Introduction [15.1]

• Coordination and agreement are fundamental to FT and DS
  • E.g., spaceship’s controllers all agree on changes in mode, etc
  • Key issue: system synchronous or asynchronous
  • Also key: how to handle failures
    • “Coping with failures is a subtle business” … build up from non-FT ones

• Contents
  • 15.2: Distributed mutual exclusion
  • 15.3: Elections
  • 15.4: Group communication (and coord./agreement with it)
  • 15.5: Agreement, especially Byzantine agreement
Failure assumptions and failure detectors [15.1.1]

• Note: simplifying assumption in Chap15 is each pair of processes connected by a **reliable channel**
  • Can build in practice as a lower layer, retransmitting dropped or corrupted messages
  • A reliable channel *eventually* delivers message to receiver (assume HW redundancy as needed)

• At any time, communication between some processes may be timely but delayed for others
  • **Network partition**, makes programming even harder
  • Bottom line: not all live processes can communicate at the same time (interval)

• Also assume by default processes fail only by crashing
  • Can’t directly detect, must infer
Figure 15.1
A network partition
Failure detectors

• **Failure detector**: a service that tracks process’ failures
  • Usually a piece/object in each process: **local failure detector**
  • Great seminal paper [Chandra and Toueg 1996]
  • Not always accurate! [why?]

• **Unreliable failure detector**: may declare (hints) *Unsuspected* or *Suspected*, based on evidence [what?] or lack thereof

• **Reliable failure detector**: always accurate in detecting a process’ failure: declares *Unsuspected* or *Failed*
  • Failed: the process has crashed
  • What might *Unsuspected* really mean?
Implementing failure detectors

• Simple scheme
  • Each process sends heartbeat message every $T$ seconds
  • Transmission time assumed to be $D$ seconds
  • If local detector not heard from process p in $T+D$ seconds, Suspected

• How to set a good timeout values $T$, $D$? Static or Dynamic?
• Synchronous system can have reliable FD [why? how?]
• Are imperfect failure detectors of any use?
Distributed Mutual Exclusion [15.2]

- Distributed processes often need to coordinate!
  - Shared purpose or goal or service (e.g., DCBlocks/GridBlox)
  - Shared resources managed by servers (Chap 16)
  - E.g., update on text files in NFS (stateless servers w/o locks)
  - Even P2P apps/services with no dedicated servers (Chapter 10)

- DME mechanism used by many applications
  - Distributed version of critical section (CS) prob., but with messages
15.2.1 Algorithms for DME

• **System model** (to start with)
  • $N$ processes $p_i$: 1, 2, …, $N$ not sharing variables
  • Assume only one critical section (simplicity; w.l.o.g.)
  • System asynchronous
  • Processes do not fail
  • Message delivery is reliable: any message sent eventually delivered, intact, exactly once

• **API**
  • `enter()` // enter critical section, blocking if necessary
  • `resourceAccesses()` // access shared resources in CS
  • `exit()` // leave critical section so others may enter
DME algorithms (cont.)

• Requirements for DME
  • **ME1** (safety): at most one process in CS at a time
  • **ME2** (liveness): requests to enter and exit CS eventually succeed

**ME2**\(\Rightarrow\) freedom from both deadlock and starvation [why?]

• Absence of starvation is a *fairness* issue
  • Also order of entry to CS
  • Happened-before can help here [how?]

• **ME3** (\(\Rightarrow\) ordering): If one request to enter the CS happened-before another, then entry to the CS is granted in that order

• How important is ME3, in theory and practice?
DME algorithms (cont.)

- Evaluation criteria:
  - Bandwidth/messages
  - Client delay (e.g. from `enter()` completing or in terms of one-way message chain)
  - Effect on system throughput
    - Rate/speed of DME can influence
    - One measure: synchronization delay between `exit()` and next `enter()`
Central server DME algorithm

- Server grants permission to enter CS
  - `enter()` sends message to server and receives reply
  - Server only sends permission when
    - No process using CS
    - Request queued and made it to the front
- Which properties does this provide:
  - **ME1** (safety)
  - **ME2** (liveness)
  - **ME3** (∴ ordering)
- Evaluation (see text for more): pretty good
- But central server can overload (no assumed failures for now) why not replicate?
Figure 15.2: Server managing a mutual exclusion token for a set of processes
Ring-based DME algorithm

- Organize processes in a logical ring
- Token passes around ring in fixed direction
- Possession of token gives permission for CS
  - If not needed, immediately pass on to logical neighbor
  - May put a time limit on how long can possess [why?]

