Implementing Fault-Tolerant Services
Using the State Machine Approach

Prof. Phillip Gibbons

Spring 2020, Lecture 13
Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial

Fred B. Schneider 1990

- Fred Schneider (Cornell)
  - NAE, AAAS, AAAS Fellow, ACM Fellow, IEEE Fellow
  - IEEE Emanuel R. Piore Award (other winners: Randy Bryant, Allen Newell, Thompson/Ritchie, Hamming, Lamport)
  - The Jean-Claude Laprie Award (other winners: Lamport, Patterson/Gibson/Katz)
  - Edsger W. Dijkstra Prize (other winners: Chandy, Lamport x3)
SigOps HoF citation (2007):
The paper that explained how we should think about replication...a model that turns out to underlie Paxos, Virtual Synchrony, Byzantine replication, and even Transactional 1-Copy Serializability.
State Machines

• Requests are processed by a state machine one at a time, in an order consistent with potential causality:
  
  O1: Processes requests by a single client in order issued
  
  O2: If request r by client c could have caused a request r’ to be made by client c’, then processes r before r’

• **Semantic Characterization**: State machine outputs are determined solely by the sequence of requests processed
  – Independent of time, other system activity

```plaintext
memory: state_machine
  var store: array[0..n] of word

read: command(loc: 0..n)
  send store[loc] to client
end read;

write: command(loc: 0..n, value: word)
  store[loc] := value
end write
end memory
```
Replicas & Coordination

- Tolerating $t$ Byzantine Failures requires $2t+1$ replicas

- Tolerating $t$ Fail-stop Failures requires $t+1$ replicas

- Replica Coordination: All replicas receive & process the same sequence of requests
  - Every nonfaulty state machine replica:
    - (agreement) receives every request
    - (order) processes the requests in the same relative order

- Relaxations:
  - (agreement) For fail-stop: read-only requests can be sent to only 1 non-faulty replica
  - (order) can be relaxed for requests that commute
Implementing Agreement

• Agreement: Every nonfaulty replica receives every request

• Designated “transmitter” processor disseminates a value to other processors such that:
  – All nonfaulty processors agree on the same value
  – If the transmitter is nonfaulty, then all nonfaulty processors use its value as the one on which they agree

• Challenge: Coping with a transmitter that fails part way through execution
Implementing Order

• Order: Every nonfaulty replica processes the requests in the same relative order

• Assign unique identifiers to requests & process in order
  – Order on IDs must conform to O1 and O2

• A request is stable at a replica once no lower-ID request from a correct client can be delivered to the replica

• Among its delivered but unprocessed requests, a replica processes the lowest-ID stable request next

• IDs can be based on Logical clocks, Synchronized real-time clocks, or Replica-generated identifiers
Using Lamport/Logical Clocks

• Logical Clock order on IDs conforms to O1 & O2

• Asynchronous setting with unbounded delays on processes/messages

• Assume Fail-stop failures & p can detect failure of q only after p has received the last message sent to p by q

• Stability Test: Every client makes periodic request to the SM; Request is stable at replica $sm_i$ if
  a request with larger timestamp has been received by $sm_i$ from every client running on a nonfaulty processor
  – Implies no request with smaller timestamp can be received from a client, faulty or not
Using Synchronized Real-Time Clocks

- **UID =** local real-time clock appended with processor id

- **Satisfy O1 provided:**
  - No client makes > 1 request between successive local clock ticks

- **Satisfy O2 provided:**
  - Degree of clock sync is better than min. message delivery time

- Define \( \Delta \) s.t. each request \( r \) gets received by each correct processor \( p \) no later than \( \text{UID}(r) + \Delta \), according to \( p \)'s clock

- **Stability Test:** \( r \) is **stable** if \( \text{UID}(r) < \tau - \Delta \), where \( \tau \) is local clock
  - \( r \) is stable if a request with \( \text{UID} > \text{UID}(r) \) has been received by every client
Using Replica-Generated Identifiers

