Dependable Systems

Coordination and Consensus

Dr. Peter Tröger

Sources:

Coulouris, George; Dollimore, Jean; Kindberg, Tim: Distributed Systems - Concepts and Design. 4. Edition. Addison Wesley, 2005. , 0321263545

Jalote, Pankaj: Fault Tolerance in Distributed Systems. Upper Saddle River, NJ, USA : Prentice-Hall, Inc., 1994.

Coordination Problems

- Collection of processes in a fault-tolerance setup typically need to agree on:
  - **Mutual exclusion** for shared resource access
    - Concurrent access to shared resource in distributed system
    - Critical section problem - like in OS, but no shared memory
  - **Election of leader**
    - Choose a process to play a particular role
    - Difference to mutual exclusion - leader must be known to everybody, and does not give away leadership explicitly after some time
  - **Ordered multicast**
    - Group of processes should receive copies of message sent to the group, sometimes with delivery and order guarantees

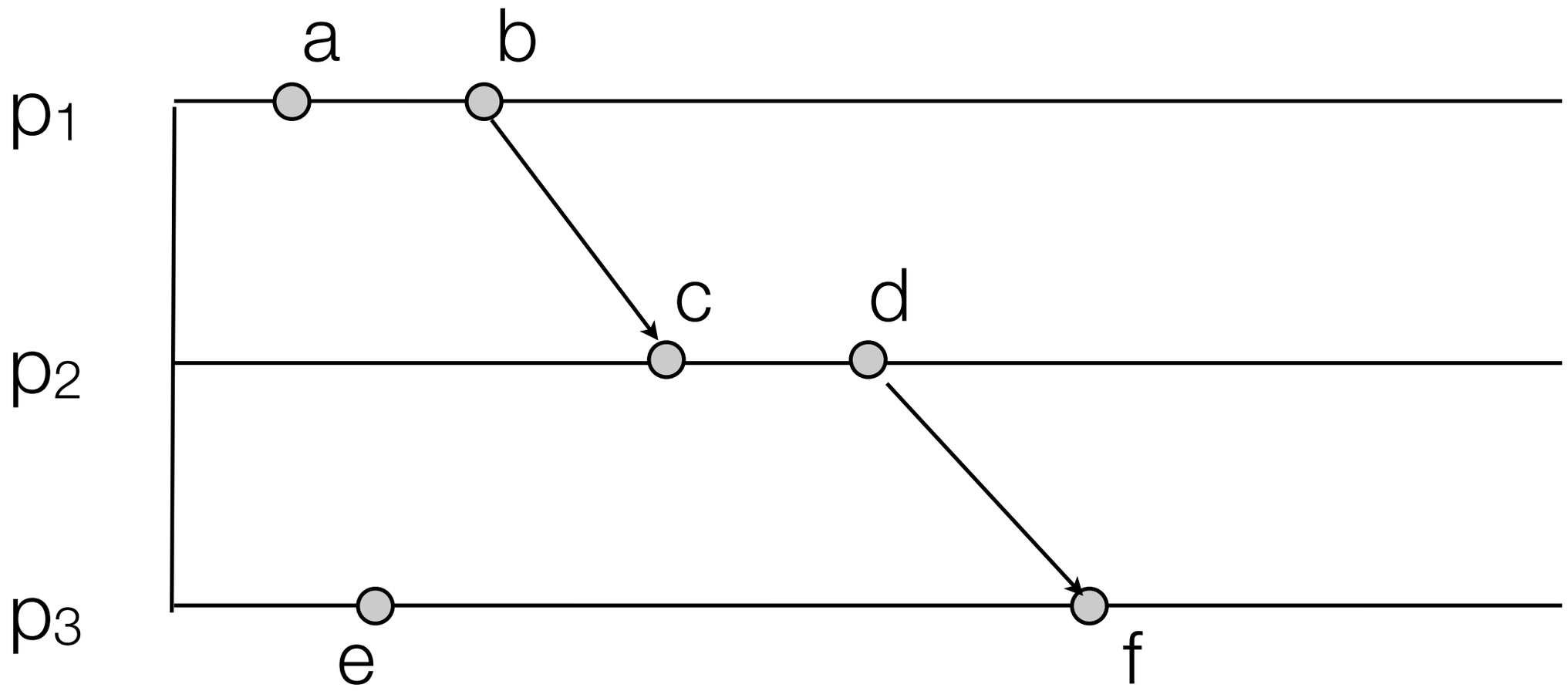
One Issue: Time

- No perfect clock synchronization in a distributed system
  - Physical time unsuitable for global event ordering
- Causal ordering with *happened-before* relation "->" can already be good enough:
  - Same process, same local clock, time of *e* smaller then of *e'*: *e* -> *e'*
  - Different processes with message exchange:
    - Event of sending a message is before the event of receiving the message
    - For any message m, *send(m) -> receive(m)*
  - If *e*, *e'* and *e"* are events with *e->e'* and *e'->e"*, then *e->e"*
- Without *happened - before*, events are *concurrent* - *e||e'*

A Good Tool: Logical clocks

- Problems with *happened-before* relation
  - Captures data flow, but demands network message
- Idea: Define logical clock L_i for each process, to capture ordering numerically
  - L_i is increments before each event is issued
    - Message sent contains L_i, receiver sets its own clock L_j to $L_j = max(L_j, L_i) + 1$
- Events on different processes might have same timestamp
  - Totally ordered logical clocks by considering process identifiers
  - $(T_i, i) < (T_j, j)$ if either $T_i < T_j$, or $T_i = T_j$ and $i<j$ - no physical significance
- *e -> e'* leads to $L(e) <= L(e')$, but $L(e) < L(e')$ does not lead to *e -> e'*
  - Clock consistency condition
  - Strict version demands more powerful clock concept

Happened-before Example



Proof: a->f and e||a
Show logical clocks, starting with a=c=e=0

Mutual Exclusion

- Assumptions from theory:
  Processes do not fail, reliable message delivery, at-most-once
- Algorithms intended to ensure exclusive access in critical section
  - No deadlocks and starvation, fairness, message passing
- Central server algorithm
  - Central server grants permission to enter critical section by providing token -> *lock service*
  - Messages: Request token, release token, grant token
  - Requests are queued on the central server, clients are therefore blocked
  - Example: File locking daemon *lockd* in NFS