- Which properties does this provide:
  - **ME1** (safety)
  - **ME2** (liveness)
  - **ME3** (ordering)

- **Evaluation**: Bandwidth? Delay? Other?
Figure 15.3
A ring of processes transferring a mutual exclusion token
DME algorithm using multicast and logical clocks

• Ricart and Agrawala [1981]

• `enter()` multicasts request message to the group
  • Only returns when reply from all processes

• Algorithm overview (details coming…)
  • Request messages have \(<T, p_i>\) in them (T is a Lamport Clock)
  • Each process tracks its CS status:
    • HELD: inside CS
    • WANTED: waiting entry
    • RELEASED: outside CS and not requesting it
Basic Idea

• If want into CS send multicast to group
  • Can enter only when have N-1 replies
• Logic with <T, p_i> ensures correctness & M1-M3
  • Lowest <T, p_i> wins ties
• Tracks own state: {WANTED, HELD, RELEASED}
Figure 15.5
Multicast synchronization

Initially
- $P_3$ not interested
- $P_1, P_2$ request simultaneously

Initially
- $P_3$ not interested
- $P_1, P_2$ request simultaneously
Figure 15.4
Ricart and Agrawala’s algorithm (at process $p_j$)

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 $<T_i, p_i>$ 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;
DME algorithm using multicast and logical clocks (cont.)

• Which properties does this provide:
  • **ME1** (safety)
  • **ME2** (liveness)
  • **ME3** (ordering)

• Evaluation (details in text…):
  • Messages?
  • Client delay?
  • Synch delay?
Voting DME algorithm

- From Maekawa 1985
- Key observation: to grant access to CS, not needed to receive OK from all processes
  - A process asking for CS is a *candidate*
  - Process sending permission is *voting* for it (sends 1 of its $M$ votes)
  - Only need a subset overlapping with all others’ subsets: *voting set*
  - Each process has $K$ votes and is in $M$ voting sets
  - Any two voting sets intersect
- Optimal solution only needs $K \sim \sqrt{N}$ and $M=K$
  - Think of a matrix…
Figure 15.6
Maekawa’s algorithm

On initialization
\[ \text{state} := \text{RELEASED}; \]
\[ \text{voted} := \text{FALSE}; \]

For \( p_i \) to enter the critical section
\[ \text{state} := \text{WANTED}; \]
Multicast request to all processes in \( V_i \);
\[ \text{Wait until} \ (\text{number of replies received} = K); \]
\[ \text{state} := \text{HELD}; \]

On receipt of a request from \( p_i \) at \( p_j \)
if (\( \text{state} = \text{HELD} \) or \( \text{voted} = \text{TRUE} \))
then
queue request from \( p_i \) without replying;
else
send reply to \( p_i \);
\[ \text{voted} := \text{TRUE}; \]
end if

For \( p_i \) to exit the critical section
\[ \text{state} := \text{RELEASED}; \]
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 \( p_k \);
\[ \text{voted} := \text{TRUE}; \]
else
\[ \text{voted} := \text{FALSE}; \]
end if
Voting DME algorithm (cont.)

• Which properties does this provide:
  • ME1 (safety)
  • ME2 (liveness)
  • ME3 (ordering)

• Evaluation (details in text…):
  • Messages?
  • Client delay?
  • Synch delay?
  • Deadlock free?
Fault Tolerance and DME

• None of previous algorithms tolerate message loss or process crashes! Consider for each…
  • What can happen when messages lost?
  • What can happen when processes crash?
• Consider how to adapt these DME algorithms to tolerate above.
• FT and coordination covered a lot more in 15.5 (consensus and related problems)
Elections [15.3]

• **Election**: choosing a unique process to play a particular role for a set of coordinating processes
  • If fail or want to retire, another election held
  • All processes must agree on the leader!

