Chapter 12: Coordination and Agreement

Introduction
- The motivation of DS design is resources sharing and cooperation. Because of the absence of global time, many problems related with time occur while system running.
- In order to solve these problems, the coordination is needed between the processes and for final agreement.

Introduction
- Collaboration behaviors in DS
  - Mutual exclusion
  - Election
  - Multicast
    - Basic, reliable, order
    - Agreement between processes
  - Consensus, byzantine agreement
- Failure assumptions
  - No failures
  - Benign failures
  - Arbitrary failures

Failure assumptions
- The real situation of the Internet
  - Network partition
  - Asymmetric routing
  - Intransitive connectivity
- Channel assumption: reliable
  - Failed link will be repaired or circumvented
- Process assumption
  - Crash failure without mention, otherwise arbitrary failure
  - Correct process: no crash failure and arbitrary failure

Failure detectors
- Unreliable failure detector
  - Inaccurate unsuspected or suspected
- Implement an unreliable failure detector
  - Each process announce its liveness every T seconds
  - Detector suspects a process if it has not receive the periodic message for D seconds
- Reliable failure detector
  - Inaccurate Unexpected
  - Failed: detect process crash
  - Require the system is synchronous

Chapter 12: Coordination and Agreement

• Introduction
• Distributed mutual exclusion
• Elections
• Multicast communication
• Consensus and related problems
• Summary
### Algorithms for mutual exclusion

- **Assumption**
  - Asynchronous, no process fail, reliable channel

- **Application level protocol**
  - `enter()`
  - `resourceAccesses()`
  - `exit()`

### Essential requirements for mutual exclusion

- **Safety**
  - At most one process may execute in the critical section at a time

- **Liveness**
  - Requests to enter and exit the critical section (CS) eventually succeed
  - Free from deadlock and starvation

- **Ordering**
  - If one request to enter the CS happened-before another, then entry to the CS is granted in that order

### Algorithms for mutual exclusion (continued)

- Evaluate the performance of the algorithms
  - **Bandwidth consumed**
    - The number of messages sent in each entry and exit operation
  - **Client Delay**
    - Incurred by a process at each entry and exit operations
  - **Throughput**
    - Synchronization delay: delay between one process exiting the critical section and the next process entering it

### The Central server algorithm

1. Request token
2. Release token
3. Grant token

- Meet safety and liveness, but not ordering
- Performance
  - Bandwidth consumption: The number of messages sent in each entry and exit operation
  - Enter(): A request message, a grant message
  - Exit(): a release message
  - Client Delay (no waiting processes)
    - The number of messages sent in each entry and exit operation
  - Synchronization delay
    - A release message + a grant message

### Ring-based algorithm

- The number of messages sent in each entry and exit operation
Ring-based algorithm

Meet safety and liveness, but not ordering

Performance
- Bandwidth consumed
  - Continuously consume network bandwidth
- Client Delay
  - Min: 0 message, when it has just received the token
  - Max: N messages, when it has just passed on the token
- Synchronization delay
  - Min: 1 message, when processes enter CS one by one
  - Max: N message, when a process enter CS continuously and no other process enter CS

Bandwidth: Enter(): N-1 multicast message, N-1 reply
Client delay: round-trip time
Synchronization delay: one message, released process will reply process in waiting queue

An algorithm using multicast and logical clocks

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;

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;

Maekawa’s voting algorithm

An algorithm using multicast and logical clocks

An algorithm using multicast and logical clocks

• Idea
  - A process enters CS if all other processes promise
    • Multicast + reply
  - Concurrence control
    • Lamport logical clock: avoid dead-lock
  - P1 and p2 want to enter CS concurrently, but p2 succeed

• Idea: a process enter CS when part of other processes promise, as long as no more than two processes enter CS
• A voting set Vi with each process pi
  - Vi ⊆ { p1, p2, ..., pn }
  - pi ∈ Vi
  - Vj ∩ Vj ≠ φ
  - | Vi | = K, K ≈ √n
  - Each process pi is contained in M of the voting sets Vi, M = K
On initialization

state := RELEASED; voted := FALSE;

For pi 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 pi at pj

if (state = HELD or voted = TRUE)
then queue request from pi without replying;
else send reply to pi; voted := TRUE;
end if