Example: NFS lockd

- RPC-based Network Lock Manager (NLM) service
  - *rpc.lockd* daemon on both client and server side
  - Application issues *fctl()* command for file locking, kernel sends request to local lock daemon (Kernel Lock Manager protocol)
  - Forwarded to remote host, 5 asynchronous operations
    - Test for lock (returns conflicting lock, or LCK_GRANTED)
    - Claim a lock (LCK_GRANTED or LCK_DENIED / LCK_BLOCKED)
    - Unlock a lock
    - Cancel a blocked lock request (client is no longer interested)
    - Grant a blocked lock (notification of server to client)

Example: NFS statd

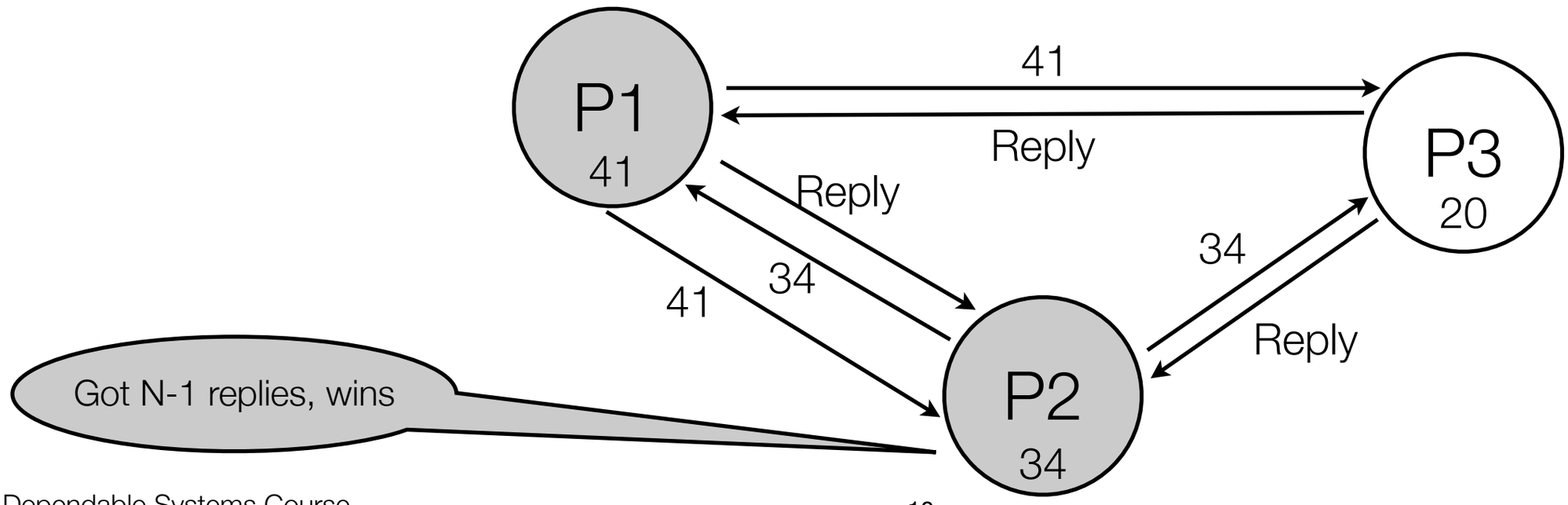
- Network Status Monitor (NSM) protocol for status change notification, implemented by *rpc.statd* daemon
- Contains fault tolerance mechanism for failing server / client
  - When lock server grants lock request, it registers a callback in *statd*
  - Invoked when the client has a status change
    - On client crash and reboot, client informs server *statd* about the issue
    - Formwarded to server *lockd*, which frees all the locks
- Same mechanism for server crash
  - Client *lockd* gets information, now tries to re-allocate all locks

Ring-based Mutual Exclusion

- Ring-based algorithm
  - Arrange agreement on mutual exclusion without additional node
  - Unidirectional ring topology, unrelated to physical interconnection
  - One token per protected resource, forwarded if process has no interest
- Performance comparison of mutex approaches
  - Consumed bandwidth as number of messages for one reservation
  - Client delay for critical section entry and exit
  - Synchronization delay between one process leaving and the next one entering the critical section

Multicast-based Mutual Exclusion

- Ricart and Agrawala, 1981
- Process demanding critical section entry multicasts a request message to N-1 other processes, can only enter after it collected N-1 replies
  - Each process keeps logical Lamport clock, has either critical section released (answers immediately with REPLY), held (delays reply), or wants it too (delays replies if own request timestamp was earlier)
- Example: Concurrent request by P1 and P2



Fault-Tolerance of Mutual Exclusion

- Lost messages
  - Not tolerated by algorithms, must be solved by other means
- Crashed process
  - Not tolerated by ring-based approaches
  - Server can tolerate client crash if it neither holds or requested the token
  - Workarounds by book-keeping (see NFS)
- Even if faulty process can be detected, system state must be re-established
  - Example: Determine current owner of the token in server-based approach

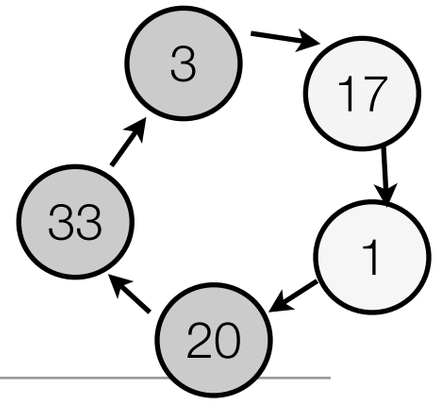
Election Algorithms

- Choose an unique process to play a particular role
  - Election is needed when the current leader fails, one node could be a better leader, or new nodes enter the system
    - One process calls the election
    - Unique choice needed, even with multiple parallel elections
  - Example: Time server in Berkeley clock synchronization algorithm
  - Example: IEEE1394 FireWire Tree Identify Protocol
    - Leader election after bus reset (node added or removed), root node acts as manager of the bus communication
    - Includes cycle detection (see „Distributed Algorithms“, N. Lynch 96)
- Timeout as failure detector might lead to wrong results

Assumptions for Leader Election

- Elected process should be the one with the largest identifier
  - Unique identifiers, totally ordered
  - Algorithm should tell all participants the leader with largest identifier
  - Processes marked as participants or non-participants
- Assumptions (Molina)
  - All nodes rely on the same algorithm
  - No bugs, no message spoofing, non-volatile node memory
  - At-most-once transfer of messages, reliable network
  - Maximum response time for nodes, otherwise failed

