Universitatea Tehnica din Cluj-Napoca
Departament Calculatoare

Distributed Systems

C7

Coordination, Agreement, Consensus

Fall 2017

Contents

Introduction
Review

- DS - collection of computing resources
  - Independent (autonomous)
  - Heterogeneous
  - Interconnected via a network
- DS user view: a single coherent system
- DS processes must coordinate their actions to agree on one or more values and relative to the shared resources

Introduction
Review

- Synchronization
  - Synchronous DS
    - Allows timeouts for process crashes
    - Bounds on max message delay
    - Max processor execution time for algorithm steps
    - Max clock drift rates
  - Asynchronous DS
    - No time assumptions
- Failure issues
  - Algorithms that detect and tolerate faults
  - New leader has to be elected when an ex leader crashes
  - Impossible to agree consensus on a shared value for asynchronous systems

Leader Election Algorithms (LEA)
Basic notions

- Leader Election algorithm
  - Chooses a unique process out of the n participation processes p_i (i = 1, 2, … N) to play a particular role
- Main requirement
  - The elected process must be unique (even when several processes are calling the election concurrently)
- Calling the election
  - Action taken by a process that initiates a particular run of the election algorithm
    - An individual process calls the election only once at a time
    - The N participation processes may call for N concurrent elections

Leader Election Algorithms (LEA)
Basic notions

- Process status
  - At any time p_i is either
    - participant (i.e. running an election algorithm) or
    - non-participant (not currently engaged in any election)
  - Each process may consider two boolean variables
    - **isDone**, becomes true when process knows that the algorithm has terminated
    - **isLeader**, becomes true when the process knows that he is the leader

Leader Election Algorithms (LEA)
Basic notions

- A leader election algorithm must satisfy the safety and liveness properties
- **Safety**: at most one p_i is leader

∀ i, j i ≠ j => not (isLeader(i) and isLeader(j))

- **Liveness**: Eventually all processes are (either leaders or not) and (at least one p_i is a leader)

∀ i => isDone(i) and ∃j isLeader(j)

Leader Election Algorithms (LEA)
Basic notions

Performance

- Time and space performance of the associated algorithms
- Total network bandwidth utilization (which is proportional to the total number of messages sent)

Leader Election Algorithms (LEA)

Basic notions

Network Topologies

- Election algorithms for different network topologies:
  - Ring (uni- or bi- directional),
  - Complete graphs,
  - Grids, etc.
- Example
  - For a spanning tree (corresponding to a complete graph) the leader can be elected as a converge cast property

Leader Election Algorithms (LEA)

Basic notions

Network Topologies

- Ring Topology
  - Network of n processes placed in a circularly ring (modulo n arithmetic)
  - Uni-directional (clockwise) – each process p_i sends messages to p_{i+1} and receives messages from p_{i-1}
  - Bi-directional – each process can send/receive messages to/from both directions

Leader Election Algorithms (LEA)

Basic notions

Network Topologies

- Anonymous rings
  - All processes are indistinguishable
  - Processes have no unique IDs
  - Processes have identical state machines with the same initial state
- No deterministic LEA in anonymous rings (not even for synchronous rings)

Leader Election Algorithms (LEA)

Basic notions

Network Topologies

- Non-anonymous rings
  - All processes are assigned identifies which must be unique and totally ordered
  - Each process knows its own identifier
  - Non-uniform rings: ring size (n) is not a priori known by the nodes
  - When algorithm terminates:
    - The process with the max id is elected and
    - All processes will know the id of the elected leader
  - LeLann-Chang-Roberts (LCR) Algorithm

LCR Algorithm

- Suitable for a collection of processes arranged in a logical ring
  - Each process p_i has a communication channel to the next process in the ring, $p_{(i+1) mod N}$
  - All messages are sent clockwise around the ring
  - Each process know little about the other processes – each knows only how to communicate to its neighbor
- Algorithm goal
  - To elect a single process called the *coordinator*, which is the process with the largest identifier
- Assumptions
  - No failures occur
  - The system is asynchronous

LCR Algorithm
Algorithm description

- Clockwise, unidirectional ring
- One or more processes may take the initiative of starting the leader election process – they send to p_{i+1} an election message containing their id
- A p_i may take part in an election process spontaneously or upon receiving a message – in this case it marks itself as a participant
- If the p_i receiving an election message has a greater id and is not already a participant, then it sends an election message with its own id to p_{i+1}.
- If its own id is smaller, it forwards the message with the id it has received
- If pi receives a message with its own id then it declares itself as the leader

LCR Algorithm for a node i

```
Variables:
  bool participant = false;
  int leaderID = null;
When initiating the election:
  send(election(i));
  participant = true;
When receiving a message election(j):
  if(j > i) send(election(j));
  if(i == j) send(leader(i));
  if((i > j) and not(participant)) {
    send(election(i));
    participant = true;
  }
When receiving a message leader(j):
  leadrID = j;
  if(i != j) send leader(j));
```


