Distributed Systems

08. State Machine Replication & Virtual Synchrony

Paul Krzyzanowski
Rutgers University
Fall 2017
Virtual Synchrony –
State Machine Replication
State machine replication

- We want high scalability and high availability
  - Achieve via redundancy
- High availability means replicated functioning components will take place of ones that stop working
  - Active-passive: replicated components are standing by
  - Active-active: replicated components are working
- Model system as a sequence of states
  - Input to a specific state produces deterministic output and a transition to a new state
    - “State”: replicated data or replicated computing
  - To ensure correct execution & high availability
    - Each process must see & process the same inputs in the same sequence
    - Obtain consensus at each state transition
State machine replication

• Replicas = group of machines = **process group**
  – Load balancing (queries can go to any replica)
  – Fault tolerance (OK if some dies; they all do the same thing)

• Important for replicas to remain consistent
  – Need to receive the same messages [usually] in the same order

• What if one of the replicas dies?
  – Then it does not get updates
  – When it comes up, it will be in a state prior to the updates
    • *Not good – getting new updates will put it in an inconsistent state*
Faults

• Faults may be
  – Fail-silent (fail-stop)
  – Byzantine (corrupted data)

• synchronous system vs. asynchronous system
  – Synchronous = system responds to a message in a bounded time
  – E.g., IP packet versus serial port transmission
  – We assume we have an asynchronous system
Agreement in faulty systems

Two army problem
  – Good processors - faulty communication lines
  – Coordinated attack
  – Infinite acknowledgement problem
Agreement in faulty systems

Byzantine Generals problem
– reliable communication lines - faulty processors
– $n$ generals head different divisions
– $m$ generals are traitors and are trying to prevent others from reaching agreement
  • 4 generals agree to attack
  • 4 generals agree to retreat
  • 1 traitor tells the 1st group that he’ll attack and tells the 2nd group that he’ll retreat
– can the loyal generals reach agreement?
Agreement in faulty systems

Byzantine Generals problem

– Solutions require:
  • $\geq(3m+1)$ participants for $m$ traitors ($2m+1$ loyal generals)
  • $m+1$ rounds of message exchanges
  • $O(m^2)$ messages
– Costly solution!

• Variation: use signed messages
  – Messages from loyal generals cannot be forged/altered
  – Traitors can still lie
  – Consensus can be achieved with $\geq(m+2)$ loyal generals
Agreement in faulty systems

• It is impossible to achieve consensus with asynchronous faulty processes
  – There is no way to check whether a process failed or is alive but not communicating (or communicating quickly enough)

• We have to live with this

• We cannot reliably detect a failed process

• **But** we can propagate our knowledge that we think it failed
  – *Take it out of the group*
Group View

- Set of processes currently in the group
- A multicast message is associated with a *group view*
- Every process in the group should have the same view
- View change
  - When a process joins or leaves the group, the group view changes
  - View change
    - Multicast message announcing the joining or leaving of a process
Virtual Synchrony

• What if a message is being multicast during a view change?
  – Two multicast messages in transit at the same time:
    • view change ($vc$)
    • message ($m$)

• Need to guarantee
  – $m$ is delivered to all processes in G before any process is delivered $vc$
  – OR $m$ is not delivered to any process in G

• Reliable multicasts with this property are virtually synchronous
  – All multicasts must take place between view changes
  – A view change is a barrier
View Changes & Virtual Synchrony

$G = \{ p \}$
$G = \{ p, q \}$
$G = \{ p, q, r, s, t \}$
$G = \{ r, s, t \}$

Time

0 10 20 30 40 50 60 70

October 2, 2017

© 2014-2017 Paul Krzyzanowski
Virtual Synchrony: implementation example

• ISIS: fault-tolerant distributed system offering virtual synchrony
  – Achieves high update & membership event rates
  – Hundreds of thousands of events/second on commodity hardware as of 2009

• Virtual synchrony
  – Provides distributed consistency
  – Applications can create & join groups & send multicasts
  – Applications will see the same events in an equivalent order
  – Group members can update group state in a consistent, fault-tolerant manner

• Who uses it?
  – ISIS: Microsoft’s scalable cluster service, IBM’s DCS system, CORBA
  – Similar models: Apache Zookeeper (configuration, synchronization, and naming service)
Implementation: Goals

- Message transmission is asynchronous
  - Machines may receive messages in different orders

- Virtual synchrony
  - Preserve the illusion that events happen in the same order
  - Uses TCP → reliable point-to-point message delivery
  - Multicasting is implemented by sending a message to each group member
  - No guarantee that ALL group members receive the message
    - The sender may fail before transmission ends
Implementation: Group Management

- **Group Membership Service (GMS)**
  - Failure detection service
  - If a process $p$ reports a process $q$ as faulty
    - GMS reports this to every process with a connection to $q$
    - $q$ is taken out of the process group and would need to re-join
  - Imposes a consistent picture on membership
When a new member joins a group
   – It will need to import the current state of the group
   – **State transfer:**
     • Contact an existing member to request a state transfer
     • Initialize the replica to that checkpoint state
     • A state transfer is treated as an instantaneous event

Problem
   – Guarantee that all messages sent to view $G_i$ are delivered to all non-faulty processes in $G_i$ before the next view change ($G_{i+1}$)
Implementation: Receiving all messages

• Make sure each process in $G_i$ has received all messages that were sent to $G_i$
  – A sender may have failed
    → there may be processes that will not receive a message $m$
  – These processes should get $m$ from somewhere else

• Let every process hold $m$ until it knows that all members of $G_i$ received it
  – Once all members received it, $m$ is stable
  – Only stable messages can get delivered to applications
  – Select an arbitrary process in $G_i$ and request it to send $m$ to all other processes
    • Delivery within the group is reliable, so this ensures that the message is stable
View change: $G_i \rightarrow G_{i+1}$

• Some process $P$ receives a view change message
  – It detected a failure or received a request from a process wanting to join or leave the group
  – $P$ forwards a copy of any unstable messages to every process in $G_{i+1}$
  – It then marks the message as stable
  – $P$ indicates it no longer has any unstable messages
  – It is ready to transition to view $G_{i+1}$ as soon as other processes are ready
  – $P$ multicasts a flush message for $G_{i+1}$
  – Waits to receive a flush message for $G_{i+1}$ from every other process
  – Then switches to the new view $G_{i+1}$
View change: $G_i \rightarrow G_{i+1}$

- Some process $Q$, still operating in view $G_i$, receives a message $m$
  - If it has already received message $m$, it discards it as a duplicate
  - Delivers $m$ (using message ordering constraints as necessary)

- When $Q$ receives a view change message, it will
  - Forward any of its unstable messages to the group
  - Send a **flush** message

  - $P$ indicates it no longer has any unstable messages
  - It is ready to transition to view $G_{i+1}$ as soon as other processes are ready
  - $P$ multicasts a **flush** message for $G_{i+1}$
  - Waits to receive a **flush** message for $G_{i+1}$ from every other process
  - Then switches to the new view $G_{i+1}$
View change summary

• Every process will
  – Send any unstable messages to all group members
  – Process received messages that are not duplicates
  – Send a flush message to the group
  – Receive a flush message from the entire group
The end