• Terminology and notation
  • **Calling an election**: initiating a particular run of the election alg.
    • One process never calls more than one at a time, but others can call too
    • Election choice must be unique despite multiple concurrent elections
  • Assume we choose the process with the largest ID (IP+port, 1/load, …)
  • **Participant**: engaged in an election (else non-participant)
  • Each $p_i$ stores $elected_i$
    • Will contain ID of elected process
    • At first initialized to special value UNDEF
Elections (cont.)

• Requirements:
  • **E1** (safety): A participant process $p_i$ has $elected_i = \text{UNDEF}$ or $elected_i = P$, where $P$ is chosen at the end of the run as the non-crashed process with the largest identifier.
  • **E2** (liveness): All processes $p_i$ participate and eventually either set $elected_i \neq \text{UNDEF}$ or crash.

  • Note: some processes may not yet be participating in a given election at a given time; they still have $elected_i$ set to winner of last election.

• Evaluating performance
  • Bandwidth/messages
  • Turnaround time (longest chain of message send times)
Ring-based election algorithm

• Chang and Roberts [1979]
• Assume no failures, but system is asynchronous
• Goal: choose a coordinator
• Initially all processes marked as non-participant
• To call election
  • Mark self as participant
  • Send election message with its ID to clockwise neighbor
Ring-based election algorithm (cont.)

- $p_j$ rec. election message from $p_i$ : compare ID with own
  - Greater: forward on message to clockwise neighbor
  - Smaller and $p_j$ not participant: pass on election message w/ own ID
  - Smaller and $p_j$ participant: don’t forward message ($p_i$ wins)
  - Equal: my ID is greatest, so I am coordinator
    - Mark self as non-participant
    - Send ELECTED message to clockwise neighbor

- Receiving an ELECTED message at $p_i$ with E-ID
  - Mark self as non-participant
  - Set $elected_i = E$-ID
  - Forward message on to clockwise neighbor
Figure 15.7
A ring-based election in progress

Note: The election was started by process 17.
The highest process identifier encountered so far is 24.
Participant processes are shown in a darker colour.
Ring-based election algorithm (cont.)

• Which requirements are met?

  • **E1** (safety): A participant process \( p_i \) has \( elected_i = \text{UNDEF} \) or \( elected_i = P \), where \( P \) is chosen as the non-crashed process at the end of the run with the largest identifier.

  • **E2** (liveness): All processes \( p_i \) participate and eventually either set \( elected_i \neq \text{UNDEF} \) or crash.

• Evaluation

  • Worst case performance if only one election?

• Notes:

  • Since does not tolerate failures not practical

  • But with a failure detector could reconstitute ring (keep multiple neighbors like Pastry and friends from Chap10 (Overlay Networks))
Bully algorithm for elections

- Garcia-Molina 1982
- Assume message delivery reliable
- Differences from ring election algorithm
  - Synchronous system, so use timeouts to detect failures
  - Ring alg. had minimal \textit{a priori} knowledge of other processes
    - Bully Alg assumes know all processes with higher IDs, can comm. w/all
- Kinds of messages
  - ELECTION: call an election (sent when timeout on process)
  - ANSWER: send response to ELECTION message
  - COORDINATOR: announces identify C-ID of elected process
Bully algorithm (cont.)

- Starting an election if highest ID: can just send COORDINATOR message (with its ID)
- Otherwise: send ELECTION msg to procs with higher IDs
  - If get no replies by timeout, send COORDINATOR msg (w/ID) to procs with lower ID
  - Else wait timeout, if no COORDINATOR msg send ELECTION
- Receiving COORDINATOR message with C-ID:
  - Set $elected_i = C$-ID
  - Treat C-ID as coordinator now
- Receiving ELECTION message:
  - Send ANSWER message
  - Call another election
Bully algorithm (cont.)

