

## Lecture 19: Fault Tolerance, Group Communication and Replicated State Machines

CSC 469H1F / CSC2208H1F  
Fall 2007  
Angela Demke Brown



## Basic Concepts & Definitions

- *Fault tolerance* is the ability of a system to continue operating in the presence of faults
- Closely-related to requirements on *dependable systems*
  - Availability: probability that system is working correctly at any given time
  - Reliability: probability that system can run continuously without failure
  - Safety: temporary faults do not lead to catastrophic failures
  - Maintainability: ease of repairing a failed system



CSC469

## Avoiding faults

- All of the standard coping with complexity stuff
  - software engineering, testing, etc...
- There are also some design "rules" that can help
  - Example: avoid situations in which things often go wrong
    - 90% utilization of file system capacity
    - minimum number of free pages
  - Example: regular maintenance
    - planned restarts: occasionally reset to clean state
    - patches/upgrades: don't leave known problems laying around
  - Example: detect problematic activity at system boundary
    - firewalls for blocking suspect traffic
- Note that this *reduces* rather than *prevents* problems



CSC469

## Masking/hiding faults

- Obvious requirement: redundancy
  - Must be able to repair broken sets of bits
    - e.g., error correction codes
  - Must be able to communicate despite broken paths
    - e.g., redundant routes, dual ported devices, etc...
  - Must be able to continue with broken servers
    - e.g., have more than one server providing same service
    - Requires *group communication* → distributed consensus



CSC469

## Recovering from faults

- Many systems are designed to tolerate a single fault
  - Must detect and recover before a second fault occurs
  - Generalizes to tolerating  $f$  faults, recovering before fault  $f+1$  occurs
- In general, requires restoring *state* of restarted process or service
  - *Checkpointing*: save state to *stable storage*
  - *Replicated state machines*: rebuild state from other group members

CSC469



## Replicated State Machines (RSMs)

- Architecture
  - Implement a service as a *state machine*
    - State variables
    - Commands
  - Replicate the state machine on different servers
  - Clients interact with sets of servers
- Rationale
  - Fault-tolerance/Availability/Reliability

CSC469



## State Machine Commands

- A message that the state machine receives
- Commands must execute atomically with respect to other commands
  - Referred to as 'linearizability'
- Commands
  - Modify state variables
  - Produce outputs
- The state/output of a state machine is completely determined by:
  - initial state
  - sequence of commands

CSC469



## RSMs & Failures

- In the case of failures
  - Clients must determine correct output of RSMs
  - RSMs are called  $t$ -tolerant
    - Fail-stop:  $t + 1$  replicas required ( $t$  correct replica sufficient)
    - Byzantine:  $2t + 1$  replicas required ( $t + 1$  correct replicas sufficient)
- Different than Broadcast/Consensus failures
  - One client must decide on result, replicas don't have to agree with each other about result

CSC469



## RSMs, Consensus & Reliable Broadcast

- Each correct replica
  - Must execute same commands in same order
  - Since all correct replicas must have the same state
  - Therefore, RSMs require Distributed Consensus to agree on order of commands
- Needs form of *group communication* called *atomic broadcast*

CSC469



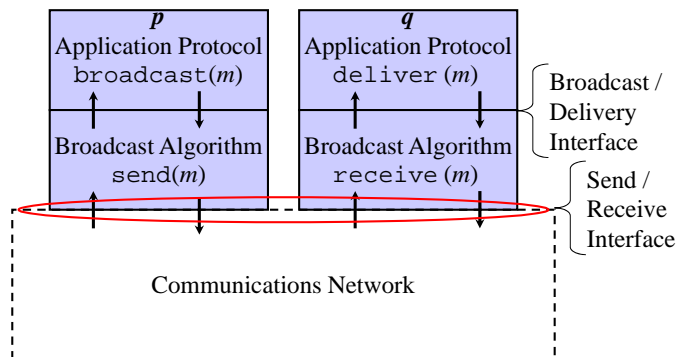
## Group communication

- In many applications processes must be able to reliably broadcast messages, so that they agree on the set of messages they deliver.
- Reliable broadcast is difficult because distributed processes do not know each other's state.
- Much of this material is taken from chapter five by Hadzilacos and Toueg in "Distributed Systems", Sape Mullender, ed.
  - Reliable broadcast taxonomy
  - Example broadcast algorithms

CSC469



## Application / Broadcast Mechanism



CSC469



## Properties of Send/Receive

- *Validity*. If  $p$  sends  $m$  to  $q$ , and both  $p$  and  $q$  and the link between them are correct, then  $q$  eventually receives  $m$ .
- *Uniform Integrity*. For any message  $m$ ,  $q$  receives  $m$  at most once from  $p$ , and only if  $p$  previously sent  $m$  to  $q$ .
- E.g. Communication with TCP

CSC469



## Properties of Broadcast/Deliver

- *Validity*: If a correct process broadcasts a message  $m$ , then all correct processes eventually deliver  $m$ .
- *Agreement*: If a correct process delivers a message  $m$ , then all correct processes eventually deliver  $m$ .
- *Integrity*: For any message  $m$ , every correct process delivers  $m$  at most once, and only if  $m$  was previously broadcast by  $sender(m)$ .

