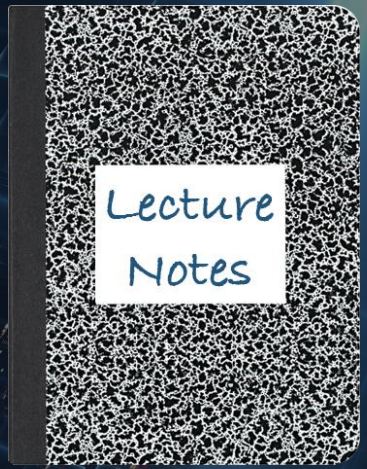
CS 417 – DISTRIBUTED SYSTEMS

Week 4: Part 3

Virtual Synchrony



Paul Krzyzanowski

Replication

We want scalability and high availability in distributed systems

- High availability

  Components can break – replicated functioning components will take place of ones that stop working

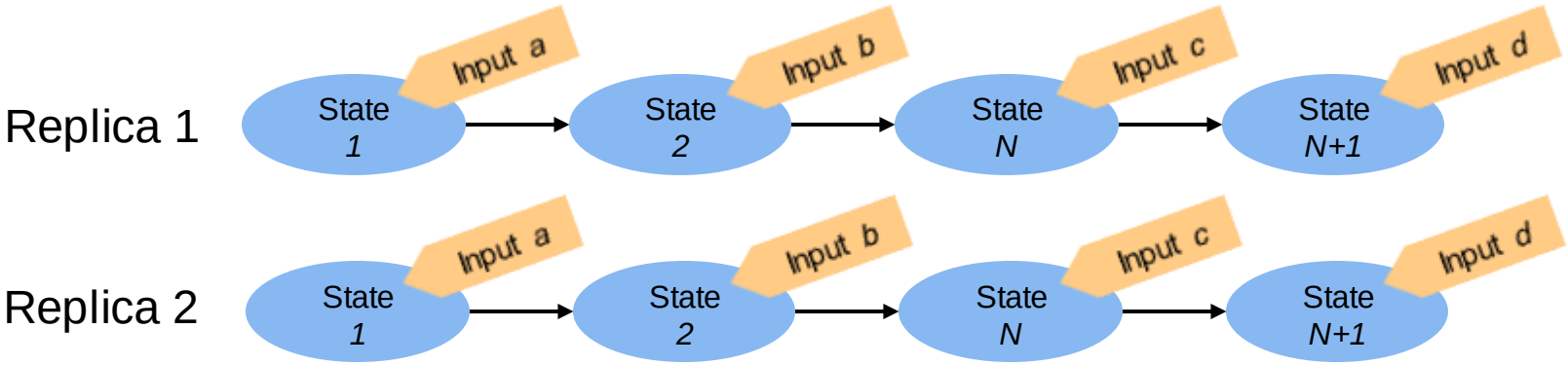
- Scalability

  In many cases, we achieve this by load balancing requests among replicated services

State machine replication

We can model a system as a sequence of states

- An input produces deterministic output and a transition to a new state
  - "State" represents data storage or computing operations that we want to replicate
- To ensure correct execution & high availability
  - Each process must see & process the same inputs in the same sequence



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 stay 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 & Delays

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

- Network transmission may be asynchronous vs. synchronous

  Synchronous = system responds to a message in a bounded time

  Asynchronous = a system that doesn't
  - E.g., IP packet versus serial port transmission