LCR Algorithm

Comments

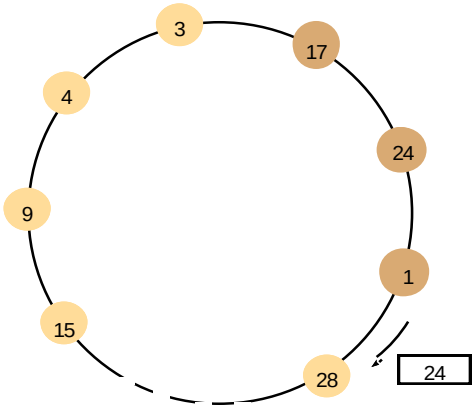
- Only the message with the largest ID travels over the whole ring and when arrives back to the originator becomes leader
- The leader send the leader(i) message; it arrives to all processes so every node knows the leader
- Time complexity O(n^2)

LCR Algorithm

Convention for the chosen process to be unique

- The elected process be chosen as one with the largest identifier
- Identifier may be any useful value (belonging to a totally ordered domain)
  - Ex. The process with the lowest computational load
  - Identifier is **<1/load, i>** where i is used to order identifiers with the same load
  - Each process has a variable **leaderID** which will contain the identifier of the elected process

A ring-based election in progress



Note: The election was started by process 17.
The highest process identifier encountered so far is 24.
Participant processes are shown in a darker colour

LCR Algorithm

Conclusion

- Limited use due to no failure toleration
- Having a reliable failure detector, when a process crashes the ring could be re-constructed and the algorithm re-executed

Bully Algorithm

- [Garcia-Molina 1982]
- Allows processes to crash during election (although it assumes that message delivery between processes is **reliable**)
- A process begins an election when it notices, through timeouts, that the coordinator has failed. Several processes may discover this concurrently
- Assumptions
  - Synchronous system, uses timeouts to detect a process failure
  - Each process knows which process have higher identifiers and it can communicate with all such processors (as opposite to ring-based algorithm)

Note on reliable communication
Reliable communication = validity + integrity

- **Validity**: Any message in the outgoing message buffer is eventually delivered to the incoming message buffer
- **Integrity**: The message received is identical to one sent, and no messages are delivered twice

Bully Algorithm

Types of messages

- *election*, sent to announce an election
- *answer*, sent in response to an *election* message
- *coordinator*, sent to announce the identity of the elected process as new coordinator

Bully Algorithm

Failure detector (reliable)

- $T = 2\ T_{trans} + T_{process}$ is the upper band between sending a message to another process (say P) and receiving a response
  - If no response within time T => P failed
- T_{trans} – max message transmission delay
- $T_{process}$ – max delay for processing a message

Bully Algorithm

- Assume a process P detects (timeout based) that the current coordinator crashed
- P takes the following steps:
  - Broadcast *election* messages to all processes with higher IDs expecting to receive *answer* (i.e. "I am alive)"
  - If no answer, P broadcasts *coordinator*
  - If P gets an *answer* from process T with a higher ID, P waits for a certain amount of time to get *coordinator* message (someone declares itself as leader); If no such message arrives, P re-broadcast the election message
  - If P gets *election* message from another process with lower ID, it replies with *answer* message and starts new election

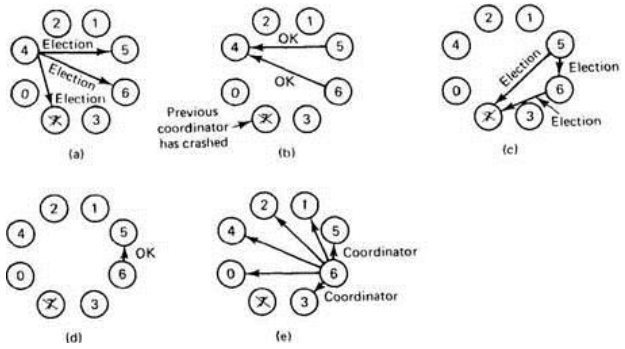
  Notes.

  (1) If P receives a message *coordinator* from a process with lower ID than itself, immediately initiates a new election (therefore the name of the algorithm)

  (2) Processes with high ID bully out of the election processes with low ID until only one process remains as coordinator

  (3) When a crashed process reboots, it issues *election* messages. Being highest ID, it wins

Bully Algorithm



The bully election algorithm.
(a) Process 4 holds an election.
(b) Processes 5 and 6 respond, telling 4 to stop.
(c) Now 5 and 6 each hold an election.
(d) Process 6 tells 5 to stop.
(e) Process 6 wins and tells everyone.

