election when it notices through a timeout that the coordinator has failed or receives an Election message
- When starting an election, send Election to all higher-numbered processes
- If no Answer is received, the election starting process is the coordinator and sends a Coordinator message to all other processes
- If an Answer arrives, it waits a predetermined period of time for a Coordinator message
- If a process knows it is the highest numbered one, it can immediately answer with Coordinator

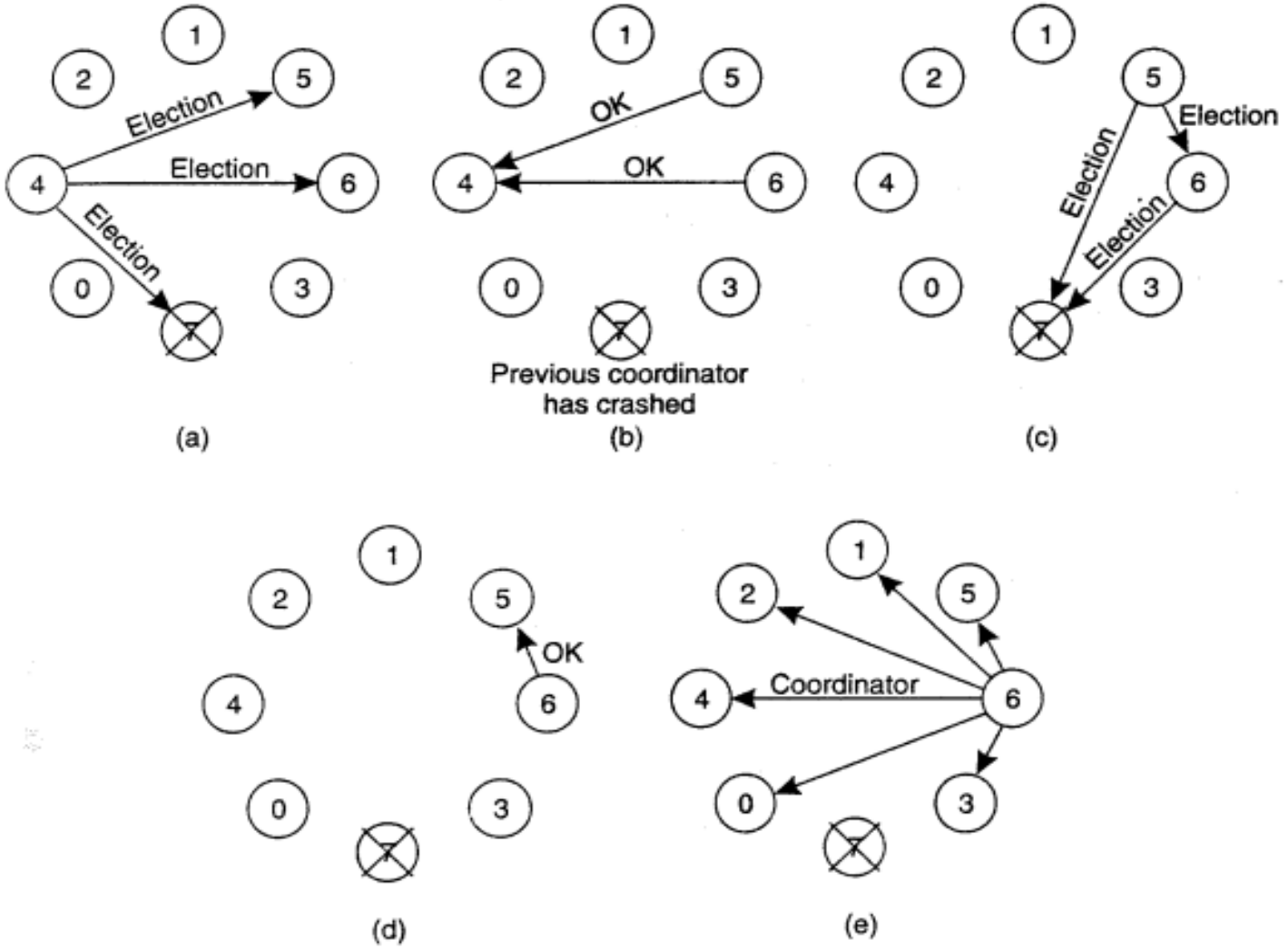


Fig. The bully election algorithm.