For pi to exit the critical section

state := RELEASED;
Multicast release to all processes in V_i;
On receipt of a release from pi at pj

if (queue of requests is non-empty)
then remove head of queue – from pj, say;
send reply to pj;
voted := TRUE;
else voted := FALSE;
end if

Deadlock example in Maekawa’s algorithm

P = {p1, p2, p3}
V1 = {p1, p2}, V2 = {p2, p3}, V3 = {p3, p1}.
Deadlock situation
1. p1, p2, p3 request to enter CS concurrently
2. p1, p2, p3 have set its own voted to TRUE, and wait for each other’s reply

Maekawa’s voting algorithm

- The improved algorithm
  - A total order of each request
  - The wait-for operation executes in accordance with the total order

Maekawa’s voting algorithm ...

- Performance
  - Bandwidth utilization
    - No release messages: 2 \sqrt{n}
    - Have release messages: 3 \sqrt{n}
    - If N>4 , 3 \sqrt{n} is better than 2(N-1)
  - Client delay: a round trip time
    - Same as multicast algorithm
  - Synchronization: a round trip time

Chapter 12: Coordination and Agreement

- Introduction
- Distributed mutual exclusion
- Elections
- Multicast communication
- Consensus and related problems
- Summary
Some concepts about *Election* algorithm

- **Election**
  - Choose a unique process to play a particular role
  - Define the *election*: choose the largest identifier
  - E.g. for electing the process with lowest load, then $id = 1/\text{load}$
- **Requirements of election algorithm**
  - **Safety**
    - A participant process $p_i$ has $\text{elected}_i = \bot$ or $\text{elected}_i = P$, where $P$ is chosen as the non-crashed process at the end of the run with the largest identifier
  - **Liveness**
    - All processes $p_i$ participate and eventually set $\text{elected}_i = \bot$ or crash

Some concepts about *Election* algorithm

- Evaluate the performance of election algorithm
  - Total bandwidth utilization
  - Turnaround time
    - The number of serialized message transmission times between the initiation and termination of a single run

A ring-based election algorithm

- Initially, every process is non-participant
- A process call an election when necessary
  - $id_{msg} = id_{local}$, send message {elect, $id_{msg}$} to the neighbor
  - The process set to be participant
- Forward the election message
  - Receiver sets to be participant if it has not been
  - forward message {elect, $\text{MAX}(id_{local}, id_{msg})$} to the neighbor
- End election when $id_{local} = id_{msg}$
  - set the process to be non-participant
  - $id_{coordinate} = id_{local}$, send message {elected, $id_{coordinate}$} to neighbor
- Forward the elected message
  - Each process set to be non-participant
  - Remember $id_{coordinate}$

A ring-based election algorithm ... continued

Evaluate the performance

- The worst turnaround case
  - A process starts an election when its anti-clockwise neighbour has the highest identifier
  - Election message reaches the neighbour: $N – 1$
  - The neighbour finds it is the coordinator: $N$
  - The neighbour announces elected message: $N$
  - So, the turnaround $= 3N – 1$
- The best turnaround $= 2N$ (highest call the election)
- Tolerate no failures

The bully algorithm

- **Assumption**
  - Synchronous system
    - Use timeout to detect a process failure
    - Reliable failure detector
  - Process can communicate with processes which have higher identifiers
    - *higher processes & lower processes*
The bully algorithm

The election of coordinator \( p_2 \) after the failure of \( p_4 \) and then \( p_3 \).

**The bully algorithm**

**Stage 1**
- \( p_1 \) initiates an election, \( p_2, p_3, p_4 \) answer.

**Stage 2**
- \( p_1 \) initiates an election, \( p_2, p_3, p_4 \) answer.
- \( p_2 \) initiates an election, \( p_3, p_4 \) answer.

**Stage 3**
- \( p_1 \) initiates an election, \( p_2, p_3, p_4 \) answer.

**Stage 4**
- \( p_1 \) initiates an election, \( p_2 \) answer.

**Initiate an election**

A process \( P \) begins an election when it notices the coordinator has failed by sending election messages to higher processes.

**Higher processes reply answer and initiate new elections**

All correct higher processes reply answers, and initiate new elections.

**Coordinator send coordinator messages to lower processes**

The highest process sends the coordinator message directly.
- If \( P \) does not receive any answers, it sends coordinator message to lower processes.
- If \( P \) has received some answers, then it waits coordinator message.
- If the message does not arrive for a period, \( P \) initiates a new election.