Source: http://www.e-reading.club/ 24

Distributed Mutual Exclusion (DME)

Motivation

- Resource sharing is important topic in DS
- DME is necessary to ensure shared resource consistency
- Critical Section problem
- DME as extension to DS specific of OS topics of avoiding race conditions and multithreaded applications
- Main assumptions
  - No shared memory among DS processors
  - Everything should be solved by message passing
- DME - fundamental issue in DS design
  - Requirement: Efficient and robust DME schemes
- **Note**. In some situations shared resources are managed by the associated servers that also provide mechanisms for mutual exclusion

Distributed Mutual Exclusion (DME)

Relevant examples

Example 1 (with server)

- Cooperative work of n users for creating a text file
- Updates should be consistent
  - Allowing one access at a time
  - The file should be locked (by the **editor server**) before making the updates

Distributed Mutual Exclusion (DME)

Relevant examples

Example 2 (no server)

- Collection of peer processes part of an ad-hoc IEEE 802.11 wireless network
  - Only one process may transmit data at a time over the shared medium
  - Processors must coordinate their work

Example 3 (no server)

- Monitoring system in a car park with processes at each entrance and exit point
  - Each process keeps a counter of free space
  - The processes must update the shared counter consistently
    - One way of solving the problem is to obtain mutual exclusion by processors only by exchanging messages between them

Distributed Mutual Exclusion (DME)

Main requirements

- Deadlock free
  - Two or more processes (sites) should not endlessly wait for messages that will never arrive
- Starvation free
  - A process (site) should not be forced to wait indefinitely to execute the Critical Section (CS) while other processes (sites) are repeatedly executing the CS (i.e. the CS requesting site should get an opportunity to execute CS in a finite time)

Distributed Mutual Exclusion (DME)

Main requirements (cont.)

- Fairness
  - The requests should be executed in the order they are made (in the order they arrive).
  - No physical clock => use of logical clocks and happens-before relationships
    Note. Fairness implies starvation free but not vice versa
- Fault Tolerance
  - A DME algorithm is fault tolerant if in the case of a failure, it continues to operate without any (prolonged) disruptions

Distributed Mutual Exclusion (DME)

Performance

- Bandwidth consumed – proportional to the number of messages sent in each *entry* and *exit* operation
- Client delay at each *entry* and *exit* operation
- Algorithm effect upon the system's throughput by measuring the synchronization delay (should be short for good performance) between one process exiting the critical section and the next process entering it

Distributed Mutual Exclusion (DME)

Assumptions

- A DS with N processes p_i that do not share variables
- The processes access common resources in a CS (for sake of simplicity we assume that there is only one CS);
  - Note. The algorithms could be extended to include more CSs.
- Asynchronous system
- Processes do not fail
- Message delivery is reliable (every sent message is eventually delivered intact and exactly once)
- The client is well behaving and spends a finite time accessing the shared resource in their CS

Distributed Mutual Exclusion (DME)

- Application protocol for executing the CS
  - *enter()* – enter CS; block if necessary
  - *resourceAccess()* – access the shared resource in the CS
  - *exit()* – leave CS and other processes may enter CS

Distributed Mutual Exclusion (DME)

Basic approaches to DME

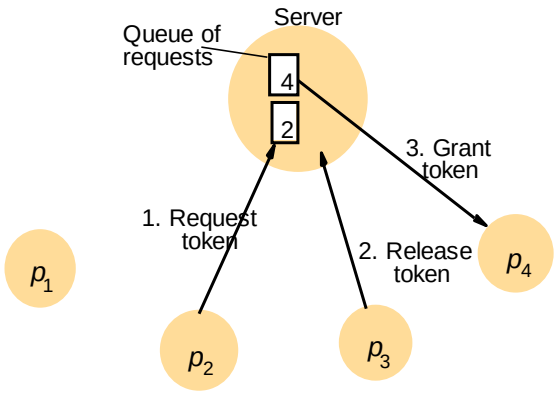
- Central control based
  - Each process equally competes for the right to use the shared resource;
  - Requests are arbiter by a central control site (process) or by distributed agreement (ex. The Central Server Algorithm)
- Token based
  - A logical token (representing the access right to the shared resource) is passed in a regular manner between processes;
  - The process which holds the token is allowed to enter the CS

Distributed Mutual Exclusion (DME)

Central Server Algorithm

- Server role – grants permission to a process to enter the CS
- To enter the CS
  - A process sends request message to server and waits for a reply
  - Server reply actually means a token granting permission to enter CS
  - If there is no process in the queue the server replies immediately
  - If other process has the token, the server places the requested process in the queue
- To exit CS
  - The process sends a message to the server giving back the token
- Granting the token
  - If the queue is non-empty the server de-queues the oldest entry in the queue, and replies to the corresponding process granting the token
- See figure