With Internet communication, we assume

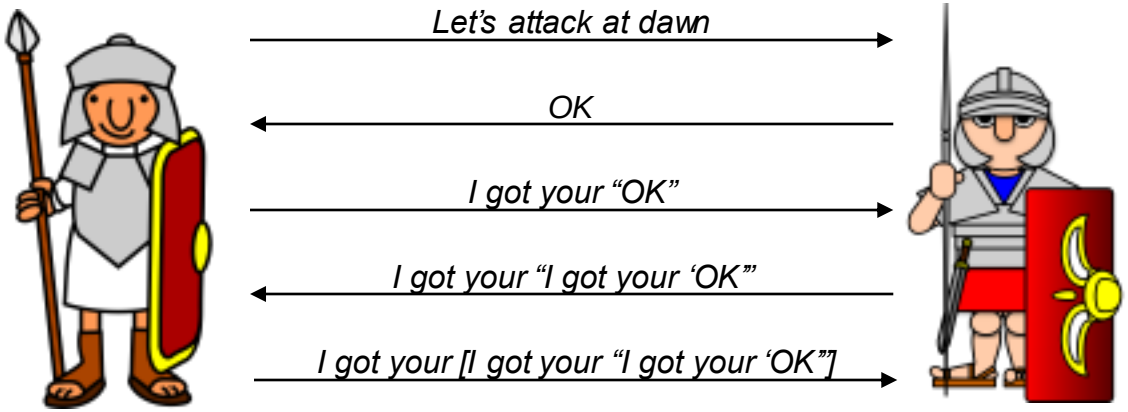
- An asynchronous network
- Non-Byzantine faults
  (assume integrity checks and, if needed, cryptographic authentication & integrity checks)

Agreement in faulty systems

Two armies problem*

- Good processors
- Asynchronous & unreliable communication lines
- Challenge: agree on an attack – be certain both sides have the message

⇒ *Infinite acknowledgment problem*



*Also called the Two Generals Problem

Agreement in faulty systems

The two armies problem demonstrates it is impossible to achieve consensus with asynchronous unreliable communication lines

– There is no way to check whether a process failed or is alive but the communications failed (or it's not 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*

Virtual Synchrony: Definition

A **virtually synchronous system** is one where

1. Even though messages are sent asynchronously, they appear to be processed in a synchronous manner.
2. The system provides guarantees about the delivery order of messages.
3. Nodes in the system have a consistent view of the group membership (i.e., which nodes are part of the system).

Even though the underlying communication between nodes is asynchronous (messages can be delayed, reordered, or lost), the system provides an abstraction layer that gives the illusion of reliable, synchronous operation.

Virtual Synchrony

Virtual Synchrony – implements atomic multicast

A message must be delivered to *every* group member

… even if the sender dies

Programmers have the abstraction that message delivery is synchronous & atomic

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 knowledge of the group 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 to a group G during a view change?

