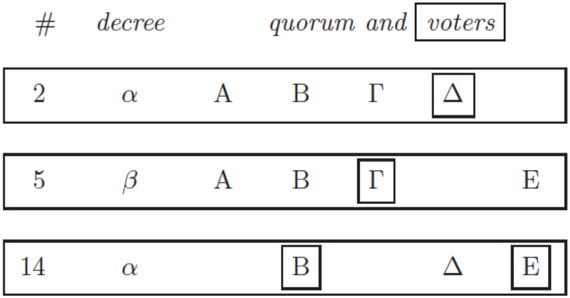
Dependable Distributed Systems – 5880V/UE

Part 4: Consensus, failure detectors, and Paxos – 2021-10-07

Prof. Dr. Hans P. Reiser | WS 2021/22

UNIVERSITÄT PASSAU



#	*decree*	*quorum and* $\boxed{\textit{voters}}$				
2	α	A	B	Γ	$\boxed{\Delta}$	
5	β	A	B	$\boxed{\Gamma}$		E
14	α		$\boxed{B}$		Δ	$\boxed{E}$

Previous lectures

Broadcast (1-to-n communication) with several semantic variants

- best effort, reliable, uniform reliable
- unordered, fbcast, cbcast, abcast, fabcast, cabcast
- atomic broadcast and consensus

Today's lecture

Consensus (theory)

- Fundamantal impossibilities, FLP

Failure detection in distributed systems

- Toueg's theory of failure detector hierarchies
- Implementing consensus with failure detectors

Towards practical fault-tolerant services

- Service replication strategies
- Paxos algorithm

Overview

Consensus

- Michael J. Fischer, Nancy A. Lynch, Mike Paterson: *Impossibility of Distributed Consensus with One Faulty Process.* J. ACM 32(2): 374-382 (1985)
- Consensus cannot be solved determistically in asynchronous systems if a process can fail

Consensus

- Michael J. Fischer, Nancy A. Lynch, Mike Paterson: *Impossibility of Distributed Consensus with One Faulty Process.* J. ACM 32(2): 374-382 (1985)
- Consensus cannot be solved determistically in asynchronous systems if a process can fail
- Why? Impossible to distinguish a crashed process from a slow process. . .

Consensus

- Michael J. Fischer, Nancy A. Lynch, Mike Paterson: *Impossibility of Distributed Consensus with One Faulty Process.* J. ACM 32(2): 374-382 (1985)
- Consensus cannot be solved determistically in asynchronous systems if a process can fail
- Why? Impossible to distinguish a crashed process from a slow process. . .
- Dead end? No, it is solvable if . . .

FLP

Fischer, Lynch and Paterson have formally proven that a correct, deterministic solution for the consensus problem does not exist if

- the system model is asynchronous, and
- at least one node may fail by crashing

In the following slides

- Sketch of the proof
- Proof itself not relevant for exam, but **the FLP result is!**

FLP

Terminology:

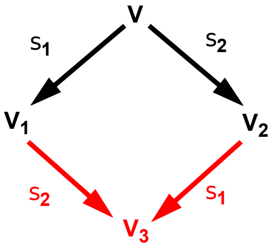
- If **all** possible executions starting in some configuration V (=global state) result in the same decision, the configuration V is called *univalent (0-valent or 1-valent)*
    - Each state V, in which a (correct) process has made a decision, must be univalent
- Otherwise, if both decisions are possible, then the configuration V is called *bivalent*

FLP

Lemma 1

- Let's consider two sequences of steps s_1 and s_2, which both can be executed in state V

If s_1 and s_2 are sequences of steps on a disjunct sets of nodes, then the order of execution of s_1 and s_2 is irrelevant


$$\begin{tikzcd} & V \arrow[dl, "s_1"'] \arrow[dr, "s_2"] & \\ V_1 \arrow[dr, "s_2"'] & & V_2 \arrow[dl, "s_1"] \\ & V_3 & \end{tikzcd}$$

FLP

Lemma 2:

If at least a single process may crash, then a bivalent initial configuration must exist

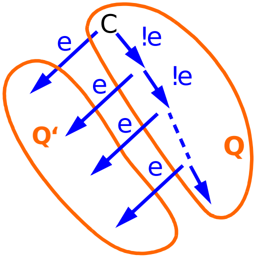
- This means that some configuration exists in which the decicion is not a priori fixed (for some executions, the decision will be 1, for others the decision will be 0)

  (Idea of proof: Construct a list of initial states in which each state differs from the next only in the local initial state of a single process. The first initial state is „all 0”, which must be 0-valent, and the final initial state in the list is „all 1”, which must be 1-valent. It is not possible to have an 0-valent initial state on the list adjunct to a 1-valent initial state, as they differ only in the local state of a single process, and would have to make a decision even if this single process failed initially.)

FLP

Sketch of proof of FLP impossibility

- Assumptions
  - C is a bivalent state
  - *e* is an arbitrary step applicable in *C* on process P_p
  - *Q* is the set of states that can be reached without applying *e*
  - Let $Q' = e(Q) = \{e(E) | E \in Q\}$
- If for each bivalent state *C* there exists a step *e* such that Q' contains a bivalent state, then you can construct an infinite sequence of steps in which the algorithm does not terminate. This would contradict the assumption of a correct algorithm.



FLP

Consequence: Lemma 3

There must exist a bivalent state C, for which Q' only contains uni-valent states.