Server managing a mutual exclusion token for a set of processes



- P2 requests to enter CS and is queued after P4
- P3 exits the CS
- P4 is granted the token

Distributed Mutual Exclusion (DME)

Central Server Algorithm

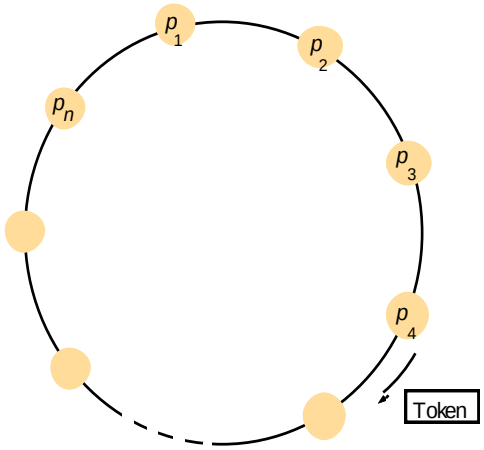
- Main disadvantages
  - Server as performance bottleneck issue
  - Server as single point of failure

Distributed Mutual Exclusion (DME)

A Ring-based Algorithm

- Totally distributed algorithm
- Arrange the N processes in a uni-directional (clockwise communication) logical ring
- How it works
  - Each process p_i has a communication channel to the next process in the ring $P_{(i+1) \bmod N}$
  - The exclusive access of a process is granted by obtained the token, passed as a message from process to process
  - If a process don't need CS - will pass the token to its neighbor process
  - If a process needs CS - will wait until receives the message, retains the token (will not pass on the message to its neighbor process)
  - When exiting the critical section - the process passes the message to its neighbor
- Note. The N processors may exchange business logic messages independent of the DME algorithm and the topology required by the algorithm

A ring of processes transferring a mutual exclusion token



Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5

Distributed Mutual Exclusion (DME)

Ricart–Agrawala [1981] using multicast and logical clocks

- N processes p_1, p_2, … p_N
- Each process has a numeric ID
- Each process keeps a Lamport logical clock
- Processes that want to enter CS multicast a message <T, p_i>
  - T is the logical clock and
  - p_i is its identifier
- Access is granted only when all other processes replied
- Each process records in a variable *state*:
  - RELEASED = process is not in the CS
  - WANTED = wants to enter CS
  - HELD = is in the CS

Ricart and Agrawala's algorithm

```
On initialization
    state := RELEASED;
To enter the section
    state := WANTED;
    Multicast request to all processes;      } request processing deferred here
    T := request's timestamp;
    Wait until (number of replies received = (N – 1));
    state := HELD;

On receipt of a request <Ti, pi> at pj (i ≠ j)
    if (state = HELD or (state = WANTED and (T, pj) < (Ti, pi)))
    then
        queue request from pi without replying;
    else
        reply immediately to pi;
    end if
To exit the critical section
    state := RELEASED;
    reply to any queued requests;
```

Distributed Mutual Exclusion (DME)

Ricart – Agrawala [1981] using multicast and logical clocks

- If a process requests entry and the state of all other processes is *RELEASED*,
  - ⇒ all processes will reply immediately to the request and the requester will obtain entry
- If some process is in the state *HELD*, then that process will not reply to requests until they finish with the critical section,
  - ⇒ the requester cannot gain entry in the meantime
- If two or more processes request entry at the same time
  - ⇒ whichever process's request bears the lowest timestamp will be the first to collect $N - 1$ replies, granting it entry next.
  - ⇒ If the requests bear equal Lamport timestamps, the requests are ordered according to the processes' corresponding identifiers.
- Notes
  - When a process requests entry, it defers processing requests from other processes until its own request has been sent and it has recorded the timestamp *T* of the request
  - This is so that processes make consistent decisions when processing requests

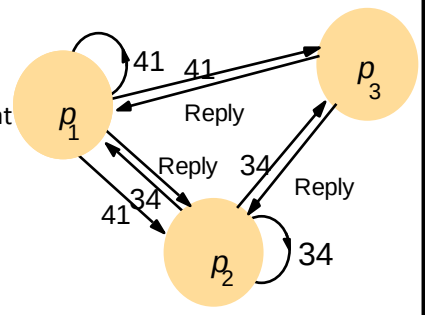
Distributed Mutual Exclusion (DME)