The processes that receive the coordinator message set \( \text{election} = \text{id coordinator} \)

**Chapter 11: Coordination and Agreement**

- Introduction
- Distributed mutual exclusion
- Elections
- Multicast communication
- Consensus and related problems
- Summary

**Multicast introduction**

- Multicast/broadcast
- Challenges
  - Efficiency
  - Bandwidth utilization
  - Total transmission time
  - E.g. delivery tree in IP multicast
- Delivery guarantees
  - Reliability
  - Ordering
- Group management
  - Processes joining and leaving group at arbitrary times

**System model**

- \( \text{Multicast}(g,m) \)
  - A process send the message \( m \) to all members of the group \( g \)
- \( \text{Deliver}(m) \)
  - Deliver the message \( m \) sent by multicast to the calling process
System model

Open and Closed groups

Basic multicast

• Basic multicast
  – A correct process will eventually deliver the message, as long as the multicaster does not crash
  – Primitives: B-multicast / B-deliver
  – Different to IP multicast in the aspect of reliability

• Implementation scheme is based on reliable one-to-one send operation
  – To B-multicast\((g, m)\): for each process \(p \in g\), send\((p, m)\);
  – On receive\((m)\) at \(p\): B-deliver\((m)\) at \(p\)

Basic multicast

• Multi-threads
  – The multicaster performs the send operations concurrently
  – The multicasting process will retransmit the dropped messages

• Ack-implosion
  – Large number of acknowledgments sent back
  – The multicasting process’s buffer will fill and drop acknowledgements

Implement basic multicast

Reliable multicast semantics

• Integrity
  – A correct process delivers a message \(m\) at most once

• Validity
  – If a correct process multicasts message \(m\) then it will eventually deliver \(m\)

• Agreement
  – If a correct process delivers message \(m\), then all other correct processes in \(\text{group}(m)\) will eventually deliver \(m\)
  • Atomicity: all or nothing

• Different from basic multicast: it is not met if the multicaster fails when it is multicasting

Implementing reliable multicast over B-multicast

On initialization

\[
\text{Received} \leftarrow \{\}\;
\]

For process \(p\) to B-multicast message \(m\) to group \(g\)

\[
\text{B-multicast}(g, m); \quad \text{// } p \in g \text{ is included as a destination}
\]

On B-deliver\((m)\) at process \(q\) with \(g = \text{group}(m)\)

\[
\text{if } (m \in \text{Received}) \text{ then}
\]

\[
\text{Received} \leftarrow \text{Received} \cup \{m\};
\]

\[
\text{if } (q \neq p) \text{ then B-multicast}(g, m); \text{ end if}
\]

\[
\text{R-deliver } m;
\]

\[
\text{end if}
\]
Implementing reliable multicast over B-multicast...continued

- **Validity**
  - A correct process will eventually B-deliver the message to itself
- **Integrity**
  - Based on B-multicast
  - Filter duplicated multicasted messages
- **Agreement**
  - If a correct process does not R-deliver m, then it must never B-deliver it, then other correct processes must never multicast it, then other correct processes must never B-deliver it
- **Expensive algorithm**
  - Each message is sent |group| times to each process

Reliable multicast over IP multicast

- **Characteristic**
  - IP multicast
    - Often successful
  - Piggy-back acknowledgment
    - A process attached an acknowledgment on the messages that they multicast to the group
  - Negative acknowledgement
    - A process sends a separate acknowledgments when it detects it has missed a message

The hold-back queue for arriving multicast messages

Reliable multicast over IP multicast ...continued

- **Evaluation**
  - Integrity
    - detection of duplicates; checksum in IP multicast
  - Validity
    - IP multicast has this property.
  - Agreement:
    - processes detect missing message,
    - retain copies of the messages they have delivered
    - retransmit

Uniform properties

- **Uniform property**
  - A property that holds whether or not processes are correct
- **Uniform agreement**
  - If a process, whether it is correct or fails, delivers message m, then all correct processes in group(m) will eventually deliver m
    - If reverse the lines “R-deliver m” and “if (q<>p) then B-multicast(g,m); end if ”, the resultant algorithm does not satisfy uniform agreement
Uniform properties

on B-deliver (m) at process q with g = group (m)
if (m ∈ Received) then
  Received := Received \cup \{m\};
if (q ≠ p) then
  B-multicast(g,m);
  R-deliver(m)
endif
endif

