Lecture 10b: Fault Tolerance, Group Communication and Replicated State Machines

CSC469
Fall 2019

Thanks to Bogdan Simion, Sam Toueg and Vassos Hadzilacos
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.
Masking/hiding faults

• Obvious requirement: redundancy
  • Must be able to repair broken sets of bits
    • e.g., error correction codes
  • Must be able to handle transient faults
    • e.g., resend message, retry disk operations
  • 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 $\rightarrow$ distributed consensus

Information redundancy
Time redundancy
Physical redundancy
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
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
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
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 (1 correct replica sufficient)
    • Byzantine: $2t + 1$ replicas required ($t + 1$ correct replicas sufficient)

• Different than Broadcast/Consensus context. Why?
  • One client must decide on result, replicas don’t have to agree with each other about result
RSMs, Consensus & Reliable Broadcast

• Each correct replica:
  • Must receive every request
    • “Agreement” requirement
  • Must execute same commands in same order
    • “Order” requirement
  • Since all correct replicas must have the same state!
  • Therefore, RSMs require Distributed Consensus to agree on the order of commands

• Needs form of group communication called atomic broadcast
Group communication

• In many applications processes must be able to reliably broadcast messages, so that they agree on the set of messages they deliver.
• Difficulty: distributed processes do not know each other’s state.
• Much of this material is based on Chapter 5 by Hadzilacos and Toueg in “Distributed Systems”, Sape Mullender, ed.
  • Reliable broadcast taxonomy
  • Example broadcast algorithms
Application / Broadcast Mechanism

\[ p \]
Application Protocol
\[ \text{broadcast}(m) \]
Broadcast Algorithm
\[ \text{send}(m) \]

\[ q \]
Application Protocol
\[ \text{deliver}(m) \]
Broadcast Algorithm
\[ \text{receive}(m) \]

Broadcast / Delivery Interface
Send / Receive Interface

Communications Network
Properties of send and 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

Liveness

Safety
Properties of broadcast/deliver

**Reliable Broadcast** satisfies the following properties:

- **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 $\text{sender}(m)$.

[Notes: Liveness and Safety]
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
  - Basically, every process sees messages in the exact same order
  - Total Order does not imply either FIFO or Causal, just the same order for everyone!
  - May be combined with any of the above delivery constraints (No Order, FIFO or Causal)
Broadcast Taxonomy

- Reliable Broadcast
- FIFO Broadcast
- Causal Broadcast
- Atomic Broadcast
- FIFO Atomic Broadcast
- Causal Atomic Broadcast

Order relations:
- Total Order
- FIFO Order
- Causal Order

University of Toronto, Department of Computer Science
Every process $p$ executes:

```plaintext
//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 executed ReliableDeliver($m$) then
        if sender($m$) $\neq$ $p$
            then
                send($m$) to all neighbors
                ReliableDeliver($m$)
```

This is a reliable broadcast algorithm (Diffusion) implemented in a distributed system.
• All correct processes take on role of broadcaster upon receipt of message

P0 starts ReliableBroadcast(m) but fails after sending to P1 and P3.
Diffusion Algorithm Illustrated

- All correct processes take on role of broadcaster upon receipt of message

On receipt of m, P1 and P3 resend to all neighbors. P2 gets 2 copies of m but resends only once.
“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
FIFO Broadcast Algorithm

• FIFO Algorithm is layered on top of Reliable Broadcast
• Each process $p$ sends a message $m$ to its neighbors, and tags it with $p$’s sender# and a sequence#.
• Each process $p$ maintains, for each other process $q_i$, the next sequence number it can FIFODeliver
• Buffers ReliableDeliver’ed messages until the sequence number indicates message may be FIFODeliver’ed
Causal Broadcast Algorithm

- Causal algorithm is layered on top of FIFO alg.
- A `CausalBroadcast` prepends the list of messages upon which $m$ causally depends, then calls `FIFOBroadcast`
- Dependent messages are the list of messages `CausalDeliver`ed since last `CausalBroadcast`.
- Buffers `FIFODeliver`ed messages until all messages upon which $m$ depends have been `CausalDeliver`ed.
Atomic Broadcast

• How do we enforce total ordering?
  • FIFO Broadcast: per-process sequence numbers
  • Causal Broadcast: dependencies sent with m
  • What happens if all (even unrelated) messages should be seen in the same order?

⇒ Atomic Broadcast is a 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
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)
Another Viewpoint/Approach

So far:

• 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
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 (Requirement):
  1. $V < R + W$
- Recommended:
  2. $V < 2 \times W$ (i.e., $W > V/2$)
- Data must contain a version number or timestamp
- If version numbers used $\rightarrow$ 2. becomes a requirement!
- 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)

D. Gifford, “Weighted Voting for Replicated Data” SOSP 1979
Example with version numbers

• \( R+W > V \) and \( W > V/2 \)
  • Read and Write quorums overlap
  • Write more than \( \frac{1}{2} \).

• To read:
  • Request version numbers until \( R \) votes are collected
  • Overlap constraint means at least 1 voter has latest version. Read it.
Example with version numbers

To write:

- Request version numbers until \( W \) votes are collected from servers with up-to-date copies.
  - May require latest write to be propagated to more servers.
  - Any set of voters with \( W \) votes must include at least 1 with latest version. Writer can detect if it needs more votes.
- Apply update to servers in write quorum.

Two concurrent writers cannot both get write quorum, since \( W > \frac{1}{2} V \).
Version numbers with $W \leq V/2$

To write:
- Request version numbers until $W$ votes are collected from servers with up-to-date copies
- Apply update to servers in write quorum
- 2 concurrent writers can both succeed in getting a write quorum
  - Suppose current version is “2”
    - P1: write(value=x, version=3)
    - P2: write(value=y, version=3)
- Reader can’t distinguish latest write
To write:

- Collect $W$ votes from servers to build write quorum (need not be up-to-date)
- Apply update to servers in write quorum
- 2 concurrent writers can both succeed in getting a write quorum
  - $P_1$: write(value=$x$, timestamp=$T_1$)
  - $P_2$: write(value=$y$, timestamp=$T_2$)
- Read quorum must include at least one server with later timestamp

May need some way to break ties, e.g., using process id.
Selecting size of R and W

- Consider the case of 3 replicas
- Three possible choices
  - $R=3$, $W=1$ (Read all, write any)
    - Fast writes at expense of reads
    - Single failure may lose most recent data
  - $R=1$, $W=3$ (Read 1, write all)
    - Fast reads at expense of writes
    - Single failure makes it impossible to write new data
  - $R=2$, $W=2$
    - Good tradeoff