Q' must contain 0-valent and 1-valent states

- As C is bivalent, there must exist reachable 0-valent and 1-valent states E_0 and E_1
- If E_0/E_1 is in Q (i.e., reachable without executing step e), the next state $e(E_0)/e(E_1)$ is part in Q' and is 0-valent/1-valent
- Otherwise (i.e., if E_0/E_1 was reached by executing e) there is a predecessor of E_0/E_1 in Q' that is 0-valent/1-valent respectively

FLP

Interim results:

- C contains a bivalent state
- There exists a Q' with only univalent states
- Q' must contain 0-valent and 1-valent states

FLP

Interim results:

- C contains a bivalent state
- There exists a Q' with only univalent states
- Q' must contain 0-valent and 1-valent states

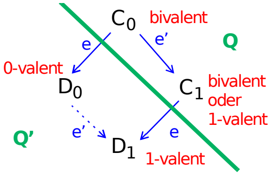
Implication:

- Step e can result in 0-valent and 1-valent states
- Therefore, there must exists two *neighbour* states C_0, C_1 in Q, for which
  - $D_0 = e(C_0)$ is 0-valent,
  - $D_1 = e(C_1)$ is 1-valent

FLP

Assumption (WLOG): $C_1 = e'(C_0)$

- C_0, C_1 are part of Q,
- D_0, D_1 are part of Q',
- D_0 is 0-valent and D_1 is 1-valent



- Steps e and e' cannot happen on different nodes, as otherwise, according to Lemma 1, $D_1 = e'(D_0)$. This is impossible, as a 1-valent state cannot be a successor of a 0-valent state
- This means that (for any correct algorithm) there must exist a node P_p that locally decides about the result. The other nodes cannot decide autonomously without P_p. If P_p crashes in state C_0, then no decision is possible.

Which basically proofs the FLP impossibility.

FLP in real life

Practical ways to circumvent the FLP impossibility

- Allow the protocol to not guarantee agreement
  - Example: Use heuristics to exclude faulty or slow nodes, guarantee agreement only among those nodes that have not been excluded
- Allow the protocol to not always terminate – “partially correct” algorithm that does not guarantee liveness properties
  - Examples: Paxos, PBFT
- Proof assumes a *deterministic model*. There might exist probabilistic algorithms (with probabilistic termination)
  - Examples: Ben Or, ABBA
- The proof assumes a completely asynchronous system model. Algorithms might exist for other system models
  - Examples: synchronous systems, partial synchronous systems, asynchronous system + unreliable failure detectors

Overview

Failure detection without distribution

- Detection can be considered reliable
  - Why?

Failure detection without distribution

- Detection can be considered reliable
  - Why?
- Various mechanisms:
  - Self-checking routines (e.g. parity checks)
  - Guardian components (e.g. memory access checkers)
  - Watch-dogs (hardware-based, software-based)

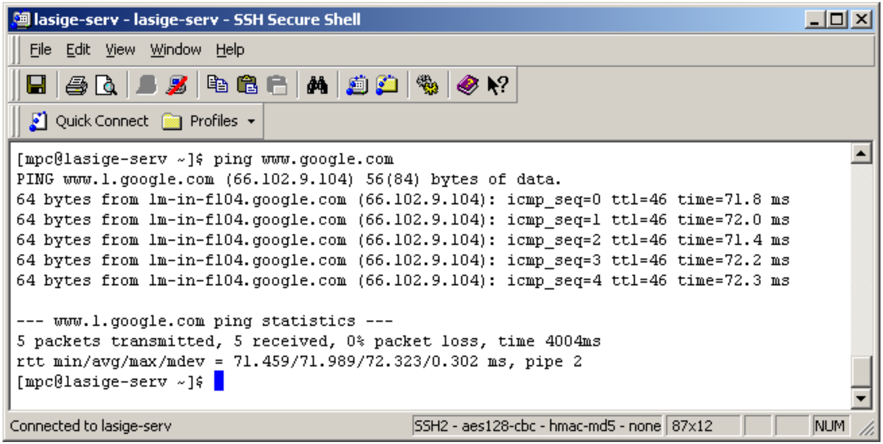
Failure detection without distribution

- Detection can be considered reliable
  - Why?
- Various mechanisms:
  - Self-checking routines (e.g. parity checks)
  - Guardian components (e.g. memory access checkers)
  - Watch-dogs (hardware-based, software-based)
- Temporal problems may still affect accuracy

Failure detection without distribution

- Detection can be considered reliable
  - Why?
- Various mechanisms:
  - Self-checking routines (e.g. parity checks)
  - Guardian components (e.g. memory access checkers)
  - Watch-dogs (hardware-based, software-based)
- Temporal problems may still affect accuracy
- From now on: distributed failure detection
  - the following: failure = crash

Failure detectors 101: ping


lasige-serv - lasige-serv - SSH Secure Shell
File Edit View Window Help
Quick Connect
Profiles

```
[mpc@lasige-serv ~]$ ping www.google.com
PING www.l.google.com (66.102.9.104) 56(84) bytes of data.
64 bytes from lm-in-f104.google.com (66.102.9.104): icmp_seq=0 ttl=46 time=71.8 ms
64 bytes from lm-in-f104.google.com (66.102.9.104): icmp_seq=1 ttl=46 time=72.0 ms
64 bytes from lm-in-f104.google.com (66.102.9.104): icmp_seq=2 ttl=46 time=71.4 ms
64 bytes from lm-in-f104.google.com (66.102.9.104): icmp_seq=3 ttl=46 time=72.2 ms
64 bytes from lm-in-f104.google.com (66.102.9.104): icmp_seq=4 ttl=46 time=72.3 ms

--- www.l.google.com ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4004ms
rtt min/avg/max/mdev = 71.459/71.989/72.323/0.302 ms, pipe 2
[mpc@lasige-serv ~]$
```

Connected to lasige-serv | SSH2 - aes128-cbc - hmac-md5 - none | 87x12 | NUM

- Typically ICMP: echo-request, echo-reply

Failure detectors (FDs)

- Ping provides *suspicions (hints)* of failure (crash)
- Failure detectors are *distributed oracles* that provide *hints*
  - local modules in computers
  - which communicate using the network