Uniform properties

Ordered multicast

• FIFO ordering
  – If a correct process issues multicast(g,m) and then multicast(g,m’), then every correct process that delivers m’ will deliver m before m’
• Causal ordering
  – If multicast(g,m) → multicast(g, m’), then any correct process that delivers m’ will deliver m before m’
• Total ordering
  – If a correct process delivers message m before it delivers m’, then any other correct process that delivers m’ will deliver m before m’
  – Causal Broadcast does not impose any order on messages not causally related

Total, FIFO and causal ordering of multicast messages

Notice the consistent ordering of totally ordered messages T1 and T2
the FIFO-related messages F1 and F2 and the causally related messages C1 and C3 – and the otherwise arbitrary delivery ordering of messages.

Display from bulletin board program

<table>
<thead>
<tr>
<th>Item</th>
<th>From</th>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>A.Hanlon</td>
<td>Mach</td>
</tr>
<tr>
<td>24</td>
<td>G.Joseph</td>
<td>Microkernels</td>
</tr>
<tr>
<td>25</td>
<td>A.Hanlon</td>
<td>Re: Microkernels</td>
</tr>
<tr>
<td>26</td>
<td>T.L’Heureux</td>
<td>RPC performance</td>
</tr>
<tr>
<td>27</td>
<td>M.Walker</td>
<td>Re: Mach</td>
</tr>
</tbody>
</table>

end
Consider a replicated database with two copies of a bank account residing at different sites. Initially, client_account = $1000

A user deposits $150 triggering a broadcast of 
\[ \text{msg1} = \{ \text{add $150 to client acnt} \} \]

At the same time, at other site, the bank initiates a broadcast of 
\[ \text{msg2} = \{ \text{add 8% interest to client acnt} \} \]

Causal Broadcast allows the two processes to deliver these updates in different order and creates inconsistency in database

Atomic Broadcast prevents such problem by providing strong message ordering or total order

**Hybrid ordered multicast**

- FIFO-total ordering
- Causal-total ordering
- Hybrid order and reliable protocol
  - Atomic multicast: a reliable totally ordered multicast

**Implementing FIFO ordering**

- \( \text{FO-multicast}/\text{FO-deliver} \)
- Algorithm
  - Similar to the algorithm of reliable multicast over IP-multicast
    - \( S_g, R_g \), Hold-back queue

**Implementing total ordering**

\( \text{TO-multicast}/\text{TO-deliver} \)

\( S_g: \text{sequencer}(g) \) maintains a group-specific sequence number

- \( \text{TO-Multicast} \)
  - A multicast a message \( m \) to group and the sequencer
  - \( \text{TO-multipost} \) a sequence number

  The sequencer multicast a sequence number for the message \( m \) to the group

- \( \text{TO-deliver} \)
  - Deliver the messages according to the sequence number
  - Bottleneck: the sequencer may become a bottleneck

**Total ordering using a sequencer**

1. Algorithm for group member \( p \)
   - On initialization: \( r_g := 0 \);
   - To TO-multicast message \( m \) to group \( g \)
     \( \text{B-multicast}(g \cup \{ \text{sequencer}(g) \}) \), \( <m, i> \);
   - On B-deliver(\( <m, i> \)) with \( g = \text{group}(m) \)
     - Place \( <m, i> \) in hold-back queue;
     - On B-deliver(\( <m, order, i, S> \) with \( g = \text{group}(m) \))
       - wait until \( <m, i> \) in hold-back queue and \( S = r_g \);
       - \( \text{TO-deliver} \) \( m \); // (after deleting it from the hold-back queue)

2. Algorithm for sequencer of \( g \)
   - On initialization: \( s_g := 0 \);
   - On B-deliver(\( <m, i> \)) with \( g = \text{group}(m) \)
     \( \text{B-multicast}(g, \langle \text{order}, i, s_g \rangle) \);
     \( s_g := s_g + 1 \);
The ISIS algorithm for total ordering

- Deliver-queue
- Group number indicate the largest message number
- Proposed number is the largest number viewed by processor
- Sender act as the sequencer
- agree with the largest number
- The smaller means missing message

\[ P_1 \quad P_2 \quad P_3 \]