- Process created to replace crashed process begins election
  - If highest ID it becomes coordinator, even though current one functioning
  - What a bully!
Figure 15.8
The bully algorithm

The election of coordinator $p_2$, after the failure of $p_4$ and then $p_3$.
Bully algorithm (cont.)

• Which requirements are met?
  
  • **E1** (safety): A participant process $p_i$ has $elected_i = \text{UNDEF}$ or $elected_i = P$, where $P$ is chosen as the non-crashed process at the end of the run with the largest identifier
  
  • **E2** (liveness): All processes $p_i$ participate and eventually either set $elected_i \neq \text{UNDEF}$ or crash

• Evaluation
  
  • Worst case performance if only one election?
Coordination and Agreement in Group Communication [15.4]

- Group comm: get message to a group of processes
  - Higher-level semantics than IP multicast (IPMC)
- Reliability properties: validity, integrity, agreement, and ordering (FIFO, causal, total)
Coordination and agreement in group communication (cont.)

- **System model**
  - Processes have 1:1 reliable channels
  - Only crash failure
  - Group comm via a multicast operation (again, >IPMC)
  - A process can belong to multiple groups
  - Some algs assume groups are closed: only members can send
  - Processes don’t lie about origin or destination of messages
  - Asynchronous system

- **APIs**
  - Multicast \((g, m)\): send message \(m\) to all members of group \(g\)
  - Deliver\((m)\): delivers a message sent to group (to queue or app)

- **Messages contain ID of sender, group**
Basic multicast [15.4.1]

- The basic building block for use in the other algorithms
  - Correct process will eventually delivery message, if multicaster does not crash
- Comparison to IPMC?

- Simple implementation
  - \( B\text{-multicast}(g,m) \): for each process \( p \) in group, send \( (p,m) \)
  - \( \text{On receive}(m) \text{ at } p \): \( B\text{-deliver}(m) \text{ at } p \)
Reliable multicast [15.4.2]

- Builds on Ch6 defns for validity, integrity, and agreement
- Properties of $R$-multicast($g,m$) and $R$-deliver($m$)
  - **Integrity**
    - Correct process $p$ delivers $m$ at most once to application
    - Delivered $m$ was supplied to R-multicast by sender($m$)
  - **Validity**: if correct $p$ multicasts $m$, then it will eventually deliver $m$
  - **(Delivery) Agreement**: if correct $p$ delivers $m$, then all other correct processes in group($m$) will eventually deliver $m$.
    - AKA *atomic* delivery (but sometimes that includes total)
  - What properties of these does B-multicast provide?
  - Do these properties in any way provide liveness?
- Simple to implement R-multicast over B-multicast
  - Process can belong to several *closed* groups
Figure 15.9
Reliable multicast algorithm

On initialization

\[ \text{Received} := \{\}; \]

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

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

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

\[ \text{if} \ (m \notin \text{Received}) \]
\[ \quad \text{then} \]
\[ \quad \text{Received} := \text{Received} \cup \{m\}; \]
\[ \quad \text{if} \ (q \neq p) \text{ then } \text{B-multicast}(g, m); \text{ end if} \]
\[ \quad \text{R-deliver } m; \]
\[ \text{end if} \]

Note: if moved up R-deliver then not \textbf{uniform agreement} (defined soon…)
Reliable multicast over B-multicast (cont.)

- Which properties does this algorithm provide?
  - **Integrity**
    - Correct process \( p \) delivers \( m \) at most once
    - Delivered \( m \) was supplied to R-multicast by sender(\( m \))
  - **Validity**: if correct \( p \) multicasts \( m \), then it will eventually deliver \( m \)
  - **Agreement**: if correct \( p \) delivers \( m \), then all other correct processes in \( \text{group}(m) \) will eventually deliver \( m \).