Failure detection

Classification of basic mechanisms:

- Heartbeat mechanism:
  - Observed component periodically sends messages
  - Usually called “I’m alive” messages or “heartbeats”
  - When these messages are missing component is considered failed
  - Heartbeats are sent spontaneously

Failure detection

Classification of basic mechanisms:

- Heartbeat mechanism:
  - Observed component periodically sends messages
  - Usually called “I’m alive” messages or “heartbeats”
  - When these messages are missing component is considered failed
  - Heartbeats are sent spontaneously
- Probe mechanism:
  - Observed component waits for a probe message and replies
  - Probe comes from the failure detector
  - Component replies with “I’m alive” message

Failure detection: properties and problems

- Consistency of distributed failure detection is desirable:
  - When a process goes down all the other processes know about it and can coordinate their actions to implement corrective measures
- **Strong Accuracy**
  - A safety requirement, specifying that no correct process is ever considered failed
- **Strong Completeness**
  - A liveness requirement, specifying that a failure must eventually be detected by every correct process
- *Possible to implement??*

Strong accuracy and strong completeness

- *Possible to implement??*

Strong accuracy and strong completeness

- *Possible to implement??*
- If “synchronous perfect channels” are available, heartbeat exchanges meet strong accuracy and strong completeness
  - Synchronous perfect channel = reliable link (no loss, no duplication, no spurious) + synchronous (upper bound on communication delay)
- Such a detector is called **perfect failure detector**:
  - If (and only if) a node crashes all correct nodes will note the absence of the heartbeat and detect the failure.

Failure detection: properties and problems

- What happens when the channel that interconnects the processes is not good enough?

Failure detection: properties and problems

- What happens when the channel that interconnects the processes is not good enough?
  - Either imperfection can be fixed by some protocol
  - Or imperfection is impossible to overcome

Failure detection: properties and problems

- What happens when the channel that interconnects the processes is not good enough?
  - Either imperfection can be fixed by some protocol
  - Or imperfection is impossible to overcome
- Communication channel is not perfect but has some mild imperfection:
  - For instance, it makes a small number k of omissions
  - Solution: transform the imperfect channel into a perfect channel, using redundancy
  - E.g., each heartbeat can be retransmitted $k + 1$ times, effectively ensuring that it is observed by all correct processes.

Failure detection: properties and problems

- Channel imperfection impossible to overcome:
  - Lack of bounds on the **number and type** of faults the communication channel may give (e.g. number of omission faults not bounded)

Failure detection: properties and problems

- Channel imperfection impossible to overcome:
  - Lack of bounds on the **number and type** of faults the communication channel may give (e.g. number of omission faults not bounded)
- Consequence:
  - Now, if a process does not receive any heartbeat message from the other process this may have two causes:
    - Because the other process has failed, or
    - Because the channel has dropped all heartbeats sent so far!

Failure detection: properties and problems

- Channel imperfection impossible to overcome:
  - The lack of bounds for the **timely behavior** of system components (processes or links) — *asynchrony*

Failure detection: properties and problems

- Channel imperfection impossible to overcome:
  - The lack of bounds for the **timely behavior** of system components (processes or links) — *asynchrony*
- Consequence:
  - No way to distinguish a missing from “extremely slow” heartbeat
  - Happens if a link can delay a message arbitrarily, or if a process can take an arbitrary amount of time to make a processing step
  - **Perfect failure detection cannot be implemented in asynchronous systems!**
  - Problem: for practical purposes, Internet “is” asynchronous

Hierarchy of FDs

- Chandra&Toueg have defined the notion of unreliable FD and a hierarchy of such FDs

T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. J. ACM, 43(2):225-267, Mar. 1996

Hierarchy of FDs

- Chandra&Toueg have defined the notion of unreliable FD and a hierarchy of such FDs

T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. J. ACM, 43(2):225-267, Mar. 1996

- Perfect failure detector $\mathcal{P}$
  - **Strong Accuracy.** No process is suspected before it crashes.
  - **Strong Completeness.** Eventually every process that crashes is permanently suspected by every correct process.

Failure detection: properties and problems

It is possible to relax the properties of FDs:

- **Weak accuracy**
  - **At least one** correct process is never suspected (by all correct processes)
- **Weak completeness**
  - Eventually every process that crashes is permanently suspected by **at least one** correct process

Failure detection: properties and problems

It is possible to relax the properties of FDs:

- **Weak accuracy**
  - **At least one** correct process is never suspected (by all correct processes)
- **Weak completeness**
  - Eventually every process that crashes is permanently suspected by **at least one** correct process

Note: Even **weak accuracy** is impossible in asynchronous systems!

Failure detection: properties and problems

It is possible to relax the properties of FDs:

- **Weak accuracy**
  - **At least one** correct process is never suspected (by all correct processes)
- **Weak completeness**
  - Eventually every process that crashes is permanently suspected by **at least one** correct process

Note: Even **weak accuracy** is impossible in asynchronous systems!

