DAD

Desenvolvimentos de Aplicações Distribuídas

Group Communication

Roadmap

- **Indirect Communication (Introduction)**
- **Introduction to Replication (Review)**
- **Group Communication**
  - Processes and groups
  - Group membership services
  - JGroups toolkit example
  - Views and view properties
  - View synchrony

Indirect Communication

- **What is Indirect Communication?**
  - communication between entities in a distributed system through an intermediary,
  - with no direct coupling between senders and receivers

  - E.g.,
    - group communication
    - publish-subscribe systems
    - message queues
    - shared memory

Indirect Communication

- **Space uncoupling**
  - sender(s) do not need to know the identity of the receiver(s) and vice-versa.
  - added fredoom in system design:
    - participants may be replaced, updated, replicated, migrated.
- **Time uncoupling**
  - sender(s) and receiver(s) do not need to exist (be active) at the same time to communicate (independent lifetimes).
    - tolerates volatile environments where participants come, go, fail, etc.
    - more than simple asynchronism
    - key: persistence of the communication channel

Indirect Communication

	Time-coupled	*Time-uncoupled*
Space coupling	*Properties*: Communication directed towards a given receiver or receivers; receiver(s) must exist at that moment in time *Examples*: Message passing, remote invocation (see Chapters 4 and 5)	*Properties*: Communication directed towards a given receiver or receivers; sender(s) and receiver(s) can have independent lifetimes *Examples*: See Exercise 15.3
Space uncoupling	*Properties*: Sender does not need to know the identity of the receiver(s); receiver(s) must exist at that moment in time *Examples*: IP multicast (see Chapter 4)	*Properties*: Sender does not need to know the identity of the receiver(s); sender(s) and receiver(s) can have independent lifetimes *Examples*: Most indirect communication paradigms covered in this chapter

Review on Replication

Replication can provide the following

- **performance enhancement**
  - e.g. several web servers can have the same DNS name and the servers are selected in turn to share the load
  - replication of read-only data is simple, but replication of changing data has overheads

- **fault-tolerance**
  - avoid single points of failure
  - guarantees correct behaviour in spite of certain faults (can include timeliness)
  - e.g., in synchronous systems:
    - if *f* of *f*+1 servers crash then 1 remains to supply the service
    - if *f* of 2*f*+1 servers have *byzantine* faults then they can supply a correct service

- **increase availability that is hindered by**
  - server failures
    - replicate data at failure-independent servers and when one fails, client may use another.
    - note that caches do not help with availability (they are incomplete).
  - network partitions and disconnected operation
    - users of mobile computers deliberately disconnect, and then on re-connection, resolve conflicts

Group Communication

- **important building block in distributed system design**
  - multicast of messages to groups of processes
- **key areas of application:**
  - reliable dissemination of information to large number of clients
    - e.g., financial industry
  - support for collaborative applications
    - to preserve common user view (e.g. games)
  - support for fault-tolerance strategies
    - consistent update of replicated data and highly available (replicated) servers
  - support for system monitoring and management
    - e.g., load balancing strategies
- **process groups are useful, e.g. for managing replicated data**
  - but replication systems need to be able to add/remove replica managers (RMs). how to support this, next…