Ricart – Agrawala [1981] using multicast and logical clocks

- See next figure
- To illustrate the algorithm, consider a situation involving three processes: *p*1 , *p*2 and *p3*
- Assumptions:
  - *p*3 is not interested in entering the critical section,
  - *p*1 and *p*2 request entry concurrently
- The timestamp of *p*1's request is 41, and that of *p*2 is 34.
- When *p*3 receives their requests, it replies immediately
- When *p*2 receives *p*1's request
  - It finds that its own request has the lower timestamp and so does not reply, holding *p*1 off.
  - However, *p*1 finds that *p*2 's request has a lower timestamp than that of its own request and so replies immediately
- On receiving this second reply, *p*2 can enter the critical section
- When *p*2 exits the critical section, it will reply to *p*1 's request and so grant it entry

Multicast synchronization

- Consider a situation involving three processes: *p*1 , *p*2 and *p3*
- Assumptions:
  - *p*3 is not interested in entering the critical section,
  - *p*1 and *p*2 request entry concurrently
- The timestamp of *p*1's request is 41, and that of *p*2 is 34.
- When *p*3 receives their requests, it replies immediately
- When *p*2 receives *p*1's request
  - It finds that its own request has the lower timestamp and so does not reply, holding *p*1 off.
- When p1 receives p2's request
  - *p*1 finds that *p*2 's request has a lower timestamp than that of its own request and so replies immediately
- On receiving this second reply, *p*2 can enter the critical section
- When *p*2 exits the critical section, it will reply to *p*1 's request and so grant it entry



Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5
© Pearson Education 2012

Distributed Mutual Exclusion (DME)

Ricart – Agrawala [1981] using multicast and logical clocks

Analysis

- Gaining entry takes 2(N-1) messages: N-1 to multicast and N-1 replies;
- Having hardware support for multicast only one message is required for the request and N-1 replies (total N messages)
- More expensive in terms of bandwidth compared to previous algorithms
- Advantage
  - Synchronization delay is only one message transmission time (both previous algorithms need a round-trip synchronization delay)

Distributed Mutual Exclusion (DME)

Ricart – Agrawala [1981] using multicast and logical clocks

Analysis (cont)

- The algorithm performance can be improved
- (1) The process that last entered the critical section and that has received no other requests for it still goes through the protocol as described, even though it could simply decide locally to reenter the critical section
- (2) Second, Ricart and Agrawala refined this protocol so that it requires *N* messages to obtain entry in the worst (and common) case, without hardware support for multicast. This is described in Raynal [1988]

Consensus

The basics

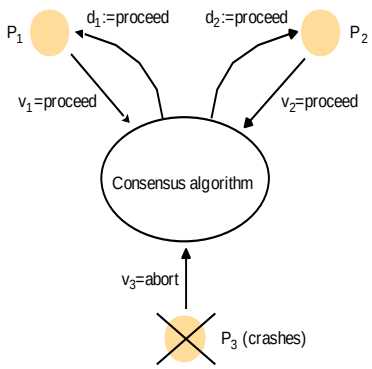
- Fundamental issue in fault-tolerant DS
- Consensus (reaching agreement) = getting all processes (e.g. servers) of a group to agree on a value based on the votes of each process
  - Allows a set of machines to work as a coherent group that survive failures
  - Important for building reliable large-scale DS
- The agreed value must be proposed by at least of one process
- Usual values { 0, 1} (or decide to take a certain action or not)
- Examples of consensus
  - Election algorithms (agree on a leader),
  - Distributed transactions (agree on commit or abort),
  - Mutual exclusion (decide who enters CS).

Consensus
System Model

- Set of p_i processes, i = 1, 2, … N
- Message passing
- Communications is reliable
- Variations: synchronism and faults
  - Processes may fail (arbitrary (Byzantine) or crash)
  - Fischer[1985]: In an **asynchronous system**, a set of processes containing (only) **one** faulty process **cannot be guaranteed to reach consensus**

Consensus
System Model

- To reach consensus in consensus algorithms
  - (undecided state) Each process proposes (e.g. using unreliable multicast) a single value v_i from a set D
  - (decided state) As a result of executing the consensus algorithm a decided value is selected; this value is assigned to a variable d_i defined in each process
- Figure
  - Three processes engaged in a consensus algorithm
  - P1 and P2 propose *proceed*;
  - P3 proposes *abort* and then crashes
  - The decided value is *proceed*



Consensus

Requirements of Consensus Algorithms (CA)

- Termination
  - Eventually each correct process sets its decision variable
- Agreement
  - The decision value of all correct processes is the same
- Integrity (or Validity)
  - If the correct processes all proposed the same value, then any correct process in the *decided* state has chosen that value
  - Integrity could be defined in many ways, depending on application

Consensus

Implementing CA

Case of Non-failing processes

- Consider for simplicity a system in which processes don't fail
- Consider the group of processes
- Each process reliably multicast its proposed value to group members
- Each process waits until it has collected all N values (including its own)
- Evaluate a function majority which returns the value that most often occurs, or a special value if no majority exists
- Properties
  - Agreement and integrity is guaranteed by the definition of majority function
  - Termination is guaranteed by the reliability of the multicast operation

