University of Freiburg, Germany
Department of Computer Science

# Distributed Systems

## Chapter 4 Coordination and Agreement

Christian Schindelhauer

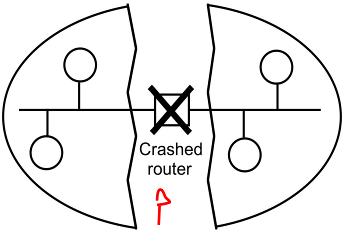
05. May 2013

# 4.1: Introduction

- Coordination in the absence of master-slave relationship
- Failures and how to deal with it
- Distributed mutual exclusion
- Agreement is a complex problem
- Multicast communication
- Byzantine agreement

## Assumptions

- Channels are reliable
- The network remains connected
- Process failures are not a threat to the communication
- Processes only fail by crashing



# Failure Detectors

- Failure detector is a service answer queries about the failures of *other* processes
- Most failure detectors are *unreliable failure detectors*
  - Returning either *suspected* or *unsuspected*
  - *suspected*: some indication of process failure
  - *unsuspected*: no evidence for process failure
- *Reliable failure detector*
  - Returning either *failed* or *unsuspected*
  - *failed*: detector has determined that the process has failed
  - *unsuspected*: no evidence for failure

## Example of an unreliable failure detector

- Each process $p$ sends a 'p is here' message to every other process every $T$ seconds
- If the message does not arrive within $T + D$ seconds then the process is reported as *Suspected*

# 4.2: Distributed Mutual Exclusion

- Problem known from operating systems (there: *critical sections*)
- How to achieve mutual exclusion only with messages

## Application-Level Protocol

| | |
|---|---|
| *enter()* | enter critical section – block if necessary |
| *resourceAccesses()* | access shared resources in critical section |
| *exit()* | leave critical section – other processes may enter |

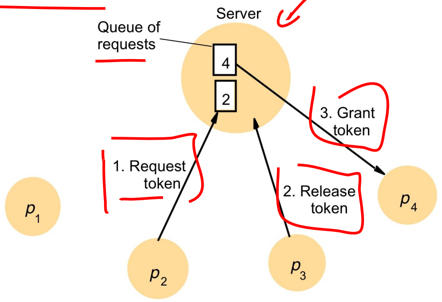
## Essential Requirements

| | |
|---|---|
| ME1: Safety ✓ | At most one process may execute the critical section at a time |
| ME2: Liveness ✓ | Requests to enter and ~~exist~~ exit the critical section eventually Finally succeed |
| ME3: $\rightarrow$ ordering | requests enter the critical section according to the *happened-before* relationship |

# Performance of algorithms for mutual exclusion

- *Bandwidth* consumed: proportional to the number of messages sent in each *entry* and *exit* operation
- *Client delay* at each *entry* and *exit* operation
- *Throughput* rate of several processes entering the critical section
- Throughput is measured by the *synchronization delay* between one process exiting the critical section and the next process entering it
- short *synchronization delay* correspond to high *throughput*

# Central Server Algorithm

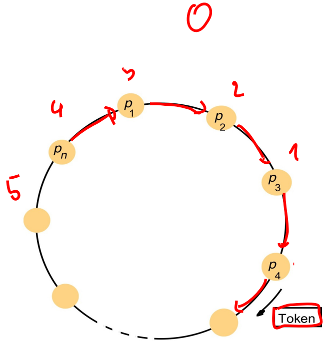



from *Distributed Systems – Concepts and Design*, Coulouris, Dollimore, Kindberg

- Simplest solution
- Request are handled by queues
- Performance
  - Entering the critical section: two messages (*request*, *grant*)
  - Leaving the critical section: one message (*release*)
- Server is performance bottleneck

# Ring Based Algorithm

this ith token




from *Distributed Systems – Concepts and Design*,

Coulouris, Dollimore, Kindberg

- Simplest distributed solution
- Arrange processes as ring (not related to physical network)
- A token (permission to enter critical section) is passed around
- Conditions ME1 (safety) and ME2 (liveness) are met
- ME3: $\longrightarrow$ ordering is not fulfilled
- Continuous consumption of bandwidth
- Synchronisation delay is between 1 and $n$ messages.