Ring-based Election (Chang & Roberts ‘79)

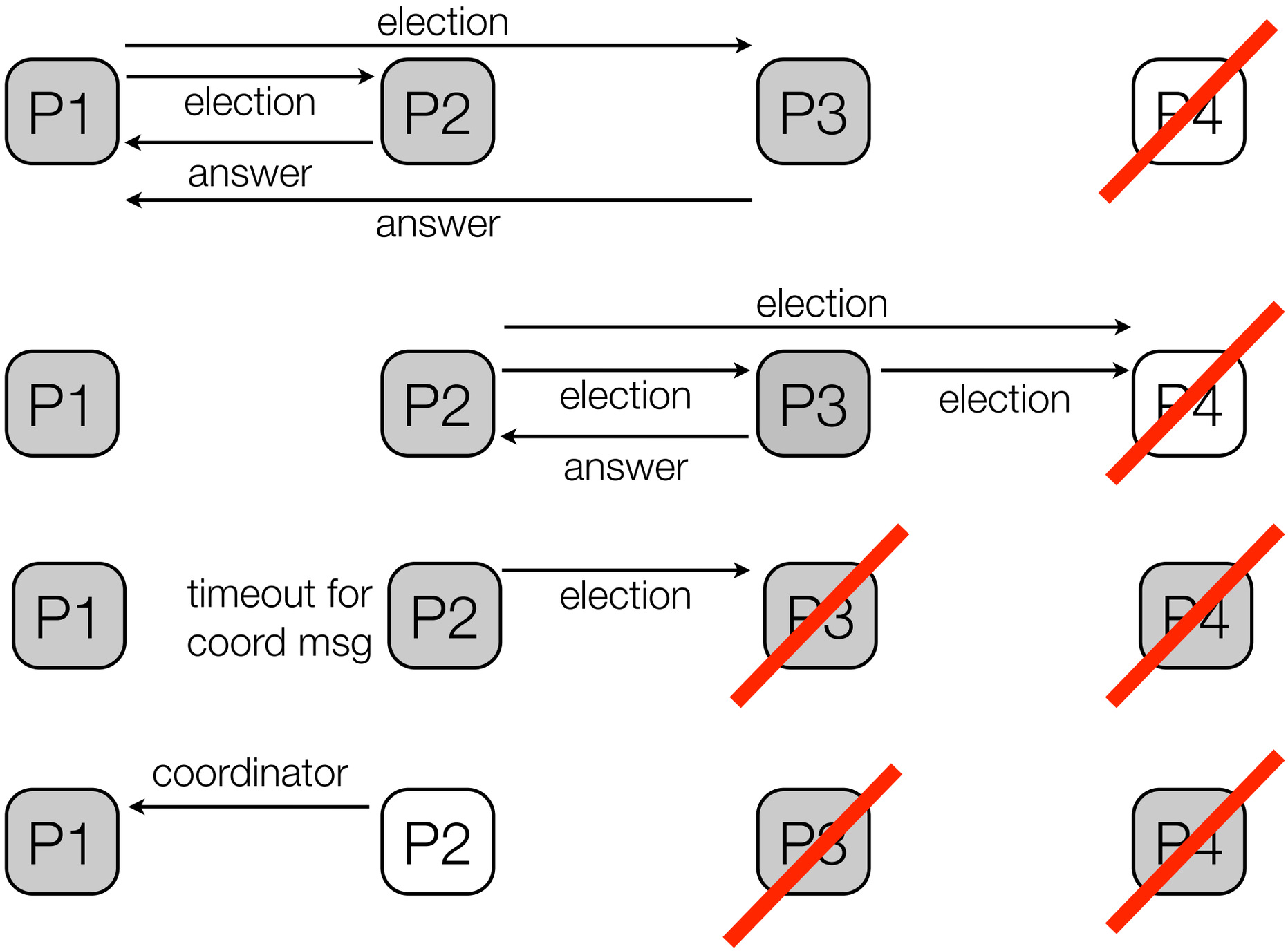


- Everybody marked as *none-participant*, one node changes its status and sends *election* message with its own id to clockwise neighbor
- On receipt of an *election* message
  - If the message node ID is greater than your own, become a *participant* and forward message
  - If smaller and node is *non-participant*, exchange contained number, become a *participant* and then forward message
  - If smaller and node is already *participant*, don‘t forward message (catch multiple elections)
  - If contained node ID is own ID and node is already *participant*, become the leader, change to *non-participant* and send *elected* message
- On receipt of an elected message, become *none-participant* and forward
- No fault tolerance, since ring is not allowed to break

Bully Algorithm (Garcia-Molina ‘82)

- Processes are allowed to crash during election (but: reliable message delivery demanded)
  - Timeout used to detect node crashes, triggers new election
  - Each process knows processes with higher numbers (similar to ring)
- *Election* message (announcement), *answer* message, *coordinator* message
- Process receiving an *election* message sends back an *answer* message and starts an own election with the higher nodes
  - Sends *election* message to higher numbers and waits for *answer* message
  - If *answer* is received, waits for *coordinator* message from this node
    - If not received, restart of election
  - When timeout occurs, sends *coordinator* message to lower numbers (win)
- Process receiving a *coordinator* message from lower ID restart election

Bully Algorithm Example



Leader Election in Practice - SMB Election (most likely outdated)

- Computer Browser Service: Central list of resources in the network
  - Netware: Periodic announcement messages, high traffic
  - Windows: Elected master browser, used by all other resources
- Domain Master Browser: Receives browser lists from master browsers
  - Typically PDC, every subnet has own master browser
- Master Browser: Master copy of network resources, periodically updated by all available machines in the subnet
- Backup Browser: Fetches copy of resource list, offers it to clients
- Browser Clients: Use special mail-slot BROWSE for queries
- Role of computer depends on operating system type and version

SMB Election

- Master browser election process triggered by *RequestElection* message to the *“\MAILSLOT\MSBROWSE“* mailslot
  - If browser recently lost a round, it looses again
  - Browser determines if it has won the round by comparing senders election version with own number (or uptime, or lexical name comparison)
- If a browser wins, it sends out a new *RequestElection* message after some delay based on the current role
  - Master Browsers and Domain Master Browsers (100ms), Backup Browsers (200-600ms), all other (800-3000ms)
  - If a browser wins 4 rounds in a row, it becomes the master browser, and sends out some announcement frame to the remaining machines