1 Message
2 Proposed Seq
3 Agreed Seq

\[ \begin{align*}
A_q^a & : \text{each process } q \text{ maintains the largest agreed sequence number it has observed so far} \\
P_q^a & : \text{each process } q \text{ maintains its own largest proposed sequence number} \\
\text{TO-multicast} & \\
& \quad - p \text{ B-multicasts } m, i > g; i \text{ is an unique identifier for } m \\
& \quad - \text{ Each process propose a sequence number} \\
& \quad \quad - \text{ Each process } q : (1) \ P_q^a = \max(A_q^a, P_q^a) + 1; (2) \text{ attach } P_q^a \text{ to } m \text{ which is in hold-back queue; (3) reply } P_q^a \text{ to } p \\
& \quad - \text{ p collect proposed } P_q^a \text{ and select the largest one } a \text{, then } B- \multicasts{<i,a>}{g} \\
& \quad - \text{ Each process } q \text{ in } g \text{ sets } A_q^a := \max(A_q^a, a), \text{ attach a to } m \text{ as sequence num} \\
\text{TO-deliver} & \\
& \quad - \text{ Deliver the messages which has agreed sequence number and be at the front of the hold-back queue}
\end{align*} \]

The ISIS algorithm for total ordering

- All correct processes ultimately agree on the same set of sequence numbers which are monotonically increasing
- Performance
  - High latency: 3 messages

Implementing causal ordering

- Vector timestamp
  - Each process maintains its own vector timestamp
- CO-multicast
  - Attach vector timestamp to the multicasted message
- CO-deliver
  - Deliver messages according to its vector timestamp

Causal ordering using vector timestamps

- When a process \( p_j \) B-delivers a message from \( p_i \) it must place it in hold-back queue before it can CO-deliver it, until
  (a) It has delivered any earlier message sent by \( p_j \)
  (b) It has delivered any message that \( p_j \) had delivered at the time it multicast the message.

Causal ordering using vector timestamps

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

On initialization
\[ V_{ij}^n(j) := 0 \quad (j = 1, 2, \ldots, N) \]

To CO-multicast message \( m \) to group \( g \)
\[ V_{ij}^n[i] := V_{ij}^n[i] + 1; \quad B-\multicasts{V_{ij}^n}{m} \]

On B-deliver(\( V_{ij}^n \)) from \( p_j \) with \( g = \text{group}(m) \) place \( <V_{ij}^n, m> \) in hold-back queue; wait until \( V_{ij}^n[j] - V_{ij}^n[j] + 1 \) and \( V_{ij}^n[k] \leq V_{ij}^n[k] \quad (k \neq j); \)

CO-deliver \( m; \quad // \) after removing it from the hold-back queue
\[ V_{ij}^n[j] := V_{ij}^n[j] + 1; \]
Chapter 12: Coordination and Agreement

- **Introduction**
- **Distributed mutual exclusion**
- **Elections**
- **Multicast communication**
- **Consensus and related problems**
- **Summary**

Consensus introduction

- **Make agreement in a distributed manner**
  - Mutual exclusion: who can enter the critical region
  - Totally ordered multicast: the order of message delivery
  - Byzantine generals: attack or retreat?
- **Consensus problem**
  - Agree on a value after one or more of the processes has proposed what the value should be
  - Consensus, Byzantine general problem, interactive consistency, totally ordered multicast
- **Failure model**
  - Process crash failure, process Byzantine (arbitrary) failure

**Definition of the consensus problem**

- **Defined variables**
  - \( p_i \): process \( i \)
  - \( v_i \): proposed value of \( p_i \)
  - \( d_i \): decision variable of \( p_i \)
- **Requirements of a consensus algorithm**
  - **Termination**
    - Eventually each correct process sets its decision variable
  - **Agreement**
    - If \( p_i \) and \( p_j \) are correct and have entered the decided state, then \( d_i = d_j \) (\( i, j = 1, 2, \ldots, N \))
  - **Integrity**
    - If the correct processes all proposed the same value, then any correct process in the decided state has chosen that value

Consensus algorithm in no-failure circumstance

- **Algorithm**
  - Each process multicasts proposed value
  - Each process collects values proposed by other processes
  - \( V = \text{majority} (v_1, v_2, \ldots, v_N) \) at each process
    - **Majority**: minimum or maximum
- **Analysis**
  - **Termination**
    - Guaranteed by the reliability of the multicast
  - **Agreement and integrity**
    - Guaranteed by the definition of **majority**
  - **How to deal with the issue when there are failures?**