# The Algorithm of Ricart and Agrawala

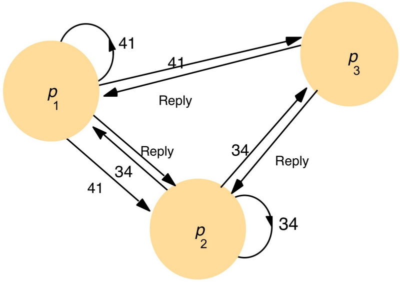
- Mutual exclusion between $n$ peer processes $p_1, p_2, \ldots, p_n$ which
  - have unique numeric identifiers
  - possess communication channels to one another
  - keep Lamport clocks attached to the messages
- Process states
  - `released`: outside the critical section
  - `wanted`: wanting to enter critical section
  - `held`: being in the critical section
- Each process `released` immediately answers a request to enter the critical section
- The process with `held` does not reply to requests until it is finished
- If more than one process requests the entry, the first one collecting the $n-1$ replies is allowed to enter the critical section.
- If the Lamport clocks of the latest messages do not differ, the numeric ID is used to break the tie.

# The Algorithm of Ricart and Agrawala

*On initialization*
    *state* := RELEASED;
*To enter the section*
    *state* := WANTED;
    Multicast *request* to all processes;
    $T$ := request's timestamp;
    *Wait until* (number of replies received = $(N - 1)$);
    *state* := HELD;

request processing deferred here

*On receipt of a request* $<T_i, p_i>$ *at* $p_j$ $(i \neq j)$
    *if* (*state* = HELD or (*state* = WANTED *and* $(T, p_j) < (T_i, p_i)$))
    *then*
        queue *request* from $p_i$ without replying;
    *else*
        reply immediately to $p_i$;
    end if
*To exit the critical section*
    *state* := RELEASED;
    reply to any queued requests;



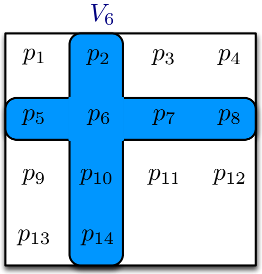

from *Distributed Systems – Concepts and Design*, Coulouris, Dollimore, Kindberg

# The Algorithm of Ricart and Agrawala

- Mutual exclusion properties
  - ME1 (safety): processes in state `held` prevent other ones from entering the CS
  - ME2 (liveness): follows from the ordering
  - ME3 (ordering): follows from the use of Lamport clocks
- Cost of gaining access: $2(n-1)$ messages
  - $n-1$ for multicast of request
  - $n-1$ for replies
- Client delay for requesting entry: a round-trip message
- Synchronization delay is one message transmission time

# Maekawa's Voting algorithm

- Reduce the number of messages by asking a subset
- For each process $p_i$ choose a *voting set* $V_i$ such that
  1. $p_i \in V_i$
  2. $V_i \cap V_j \neq \emptyset$ for all $i, j$
  3. $|V_i| = k$ for all $i$ (fairness)
  4. Each process occurs in at most $m$ voting sets
- Minimal choice of $\max\{m, k\}$ is $k, m \in \Theta(\sqrt{n})$.
- The optimal solution can be approximated by placing all nodes in a square matrix and choosing the row and column as voting set.




from *Distributed Systems – Concepts and Design*, Coulouris, Dollimore, Kindberg

# Maekawa's Voting algorithm

*On initialization*
  *state* := RELEASED;
  *voted* := FALSE;

*For $p_i$ to enter the critical section*
  *state* := WANTED;
  Multicast *request* to all processes in $V_i$;
  *Wait until* (number of replies received = $K$);
  *state* := HELD;

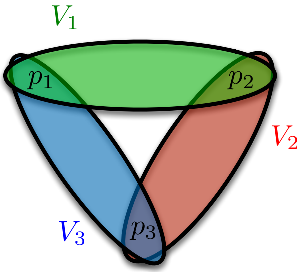
*On receipt of a request from $p_i$ at $p_j$*
  *if* (*state* = HELD *or voted* = TRUE)
  *then*
    queue *request* from $p_i$ without replying;
  *else*
    send *reply* to $p_i$;
    *voted* := TRUE;
  *end if*

*For $p_i$ to exit the critical section*
  *state* := RELEASED;
  Multicast *release* to all processes in $V_i$;