SMB Problems

- Worst case machine removal time of 51 minutes
  - Backup browser fetches updated list every 15 minutes
  - Computer announces itself in intervals (1 - 12 minutes)
    - After three left out intervals, computer is removed
- Too many election processes (e.g. by faulty Win95 clients) were able to make the master browser unavailable
- Problem with broadcast messages over subnets - WINS server
  - Own naming services for NETBIOS networks
  - Meanwhile ActiveDirectory / DNS-based approaches

Multicast Communication

- Group communication
  - Group of processes should receive copies of message sent to the group, sometimes with delivery and order guarantees
  - Process may join and leave group at arbitrary times
  - Many research implementations: V-System (1985), Chorus (1988), Amoeba (1991), Isis (1993), Horus (1996), Totem (1996), Transis (1996)
- Example: Database replication across several sites, query always routed to the nearest copy
  - Two concurrent updates, events arrive at different order at two sites (add $100 to account - increase account by 1% interest)
  - Demands totally-ordered multicast - all events should happen everywhere in the same order
- Crash-fault assumption

Group Communication

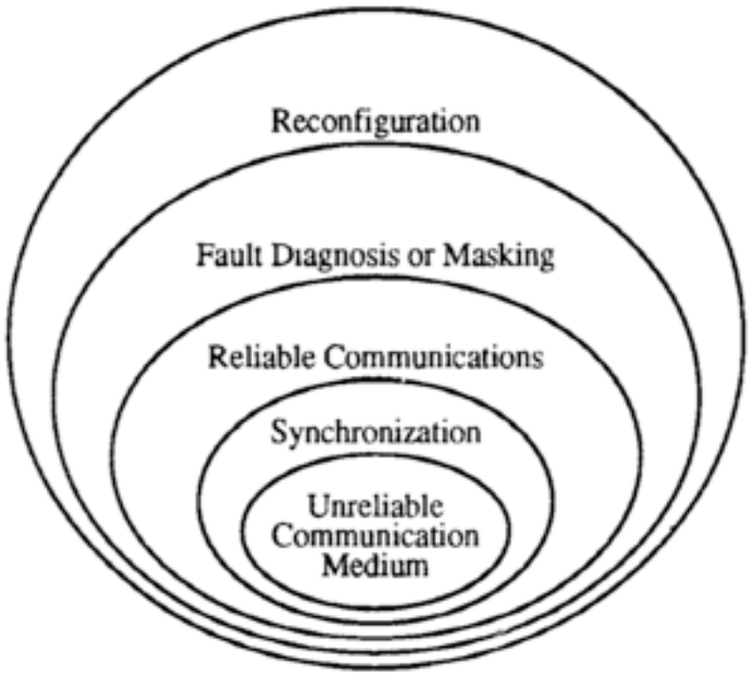
- System model: reliable communication, processes may crash, messages carry sender ID, closed groups vs. open groups
  - *multicast (group, message)*
  - *deliver (message)*
- Delivery to application separated from receive event
  - Re-ordering of incoming messages to fulfill some order guarantees
- Different implementation strategies
  - Hardware support (multicast addresses)
  - Broadcast communication with filtering
  - Point-to-point to all members

Message Ordering

- FIFO Order
  - If correct member executes multicast(m_1) before multicast(m_2), then no other correct member delivers m_2 before m_1
- Causal Order
  - If correct member executes multicast(m_2) after receipt of m_1, then no other correct member delivers m_2 before m_1
  - If multicast(m_1)->multicast(m_2), then deliver(m_1)->deliver(m_2) (Happens-before)
- Total (Agreed) Order
  - Message delivery order is the same at all members in the group
- Total FIFO and total causal order

Consensus Problems

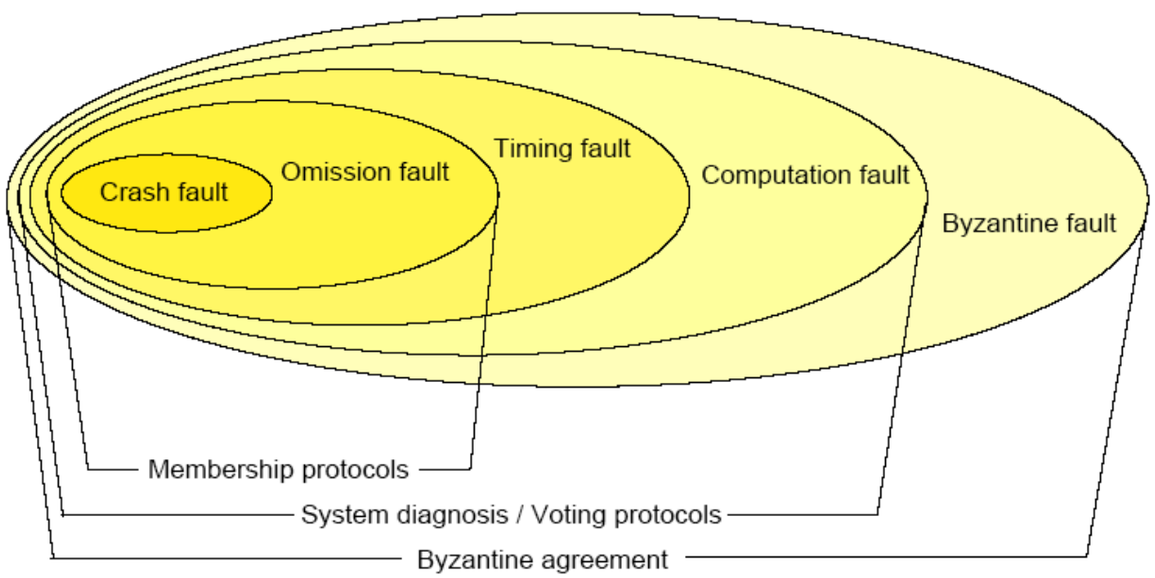
- Analysis of dependability issues in distributed systems on a high abstraction level
  - Independent nodes
  - Only message exchange
  - Nodes fail according to different fault models
- **Consensus problems**
  - Set of error-free nodes agrees on some system state
  - Is not influenced by the behavior of the faulty nodes
  - Can be understood as „coordination in the error case“
  - Examples: Resource management, scheduling, fault diagnosis, synchronization, reliable broadcast, reconfiguration, scheduling, voting, group membership, ...