The byzantine generals problem

- **Byzantine generals problem**
  - One commander order “attack” or “retreat”, other generals execute the order
  - There are treacherous generals
  - All correct generals execute the same order
The byzantine generals problem

- **Requirements of the algorithm**
  - **Termination**
    - Eventually each correct process sets its decision variable
  - **Agreement**
    - If \( p_i \) and \( p_j \) are correct and have entered the decided state, then \( d_i = d_j \) (\( i, j = 1, 2, \ldots, N \))
  - **Integrity**
    - If the commander is correct, then all correct processes decide on the value that the commander proposed

Interactive consistency

- **Agree on a vector of values**
  - **Decision vector**: each element represents the decided value of each process
- **Requirements of the algorithm**
  - **Termination**
    - Eventually each correct process sets its decision variable
  - **Agreement**
    - The decision vector of all correct processes is the same
  - **Integrity**
    - If \( p_i \) is correct, then all correct processes decide on \( v_i \) as the \( i \)th component of their vector

Relate consensus to other problems

- **Definitions**
  - \( C(v_1, v_2, \ldots, v_N) \)
    - decision state of \( p_i \) in consensus problem
  - \( BG_i(j, v) \)
    - decision state of \( p_i \) in byzantine general problem, in which \( p_j \) is the commander
  - \( IC_i(v_1, v_2, \ldots, v_N)[j] \)
    - the \( j \)th decision state of \( p_i \) in interactive consistency problem

Consensus in a synchronous system

- **Definitions**
  - \( \alpha \cup \beta \times \Gamma \)
    - decision state of \( p_i \) in consensus problem
  - \( BG_i(j, v) \)
    - decision state of \( p_i \) in byzantine general problem, in which \( p_j \) is the commander
  - \( IC_i(v_1, v_2, \ldots, v_N)[j] \)
    - the \( j \)th decision state of \( p_i \) in interactive consistency problem

Consensus in a synchronous system

<table>
<thead>
<tr>
<th>( p_1 )</th>
<th>( p_2 )</th>
<th>( p_3 )</th>
<th>( p_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p_2 \rightarrow ) crash</td>
<td>( p_1 \rightarrow ) crash</td>
<td>( p_2 \rightarrow ) crash</td>
<td>( p_2 \rightarrow ) crash</td>
</tr>
<tr>
<td>( p_2 \rightarrow {1,2,3,4} )</td>
<td>( p_3 \rightarrow {1,2,3,4} )</td>
<td>( p_3 \rightarrow {1,2,3,4} )</td>
<td>( p_3 \rightarrow {1,2,3,4} )</td>
</tr>
<tr>
<td>Round 1</td>
<td>Round 2</td>
<td>Round 3</td>
<td>Round 3</td>
</tr>
</tbody>
</table>

Algorithm for process \( p_i \in G \); algorithm proceeds in \( f + 1 \) rounds

On initialization

- \( Values_i^0 := \{v_i\}; \ Values_i^0 := \{\} \)

In round \( r \) (\( 1 \leq r \leq f + 1 \))

- B-multicast(\( \Gamma \), \( Values_i^r - Values_i^{r-1} \)), // Send only values that have not been sent
- \( Values_i^{r+1} := Values_i^r \)
- while (in round \( r \))
  - \( \text{On B-deliver}(V_j) \) from some \( p_j \)
    - \( Values_i^{r+1} := Values_i^{r+1} \cup V_j \)
- After \( (f+1) \) rounds
  - Assign \( d_i := \min(Values_i^{f+1}) \)
Consensus in a synchronous system ...continued

• Analysis
  – Termination: ensured by synchronous system
  – Agreement and integrity
    • Assume: \( p_i \) holds a value \( v \) that \( p_j \) does not hold
    • \( P_{s_i} \) which sends \( v \) to \( p_i \) crash before it sends \( v \) to \( p_j \)
    • \( P_{s_j} \) which sends \( v \) to \( P_{s_i} \) crash before it sends \( v \) to \( p_j \)
    • ... 
    • \( P_{s_{f+i}} \) which sends \( v \) to \( P_{s_j} \) crash before it sends \( v \) to \( p_j \)
    • But, there are only \( f \) crashed processes
    • So, the sets of values that \( p_i \) and \( p_j \) hold are same
    • So, \( \text{minimum}(Values^{i, j}) \) are same