*On receipt of a release from $p_i$ at $p_j$*
  *if* (queue of requests is non-empty)
  *then*
    remove head of queue – from $p_k$, say;
    send *reply* to $p_k$;
    *voted* := TRUE;
  *else*
    *voted* := FALSE;
  *end if*

# Maekawa's Voting algorithm

- Mutual exclusion properties
  - ME1 (safety): follows from the intersections of $V_i$ and $V_j$
  - ME2 (liveness): not guaranteed.
- Sanders improved this algorithm to achieve ME2 and ME3 (not presented here)
- Cost
  - $2k$ per entry to the critical section
  - $k$ for exit
  - $O(\sqrt{n})$ messages
- Client delay for requesting entry: a round-trip message
- Synchronization delay is a round-trip message



# Mutual Exclusion

## Fault Tolerance

- What happens when messages are lost
- What happens when process crashes

- All of the above algorithms presented fail
- We will revisit this problem

# 4.3: Elections

## Election Algorithm

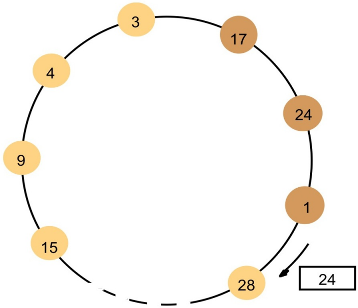
- An algorithm for choosing a unique process from a set of processes $p_1, \ldots, p_n$.
- A process *calls the election* if it initiates a run of an election algorithm
- Several elections could run in parallel where subset of processes are *participants* or *non-participants*.
- We assume processes have numeric IDs and that wlog. the process with the highest will be chosen.

## Requirements

| | |
|---|---|
| E1: Safety | During the run each participant has either $\text{elected}_i = \perp$ or $\text{elected}_i = P$, where $P$ is the non-crashed process with the largest ID |
| E2: Liveness | All participating processes $p_i$ eventually set $\text{elected}_i \neq \perp$ or crash. |

# Ring-Based Election: Algorithm of Chang and Roberts

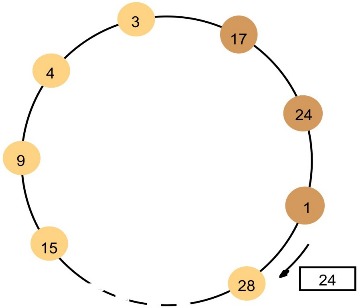
- Each process $p_i$ has a communication channel to the next process in the ring $p_{(i+1) \bmod n}$
- Messages are sent clockwise
- Assumption: no failures occur
- Non-participants are marked
- When a process receives an election message, it compares the identifier
  - If the arrived ID is greater, it forwards it
  - if the arrived ID is smaller and the process participates, it replaces it with its ID
  - if the arrived ID equals the process ID, the process is elected and sends an elected message around (with its ID).




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

# Ring-Based Election: Algorithm of Chang and Roberts

- E1 (Safety): follows directly
- E2 (Liveness): follows in the absence of crashes and communication errors
- Worst-case performance if a single node participates in the process
- Time: $3n - 1$ messages for the election
- Not very practical algorithm fault-prone and high communication overhead
- assumes a-priori knowledge (ring topology)



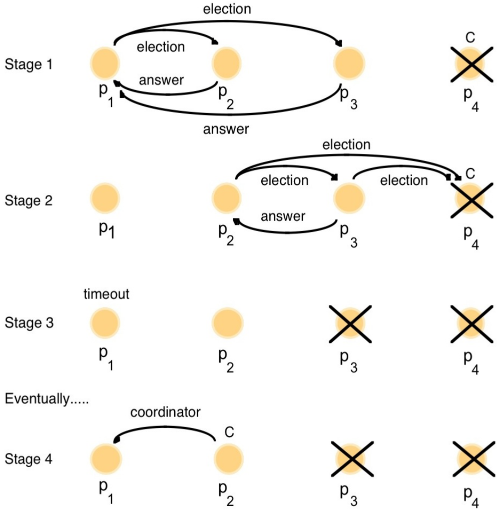

Note: The election was started by process 17.
The highest process identifier encountered so far is 24.
Participant processes are shown darkened