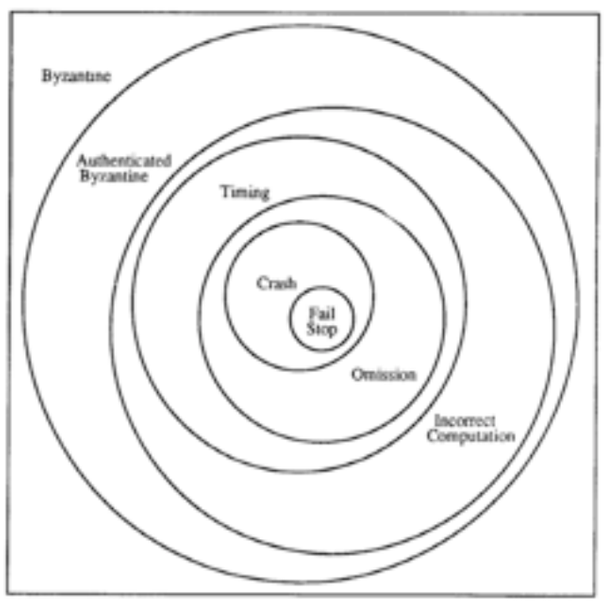


[Barborak et al.]

Fault Model (Flaviu Cristian)

- Originates from hardware background, meanwhile adopted to software
  - How many faults of different classes can occur
- Process as black box, only look on input and output messages
- Link faults are mapped to the participating nodes
- Timing of faults: Fault delay, repeat time, recovery time, reboot time, ...





Fault Types

- Fail Stop Fault : Processor stops all operations, notifies the other ones
- Crash Fault : Processor looses internal state or stops without notification
- Omission Fault : Processor will break a deadline or cannot start a task
  - Send / Receiver Omission Fault: Necessary message was not not sent / not received in time
- Timing Fault / Performance Fault : Processor stops a task before its time window, after its time window, or never
- Incorrect Computation Fault : No correct output on correct input
- Byzantine Fault / Arbitrary Fault : Every possible fault
  - Authenticated Byzantine Fault : Every possible fault, but authenticated messages cannot be tampered

Failure Detectors

- Coordination research typically concentrates on crash or byzantine failures
  - Detection of process crash by *failure detector* [Chandra and Toueg 96]
  - Processes query if a particular other process failed
- Unreliable failure detector states *unsuspected* or *suspected*
  - Decision maybe based on message arrival rate
  - Example implementation: pair-wise heartbeat
  - Too short timeout (many wrongly suspected) vs. too long timeout (many wrongly unsuspected)
- Reliable failure detector states *unsuspected* or *failed*
  - Implementation demands upper limit for network transfer

System-Level Diagnosis

- Complex hardware uses multiple approaches in different locations
  - Processor hardware: Signature analysis, watchdogs
  - Memory hardware: Error-correcting codes, parity
  - Network hardware: Checksumming, CRC, ...
- Concurrent detection is typically combined with component-level error correction
- Packaging determines some level of fault localization
- **Concurrent diagnosis** needs to consider component dependencies and relation
  - Parallel vs. localized diagnosis
  - Centralized vs. distributed diagnosis
- Many algorithm proposals for **consistent system-level diagnosis -> consensus**

System-Level Diagnosis

- Basic idea: Avoid steep costs of NMR and special testing hardware, by letting processing elements test each other for correct functioning
  - Test exists to check component A from component B
  - ‚Bad‘ component can be detected safely, if the tester works
    - Problem: Faulty components do not deliver correct test results
- Definition: A system is **t-diagnosable** if any distribution of t faults is diagnosable (detectable **and** locatable)
  - Assumes existence of an external supervisor that safely collects the results
  - Also used for systems with correctness demands in timing behavior
- Tests are assumed to be exhaustive for the given fault model
- Faults are assumed to be permanent (until the diagnosis is finished)

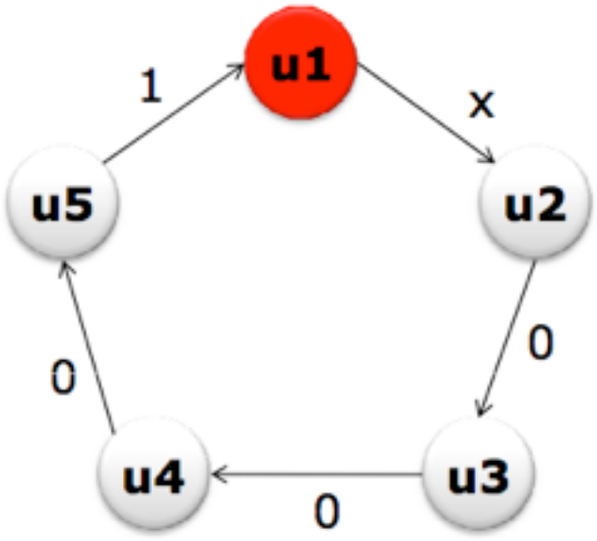
Preparata Metze Chien (PMC) Model

- Published in 1967 as algorithm for diagnosis in multiprocessor systems
- System is decomposed into n (replaceable) units
  - Units don‘t have to be identical, $U=u_1, u_2, u_3, ..., u_n$
  - Each unit is either ‚faulty‘ (1) or ‚fault-free‘ (0), nodes test each other
  - Centralized component collects all test results for the system diagnosis
- Result of all pair-wise tests is called **syndrome -** the base for fault location
  - If the testing node is fault-free, it returns the correct status
  - If the testing node is faulty, the result can be arbitrary
  - No unit tests itself

PMC Model

- Non-failing supervisor collects syndrome
- All syndroms are different, so fault can be detected and located

Faulty Unit	Possible Syndrom				
1	0	0	0	0	1
	1	0	0	0	1
2	1	0	0	0	0
	1	1	0	0	0
3	0	1	0	0	0
	0	1	1	0	0
4	0	0	1	0	0
	0	0	1	1	0
5	0	0	0	1	0
	0	0	0	1	1