- Other comments on algorithm?
Reliable multicast over IPMC

• Alternate impl.: use IPMC, piggybacked ACKS, and NACKS
  • Observation: IPMC is efficient, and usually successful
  • No separate ACKs, piggyback on messages multicasted to group
  • Send a NACK only when detect missed a message
  • Assume groups closed

• Basic idea
  • \( p \) tracks seqns \( S[p,g] \) and last delivered \( R[q,g] \)
  • \( R\text{-multicast}(g,m) \) piggybacks on IPMC msg \( S[p,g]++ \) and all \( R[q,g] \)
  • \( R\text{-deliver}(m) \) delivers \( m \) w/seqn \( S \) from \( p \) when \( S=(R[p,g]++)+1 \)
    • Otherwise queues it in holding queue
    • Learn about missing messages this way, can send NACK
    • \( R\text{-multicast}(g,m) \) code must buffer \( m \) for some time at all processes
Figure 15.10
The hold-back queue for arriving multicast messages

- Not strictly necessary for reliability property
- But simplifies algorithm
- Also later helps provide ordered delivery
Reliable multicast over IPMC (cont.)

• Which properties does this algorithm provide?

  • **Integrity**
    • Correct process \( p \) delivers \( m \) at most once
    • Delivered \( m \) was supplied to R-multicast by sender(\( m \))

  • **Validity**: if correct \( p \) multicasts \( m \), then it will eventually deliver \( m \)

  • **Agreement**: if correct \( p \) delivers \( m \), then all other correct processes in \( group(m) \) will eventually deliver \( m \).

• Other comments on algorithm?
Uniformity

• Agreement so far only dealt w/ correct processes: never fail

• **Uniform properties**: hold whether or not processes are correct or not
  
  • **Uniform agreement**: if a process, whether correct or fails, delivers message $m$, then all correct processes in $\text{group}(m)$ will eventually deliver $m$
  
  • Does Fig 15.9 provide uniformity: if crash after R-deliver?

• Why care about dead processes’ behavior anyway?
Ordered multicast [15.4.3]

• B-multicast delivers a message to group members in an arbitrary order.

• Some apps need more than that.

  • **FIFO ordering**: if a correct process issues \( \text{multicast}(g,m) \) and then \( \text{multicast}(g,m') \), every correct process will deliver \( m \) before \( m' \).

  • **Causal ordering**: if \( \text{multicast}(g,m) \rightarrow \text{multicast}(g,m') \), where \( \rightarrow \) is the happened-before relationship induced only by messages sent between the members of \( g \), then any correct process that delivers \( m' \) will deliver \( m \) before \( m' \).

    • Does Causal imply FIFO?

  • **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 \( m' \).

• Note: for now assume process only in one group … later extend
Notice the consistent ordering of totally ordered messages $T_1$ and $T_2$, the FIFO-related messages $F_1$ and $F_2$ and the causally related messages $C_1$ and $C_3$ – and the otherwise arbitrary delivery ordering of messages.
Ordered multicast (cont.)

• Ordering does not assume or imply reliability!
  • Reliable (all-or-none) and total AKA “atomic broadcast” sometimes
    • Called atomic+total often called ABCAST
  • Also reliable versions of FIFO, causal, and some hybrid orderings

• Performance
  • Very expensive and not largely scalable
  • E.g., some have proposed application-specific message semantics to define orderings [Cheriton and Skeen 1993, Pedone and Schiper 1999]
    • VERY interesting papers for student presentations in 562 (fault-tolerant computing)
Example: bulletin board system

- **App**: users post messages
- Each user has a local process delivering to user
- Each topic has its own process group
  - User posts: multicasts to others
  - Receive message: deliver in “right” order
- What ordering (if any) is desirable here?
Figure 15.12
Display from bulletin board (AKA discussion forum) 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>

- FIFO at least desirable
- Causal: needed so “Re:” comes after original (23 → 27)
- Total: numbers consistent (and useable as message IDs)
- Note: USENET does not provide (full) causal or any total
Implementing FIFO ordering

• Use a per-sender sequence number
• As with R-multicast, $S[p,g]$ and $R[q,g]$ kept at $p$, for all $q$ in $g$

$p$ calls $FO-multicast(g,m)$:
  • Piggyback $S[p,g]++$ onto $m$
  • Call $B-multicast(g,m)$