# The Bully Algorithm of Garcia & Molina

- The distributed system is assumed to be synchronous
  - i.e. after a timeout period $T$ a missing answer is interpreted as crash
  - reliable failure detector
  - fail-stop model
- Message types
  - *election*: Announces an election
  - *answer*: Answers *election* message (contains ID)
  - *coordinator*: Announces the identity of the elected process
- Any process may trigger an *election*
- Every process receiving an *election* messages sends an *answer* and starts a new one (if it has not started one before).
- If a process knows it has the highest ID (based on the answers) it sends the *coordinator* message to all processes
- If answers of lower IDs fail to arrive within time $T$ the sender considers itself a coordinator and sends the *coordinator* message

# The Bully Algorithm of Garcia & Molina

- If a process receives an *election* message it sends back an *answer* messages and begins another election — if it has not begun an election
- If a process knows it has the highest ID it sends the *coordinator* message
- New arriving processes with higher ID „bully" existing cordinators



# The Bully Algorithm of Garcia & Molina

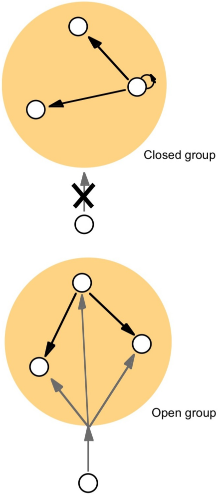
- E2: liveness condition is guaranteed if messages are transmitted reliably
- E1: safety condition: Not guaranteed if processes are replaced by processes with the same identifier
- different conclusions on which is the coordinator process
- E1 not guaranteed if the timeout value is too small
- In the worst case the algorithm needs $O(n^2)$ messages for $n$ processes

# 4.4: Multicast communication

- With a single call of $\mathit{multicast}(g, m)$ a process sends a message to all members of the group $g$
- Using $\mathit{deliver}(m)$, received messages are delivered on participating processes
- *Efficiency*
  - Number of messages, transmission time
- *Delivery guarantees*
  - ordering
  - receipt
  - e.g. IP Multicast does not guarantee ordering of success

# 4.4: Multicast communication

- *System Model*
  - *multicast*$(g, m)$: sends the message $m$ to all members of group $g$
  - *deliver*$(m)$: delivers a message to the process (message has been received by lower level)
  - *sender*$(m)$: sender of a message $m$ (within the message header)
  - *group*$(m)$: group of a message $m$ (within the message header)
- Allowed senders
  - closed group: senders must be members of a group
  - open group: any process can send a message to the group



## Basic Multicast

- *B-multicast*$(g, m)$: for each process $p \in g$, *send*$(p, m)$
- *B-deliver*$(m)$: if message $m$ is received at $p$ return the message $m$

*Ack Implosion*

- if too many processes participate
- if *send* uses acknowledgments, some of them could be dropped
- then the messages could be retransmitted
- further *acks* are lost due to full buffers etc.

## Reliable Multicast

- *Safety: Integrity*
  - Every message is delivered at most once
  - Receiver of $m$ is a member of $group(m)$
  - Sender has initiated a $multicast(g, m)$
- *Liveness: Validity*
  - If a correct process multicasts a messages then it eventually delivers $m$ (to itself)
- *Agreement*
  - If a correct process delivers $m$ then all other processes eventually deliver $m$

## Implementing Reliable Multicast over Basic Multicast

*On initialization*
  *Received* := {};

*For process p to R-multicast message m to group g*
  *B-multicast(g, m);*   // $p \in g$ is included as a destination

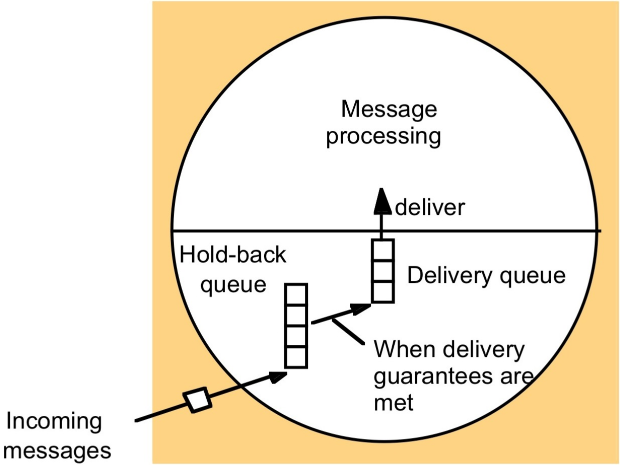
*On B-deliver(m) at process q with g = group(m)*
  *if* ($m \notin$ *Received*)
  *then*
    *Received* := *Received* $\cup$ {*m*};
    *if* ($q \neq p$) *then B-multicast(g, m); end if*
    *R-deliver m;*
  *end if*

Each message needs to be sent $|g|$ times!