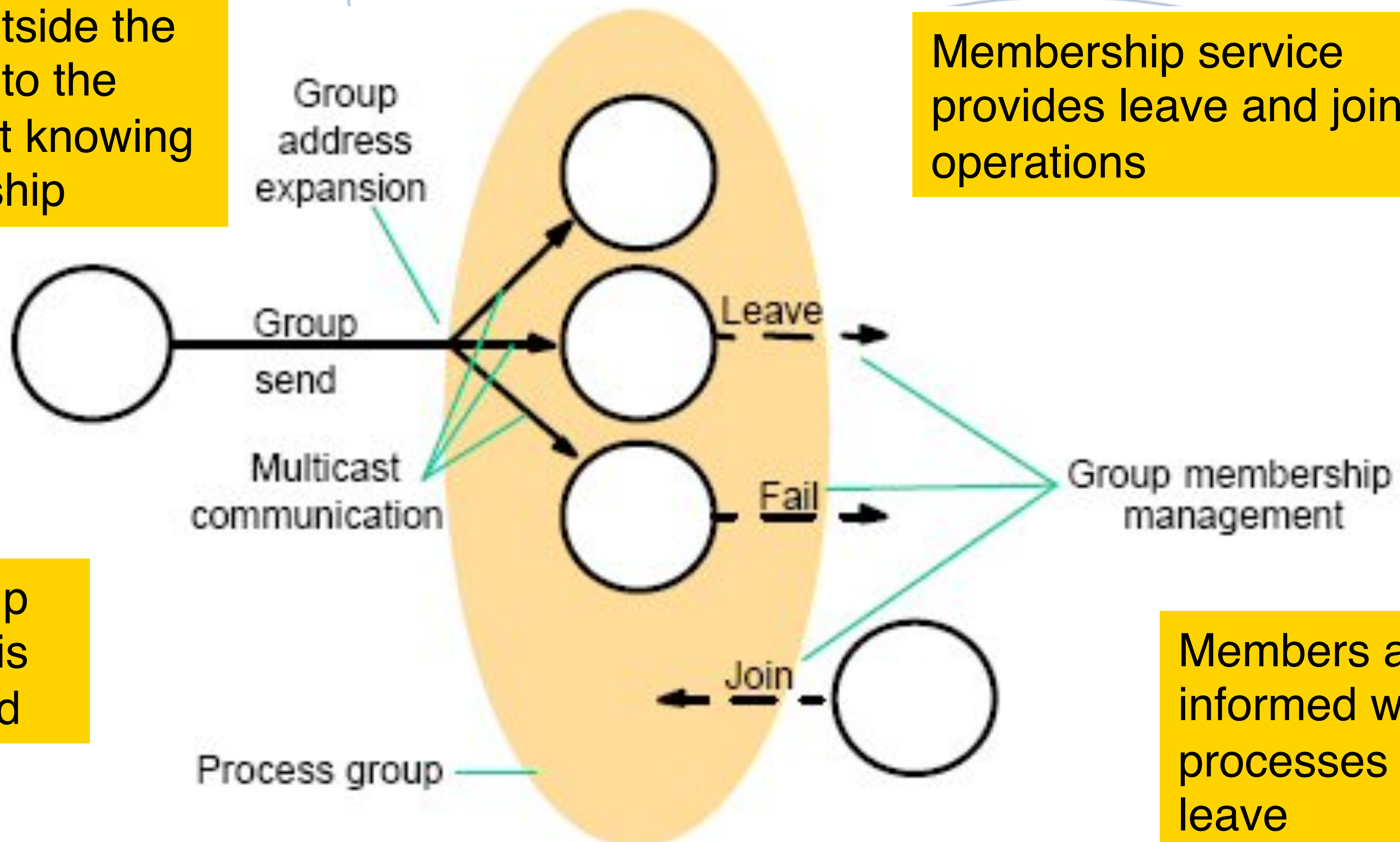
Group Communication Services

- **group membership service provides:**
  - interface for **adding/removing members**
    - create, destroy process groups, add/remove members.
    - a process can generally belong to several groups.
  - implements a **failure detector**
    - which monitors members for failures (crashes/communication),
    - and excludes them when unreachable
  - notifies members of **changes in membership**
    - new process added or process removed
  - **expands group addresses**
    - multicasts addressed to group identifiers and service translates it to the current list of members
    - coordinates delivery when membership is changing

