Distributed Systems

08. State Machine Replication & Virtual Synchrony

Paul Krzyzanowski
Rutgers University
Spring 2020
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

• We can model a system as a sequence of states
  – Input to a specific state produces deterministic output and a transition to a new state
    • “State” represents replicated data storage or replicated computing operations
  – 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 will not get updates
  – When it comes up, it will be in a state prior to the updates
    = **stale state**
    • Not good – getting new updates will put it in an inconsistent state
Faults

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

• Our network may be an asynchronous system vs. a synchronous system
  
  *Synchronous* = system responds to a message in a bounded time
  *Asynchronous* = a system that doesn’t

  – E.g., IP packet versus serial port transmission
  – We assume we have an asynchronous system
Agreement in faulty systems

Two armies problem

– Good processors
– Asynchronous & unreliable communication lines
– Coordinated attack
– Infinite acknowledgement problem
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 must 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 the $vc$
  – OR else $m$ 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

\[ G = \{ p \} \]
\[ G = \{ p, q \} \]
\[ G = \{ p, q, r, s, t \} \]
\[ G = \{ r, s, t \} \]
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
- 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)
Goals

Message transmission is asynchronous
• Machines may receive messages in different order

Virtual synchrony
• Preserve the *illusion* that events happen in the same order
  – Use a hold back queue & deliver messages to the application in a consistent order
Group Management

Group Membership Service (GMS)

• Failure detection service
• Keeps track of the definitive list of who’s in each group
  – 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 of group membership for everyone
Sending & receiving messages

• Sending
  – Uses TCP → reliable point-to-point message delivery
  – Multicasting is implemented by sending a message to each group member

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

• Receiving: hold-back & delivery
  – Every process that receives a message $m$ holds it until it knows that all members of $G_i$ received it
  – Every process that receives a message sends an acknowledgement to the sender
  – When the sender receives all acknowledgements, $m$ is stable
  – Only stable messages can get delivered to applications
    • Optimization: receivers can acknowledge groups of messages; senders can confirm groups of stable messages
Sender failure

• A sender may die before all messages are sent (or acknowledged)
  – These messages are **unstable** and remain in the hold-back queue at each receiver that got the message

• When the death of the sending process is detected, the GMS issues a **view change** and removes the process from the group
  – During this view change:
    • All unstable messages need to be sent to all remaining group members
    • … and delivered to the applications (since they will now be stable)

• This enforces the **atomic multicasting** property (all or none)
Joining a group & state transfer

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 the latest state from its last checkpointed state  
  – A state transfer is treated as an instantaneous event  
    • No other processing takes place until this is complete

• **Message delivery during a view change**  
  – Guarantee that all messages sent to view $G_i$ are delivered to all non-faulty processes in $G_i$ before the next view change ($G_{i+1}$)
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 each of these messages 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: $G_i \rightarrow G_{i+1}$

Receiving a view change

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
– Multicast a $\text{flush}$ message for $G_{i+1}$
– Waits to receive a $\text{flush}$ message for $G_{i+1}$ from every other process
– Then switches to the new view $G_{i+1}$

$\text{flush} = \text{barrier}$
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