# Implementing Reliable Multicast over IP Multicast

- *R-multicast*$(g, m)$ for sending process $p$
  - Sender increments a (sending) sequence number $S_g^p$ for group $g$ after each messages
  - Sequence number sent with message
  - Acknowledgements of all received messages with $\langle q, R_g^q \rangle$ are piggy backed with message
  - Negative Acknowledgments: by received sequence number $R_g^q$ causes retransmission of message
- *R-deliver*$(g)$ for receiving process $q$
  - $R_g^q$ is the sequence number of the latest message it has delivered.
  - it is send with each acknowledgment and allows the sender (and all receivers) to learn about missing messages
  - Process $q$ *delivers* a message $m$ (with piggy backed $S$) only if $S = R_g^q + 1$.
  - messages with $S > R_g^q + 1$ are kept in a hold-back queue
  - messages with $S < R_g^g + 1$ are erased
  - After delivery $R_g^q := R_g^q + 1$

# Hold-Back Queue for Arriving Multicast Messages

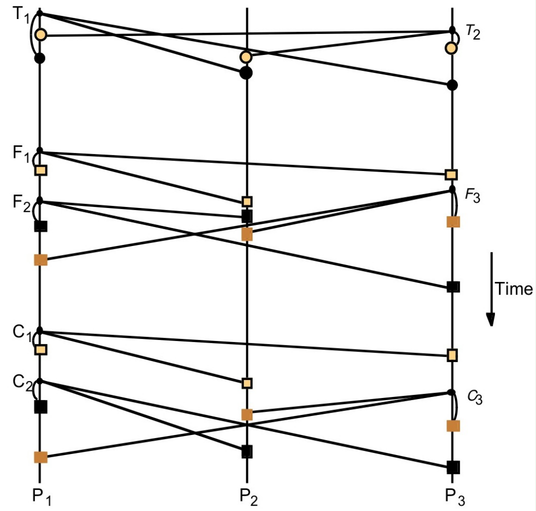


## Ordered Multicast

- *FIFO Ordering*
  - If a process casts $\texttt{multicast}(g, m)$ before $\texttt{multicast}(g, m')$
  - then $m$ is delivered before $m'$
  - in each process of group $g$
- *Causal Ordering:*
  - If $\texttt{multicast}(g, m) \rightarrow \texttt{multicast}(g, m')$
  - then $m$ is delivered before $m'$
  - $\rightarrow$ is based only on messages within the group $g$
- *Total Ordering:*
  - If a process delivers $m$ before $m'$
  - then $m$ is delivered before $m'$ on any other process of $g$

# Total, FIFO and Causal Ordering

- *Total Ordering*
- *FIFO Ordering*
- *Causal Ordering*



## Bulletin Board

| Bulletin board: *os.interesting* | | |
|---|---|---|
| Item | From | Subject |
| 23 | A.Hanlon | Mach |
| 24 | G.Joseph | Microkernels |
| 25 | A.Hanlon | Re: Microkernels |
| 26 | T.L'Heureux | RPC performance |
| 27 | M.Walker | Re: Mach |
| end | | |

- *FIFO Ordering*
- *Causal Ordering*
- *Total Ordering*

# Implementing FIFO Ordering Multicast

- Use sequence numbers for each message
  - $S_g^p$ for each sender process $p$ and group $g$
  - $R_g^p$ for the last message delivered to process $p$ of group $g$
- Multiast over IP Multicast satsifies FIFO ordering
- Essential components for FIFO ordering:
  - Sender piggy backs $S_g^p$ on the message
  - Receiver checks wether received message satisfies $S = R_g^q + 1$
  - and delivers $m$ and sets $R_g^q := R_g^q + 1$.
  - if $S > R_g^q + 1$ it puts $m$ into the hold-back queue
- In combination of a reliable multicast we obtain a reliable FIFO ordering multicast algorithm

