Implementing Fault-Tolerant Services using the State Machine Approach: A Tutorial

Phil Gibbons
15-712 F15
Lecture 21

Today’s Reminders

• Interim Project Reports on Friday
  - Summary of the motivation for the project, and how it
    compares with related work. Has anything changed based
    upon feedback from your proposal or as a result of your
    research so far?
  - A re-iteration of your proposed goals, what progress you have
    made to date on those goals, and what your timeline is for
    accomplishing the rest of them by the end of the semester.
  - A list of the major components of your evaluation plan.
    What’s been completed and what did you learn?
  - What are the current concerns with this project and its
    progress? Any stumbling blocks discovered?

• Only 3 summaries left ☺

Implementing Fault-Tolerant Services using the State Machine Approach: A Tutorial
[ACM Computing Surveys, 1990]

• Fred Schneider (Cornell)
  - AAAS Fellow, ACM Fellow, IEEE Fellow, NAE
  - IEEE Emanuel R. Piore Award
    (other winners: Randy Bryant,
     Allen Newell, Thompson/Ritchie,
     Lamport, Hamming)

“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.”
– SigOps HoF citation

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

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, and
    (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 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 the minimum message delivery time
- Define $\Delta$ s.t. each request $r$ gets received by every correct processor no later than $\text{UID}(r) + \Delta$
- **Stability Test:** $r$ is stable if local clock is $\Delta$ & $\text{UID}(r) < r - \Delta$
  - $r$ 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 UIDs for a request & then one is selected
  - $\text{CandUID}(s_{mi}, r) \leq \text{UID}(r)$
  - if see $r'$ after accept $r$, then $\text{UID}(r) < \text{CandUID}(s_{mi}, r')$
- **Stability Test:** accepted request $r$ is stable provided
  - No $r'$ has (i) been seen by $s_{mi}$ & (ii) not been accepted by $s_{mi}$
  - $\text{CandUID}(s_{mi}, r) \leq \text{UID}(r)$

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 from $2t+1$
  - If client on same processor as replica, then just ask replica

Tolerating Faulty Clients

- **Replicate the client**
- **Defensive programming**

```plaintext
acquire:
  command
  if user != $\Phi$
    send OK to client;
    time_granted := TIME;
    schedule
    (makes.timeout, time_granted)
    for $i$
      user := $\Phi$
      waiting := $\Phi$
      if when_granted := $\Phi$
        when_granted := integer
        if when_granted $\neq$
          time_granted := $\Phi$
          send OK to head(waiting);
          user := head(waiting);
          waiting := tail(waiting)
      fi
  end acquire
```

```plaintext
release:
  command
  if user != $\Phi$
    make.timeout, time_granted
    if when_granted $\neq$
      time_granted := $\Phi$
      send OK to head(waiting);
      user := head(waiting);
      time_granted := TIME;
      waiting := tail(waiting)
    fi
  end release
```
Using Time to Make Requests

• After predetermined amount of time passes, assume "default" value is sent

Reconfiguration

• Remove faulty replicas & Add working replicas

  • $P(t) = $ number of processors, of which $F(t)$ are faulty

  • Combining Condition: $P(t) - F(t) > \text{Enuf}$ for all $t \geq 0$
    - $\text{Enuf} = P(t)/2$ 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
Wednesday’s Class

Paxos Made Simple
Leslie Lamport

[Sigact News, 2001]