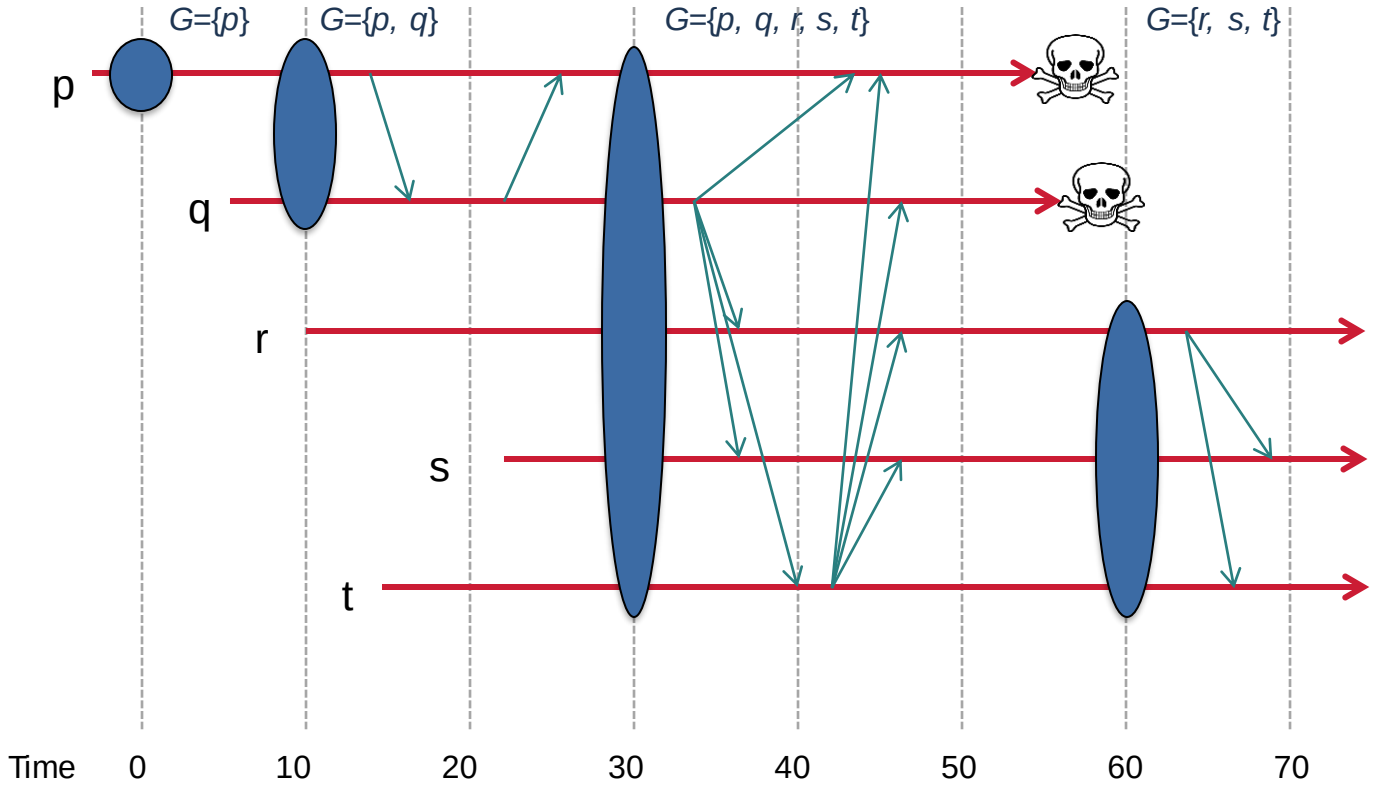
- Two multicast messages in transit at the same time:
  - view change (***vc***)
  - message (***m***)

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

- Need to guarantee
  - ***m*** is delivered to *all* processes in ***G*** before any process is delivered the ***vc***
  - OR ***m*** is delivered *all* processes in ***G*** after every 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



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
- Applications can create & join groups & send multicasts
  - Provides distributed consistency in the event of failure
  - 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?

- Microsoft's Scalable Cluster Service, IBM's DCS system, CORBA
- Similar models influenced by virtual synchrony:
  Apache Zookeeper (configuration, synchronization, and naming service)

Goals

Assume message transmission is asynchronous

- Machines may receive messages in different sequences

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
  - ***p*** tells the GMS
  - GMS reports this to every process with ***q*** in its view
  - ***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
  - Multicasting is implemented by sending a UDP message to each group member or IP multicast
  - TCP not used because it doesn't send acknowledgments to the user process
  - Isis handles its own acknowledgments

- 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 acknowledgment to the sender
  - When the sender receives all acknowledgments, *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
  - **View change**
    - All unstable messages must be sent to all remaining group members
      … and then delivered to the applications (since they will now be stable)

- This enforces the **atomic multicasting** property (all or none)

View change: $G_i \rightarrow G_{i+1}$

Some process *P* receives a ***view change*** message

A possible failure was detected, or it received a request from a process joining or leaving 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}

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 atomic (instantaneous) event
    - No other processing takes place until this is complete

Virtual synchrony summary

- Message delivery
  - IP multicast or multiple UDP unicasts to implement multicast
- Message receipt
  - Each message is acknowledged but remains unstable
  - Sender informs group members when all ACKs have been received ⇒ stable
- Failure
  - Take failed process out of the group; initiate a view change

View change: 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
- Wait until it receives *flush* messages from the entire group

The End