Therefore:

- **Eventual strong accuracy**
  - **There is a time** after which (all) correct processes are never considered failed by any correct processes
- **Eventual weak accuracy**
  - **There is a time** after which some correct process is never considered failed by any correct processes

Eventually weak failure detection $\diamond\mathcal{W}$

The *weakest*, defined in terms of two properties:

- **Eventual weak accuracy**
  - **There is a time** after which **some correct process** is never considered failed by any correct processes
- **Weak completeness**
  - Eventually every process that crashes is permanently suspected by **at least one** correct process

Eventually weak failure detection $\diamond\mathcal{W}$

The *weakest*, defined in terms of two properties:

- **Eventual weak accuracy**
  - **There is a time** after which **some correct process** is never considered failed by any correct processes
- **Weak completeness**
  - Eventually every process that crashes is permanently suspected by **at least one** correct process

- Consensus is solvable in an asynchronous system with an eventually weak FD
  - Notice that this is a system model different from the asynchronous system model; the eventually weak FD is not implementable in the latter
  - Proved by Chandra, Hadzilacos and Toueg

Hierarchy of failure detectors

- Note: the diamond $\diamond$ stands for "eventually"

Completeness	**Accuracy**			
	Strong	Weak	Eventually strong	Eventually weak
Strong	*Perfect* $\mathcal{P}$	*Strong* $\mathcal{S}$	*Eventually perfect* $\diamond\mathcal{P}$	*Eventually Strong* $\diamond\mathcal{S}$
Weak	$\mathcal{Q}$	*Weak* $\mathcal{W}$	$\diamond\mathcal{Q}$	*Eventually weak* $\diamond\mathcal{W}$

Weakest failure detector revisited

Completeness	Accuracy			
	Strong	Weak	Eventually strong	Eventually weak
Strong	*Perfect* $\mathcal{P}$	*Strong* $\mathcal{S}$	*Eventually perfect* $\diamond\mathcal{P}$	*Eventually Strong* $\diamond\mathcal{S}$
Weak	$\mathcal{Q}$	*Weak* $\mathcal{W}$	$\diamond\mathcal{Q}$	*Eventually weak* $\diamond\mathcal{W}$

- w/ crash faults: trivial to obtain *strong completeness* from *weak completeness* (so using $\diamond\mathcal{S}$ is basically the same as $\diamond\mathcal{W}$)

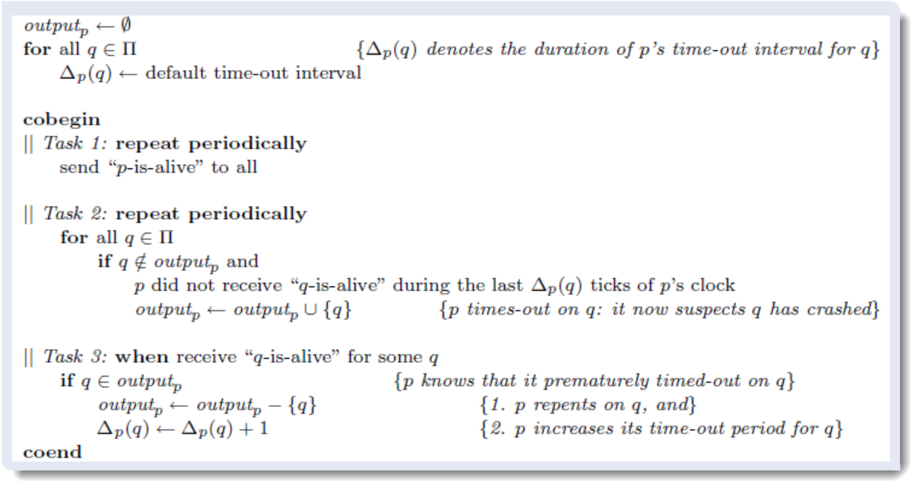
Implementing a FD of class $\diamond\mathcal{P}$

(that also implements weaker FDs: $\diamond\mathcal{W}$, $\diamond\mathcal{S}$)

Consider the following system model:

- **Partial synchrony**
  - For every run *S* there is a Global Stabilization Time (GST) after which there are bounds on the relative process speeds and communication delays
  - (both GST and the bounds are unknown)
- Reliable channels
- Processes can fail by crashing
- Explicar que Partial synchrony é modelo intermédio entre synchrony and asynchrony e que é um modelo "razoável"

Implementing a FD of class $\diamond\mathcal{P}$



```
$output_p \leftarrow \emptyset$
for all $q \in \Pi$                                   {$\Delta_p(q)$ denotes the duration of $p$'s time-out interval for $q$}
    $\Delta_p(q) \leftarrow$ default time-out interval

cobegin
|| Task 1: repeat periodically
    send "$p$-is-alive" to all

|| Task 2: repeat periodically
    for all $q \in \Pi$
        if $q \notin output_p$ and
            $p$ did not receive "$q$-is-alive" during the last $\Delta_p(q)$ ticks of $p$'s clock
            $output_p \leftarrow output_p \cup \{q\}$          {$p$ times-out on $q$: it now suspects $q$ has crashed}

|| Task 3: when receive "$q$-is-alive" for some $q$
    if $q \in output_p$                          {$p$ knows that it prematurely timed-out on $q$}
        $output_p \leftarrow output_p - \{q\}$                {1. $p$ repents on $q$, and}
        $\Delta_p(q) \leftarrow \Delta_p(q) + 1$                  {2. $p$ increases its time-out period for $q$}
coend
```

Solving consensus with $\diamond\mathcal{S}$

- Impossible in asynchronous systems, but solvable with $\diamond\mathcal{S}$
- Consensus:

- *Termination:* Every correct process eventually decides some value
- *Agreement:* No two correct processesss decide on different values v, v'
- *Validity:* If a correct process decides v, then v was previously proposed by some process.
- *Integrity:* No process decides twice

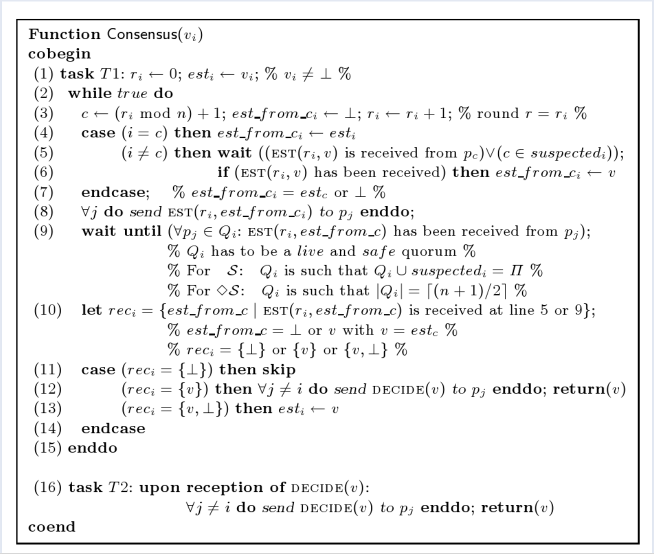
Solving consensus with $\diamond\mathcal{S}$