Consensus

Implementing CA

Case of failing processes

- Consider arbitrary way (Byzantine) of processes fail (Byzantine generals problem)
- Faulty processes may communicate random values to others (bug or malicious processes)
- The decided value will be the one agreed by the correct processes; If a commander process is present (and is correct) than the value proposed by the commander is decided
- Consensus algorithms make progress when any majority of their processes are available
  - Ex. A cluster of 5 servers continue to operate even if 2 servers fail. If more servers fail they stop making progress (but will never return an incorrect result)

Consensus

Fischer-Lynch-Patterson (1985)

- No consensus can be guaranteed in an asynchronous communication system in the presence of any failures
- Internet is asynchronous
- Practical implications
  - Distributed transaction commit
  - Consistent replication
  - Reliable ordered multicast communication in groups

Consensus

Fischer-Lynch-Patterson (1985)

- How to avoid consequences – tricks to apply synchronous algorithms
  - Fault masking – assume that failed processes always recover and then reintegrate them in the group
    - If you (algorithm) haven't heard from a process, wait longer
    - A round terminates when every expected message is received
  - Failure detectors (FD): build a FD that can determine if a process has failed

Consensus

Fischer-Lynch-Patterson FLP (1985)

- FD – How to detect process failure
  - Timeout, ping, heartbeats, beacons
  - Recovery notifications
- Accuracy of FD
- Is FD live?
  - In an asynchronous system a FD could be accurate or live (but not both)
  - FLP actually says that it is impossible for an asynchronous system to agree on anything with accuracy and liveness

Consensus

- **Safety** [Lamport 1977]
  - A **safety** property asserts that nothing bad happens during system/program execution (ERROR/STOP-DEADLOCK states are not reachable)
  - Example: In case of a sequential program safety means that the program will never produce a wrong result ("partial correctness")
- **Liveness** [Lamport 1977]
  - A **liveness** property asserts that something good eventually happens (but we don't know when) (i.e. makes progress)
  - Example: In case of a sequential program liveness means that the program will produce a result ("termination")
- **Progress**
  - A progress property asserts that it is always the case that an action is eventually executed (**progress** is opposite to **starvation**)
- **Starvation**
  - A concurrent programming situation in which an action is never executed

Consensus

Current approach for FT services

- Consensus algorithms allow a collection of machines to work as a coherent group that can survive the failures of some of its members.
- Because of this, they play a key role in building reliable large-scale software systems
- Consensus (typically) arises in the context of replicated state machines - a general approach to building fault-tolerant systems
- In this approach, state machines on a collection of servers compute identical copies of the same state and can continue operating even if some of the servers are down
- Each server has a state machine and a log
- The state machine is the component that we want to make fault-tolerant
- It will appear to clients that they are interacting with a single, reliable state machine, even if a minority of the servers in the cluster fail
- Each state machine takes as input commands from its log
- The consensus algorithm must ensure that each state machine processes the same series of commands and thus produces the same series of results and arrives at the same series of states

Consensus

Current approach for FT services

- Replicated State Machines (RSM) for fault-tolerant services
  - It is achieved by replicating the servers that fail independently
    - Physical and electrical independent processors that execute the servers
  - Algorithms to coordinate client interactions with server replicas
- A State Machine (SM) is defined by:
  - a set of States,
  - a set of Inputs,
  - a set of Outputs,
  - a transition function (Input X State -> State),
  - an output function (Input X State -> Output) and
  - a Start state.
- A state is stable until a new input (service request) is received and the output is transmitted to the caller (client).
- Multiple copies of deterministic state machines are replicated to servers
  - They should receive the same series of inputs (requests) in the same order, thus generating the same transitions and the same outputs
  - The majority vote decides the correct output
    - A system that supports F failures must have 2F + 1 replicas
    - Additional discussion is necessary when particular system aspects are considered such as: Fail-Stop Faults, Byzantine Faults, Malicious attacks, cryptographic or non-cryptographic techniques, etc.

Consensus

Current approach FT services

How it works

1. Distribute copies of the same state machines on multiple independent servers
2. Get client requests (inputs to the SMs)
3. **Determine an ordering of the inputs**

   Why necessary (we have a DS and the inputs may arrive in different order over the communication network)?