$p$ receives $m$ from $q$ with sequence $S$
  • $R=R[q,g]++$
  • IF $S = R+1$: $FO-deliver(m)$ to $p$
  • ELSE if $S > (R+1)$: put in holding queue until ready
  • ELSE: discard // duplicate, $S <= R$

• Can use any implementation of $B$-multicast
• If use R-multicast, then have reliable FIFO
• Note: above only works if groups are non-overlapping
Implementing total ordering

• **TO-multicast**(g,m) and **TO-deliver**(m)
  - Basic idea: assign TO-IDs for each multicast message
  - Similar to FIFO, but track *group-specific IDs, not process-specific*
  - Two main algorithms: sequencer proc. and distributed agreement

• TO sequencer process idea (Kaashok on Amoeba Dist OS)
  - Main process that assigns the TO-ID(m)

  • **TO-multicast**(g,m)
    - attaches unique ID to m, id(m)
    - B-multicast(g,m) and to sequencer(g)
    - sequencer(g) assigns TO-ID(m)
    - Sequencer does B-multicast to group to tell TO-ID(m)
    - Group members now know when to deliver m (wait until at $f+1$ processes)

• Evaluation? Comments?
Figure 15.13
Total ordering using a sequencer

1. Algorithm for group member \( p \)

   \textit{On initialization:} \( r_g := 0; \)

   \textit{To TO-multicast message} \( m \) \textit{to group} \( g \)
   \( \quad B\text{-multicast}(g \cup \{ \text{sequencer}(g) \}, \langle m, i \rangle); \)

   \textit{On B-deliver} \( \langle m, i \rangle \) \textit{with} \( g = \text{group}(m) \)
   \( \quad \) Place \( \langle m, i \rangle \) in hold-back queue;

   \textit{On B-deliver} \( m_{\text{order}} = \langle \text{“order”}, i, S \rangle \) \textit{with} \( g = \text{group}(m_{\text{order}}) \)
   \( \quad \) wait until \( \langle m, i \rangle \) in hold-back queue and \( S = r_g; \)
   \( \quad \) \textit{TO-deliver} \( m; \) \( \quad // (after deleting it from the hold-back queue) \)
   \( \quad r_g = S + 1; \)

2. Algorithm for sequencer of \( g \)

   \textit{On initialization:} \( s_g := 0; \)

   \textit{On B-deliver} \( \langle m, i \rangle \) \textit{with} \( g = \text{group}(m) \)
   \( \quad B\text{-multicast}(g, \langle \text{“order”}, i, s_g \rangle); \)
   \( \quad s_g := s_g + 1; \)
Total ordering via distributed agreement (ISIS)

• Basic Idea
  1. Process $p$ B-multicast message $m$ to members (open or closed)
  2. Receiving processes propose a sequence number
     1. Tracks agreed $A[q,g]$ and its proposed so far $P[q,g]$
  3. Processes agree on TO-ID($m$)

• Details
  1. $p$ calls $B$-multicast($m$, $id(m)$), where $id(m)$ globally unique
  2. Each proc $q$ replies to $p$ w/ $P[q,g] = \text{MAX}(A[q,g], P[q,g]) + 1$
  3. $p$ collects sequence numbers and chooses the largest one, $a$
  4. $p$ calls $B$-Multicast($g$, $id(m)$, $a$)
  5. All processes now know $a$ is TO-ID($m$)

• Evaluation? Comments? (more details in text….)
Figure 15.14
The ISIS algorithm for total ordering

Note: here $P_1$ is both sender(m) and sequencer(g)
Implementing causal ordering (ISIS)

- Each process maintains its own vector time, \( V[q] \)
  - Tracks the number of events it has seen from each process that happened-before the message about to be multicasted

- \( CO\text{-multicast}(m,g) \) at \( p \):
  - \( V[p]++ \)
  - \( B\text{-multicast}(g,m, id(m), V) \)