Multicast Communication to Groups with Dynamic Membership

A process outside the group sends to the group without knowing the membership

Membership service provides leave and join operations



The group address is expanded

Members are informed when processes join/ leave

Failure detector reveals failures and evicts failed processes from the group

Key distinctions on Groups

- **Open and closed groups**
  - whether non-members can send messages to the group
- **Overlapping and non-overlapping groups**
  - whether processes may belong to mutiple groups
- **Synchronous and asynchronous**
  - whether both environments should be supported by the group communication middleware

Group Membership Service (1)

- **interface for group membership changes**
  - create, destroy groups
  - add, remove processes from groups
- **implements a failure detector**
  - which monitors members for failures (crashes/ communication),
  - and excludes them when unreachable (or suspected)
- **notifies of membership changes to group members**

Group Membership Service (2)

- **group address expansion**
  - multicasts addressed to group identifiers,
    - ensures space decoupling
  - coordinates delivery when membership is changing
- **Example:**
  - IP multicast allows members to join/leave and performs address expansion, but not the other features

Group Membership Service (3)

- **A full membership service maintains group views:**
  - lists of members, ordered e.g. as members join group.
- **A new group view is generated:**
  - each time a process joins or leaves the group.
- **View delivery:**
  - The idea is that processes can ***'deliver views'*** (like delivering multicast messages).
  - ideally we would like all processes to get the same information in the same order relative to the messages.

Group Membership Service (4)

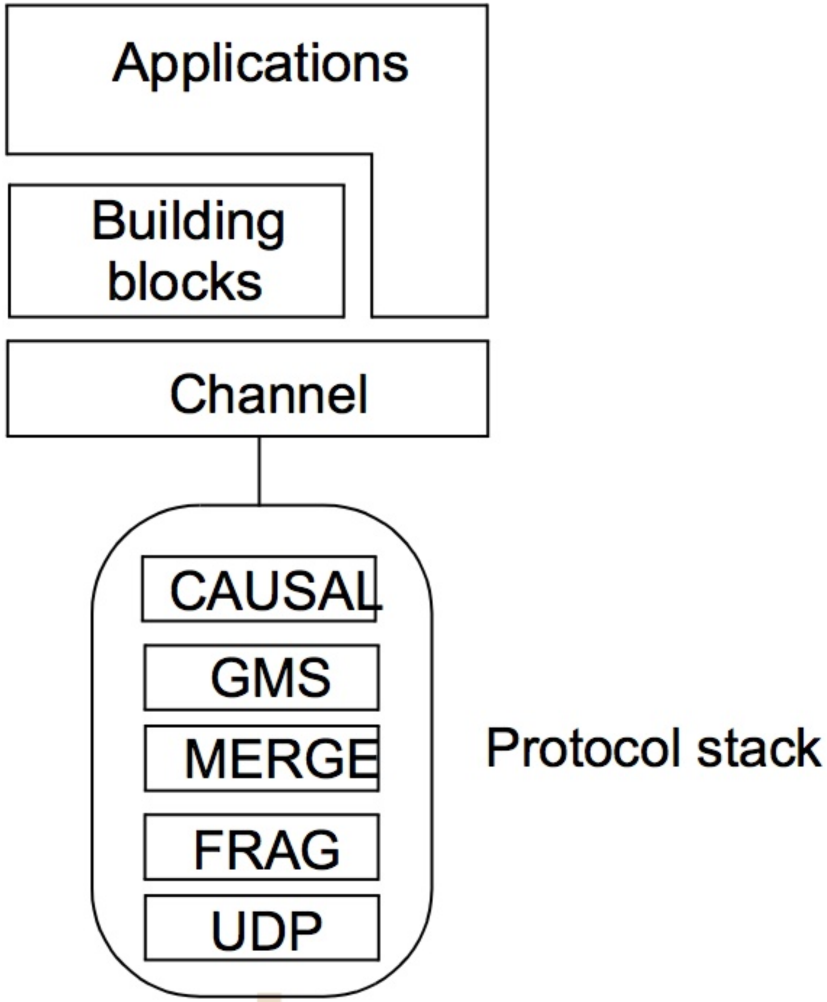
- **View-synchronous group communication with reliability:**
  - all processes agree on the ordering of messages and membership changes,
  - a joining process can safely get state from another member.
  - or if one crashes, another will know which operations it had already performed
  - This work was done in the ISIS system (Birman)