- Impossible in asynchronous systems, but solvable with $\diamond\mathcal{S}$
- Consensus:

- *Termination:* Every correct process eventually decides some value
- *Agreement:* No two correct processesss decide on different values v, v'
- *Validity:* If a correct process decides v, then v was previously proposed by some process.
- *Integrity:* No process decides twice

- Mostefaoui & Raynal 99
- Threshold: f = $\lfloor (n-1)/2 \rfloor$

Solving consensus with $\diamond\mathcal{S}$



```
Function Consensus($v_i$)
cobegin
(1)  task $T1$: $r_i \leftarrow 0$; $est_i \leftarrow v_i$; % $v_i \neq \bot$ %
(2)    while $true$ do
(3)      $c \leftarrow (r_i \bmod n) + 1$; $est\_from\_c_i \leftarrow \bot$; $r_i \leftarrow r_i + 1$; % round $r = r_i$ %
(4)      case ($i = c$) then $est\_from\_c_i \leftarrow est_i$
(5)           ($i \neq c$) then wait (($\textsc{est}(r_i, v)$ is received from $p_c$)$\vee$($c \in suspected_i$));
(6)                         if ($\textsc{est}(r_i, v)$ has been received) then $est\_from\_c_i \leftarrow v$
(7)      endcase;   % $est\_from\_c_i = est_c$ or $\bot$ %
(8)      $\forall j$ do send $\textsc{est}(r_i, est\_from\_c_i)$ to $p_j$ enddo;
(9)      wait until ($\forall p_j \in Q_i$: $\textsc{est}(r_i, est\_from\_c)$ has been received from $p_j$);
                   % $Q_i$ has to be a live and safe quorum %
                   % For $\mathcal{S}$:  $Q_i$ is such that $Q_i \cup suspected_i = \Pi$ %
                   % For $\diamond\mathcal{S}$:  $Q_i$ is such that $|Q_i| = \lceil (n+1)/2 \rceil$ %
(10)     let $rec_i = \{est\_from\_c \mid \textsc{est}(r_i, est\_from\_c)$ is received at line 5 or 9$\}$;
                   % $est\_from\_c = \bot$ or $v$ with $v = est_c$ %
                   % $rec_i = \{\bot\}$ or $\{v\}$ or $\{v, \bot\}$ %
(11)     case ($rec_i = \{\bot\}$) then skip
(12)          ($rec_i = \{v\}$) then $\forall j \neq i$ do send $\textsc{decide}(v)$ to $p_j$ enddo; return($v$)
(13)          ($rec_i = \{v, \bot\}$) then $est_i \leftarrow v$
(14)     endcase
(15)   enddo

(16) task $T2$: upon reception of $\textsc{decide}(v)$:
                   $\forall j \neq i$ do send $\textsc{decide}(v)$ to $p_j$ enddo; return($v$)
coend
```

Overview

1. FLP impossibility
2. Failure Detectors
   - Theory: Definition
   - Chandra&Toueg hieararchy of FDs
   - Consensus with $\diamond\mathcal{S}$
3. Replication
   - Active replication / state machine replication
   - Primary-backup approach / passive replication
4. Paxos
   - Background
   - Algorithm

Replication: basic idea

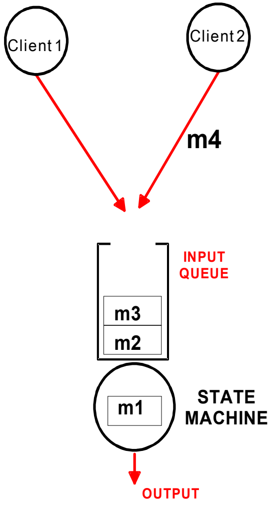
Client-server model

- Service is replicated on several servers
- . . . but client remains under the illusion that it contacts a single server
- If replicas fail independently, the service tolerates such a fault

Overview

State machine

- Execution model
  - **state** = set of state variables
  - **commands** modify state variables and produce outputs
- Characteristics
  - confinement – atomic commands
  - fault tolerance – easy replication – why?



State machine replication

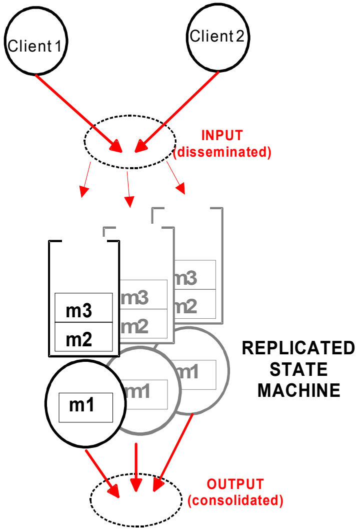
- Decentralized fault-tolerant applications may run replicated pieces of code which should behave in the same way
- **Replica determinism**:
  - Two replicas, departing from the **same initial state** and subject to the **same sequence of inputs**, reach the **same final state** and produce the **same sequence of outputs**

State machine replication

- Decentralized fault-tolerant applications may run replicated pieces of code which should behave in the same way
- **Replica determinism**:
  - Two replicas, departing from the **same initial state** and subject to the **same sequence of inputs**, reach the **same final state** and produce the **same sequence of outputs**
- **Atomic broadcast**:
  - Guarantees “same sequence of inputs” objective
  - The rest lies with the replica itself

