Distributed Systems

09. State Machine Replication & Virtual Synchrony

Paul Krzyzanowski
Rutgers University
Fall 2016

 1

State machine replication

 2

State machine replication

- We want high scalability and high availability
  – Achieve this via redundancy
- Replicated components will take place of ones that stop working
  – Active-passive: replicated components are standing by
  – Active-active: replicated components are working
- Replicated state machine
  – State machine = program that takes inputs & produces outputs & holds internal state (data)
  – Replicated = run concurrently on several machines
  – If all replicas get the same set of inputs in the same order, they will perform the same computation and produce the same results
  – To ensure correct execution & high availability
    - Each process must see & process the same inputs in the same sequence
    - Obtain consensus at each state transition

 3

State machine replication

- Replicas = group of machines = **process group**
  – Load balancing (queries can go to any replica)
  – Fault tolerance (OK if some die; they all do the same thing)
- Important for replicas to remain **consistent**
  – Need to receive the same messages [usually] in the same order (causally related messages)
- 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*

 4

Faults

- Faults may be
  – **fail-silent**: the system does not communicate
    - fail-stop: a fail-silent system that remains silent
    - fail-recover: a fail-silent system that comes back online
  – **Byzantine**: the system communicates with bad data
- synchronous system vs. asynchronous system
  – Synchronous = system responds to a message in a bounded time
  – Asynchronous = no assurance of when a message arrives
  – E.g., IP packet versus serial port transmission
    - IP network = asynchronous
- In a distributed system, we assume processes are:
  – **Concurrent, asynchronous, failure-prone**

 5

Agreement in faulty systems

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

 6

Agreement in faulty systems

It is impossible to achieve consensus with asynchronous faulty processes

– There is no foolproof 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
  – Moreover, the the system might recover
- *But* we can propagate knowledge that we *think* it failed
  – *Take it out of the group (even if it is alive)*
  – If it recovers, it will have to re-join

Virtual Synchrony

Virtual Synchrony is a software model

Model for group management and group communication

– A process can join or leave a group
– A process can send a message to a group
  - Message ordering requirements defined by programmer

Atomic multicast

"A message is either delivered to *all* processes in the group or to *none*"

Group View

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 group view
- When a process joins or leaves the group, the *group view* changes

View change

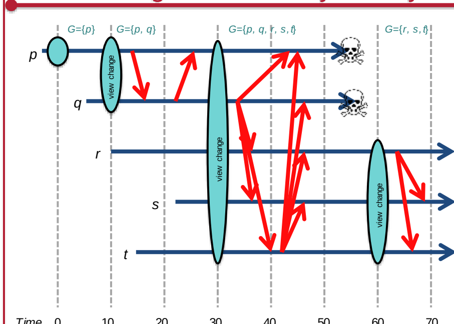
- View change =
  Multicast message announcing the joining or leaving of a process
- Timeouts lead to failure detection
  – Group membership change ⇒ the dead member is removed from the group

Events

Group members receive three types of events

1. New message received
2. View change: group membership change
3. Checkpoint request
   - Dump the state of your system so a new process can read it

View Changes & Virtual Synchrony



A view change is a barrier

- 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 "all or nothing" semantics
  - *m* is delivered to all processes in G before any process is delivered a *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

recall the distinction between **receiving** a message and **delivering** it to the application

Virtual Synchrony: implementation example

- ISIS toolkit: 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?
  - New York Stock Exchange, Swiss Exchange, US NAVY AEGIS, etc.
  - Similar models:
    - Microsoft's scalable cluster service, IBM's DCS system, CORBA
    - Apache Zookeeper (configuration, synchronization, and naming service)

Implementation: Goals

- Message transmission is asynchronous (e.g., IP)
  - Machines may receive messages in different order
- 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 will need to re-join
- Imposes a consistent view of membership to all members

Implementation: State Transfer

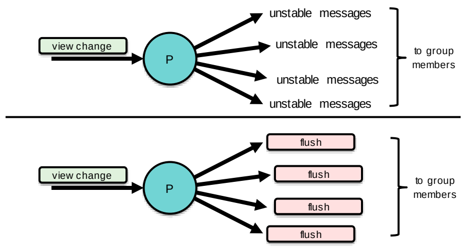
- When a new member joins a group
  - It will need to import the current state of the group
  - **State transfer**:
    - Contact an existing member to request a state transfer – *checkpoint request*
    - Initialize the new member (replica) to that checkpoint state
- Important – enforce the group view barrier
  - A state transfer is treated as an instantaneous event
  - 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})

Ensuring all messages are received

- All messages sent to G_i must be delivered to all non-faulty processes before a view change to G_{i+1}
- But what if the sender failed?
  - Each process stores a message until it know all members received it
  - At that time, the message is **stable**

View Change

Stable message = received (acknowledged) by all group members
Every process holds a message until it knows that it has been received by the group



View change complete when each process receives a *flush* message from every other process in the group

View change summary

- Every process will
  - Send any unstable messages to all group members
    - Wait for acknowledgements
  - Deliver any received messages that are not duplicates
  - Send a ***flush*** message to the group
  - Receive a ***flush*** message from every member of the group

- Benefits
  - No need for a single master that propagates its updates to replicas
  - Not transactional – not limited to one-at-a-time processing
  - Message ordering is generally causal within a view – more efficient than imposing total ordering

The End