5.5.2 RING ALGORITHM

- Two types of messages:
  - Election: forward election data
  - Coordinator: announce elected coordinator
- Processes ordered in ring
- A process begins an election when it notices through a timeout that the coordinator has failed.
- Sends message to first neighbor that is up
- Every node adds own id to Election message and forwards along the ring
- Election finished when originator receives Election message again
- Forwards message on as Coordinator message

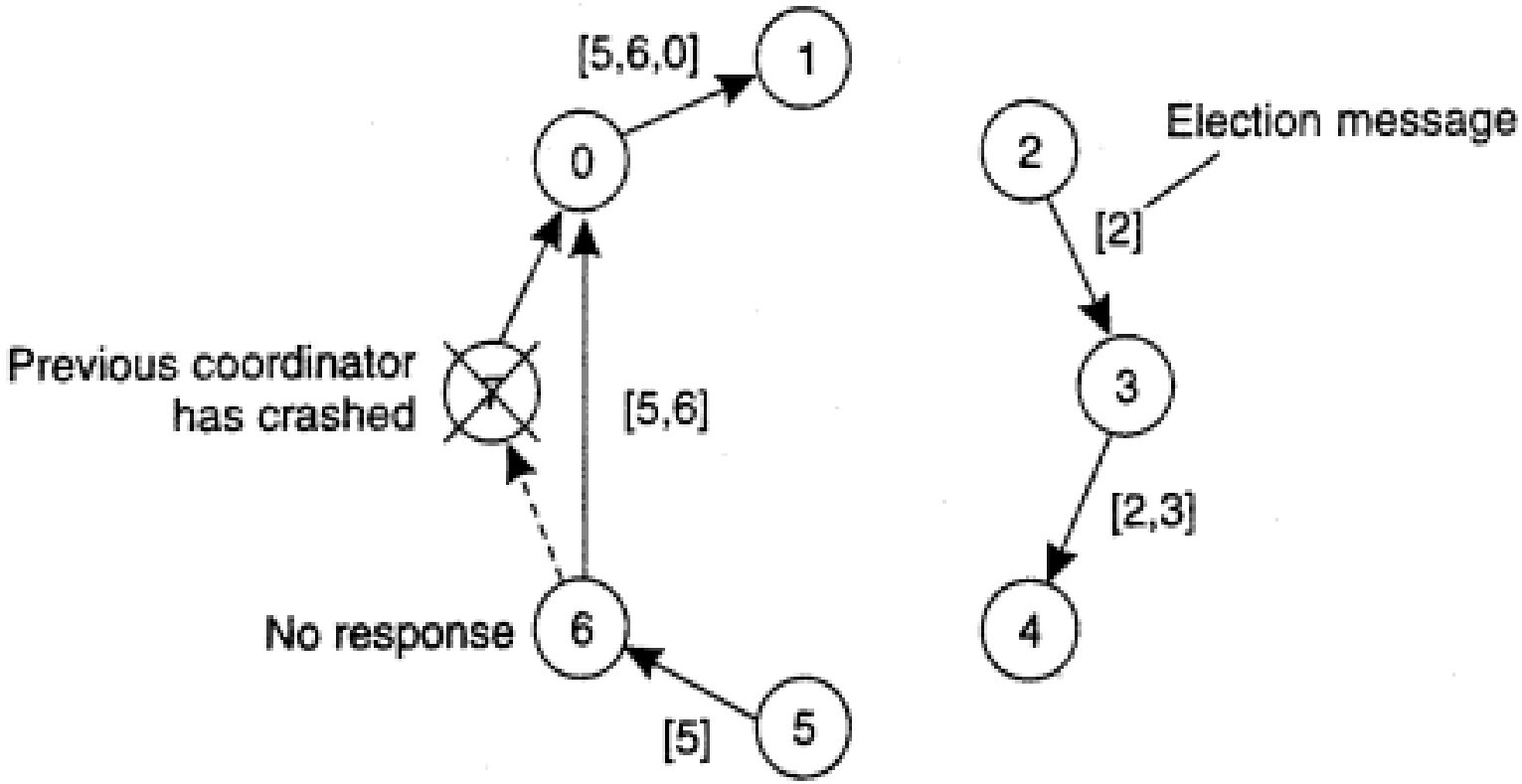


Fig. Election algorithm using a ring