From BG to IC & From IC to C

• From BG to IC
  – Run BG \( N \) times, once with each process \( p_i \) acting as the commander
  – \( IC_i(v_1,v_2,...,v_N)[j] = BG(j,v), (i,j = 1, 2, ..., N) \)
• From IC to C
  – \( C(v_1,v_2,...,v_N) = \text{majority}(IC_i(v_1,v_2,...,v_N)[1],..., IC_i(v_1,v_2,...,v_N)[N]) \)

From C to BG

• The commander \( p_j \) sends its proposed value \( v \) to itself and each of the remaining processes
• All processes run \( C \) with the values \( v_1,v_2,...,v_N \) that they receive (\( p_j \) may be faulty)
• \( BG_i(j,v) = C(v_1,v_2,...,v_N), (i = 1, 2, ..., N) \)

The Byzantine generals problem in a synchronous system

• Arbitrary failures
  – \( f \) of the \( N \) process exhibits arbitrary failures
  • \( N \leq 3f \)
    – There is no solution to reach an agreement
  • \( N \geq 3f + 1 \)
    – There is solutions
    – Lamport[1982] give their solution

Impossibility with three processes

• If there is a solution, according to integrity condition, \( p_2 \) will choose \( 1:v \) in the left scenario
• \( p_2 \) can not identify the two scenario, so \( p_2 \) will choose \( 1:w \) in the right scenario
• By symmetry, \( p_3 \) will also choose \( 1:x \) in the right scenario
• So, contradict the agreement condition
• Reach agreement
  – If the generals digitally sign their messages, Byzantine agreement can be reached for 3 generals, with one of them faulty
Impossibility with \( N \leq 3f \)
- \( n_1, n_2, n_3 \)
  - Divide \( N \) into 3 groups
  - \( N = n_1 + n_2 + n_3 \) and \( n_1, n_2, n_3 \leq N/3 \)
- \( p_1, p_2, p_3 \)
  - Let \( p_1, p_2, p_3 \) simulate the behaviors of \( n_1, n_2 \) and \( n_3 \)
  - Since \( N \leq 3f \), so there is one process is faulty
- If there is a solution …
  - Reach agreement among the \( N \) entities
- Then, there is a solution among \( p_1, p_2 \) and \( p_3 \)
  - Contradict the impossibility of 3 processes

Solution with one faulty process
- Two rounds algorithm for \( N=4, f=1 \)
  - The commander sends a value to each of the lieutenants
  - Each of the lieutenants sends the value it received to its peers
  - \( d_i = \text{majority}(\text{received values}) \)
- Illustration
  - Left scenario
    - \( d_2 = \text{majority}(v, u, v) = v \)
    - \( d_4 = \text{majority}(v, v, w) = v \)
  - Right scenario
    - \( d_2 = d_3 = d_4 = \text{majority}(u, v, w) = \bot \)

Four byzantine generals

Performance discussion
- Questions
  - How many message rounds does it take?
  - How many messages are sent, and of what size?
- Lamport algorithm
  - \( f + 1 \) rounds
  - \( O(N^{f+1}) \) messages
- Conclusion from Fischer and Lynch[1982]
  - At least \( f+1 \) rounds message in deterministic algorithm

Impossibility in asynchronous systems
- No algorithm can guarantee to reach consensus in an asynchronous system
  - Can not tell a crashed process from a slow one
- Masking faults
  - Recover from crash to mask crash failure
- Consensus using failure detectors
- Consensus using randomization
  - Introduce an element of chance in the process’s behavior

Chapter 12: Coordination and Agreement
- Introduction
- Distributed mutual exclusion
- Elections
- Multicast communication
- Consensus and related problems
- Summary
Summary

• Distributed mutual exclusion
  – Central server
  – ring-based algorithm
  – multicast-based algorithm using logical clocks
  – Maekawa’s voting algorithm

• Elections
  – Ring-based algorithm
  – bully algorithm

Summary

• Multicast communication
  – Basic multicast
  – Reliable multicast
    • Over basic multicast, over IP multicast
  – Ordered multicast
    • FIFO delivery ordering, total delivery ordering, causal delivery ordering

• Consensus
  – Consensus
  – Byzantine generals
  – interactive consistency