# Implementing Total Ordering Multicast with a Sequencer

- A sequencer is an extra process taking care about ordering
- A sender process sends message with unique ID *i* to sequencer
- Sequencer marks message with ordering and multicasts the message

1. Algorithm for group member *p*

*On initialization:* $r_g := 0$;

*To TO-multicast message m to group g*
    *B-multicast*(*g* ∪ {*sequencer*(*g*)}, <*m*, *i*>);

*On B-deliver*(<*m*, *i*>) *with g* = *group*(*m*)
    Place <*m*, *i*> in hold-back queue;

*On B-deliver*($m_{order}$ = <"order", *i*, *S*>) *with g* = *group*($m_{order}$)
    wait until <*m*, *i*> in hold-back queue and $S = r_g$;
    *TO-deliver m*; // (after deleting it from the hold-back queue)
    $r_g = S + 1$;

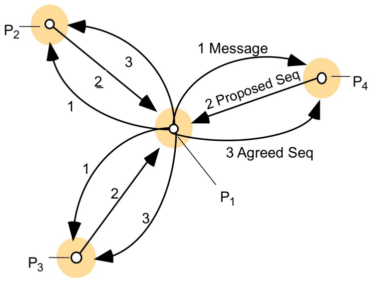
2. Algorithm for sequencer of *g*

*On initialization:* $s_g := 0$;

*On B-deliver*(<*m*, *i*>) *with g* = *group*(*m*)
    *B-multicast*(*g*, <"order", *i*, $s_g$>);
    $s_g := s_g + 1$;

# Implementing Total Ordering Multicast using ISIS

- Used in the ISIS toolkit of Birman & Joseph
- Each participating process proposes a sequence number for a messages
  - All proposed message numbers are unique
  - The sender chooses the maximum of all proposals and sends this information (piggy backed with the next messages)
  - This agreed sequence number defines the ordering of the hold-back-queue
  - The smallest elements of the hold-back queue can be delivered as the first element
- Does not imply causal nor FIFO ordering



# Implementing Causal Ordering

- Uses vector clocks to keep causal ordering (piggy backed to messages)
- Vector clock $V_i^g[i]$ counts all multicast messages of process $i$ in group $g$
- hold-back queue reflects vector clocks

Algorithm for group member $p_i$ $(i = 1, 2 \ldots, N)$

*On initialization*

$V_i^g[j] := 0$ $(j = 1, 2 \ldots, N)$;

*To CO-multicast message m to group g*

$V_i^g[i] := V_i^g[i] + 1$;

*B-multicast*$(g, \langle V_i^g, m\rangle)$;

*On B-deliver*$(\langle V_j^g, m\rangle)$ *from* $p_j$, *with* $g = group(m)$

place $\langle V_j^g, m\rangle$ in hold-back queue;

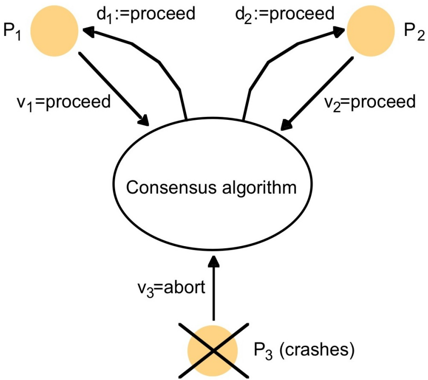
wait until $V_j^g[j] = V_i^g[j] + 1$ and $V_j^g[k] \le V_i^g[k]$ $(k \ne j)$;

*CO-deliver* $m$; // after removing it from the hold-back queue

$V_i^g[j] := V_i^g[j] + 1$;

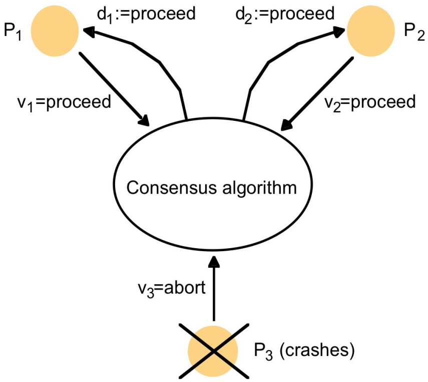
# 4.5: Consensus

- $n$ processes $p_1, \ldots, p_n$
- at most $f$ processes have arbitrary (Byzantine) failures
- Every process starts in the *undecided*state and *proposes* a value $v_i$
- Eventually all correct processes $p_i$
  - choose the *decided* state
  - and choose the same value $d_i \in \{v_1, \ldots, v_n\}$
  - and stay in this state