   Most difficult step – consensus order algorithms being usually necessary
4. Execute inputs in the chosen order on each SM and generate Output
5. Reply to clients with SM Output
   1. Faulty outputs must be filtered out
   2. If no majority => the unique output should be: FAIL
6. Monitor replicas for differences in State and Output
   1. Non faulty replicas will always generate the same state and output
   2. A replica with state and output different from majority is declared Faulty

Consensus

Current approach for FT services - Paxos

- Paxos [1989] - Asynchronous Consensus Algorithm in a network of un-reliable processors
  - Named after a lost civilization (fictional) legislative consensus system used on Paxos island (Greece)
- Paxos solves consensus and may be used as algorithm for consensus order
- Paxos require a single leader to ensure liveness, i.e. one of the replicas must be leader long enough to achieve consensus on the next operation of the SM
- System behavior is unaffected if the leader changes after every instance, or if the leader changes multiple times per instance. The only requirement is that one replica remains leader long enough to move the system forward.
- In general, a leader is necessary only when there is disagreement about which operation to perform and if those operations conflict in some way (for instance, if they do not commute)
- When conflicting operations are proposed, the leader acts as the single authority to set the record straight, defining an order for the operations, allowing the system to make progress.

Consensus

Current approach for FT services - Paxos

- FLP: No asynchronous consensus algorithm can guarantee both safety and liveness
- Paxos guarantees **safety** (consistency) but not guarantees **liveness (**i.e. making **progress)**
- Paxos is usually used where durability is required
  - For example, to replicate a file or a database, in which the amount of durable state could be large
  - The protocol attempts to make progress even during periods when some bounded number of replicas are unresponsive. There is also a mechanism to drop a permanently failed replica or to add a new replica.

Consensus

Current approach for FT services - Paxos

- Family of Paxos protocols
  - Basic Paxos
  - Multi-Paxos
  - Cheap Paxos
  - Fast Paxos
  - Generalized Paxos
  - Byzantine Paxos

Consensus

Current approach for FT services

RAFT [Ongaro, Ousterhout]

- https://raft.github.io/
- Improved Paxos
- RAFT - a generic way to distribute a state machine to a cluster of computing systems, ensuring that each node in the cluster agrees upon the same series of state transitions
- Achieves consensus via an Elected Leader
- Applies to cluster of servers operating based on Replicated State Machines (RSM) and logs
- Each server in the cluster is either: leader, candidate or follower
- Leader:
  - Replicates the log to the followers
  - Regularly sends heartbeat messages to followers (informing them that it is alive)
  - Each follower sets a timeout (150.. 300 ms) for receiving messages from the leader
  - Timeout is reset after receiving the heartbeat
  - If no message is received, the follower changes its status to candidate and starts a new leader election
- RSM are the basics for solving a variety of fault-tolerance problems in DS
- Examples
  - Large scale systems having single cluster leader such as GFS (Google File System), HDFS (Hadoop File System), RamCloud use a separate replicated state machine to manage leader election and store configuration information that must survive leader's crash
  - Google Chubby (lock service for loosely coupled services)
- Each server has an associated state machine that executes commands that reside in a log of commands
- See figure

Consensus

RAFT [Ongaro, Ousterhout]

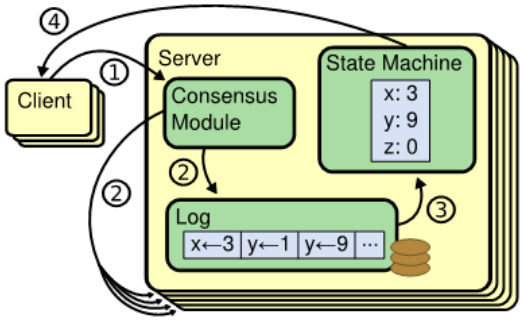


Figure 1:Replicated state machine architecture. The consensus algorithm manages a replicated log containing state machine commands from clients. The state machines process identical sequences of commands from the logs, so they produce the same outputs.

Source: Ongaro, Ousterhout]

Consensus

RAFT [Ongaro, Ousterhout]

- Each server stores a log containing a series of commands, which its state machine executes in order.
- Each log contains the same commands in the same order, so each state machine processes the same sequence of commands. Since the state machines are deterministic, each computes the same state and the same sequence of outputs.
- Keeping the replicated log consistent is the job of the consensus algorithm.
- The consensus module on a server (previously elected as leader) receives commands from clients and adds them to its log.
- It communicates with the consensus modules on other servers to ensure that every log eventually contains the same requests in the same order, even if some servers fail.
- Once commands are properly replicated, each server's state machine processes them in log order, and the outputs are returned to clients.
- As a result, the servers appear to form a single, highly reliable state machine

Consensus

RAFT [Ongaro, Ousterhout]

- Raft Implementation – Leader election and its role
  - A leader is elected
  - The leader has complete responsibility for the management of the replicated log
  - The leader accepts log entries from clients, replicates them on other servers, and tells servers when it is safe to apply log entries to their state machines.
  - Having a leader simplifies the management of the replicated log. For example, the leader can decide where to place new entries in the log without consulting other servers, and data flows in a simple fashion from the leader to other servers.
  - A leader can fail or become disconnected from the other servers, in which case a new leader is elected.