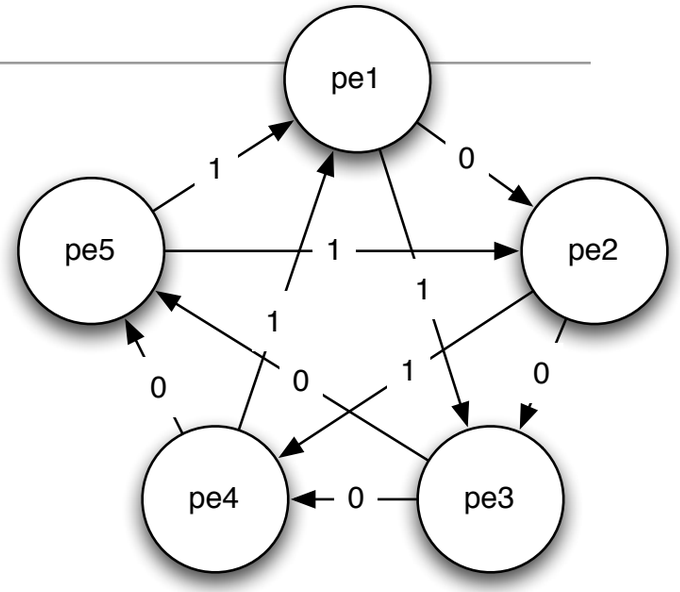


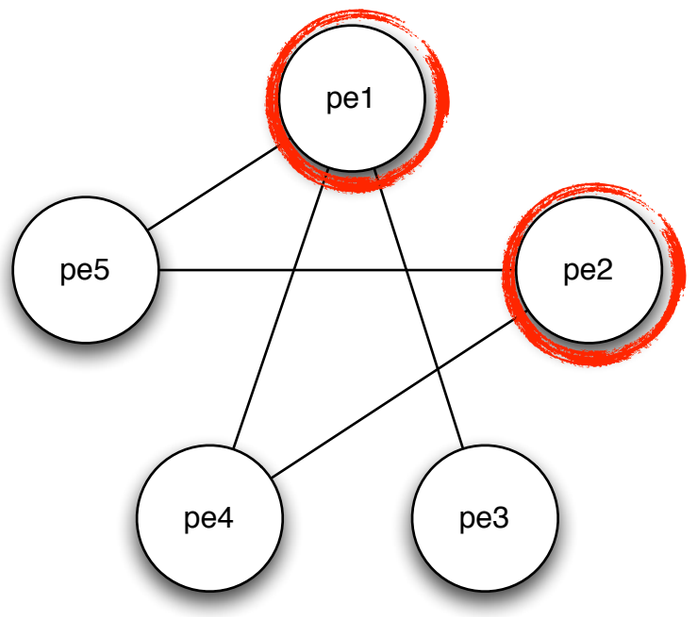
PMC Model

- A system is **t-diagnosable** with **PMC** if each node is tested by at least **t** other nodes and **n >= 2t+1**
- Example: Test ring with three nodes is 1-diagnosable, but not 2-diagnosable
- Strict assumptions in PMC
  - All faults are permanent faults
  - A fault free processor can always determine accurately the test result
  - A central perfect arbitration unit exists
  - Not more than t processors may be faulty in one diagnosis round

Diagnosis with the PMC Model

- Diagnosis algorithm by Dahbura & Masson (1984)
  - Best solution in terms of worse-case efficiency $O(n^{2.5})$
  - Convert test graph into *disagreement graph*
    - Choose a processing element, and assume it is fault free
    - Draw edge to processing elements that are then understood as faulty
    - Repeat for all nodes
- Edges represent *implied faulty set* from PEs viewpoint
- Find *minimum vertex cover* of the result
  - Minimal set of nodes so that each edge has one of them as end vertex -> represent the faulty units





Two Generals Problem

- Credited to Jim Gray, developed by Akkoyunlu et al.
- Thought experiment for coordination of an unreliable link
  - Two armies with generals prepare attack on city, each on a hill
  - Only communication path through a valley, occupied by city defenders
  - Agreement on attack exists, but not on time
  - Acknowledgment of receipt is demanded to be sure
- Proven that no algorithms can be designed to solve the problem
- Use cases
  - Electing a coordinator, decide upon transaction commit, synchronization

Two Generals Problem

- Proof scenario: Sending acknowledgment for the acknowledgment for the acknowledgment ... (every message could be modified)
- Termination on number of steps makes the last message the relevant one
  - But sender of last message still lacks trustworthy acknowledgment
    - Even with non-faulty participants, trustworthy agreement cannot be reached
      - Models the unreliable link problem
- Arbitrary error states are getting more relevant in practice
  - Malicious attacks, software errors, non-deterministic hardware / software stacks
- Real world solution: Lower probability of message tampering (repeated send, informational redundancy etc.)

Byzantine Generals

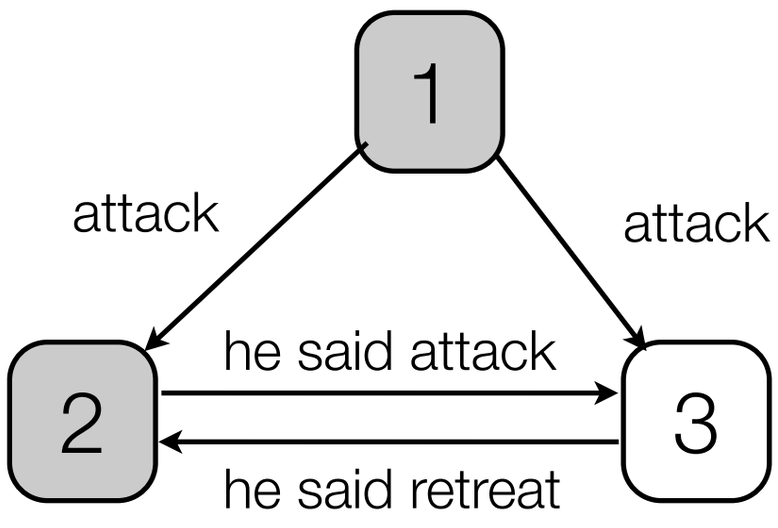
- *Lamport, L., Shostak, R., and Pease, M. 1982. The Byzantine Generals Problem. ACM Trans. Program. Lang. Syst. 4, 3 (Jul. 1982), 382-401*
- Coined the term „Byzantine failures“ - system components fails in arbitrary way
  - Incorrect or inconsistent results, altered node state, silence, message creation, ...
- Consensus problem, generalization of the *Two Generals‘ Problem*
  - Generals of the Byzantine army (on hills) must decide upon attack
  - Presence of traitors amongst the generals
    - Trick some generals into attacking
    - Force a decision that is not consistent with the generals' desires
    - Confusing some generals so that they are unable to make up their minds
  - *Loyal generals* must find unanimous agreement on their strategy