State machine replication (active replication)

- Also called *state machine approach*
- Replicated state machine:
  - Servers start in same state
  - All replicas execute same sequence of input commands, in same order
  - All follow same sequence of state/outputs
  - **Achieves error masking**
  - Determinism mandatory
- Message ordering:
  - Total order of commands to replicas
  - Same commands in same order => same results



State machine replication (active replication)

Properties that specify the problem:

- SMA1 Initial state. All servers start in the same state.
- SMA2 Agreement. All servers execute the same commands.
- SMA3 Total order. All servers execute the commands in the same order

State machine replication (active replication)

- Agreement and total order => atomic broadcast (to all servers)

State machine replication (active replication)

- Agreement and total order => atomic broadcast (to all servers)
- Client-server protocol
  - Send to all servers; then all servers atomically broadcast OR
  - Send to one server; then server atomically broadcasts; if no reply obtained in a timeout, send to *f* servers etc.

State machine replication (active replication)

- Agreement and total order => atomic broadcast (to all servers)
- Client-server protocol
  - Send to all servers; then all servers atomically broadcast OR
  - Send to one server; then server atomically broadcasts; if no reply obtained in a timeout, send to *f* servers etc.
  - . . . at last, servers send output to client
    client consolidates output, e.g., by voting

How many servers do we need?

Number of replicas relevant for

- Input dissemination
- Output consolidation

How many servers do we need?

Number of replicas relevant for

- Input dissemination
- Output consolidation

Input dissemination

- Typically $2f + 1$ replicas for crash model ($f + 1$ in specific cases)
- Typically $3f + 1$ replicas for BFT model

How many servers do we need?

Number of replicas relevant for

- Input dissemination
- Output consolidation

Input dissemination

- Typically $2f + 1$ replicas for crash model ($f + 1$ in specific cases)
- Typically $3f + 1$ replicas for BFT model

Output consolidation

- $f + 1$ replicas in crash model (single reply is correct)
- $2f + 1$ replicas in BFT model (majority voting)

State machine replication: issues

- Non-deterministic component
  - State and behavior depend not only on the sequence of commands it executes but also on local parameters that cannot be controlled.

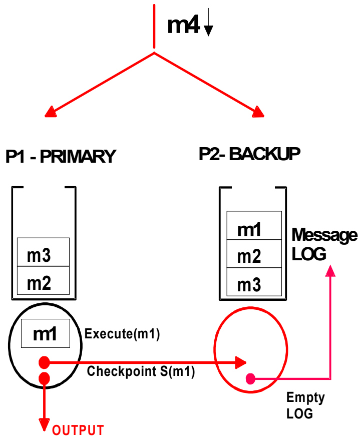
State machine replication: issues

- Non-deterministic component
  - State and behavior depend not only on the sequence of commands it executes but also on local parameters that cannot be controlled.
- Many mechanisms can cause a non-deterministic behavior:
  - Non-deterministic constructs in programming languages such as the Ada select statement;
  - Scheduling decisions; resource sharing with other processes;
  - Readings from clocks or random number generators; etc.
- State of two non-deterministic replicas may diverge even when they execute same sequence of inputs

Overview

Primary-backup approach (passive replication)

- Passive replication
  - Only Primary executes (and it decides the order)
  - Uses leader-election algorithm (no atomic broadcast)
  - Supports preemption and non-determinism (active rep. doesn't)
- State transferred to Backup(s)
  - Inter-replica state-level synchronization (checkpoints)
  - Backups log commands until checkpoint received
  - Primary fails: Backup takes over
  - Potentially long takeover-glitch
- Reliable unordered message diffusion (rbcast)



Primary-backup approach: Algorithm

Part executed by the primary only

Upon receiving a request from a client:
Execute the request
Forward the state update (if any) to the backups
Send reply to the client

Part executed by the backups only

Upon receiving a request
Store it in the log

Upon receiving a state update
Update the state and clean the corresponding request from the log

Upon detection of faulty primary
Elect a new primary
If it is the new primary, execute and reply to any requests in the log

Primary-backup approach (passive replication)

Fault model is important:

- If the network guarantees that:

  primary sends $m \Rightarrow$ backup eventually receives m, even if primary fails

  then the algorithm is *nonblocking* (reply to client sent without delay)

- If the network commits omission faults (state update messages may get lost), the primary must make sure that the backup(s) receive(s) the state update, before a reply is sent to the client
  - Explicit confirmation from backups to primary (2x round trip delay)
  - Or: backup sends reply to client after receiving update
    (1x round trip delay)

Primary-backup approach (passive replication)

Fault model is important (cont'd):

- Alternative:
  - Use total order broadcast for request, assume deterministic replicas
  - Sufficient to send state updates only periodically (not necessary to send after each client request)
  - If backup becomes the new primary, it (re-)executes all pending requests after the last received checkpoint
- Another (weak) alternative:
  - Relax consistency guarantees in case of a primary failure
  - I.e., accept that some client requests possibly get "lost" in case of a primary failure

Overview

Quick history of the Paxos algorithm

From Leslie Lamport's web page:

The Part-Time Parliament
ACM Transactions on Computer Systems 16, 2 (May 1998), 133-169. Also appeared as SRC Research Report 49. This paper was first submitted in 1990, setting a personal record for publication delay.

(. . .) Inspired by my success at popularizing the consensus problem by describing it with Byzantine generals, I decided to cast the algorithm in terms of a parliament on an ancient Greek island. (. . .) I submitted the paper to TOCS in 1990. All three referees said that the paper was mildly interesting, though not very important, but that all the Paxos stuff had to be removed. I was quite annoyed at how humorless everyone working in the field seemed to be, so I did nothing with the paper.