Consensus

RAFT [Ongaro, Ousterhout]

- Raft Implementation – Server types
  - At any given time each server is in one of three states: leader, follower, or candidate.
  - In normal operation there is exactly one leader and all of the other servers are followers.
  - Followers are passive: they issue no requests on their own but simply respond to requests from leaders and candidates.
  - The leader handles all client requests (if a client contacts a follower, the follower redirects it to the leader).
  - The third state, candidate, is used to elect a new leader

Consensus

RAFT [Ongaro, Ousterhout]

- Raft Implementation – Triggering leader election
  - Heartbeat mechanism is used to trigger leader election
  - Leaders send periodic heartbeats (AppendEntries RPCs that carry no log entries) to all followers in order to maintain their authority.
  - If a follower receives no communication over a period of time called theelection timeout, then it assumes there is no vi-able leader and begins an election to choose a new leader

Consensus

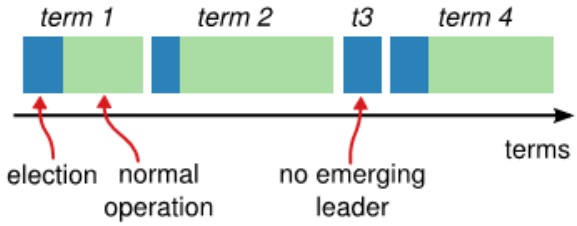
RAFT [Ongaro, Ousterhout]

- Raft Implementation – Timing
  - Raft divides time into terms of arbitrary length, as shown in Figure. Terms are numbered with consecutive integers.
  - Each term begins with an election, in which one or more candidates attempt to become leader (as described)
  - If a candidate wins the election, then it serves as leader for the rest of the term.
  - In some situations an election will result in a split vote. In this case the term will end with no leader; a new term (with a new election) will begin shortly.
  - Raft ensures that there is at most one leader in a given term

Consensus

RAFT [Ongaro, Ousterhout]

- Raft Implementation – Timing



Consensus

RAFT [Ongaro, Ousterhout]

- Raft Implementation – Communication
  - Raft servers communicate using remote procedure calls (RPCs)
  - The basic consensus algorithm requires only two types of RPCs.
    - RequestVote RPCs are initiated by candidates during elections and
    - AppendEntries RPCs are initiated by leaders to replicate log entries and to provide a form of heartbeat.
  - A third RPC is for transferring snapshots between servers.
  - Servers retry RPCs if they do not receive a response in a timely manner, and they issue RPCs in parallel for best performance

Consensus

RAFT [Ongaro, Ousterhout]

- Raft Implementation – Log replication
  - Once a leader has been elected, it begins servicing client requests.
  - Each client request contains a command to be executed by the replicated state machines.
  - The leader appends the command to its log as a new entry, then is-sues AppendEntries RPCs in parallel to each of the other servers to replicate the entry.
  - When the entry has been safely replicated (as described below), the leader applies the entry to its state machine and returns the result of that execution to the client.
  - If followers crash or run slowly, or if network packets are lost, the leader retries Append-Entries RPCs indefinitely (even after it has responded to the client) until all followers eventually store all log entries

Consensus

RAFT [Ongaro, Ousterhout]

- Raft Implementation – Log organization
- Each log entry stores a state machine command along with the term number when the entry was received by the leader.
- The term numbers in log entries are used to detect inconsistencies between logs and to ensure some of the properties such as Election-Safety, State Machine Safety, Leader Completeness.
- Each log entry also has an integer index identifying its position in the log
- The leader decides when it is safe to apply a log entry to the state machines; such an entry is called committed.
- Raft guarantees that committed entries are durable and will eventually be executed by all of the available state machines.
- A log entry is committed once the leader that created the entry has replicated it on a majority of the servers
- This also commits all preceding entries in the leader's log, including entries created by previous leaders

Consensus

RAFT [Ongaro, Ousterhout]

- Raft Implementation – Consistency
  - During normal operation, the logs of the leader and followers stay consistent, so the AppendEntries consistency check never fails.
  - However, leader crashes can leave the logs inconsistent (the old leader may not have fully replicated all of the entries in its log). These inconsistencies can compound over a series of leader and follower crashes.
  - A follower may be missing entries that are present on the leader, it may have extra entries that are not present on the leader, or both.
  - Missing and extraneous entries in a log may span multiple terms
  - In Raft, the leader handles inconsistencies by forcing the followers' logs to duplicate its own. This means that conflicting entries in follower logs will be overwritten with entries from the leader's log.
  - To bring a follower's log into consistency with its own, the leader must find the latest log entry where the two logs agree, delete any entries in the follower's log after that point, and send the follower all of the leader's entries after that point.
  - All of these actions happen in response to the consistency check performed by AppendEntries RPCs