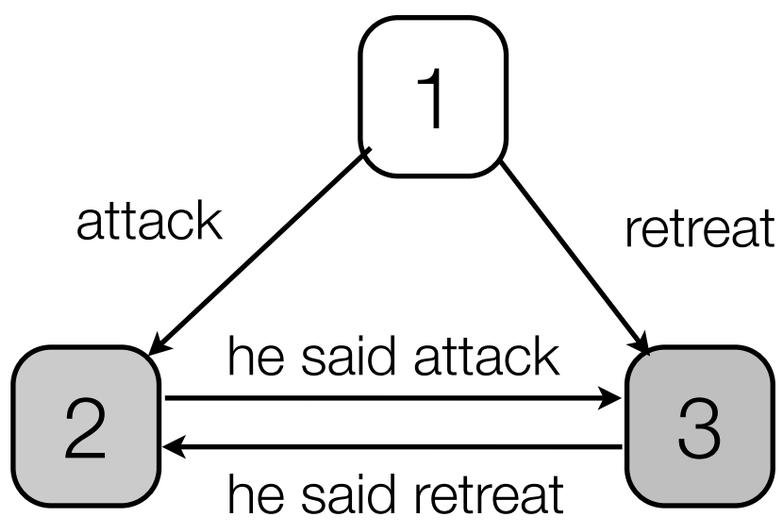
Byzantine Generals

- Termination condition: Each correct process decides on something in finite time
- Agreement condition: Decision value of all correct processes is the same
- Integrity condition: If the commander is correct, all processes have agreement
- Demands on communication mechanisms
  - Synchronous: Perfectly synchronized clocks, delivery in one time unit
  - Authenticated: Identity of sender is known to the receiver
  - Point-to-point: Communication structure is a complete graph
- Generalized impossibility result by Pease et.al.
  - Let f be the maximum number of faulty processes in a system of n processes. As long as **n<=3f**, there is no algorithm to solve the byzantine generals problem.

Three Generals - „n<=3f“

- n=2, f=1 : clear
- For n=3, f=1, it is also impossible to reach consensus
  - Left pic: P2 cannot figure out who the traitor is, can‘t decide for activity of P1
  - Right pic: P2/P3 cannot decide for the same activity as P3/P2
  - Violates agreement condition, since correct nodes should do the same action

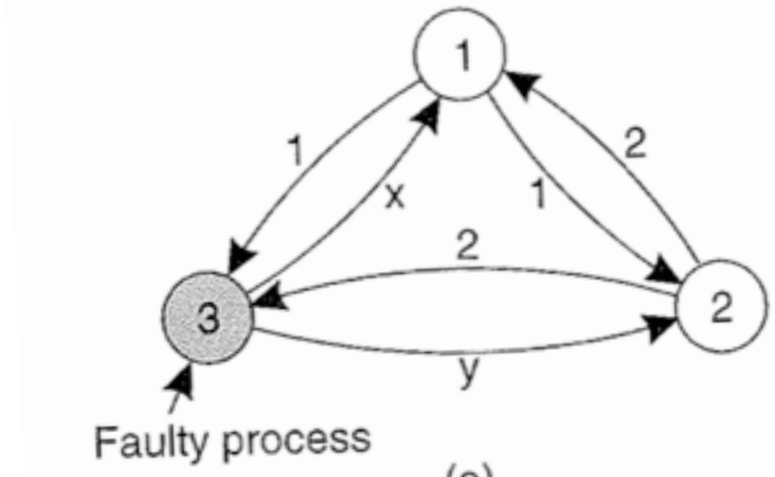




Byzantine Generals

- Many impossibility proofs with lower bounds for specialized conditions
- Algorithm example
  - Assume fully connected graph
  - n=#nodes, f=#faulty nodes
  - n+1 rounds
    - Round 0: Every node sends every other node his value.
    - Round i: Node i sends all responses it got to all other nodes
    - After all rounds, majority voting on other nodes values per node

Byzantine Generals - Example



(a)

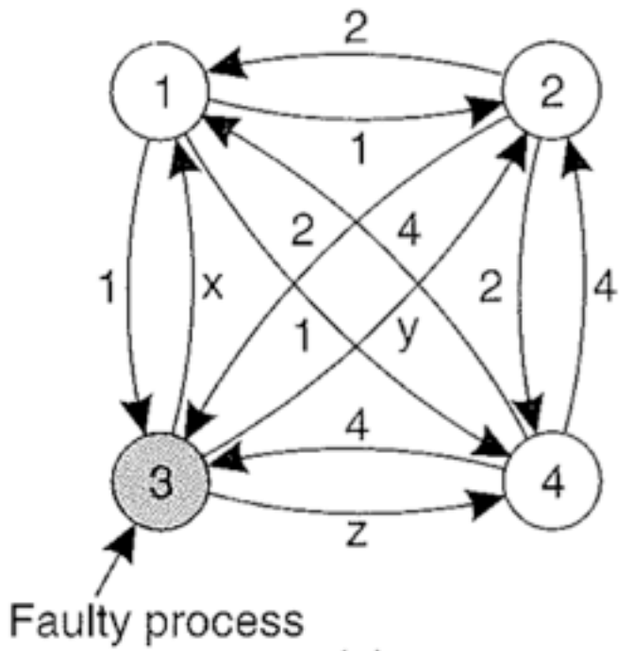
1 Got(1, 2, x)
2 Got(1, 2, y)
3 Got(1, 2, 3)

(b)

1 Got	2 Got
(1, 2, y)	(1, 2, x)
(a, b, c)	(d, e, f)

(c)

No majority decision possible



(a)

1 Got(1, 2, x, 4)
2 Got(1, 2, y, 4)
3 Got(1, 2, 3, 4)
4 Got(1, 2, z, 4)

(b)