(. . .) the one exception in this dismal tale was Butler Lampson

(. . .) De Prisco, Lynch, and Lampson published their version of a specification and proof. Their papers made it more obvious that it was time for me to publish my paper. So, I proposed to Ken Birman, who was then the editor of TOCS, that he publish it. (. . .)

Essential Paxos literature

- Lamport: *The Part-Time Parliament*. Technical Report, DEC Systems Research Center, Sept. 1989; also in: ACM Transactions on Computer Systems, vol. 16., no. 2, 1998
- Lampson: *How to Build a Highly Available System Using Consensus.* Proc. 10th Int. Workshop on Distributed Algorithms (WDAG 96)
- Roberto De Prisco: *Revisiting the PAXOS Algorithm.* M.S. thesis, MIT, June 1997; published as: R. De Prisco, B. Lampson, N. Lynch: *Revisiting the PAXOS Algorithm.* Proc. 11th Int. Workshop on Distributed Algorithms (WDAG 97)
- Lamport: *Paxos Made Simple.* ACM SIGACT News (Distributed Computing Column) 32, 4, Dec. 2001
- Jonathan Kirsch, Yair Amir: *Paxos for System Builders.* LADIS'2008

Why Paxos matters

- Paxos solves active replication despite any number of crash failures and recoveries
  - Main problem: define a total order of clients' requests
  - It's core is a consensus algorithm
- There are several variations by Lamport and others
  - We consider the crash+recovery and message passing
  - There are variants even for Byzantine failures (actually, many)
- We consider Paxos for System Builders (PSB)
  - Amir and Kirsch
  - Because the original Paxos skips several aspects important for implementation

System model

- Links do not corrupt messages but may omit some
- Crash failures + recoveries
- Partial synchrony. . .
- . . .but: algorithm is **safe** even if system is asynchronous; partial synchrony needed only for **liveness**
  - This is an important design principle

Main idea

- A fixed number of servers
- There is a leader that assigns sequence numbers to clients' requests
- If the leader is suspected of being crashed, a new one is elected
- Quorum needed for progress: $\lfloor N/2 \rfloor + 1$ servers

- The original algorithm considered proposers, acceptors and learners; PSB considers all servers play the 3 roles

Main idea: more details

- Leader *periodically* tries to commit *i*-th message
  - Operation may fail due to crashed processes or lost messages
- If leader changes (suspected crash), the new leader tries to commit the same message (identified by sequence number) with a new *view number*
- Unreliable failure detection may (temporarily) lead to multiple active leader processes
- Multiple leaders may commit a message with specific sequence number multiple times, but in this case the algorithm makes sure that the messages are identical

Normal case operation (I)

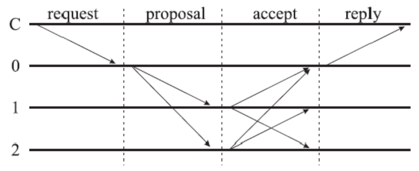


Figure 1: Paxos normal-case operation. Client C sends an update to the leader (Server 0). The leader sends a PROPOSAL containing the update to the other servers, which respond with an ACCEPT message. The client receives a reply after the update has been executed.

- The *PROPOSAL* message contains the sequence number
- The sequence number is accepted by a server when it received $\lfloor N/2 \rfloor + 1$ *ACCEPT* messages (the leader's *PROPOSAL* also counts as *ACCEPT*)

Normal case operation (II)

- Execution is done in order of sequence number
  - Network can reorder messages, so requests may be accepted in an order different from the sequence numbers
  - In that case, execution of early requests has to wait for late requests
- Servers may crash and recover
  - When they recover they have to know in what state they were, so:
  - Leader must synch to disk before sending *PROPOSAL*
  - The others must synch to disk before sending *ACCEPT*

Leader failure (I)

Two aspects:

- Leader election
- Prepare next sequence number, i.e., finding the last sequence number assigned by last leader

Leader failure (II)

Leader election (one of several possible protocols)

- When a server executes a request, it restarts a timer
  - If leader has no requests to order, periodically sends an empty PROPOSAL; servers restart the timer when they receive it
- If timer expires (no request to execute), server suspects the leader of being faulty $\rightarrow$ sends a VIEW-CHANGE message to all servers (with the next view number, V)
- When a server receives $\lfloor N/2 \rfloor + 1$ VIEW-CHANGE messages with V, it picks deterministically a new leader (e.g., V mod N)

Partial synchronous time model

- Timeout should be increased each time

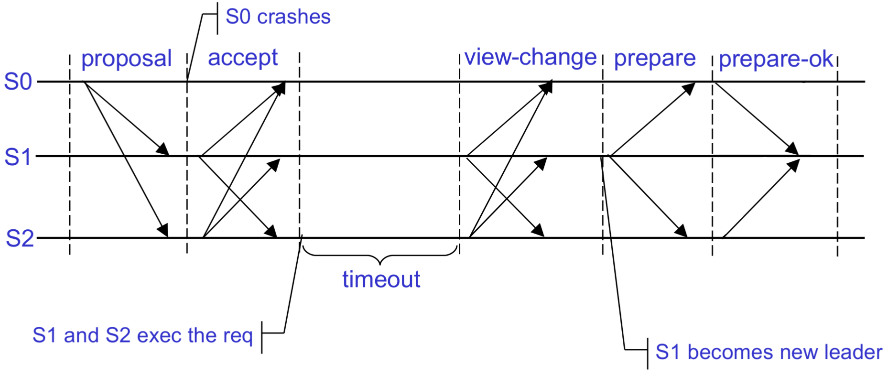
Leader failure (III)

Prepare next sequence number

- After election, the new leader sends a PREPARE message to all servers
  - Asking for the last request they accepted
- Each server replies with a PREPARE-OK message
  - Promising not to accept earlier proposals
  - Immediately before synchs to disk
- When leader gets reply from majority of servers:
  - Picks the highest sequence number received and adds 1 to be the next one;
  - Goes to normal case operation
- When a server accepts the first request from the new view, it implicitly becomes aware of which was the last one of the previous view