• State machine replicas propose candidate UID for a request & then one is selected (as part of Agreement protocol)
  – CandUID(sm_i, r) ≤ UID(r)
  – if see r’ after accept r, then UID(r) < CandUID(sm_i, r’)

• Stability Test: accepted request r is stable provided
  – No r’ has (i) been seen by sm_i & (ii) not been accepted by sm_i
    s.t. CandUID(sm_i, r’) ≤ UID(r)

• Advantage?
  – Not all processors need to communicate

• How to satisfy O1 & O2?
  – Client waits for its request to be accepted by all non-faulty replicas before starting its next request
Discussion: Summary Question #1

- **State the 3 most important things the paper says.** These could be some combination of their motivations, observations, interesting parts of the design, or clever parts of their implementation.
Tolerating Faulty Output Devices

• Outputs used outside the system
  – Use 2t+1 devices for t-resilience to Byzantine failures

• Outputs used inside the system
  – Client waits until receives t+1 identical responses
  – If client on same processor as replica, then just ask replica
Tolerating Faulty Clients

• Replicate the client

• Defensive programming:

```plaintext
release:
   command
   if user ≠ client → skip
   □ waiting = Φ ∧ user = client →
      user := Φ
   □ waiting ≠ Φ ∧ user = client →
      send OK to head(waiting);
      user := head(waiting);
      waiting := tail(waiting)
   fi
end release

acquire:
   command
   if user = Φ →
      send OK to client;
      time_granted := TIME;
      schedule
      (mutex.timeout, time_granted)
   for + B
   □ user ≠ Φ → waiting := waiting ∘ client
   fi
end acquire

timeout:
   command (when_granted : integer)
   if when_granted ≠
      time_granted → skip
   □ waiting = Φ ∧ when_granted =
      time_granted → user := Φ
   □ waiting ≠ Φ ∧ when_granted =
      time_granted →
      send OK to head(waiting);
      user := head(waiting);
      time_granted := TIME;
      waiting := tail(waiting)
   fi
end timeout
```
• Describe the paper's single most glaring deficiency. Every paper has some fault. Perhaps an experiment was poorly designed or the main idea had a narrow scope or applicability.
Using Time to Make Requests

• After predetermined amount of time passes, assume “default” value is sent
Reconfiguration

- Remove faulty replicas & Add working replicas

- \( P(\tau) = \) number of processors, of which \( F(\tau) \) are faulty

- Combining Condition: \( P(\tau) - F(\tau) > Enuf \) for all \( \tau \geq 0 \)
  - \( Enuf = P(\tau)/2 \) with Byzantine, or 0 with only Fail-stop

- Removing faulty processors can improve system performance (reduces agreement costs)

- Can have configurator for each client / replica / device
  - For fail-stop, check failure-detection element
  - For Byzantine, not always possible to detect failures, but can try by comparing across peers and history
Integrating a Repaired Client

• Need to have correct state when added

• For Client to join immediately after request r*, send it the state after r* & before sending any output by requests with UID larger than UID(r*)
  – If self-stabilizing (from k previous inputs) then instead just run the client on k inputs prior to r*
  – With Byzantine, client awaits t+1 identical copies of state
Integrating a Repaired SM Replica

• Send values of state variables & copies of pending requests

• $sm_i$ also may need to forward requests to $sm_{new}$

• Fail-stop + Logical Clocks
  – $sm_i$ relays requests $r$ from each client $c$ where $UID(r) < UID(r_c)$, where $r_c$ is first direct request from $c$ to $sm_{new}$

• Fail-stop + Real-time Clocks
  – $sm_i$ relays requests received within next duration $\Delta$

• Byzantine: await $t+1$ identical copies

• Stability Test during Restart: must await all relayed requests before processing any direct requests
Discussion: Summary Question #3

• Describe what conclusion you draw from the paper as to how to build systems in the future. Most of the assigned papers are significant to the systems community and have had some lasting impact on the area.
Friday’s Paper

Fault Tolerance (II)

“Paxos Made Simple”
Leslie Lamport 2001