- When \( p_i \) B-delivers \( m \) from \( p_j \), puts in holdback queue before can CO-deliver it
  - Must ensure all happened-before messages have arrived
  - \( p_j \) waits until
    - It has delivered any earlier message sent by \( p_j \)
    - It has delivered any message \( p_j \) had delivered before it sent \( m \)
Figure 15.15
Causal ordering using vector timestamps

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

On initialization
\[ V^g_i[j] := 0 \quad (j = 1, 2, ..., N); \]

To CO-multicast message \( m \) to group \( g \)
\[ V^g_i[i] := V^g_i[i] + 1; \]
\[ B\text{-multicast}(g, <V^g_i, m>); \]

On B-deliver\(<V^g_j, m>\) from \( p_j \), with \( g = \text{group}(m) \)
place \(<V^g_j, m>\) in hold-back queue;
wait until \( V^g_j[j] = V^g_i[j] + 1 \) and \( V^g_j[k] \leq V^g_i[k] \) \((k \neq j)\);
CO-deliver \( m \); // after removing it from the hold-back queue
\[ V^g_i[j] := V^g_i[j] + 1; \]
Discussion

• Many possible global orderings (see text): global FIFO, global causal, pairwise total, global total, overlapping groups

• So far, did not give algorithm guaranteeing both reliable and total ordered delivery! [Why?]
Consensus and related problems [15.5]

• Similar problems here: consensus, Byzantine generals, interactive consistency … plus earlier DME, and total ordering … all fundamentally agreement.

• Exploring 3 variations deeper
  • Byzantine generals
  • Interactive consistency
  • Totally ordered multicast
  • …. Plus

• Impossibility result [FLP85]
• Practical algorithms “circumventing” [FLP85]
As before, collection of $N$ processes (only message passing)

Consensus must be reached even with faults

Communication channels reliable

Processes may fail: crash, Byzantine (up to $f$ of $N$)
  - And if digitally sign or not (can’t successfully lie about what another process told you); default is no
Definition of consensus problem

• Each proc $p_i$ (i=1,2,…N)
  • Begins in **undecided** state
  • **Proposes** value $v_i$ from set $D$
  • Exchanges values with others
  • Sets **decision variable** $d_i$, entering **decided** state can’t change
Figure 15.16
Consensus for three processes

Consensus algorithm

P₁

P₂

P₃ (crashes)

v₁=proceed

d₁:=proceed

d₂:=proceed

v₂=proceed

v₃=abort
Requirements for consensus algorithm

• Every execution of it always provides:
  • **Termination**: eventually each correct process sets its decision variable
  • **Agreement**: the decision value of all correct processes is the same: if \( p_i \) and \( p_j \) are correct and have entered the decided state, then \( d_i = d_j \) for all \( i, j \)
  • **Integrity**: If the correct processes all proposed the same value, then any correct process in the decided state has chosen that value
    • AKA validity in the literature
    • Weaker variation: decision value a value that some, not all, propose [use?]

• Simple without process failures … multicast, wait for all, all choose majority\((v_1, v_2, \ldots, v_N)\), UNDEF if no majority
  • Could use minimum, maximum, … for some apps and data types
Requirements for Byzantine generals problem

• Three or more generals agree to attack or retreat, one (distinguished process, the commander) issues orders, one or more faulty
  • Different from other flavors of consensus: distinguished process proposes value (most others are peer-to-peer)

• Every execution of it always provides:
  • **Termination (same)**: eventually each correct process sets its decision variable
  • **Agreement (same)**: the decision value of all correct processes is the same: if \( p_i \) and \( p_j \) are correct and have entered the decided state, then \( d_i = d_j \) for all \( i, j \)
  • **Integrity**: If the commander is correct, then all correct processes decide on the value the commander proposed
    • Note: commander need not be correct, no agreement then
Requirements for interactive consistency

• Every process proposes a value, agree on a vector of values
• Every execution of it always provides:
  • **Termination** (same): eventually each correct process sets its decision variable
  • **Agreement**: the decision value of all correct processes is the same
  • **Integrity**: If $p_i$ the correct, all correct processes agree on $v_i$ as the $i$th component of the vector
Equivalence of the fundamental problems