## Consensus Problem

- *Termination:* Eventually each correct process $p_i$ is *decided* by setting variable $d_i$
- *Agreement:* The decision value $d_i$ of all correct processes is the same
- *Integrity:* If all correct process proposed the same value $v$, then $d_i = v$ for all correct $p_i$




- Possible decision functions: *majority, minimum, maximum, ...*
- Byzantine failures can cause irritating and adversarial messages
- System crashes may not be detected

# Byzantine Generals Problem

- $n$ generals have to agree on attack or retreat
- one of them is the commander and issues the order
- at most $f$ generals are traitors (possibly also the commander) and have adversarial behavior
- all correct generals have eventually to agree on the commander decision if he acts correctly

## Consensus Problem

- *Termination:* Eventually each correct process $p_i$ is *decided* by setting variable $d_i$
- *Agreement:* The decision value $d_i$ of all correct processes is the same
- *Integrity:* If the commander is correct then all correct processes choose the commander's proposal

# Interactive Consistency

- $n$ processes need to agree on a *vector* of values
- Each process proposes a value $v_i$
- A correct processes eventually decide on a vector $d_i = \{d_{i,1}, \ldots, d_{i,n}\}$ where

$$d_{i,j} = v_j \quad \text{if } p_j \text{ is correct}$$

## Interactive Consistency

- *Termination:* Eventually each correct process $p_i$ is *decided* by setting variable $d_i$
- *Agreement:* The decision value $d_i$ of all correct processes is the same
- *Integrity:* If the $p_j$ is correct then all correct processes $p_i$ set $d_{i,j} = v_j$

# The Relationship between Consensus Problems

Assume solutions to Consensus (C), Byzantine generals (BG), interactive consistency (IC)

$$
\begin{aligned}
C_i(v_1, \ldots, v_n) &= \text{consensus decision value of } p_i \text{ for proposals } v_i \\
BG_i(j, v) &= \text{BG decision value of } p_i \text{ for commander } p_j \text{ proposal } v_j \\
IC_i(v_1, \ldots, v_n)[j] &= j\text{-th position of interactive consistency decision vector of } p_i \text{ for proposals } v_i
\end{aligned}
$$

## Solving *IC* from *BG*

- In parallel $n$ Byzantine generals problems are solved
- each process $p_j$ acts as commander once

$$IC_i(v_1, \ldots, v_n)[j] = BG_i(j, v)$$

# The Relationship between Consensus Problems

## Solving $C$ from $IC$

- *majority* returns the most often parameter or $\perp$ if no such value exists
- for all $i = 1, \ldots, n$

$$C_i(v_1, \ldots, v_n) = \mathit{majority}(IC_i(v_1, \ldots, v_n)[1], \ldots, IC_i(v_1, \ldots, v_n)[n])$$

## Solving $BG$ from $C$

- The commander $p_j$ sends its proposed value to itself and each other process
- All processes run consenus with the values $v_1, \ldots, v_n$ received from the commander
- for all $i = 1, \ldots, n$

$$BG_i(j, v) = C_i(v_1, \ldots, v_n)$$

# Consensus in a Synchronous System

- Assume that there are no arbitrary (Byzantine) errors
- Given a synchronous distributed systems (fail-stop model)
- Use basic multicast for $f + 1$ rounds
- Multicast all known values of all participants
- $Values_i^r$ denotes the set of proposed variables at the beginning of round $r$
- Reduce communication overhead by multicasting only freshly arrived variables $Values_i^r - Values_i^{r-1}$
- Choose the minimum of all known values as final value

# Consensus in a Synchronous System

Algorithm for process $p_i \in g$; algorithm proceeds in $f+1$ rounds

*On initialization*

$Values_i^1 := \{v_i\};\ Values_i^0 = \{\};$

*In round* $r$ $(1 \le r \le f+1)$

$B\text{-}multicast(g,\ Values_i^r - Values_i^{r-1})$; // Send only values that have not been sent

$Values_i^{r+1} := Values_i^r;$

*while* (in round $r$)