CSC469



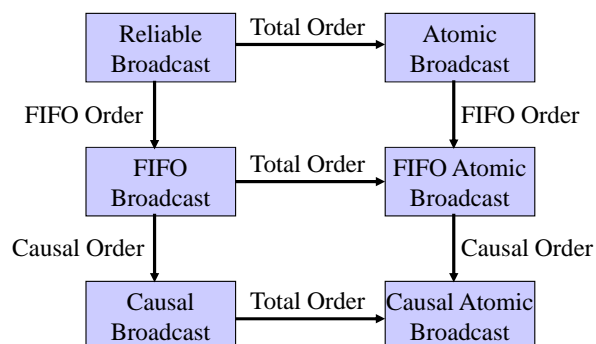
## Message Order

- *Unordered*: no guarantees on delivery order
- *FIFO Order*: If a process broadcasts a message  $m$  before it broadcasts a message  $m'$ , then no correct process delivers  $m'$  unless it has previously delivered  $m$ .
- *Causal Order*: If the broadcast of a message  $m$  causally precedes the broadcast of a message  $m'$ , then no correct process delivers  $m'$  unless it has previously delivered  $m$ .
- *Total Order*: All correct processes deliver messages in the same order
  - May be combined with any of the above delivery constraints

CSC469



## Broadcast Taxonomy



CSC469



## Reliable Broadcast Alg. (Diffusion)

Every process  $p$  executes:

```

//to reliably broadcast messages
ReliableBroadcast(m):
  //make m unique
  tag m with sender(m), sequence_number(m)
  send(m) to all neighbors including p

//event loop for receive events
upon receive(m) do
  if p has not previously executed ReliableDeliver(m)
  then
    if sender(m) ≠ p
    then
      send(m) to all neighbors
      ReliableDeliver(m)
  
```

CSC469



## "Diffusion" Algorithm Considered

- Works in synchronous or asynchronous system
- Assumes network does not partition
- Failures assumed to be fail-stop
- Floods the network
  - especially if processes are highly connected

CSC469



## FIFO Broadcast Algorithm

- FIFO Algorithm is layered on top of Reliable Broadcast
- Each process  $p$  maintains, for each other process  $q_i$ , the next sequence number it can **FIFO**Deliver
- Buffers **Reliable**Delivered messages until the sequence number indicates message may be **FIFO**Delivered

CSC469



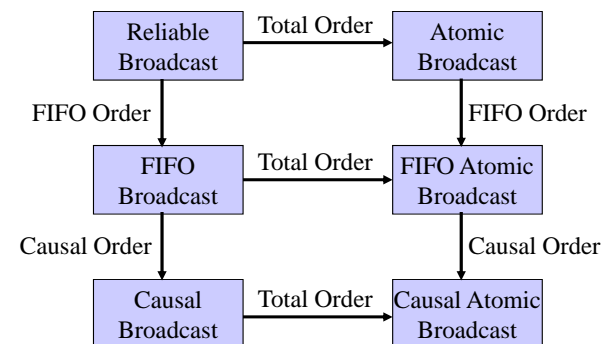
## Causal Broadcast Algorithm

- Causal algorithm is layered on top of FIFO alg.
- **CausalBroadcast** prepends list of messages upon which  $m$  causally depends then calls **FIFOBroadcast**
- Dependent messages is the list of messages **CausalDelivered** since last **CausalBroadcast**.
- Buffers **FIFO**Delivered messages until all messages upon which  $m$  depends have been **CausalDelivered**.

CSC469



## Broadcast Taxonomy



CSC469



## Atomic Broadcast

- Atomic Broadcast is form of Distributed Consensus
  - Therefore no deterministic, asynchronous algorithm
  - Synchronous algorithms for various failure models exist
- Other Atomic Broadcast algorithms can be built on top of Atomic Broadcast with similar limitations
  - FIFO Atomic Broadcast
  - Causal Atomic Broadcast

CSC469



## Schneider Tutorial on RSMs

- Not distinguished for clarity of assumptions/model of failure and synchrony
  - But better than any other paper as an introduction to RSMs
- Ties together:
  - Broadcast, consensus,
  - logical clocks, clock synchronization
  - leases, heart beats, failure detectors,
  - group membership (reconfiguration),
  - recovery (managing configuration)

CSC469



## Another Viewpoint/Approach

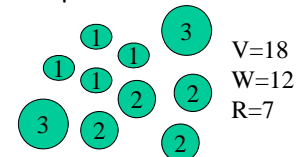
- Distributed Consensus
  - Servers communicate amongst themselves to reach agreement on state.
- Reliable Broadcast
  - Servers communicate amongst themselves to order messages
- What can clients do?
  - Clients can read and write to sets of servers in a consistent manner
  - Storing/restoring the state variables to servers & implementing a state machine locally is similar to RSMs

CSC469



## Voting

- Let  $V$  be the number of votes in the system
- Let  $W$  be the number of votes required to write
- Let  $R$  be the number of votes required to read
- Overlap Constraint:
  - $R + W > V$
- Recommend:
  - $2 \cdot W > V$
  - $R + W < V + \epsilon$
- **Data must contain a version number or timestamp**
- If constraints are met, then data will remain consistent.
- Note that votes can be arbitrarily assigned to servers in the system (I.e. weights can be assigned to servers)

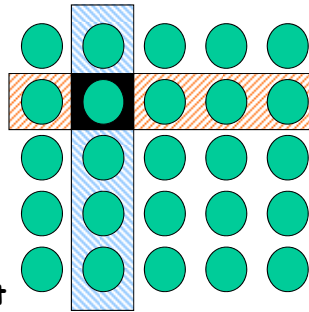


CSC469



## Quorums

- Quorums are a generalization of voting.
  - Organize servers into logical structures.
- Overlap constraint
  - Every write quorum must overlap with every read quorum
  - Example: writes must go to a column, reads must get a row.
- Note that voting **does not imply majority**



CSC469