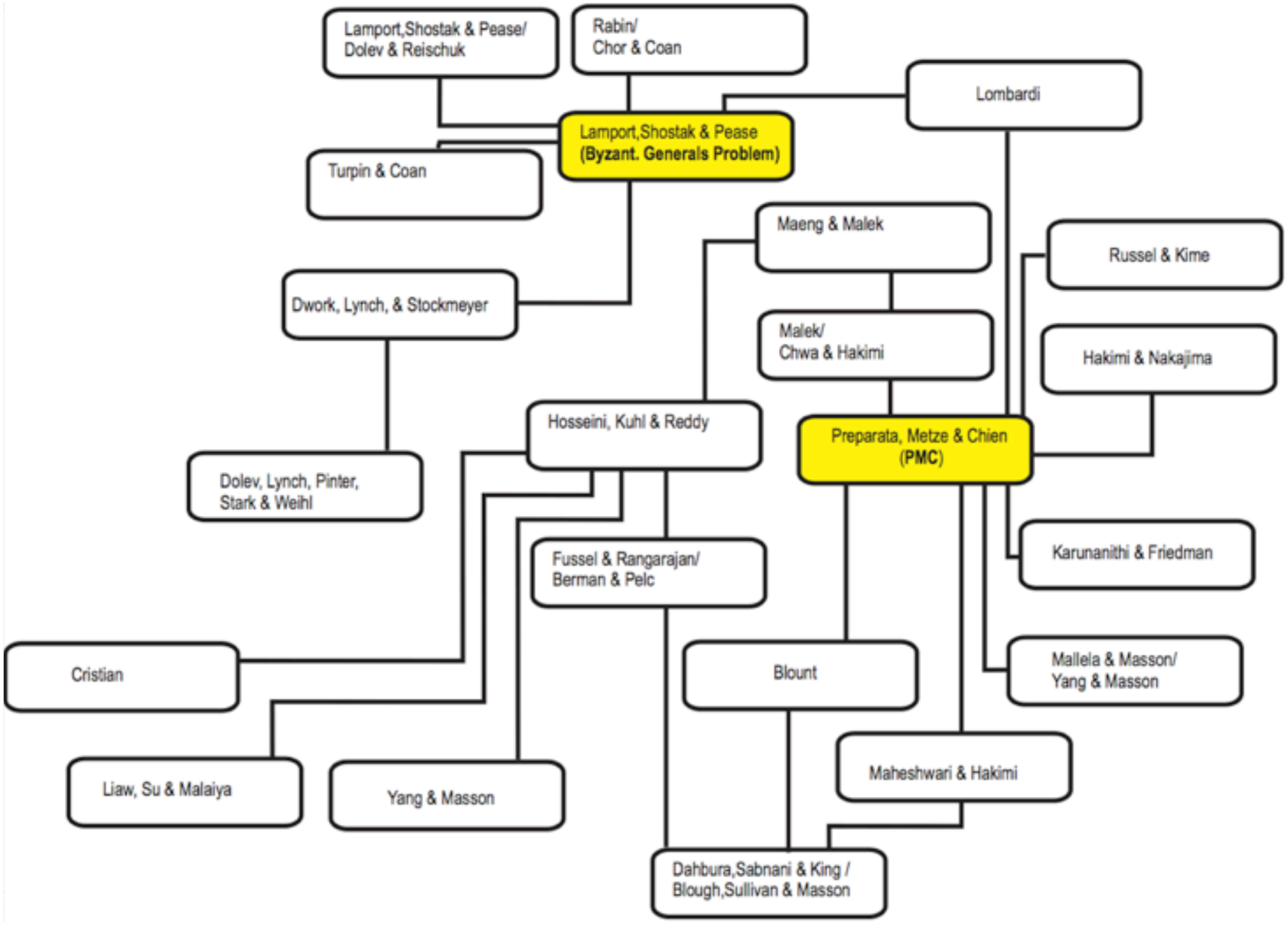
1 Got	2 Got	4 Got
(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

(c)

Byzantine Generals

- Byzantine agreement becomes simpler if messages are authenticated or signed
  - A loyal general's signature cannot be forged, and any alteration of the contents of his signed messages can be detected
  - Anyone can verify the authenticity of a general's signature
  - Allows to identify differing answers from illoyal partners
- Several initial implementation attempts, mostly to costly in terms of messages
- Today available in replication support libraries, based on optimized algorithms

Consensus Problem Families [Werner]



Distributed Snapshot

- Record **consistent global state** of an asynchronous system
  - No communication failures, all messages arrive in order
  - Fully connected communication graph, any process may initiate the snapshot
  - Useful for checkpointing, consistency of distributed information, barriers
- Easy approach with global physical clock
  - Set common snapshot time on all nodes
  - Extend messages with time stamps
  - Save local snapshot on time, save all later arriving messages with time stamp before snapshot time
- Without global time ?

Distributed Snapshot

- *Chandy-Lamport algorithm*
  - Initiator saves local state and sends message with marker to all others
  - On marker, if node did not save own state so far
    - Save state, send markers to all outgoing channels
  - On marker, if node saved state already
    - Record all messages between local state saving and marker arrival
    - Expresses responsibility of the node to save the channel state
  - Terminate if marker was received on every incoming channel
    - Send recorded local state and saved messages to initiator
- Marker acts as delimiter on FIFO channel
- Snapshot creation could be started in parallel, add version number to marker

Example: Chubby

- *Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. OSDI 06*
- File and record locking for reads and writes
  - Standard performance optimization problem in databases and file servers
  - Clients in a loosely-coupled system synchronize their activities with the lock service
- *Chubby cell* - one instance of the lock service, typically per data center
- Used for leader election
  - Google File System determines GFS master server with Chubby lock
  - Google Bigtable selects a master and permits clients to find the master by Chubby
- Distributed consensus problem in an asynchronous network

Example: Chubby

- Chubby cells use distributed consensus protocol to elect a master
  - Solved by the Paxos protocol [Lamport]
  - Master must obtain votes from majority of participants
  - Promise for minimum election time - *master lease*
- Replicas only copy updates from the master
- Clients find the master by asking their replica, only talk to master then
  - Write requests are acknowledged when a majority of replicas are updated
  - Read requests are answered by the elected master
- Intended for coarse-grained locks (days / weeks)
- Each Chubby lock identified by virtual Unix path, reader-writer-lock semantics

Paxos

- Chubby replica has fault-tolerant log, synchronized by Paxos algorithms
  - Consensus between set of processes (replicas) to agree on a value
  - Replicas may crash, recover, drop messages
  - Three phases in the algorithm (might be repeated)
    - Elect a replica to be the coordinator
    - Step 1: Coordinator broadcasts an *accept message*
    - Step 2: Other replicas either *acknowledge* or *reject* it
    - On acknowledge from majority, consensus has been reached
    - Step 3: Coordinator broadcasts *commit message* to notify replicas
- Paxos supports multiple coordinators that propose even different values