{

*On B-deliver*$(V_j)$ *from some* $p_j$

$Values_i^{r+1} := Values_i^{r+1} \cup V_j;$

}

*After* $(f+1)$ *rounds*

Assign $d_i = minimum(Values_i^{f+1});$

# Consensus in a Synchronous System

- There are no arbitrary errors only processes that crash and are correctly detected
- Given a synchronous distributed systems (fail-stop model)
- Correctness
  - Assume that two processes $p_i$ and $p_j$ have different values at round $r$
  - Then, in round $r-1$ at least one process $p_k$ has sent different values to $p_i$ and $p_j$
  - Then, $p_k$ has crashed in this round
  - Since the number of crashes is limited to $f$ there are not enough crashes to cover each of the $f+1$ rounds

# Byzantine Generals Problem in a Synchronous System

- Assume that there are **Byzantine** errors
- Given a synchronous distributed system
  - crashes are detected
  - other wrong behavior can not detected, e.g. strange messages
- messages are not (digitally) signed
- at most $f$ faulty processes

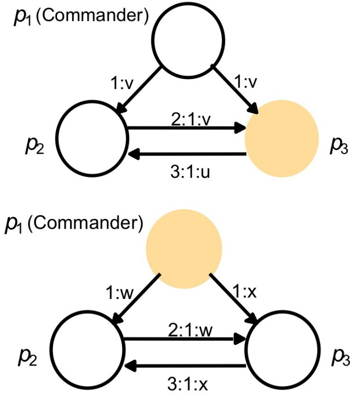
## Impossibility of a solution of the Byzantine generals problem [Lamport, Shostak, Pease 1982]

- The byzantine generals problem cannot be solved for $n = 3$ and $f = 1$.
- The byzantine generals problem cannot be solved for $n \leq 3f$.

# Byzantine Generals Problem in a Synchronous System

## Impossibility of a solution of the Byzantine generals problem for $n = 3$

- The byzantine generals problem with arbitrary failures cannot be solved for $n = 3$ and $f = 1$ in a synchronous system.
  - a faulty commander sending different values to his generals
  - cannot be distinguished from a faulty general forwarding wrong values



# Solution of the Byzantine Generals Problem

- Assume that there are **Byzantine** errors
- Given a synchronous distributed system
- messages are not (digitally) signed
- at most $f$ faulty processes

## Solution of the Byzantine generals problem [Pease, Shostak, Lamport 1980]

- The byzantine generals problem can be solved for $n = 4$ and $f = 1$.
- The byzantine generals problem can be solved for $n \geq 3f + 1$.

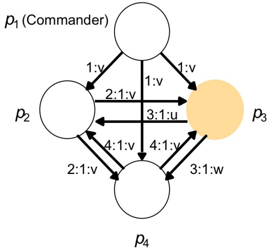
# Solution for Four Generals and One Faulty Process

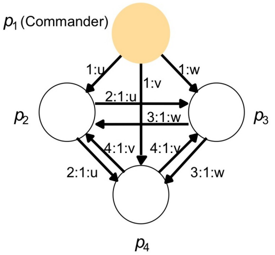
- The byzantine generals problem can be solved for $n \geq 4$ and $f = 1$.

## Algorithm of Pease et al.

1. The commander sends a value to all other generals (lieutenants)
2. All lieutenants send the received value to all other lieutenants
3. The commander chooses its value; the lieutenants compute the majority of all received values

- Since $n \geq 4$ the majority function always can be computed if at most one process is faulty
- If the commander crashes very early then all lieutenants agree on $\perp$



# More About the Byzantine Generals Problems

- For $f > 1$ the algorithm can be used recursively
  - Complexity: $f+1$ rounds and $O(n^{f+1})$ messages
  - The time complexity of $f+1$ rounds is optimal
- With the help of signed messages
  - any number of faulty generals $f < n$ can be dealt with
  - with signed messages the Byzantine Generals problem can be solved in $f+1$ rounds with $O(n^2)$ messages [Dolev & Strong 1983]
- For asynchronous systems with crash failures
  - No algorithm can reach consensus even if only **one processor** is faulty [Fischer, Lynch, Paterson 1985]
  - Each algorithm that tries to reach consensus can be confronted with a faulty process which influences the result if it continues (instead of crashing)

End of Section 4