- Problems are equivalent: consensus (C), Byzantine generals (BG), and interactive consistency (IC)
  - See text for details: expressing one in terms of the other
  - Also total order (TO), e.g. consensus on sequence# for a message
- For all, it is reasonable to consider them in terms of
  - Failure model: arbitrary or crash of process
  - Boundedness: synchronous or asynchronous DS
Consensus in a synchronous system [15.5.2]

- Algorithm by Dolev and Strong [1983]
  - $f+1$ rounds of collecting info from each other via $B$-multicast
    - In any round a process could crash sending to some but not all processes
    - Fundamental limitation for consensus even with crash failures
  - Modified Integrity property: if all processes (correct or not) proposed the same value, then correct processes in decided state choose it
    - Because only assuming crash failures, any value sent is correct
    - Allows use of the MINIMUM function to choose decision value
  - $values[r,i]$ holds set of proposed values known to $p_i$ at start of round $r$
  - Rounds limited by timeout
Algorithm for process $p_i \in g$; algorithm proceeds in $f + 1$ rounds

On initialization
$$\text{Values}_i^1 := \{v_i\}; \text{ Values}_i^0 = \{};$$

In round $r$ ($1 \leq r \leq f + 1$)
$$\text{B-multicast}(g, \text{ Values}_i^r - \text{ Values}_i^{r-1});$$ // Send only values that have not been sent
$$\text{Values}_i^{r+1} := \text{ Values}_i^r;$$

while (in round $r$)

{} 

On B-deliver($V_j$) from some $p_j$
$$\text{Values}_i^{r+1} := \text{Values}_i^{r+1} \cup V_j;$$

{} 

After $(f + 1)$ rounds
Assign $d_i = \text{minimum} (\text{Values}_i^{f+1});$
Byzantine generals problem in a synchronous system [15.5.3]

• System model
  • Processes can fail arbitrarily
  • Communication channels are pairwise and private
    • I.e., a process can’t snoop and then determine another process is lying
    • No process can inject a message into the channel

• Need $\geq 3f+1$ processes to tolerate $f$ failures with unsigned messages

• Need $\geq f+1$ rounds for both crash and arbitrary process failure [why?]

• Scenario: commander sends order to lieutenants, who then agree on what they were ordered to do

• Notation: $x:y:z$ means $p_x$ says $p_y$ said value $z$. 
Figure 15.18
Three Byzantine generals

Faulty processes are shown coloured

$p_2$ votes on?  
[v,u]

$p_2$ votes on?  
[x,w]

$p_2$ can’t tell who failed (whose value to ignore); could if messages signed
Figure 15.19
Four Byzantine generals

Faulty processes are shown coloured

- MAJORITY in correct processes chooses \( v \) (left) or UNDEF (right)
- Complexity: \( f+1 \) rounds \( O(N^{f+1}) \) messages, later \( O(N^2) \) signed
- Implicit timeout (not shown) turns lack of vote into UNDEF
- Ergo simple majority fine
Impossibility in asynchronous systems

• Assumed so far: rounds of messages, can set a timeout and assume failed

• In asynch. system, can’t be guaranteed to reach consensus with even 1 process crash failure [FLP85]
  • Can’t distinguish a crashed process from a slow one
    ➔ no solution to Byzantine generals, interactive consistency, totally ordered multicast

• Workaround #1: Mask faults (see [2.4.2])
  • Use persistent storage of state & process restart
  • Takes longer but still works
Impossibility in asynchronous systems (cont.)

• Workaround #2: using “perfect by design” failure detectors
  • Declare the unresponsive process to have failed
  • Remove from the group
  • Ignore any messages from it
  • Analysis?

• Workaround #3: use *eventually weak failure detectors*
  • [Chandra and Toueg 1996], with reliable coms and <half crashed
  • *Eventually weak complete*: each faulty process is *eventually suspected permanently* by *some* correct process
  • *Eventually weak accurate*: after some point in time, at least one correct process is *never suspected* by *any* correct process
  • Adaptive timeout scheme (15.1) can come close to this

• W. #4: consensus w/randomization (confuse adversary)