A leader change

Example: leader sends PROPOSAL then crashes



Missing requests

- Messages may be omitted so a server may accept request r without being able to accept $r - 1$
  - Stops the execution of messages
- Solution: ask others for missing messages

Recovery

When a server recovers

- Restores its state from the disk
- Starts a leader election

- May have to wait some time to restart executing requests

Ensuring safety

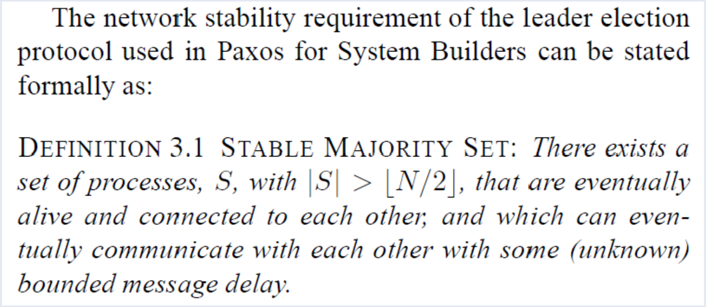
- Not possible to execute 2 different requests with the same sequence number:
  - In the same view: $\lfloor N/2 \rfloor + 1$ ACCEPTs (incl. 1 PROPOSE) needed; no server sends 2 for the same seq number; $\lfloor N/2 \rfloor + 1 + \lfloor N/2 \rfloor + 1 > N$
  - In different views: $\lfloor N/2 \rfloor + 1$ PREPARE-OK (incl. 1 PREPARE); at least 1 must have accepted a request that was executed by someone
- Not possible to execute a request without executing the previous
  - Imposed locally by each server

Ensuring liveness

Progress depends on

- Ability to elect a leader
- Quorum of $\lfloor N/2 \rfloor + 1$ messages/servers (c.f. normal operation)

PSB:



The network stability requirement of the leader election protocol used in Paxos for System Builders can be stated formally as:

DEFINITION 3.1 STABLE MAJORITY SET: *There exists a set of processes, S, with $|S| > \lfloor N/2 \rfloor$, that are eventually alive and connected to each other, and which can eventually communicate with each other with some (unknown) bounded message delay.*

Implementing Paxos

Several issues:

- UDP+IP multicast or TCP/IP?
- Flow control
- Timeouts: “large” or “short”?
- Aggregation (batching)?

Next: some results with PSB

- Y. Amir, J. Kirsh, Paxos for System Builders: An Overview, LADIS 2008

Throughput (no aggregation)

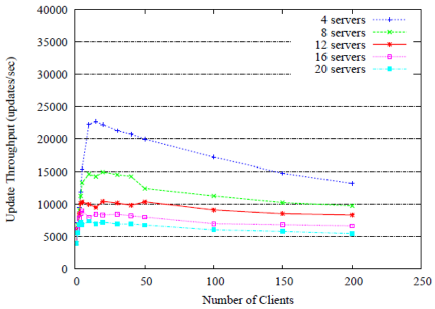


Figure 2: Update Throughput, No Disk Writes

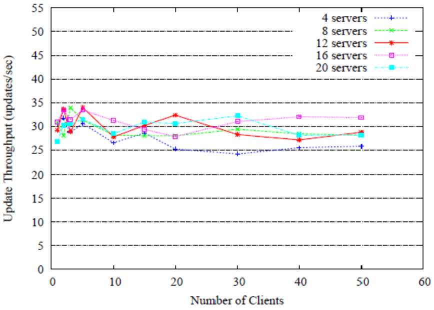


Figure 3: Update Throughput, Synchronous Disk Writes

Throughput (with aggregation)

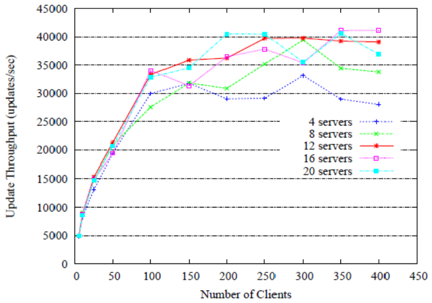


Figure 4: Update Throughput, Aggregation, No Disk Writes

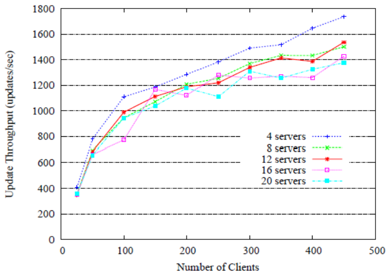
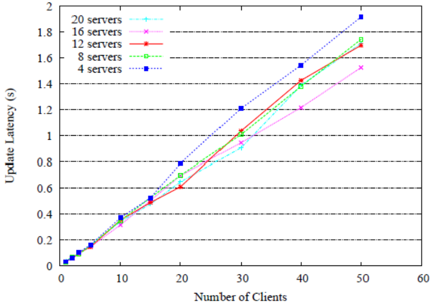
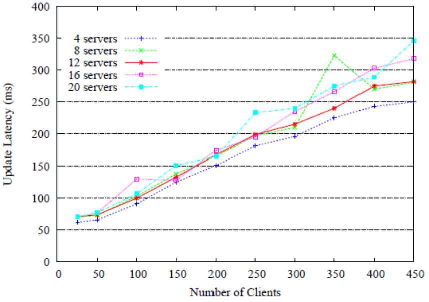


Figure 5: Update Throughput, Aggregation, Synchronous Disk Writes

Latency





ure 6: Update Latency, No Aggregation, Synchronous Disk Wr Figure 7: Update Latency, Aggregation, Synchronous Disk Writ