Group Membership Service (5)

- **Group views**
  - lists of current group members
  - process "suspected"
    - exclusion from group view
    - if process not failed, or recovered, it needs to re-join group
    - false suspicion reduces effectiveness of group
- **Network partitions**
  - occur when routers or links fail such that two subnets emerge that are no longer connected to each other
  - group management in the presence of partitions
    - **primary-partition:** at most one sub-group survives, remaining processes told to suspend
    - **partitionable:** subgroups survive as independent multicast groups

Example: JGroups toolkit

- **Architecture**



Example: JGroups toolkit (send)

```
import org.jgroups.JChannel;
public class FireAlarmJG {
public void raise() {
    try {
        JChannel channel = new JChannel();
        channel.connect("AlarmChannel");
        Message msg = new Message(null, null, "Fire!");
    channel.send(msg);
    }
    catch(Exception e) {
    }
}
```

Example: JGroups toolkit (receive)

```
import org.jgroups.JChannel;

  public class FireAlarmConsumerJG {
     public String await() {
     try {
           JChannel channel = new JChannel();
           channel.connect("AlarmChannel");
           Message msg = (Message) channel.receive(0);
           return (String) msg.GetObject();
     } catch(Exception e) {
           return null;
     }
  }
}
```

View Delivery

- treat each member of a group in a consistent way
  - when/during group membership changes
- necessary to relieve programmer:
  - from querying state of all other group members before making a send decision
- group management service delivers sequence of views to members, e.g.
  - v0(g) = {p}, v1(g) = {p, p'}, v2(g) = {p}, ...
- system imposes an ordering
  - on the possibly concurrent view changes
- receiving vs. delivering a view
  - view is placed in hold-back queue as for multicast until all members agree to deliver the view

View Delivery (2)

- **properties of view delivery**
  - **order**: if some process delivers two views in some order, then all processes in the group will do so
  - **integrity**: if some p delivers view of group g, then $p \in g$
  - **non-triviality**
    - if q joins group and becomes indefinitely reachable,
    - then q will eventually be included in all views
  - if the group partitions,
    - then eventually views delivered in one partition will exclude views delivered in other partitions

View Synchronous Group Communication (1)

- **extends reliable multicast semantics to view delivery**
  - guarantees not only the above properties for view delivery, but also <u>includes guarantees on the delivery of multicast messages</u>
- **for simplicity**
  - we exclude the possibility of network partitioning:
    - only a single group can exist at any point in time
- **guarantees/properties provided**
  - **agreement**
  - **integrity**
  - **validity**

View Synchronous Group Communication (2)

- **Guarantees:**
  - **agreement**:
    - correct processes deliver same sequence of views
      - (starting from the view in which they join the group), and
    - the same set of messages in any given view.
    - conclusion:
    - if a correct process delivers message **m** in view **v(g)**, then all other correct processes that deliver **m** also do so in the view **v(g)**.
  - ***uniform* agreement**
    - if any process delivers message **m** in view **v(g)**, then all other correct processes that deliver **m** also do so in the view **v(g)**.

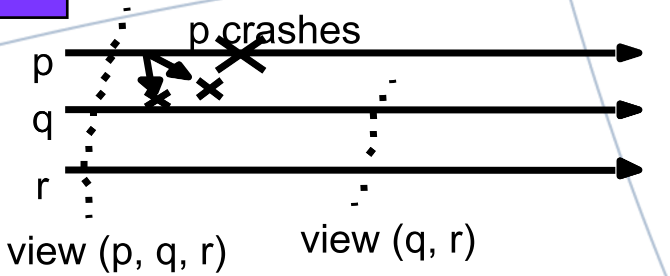
View Synchronous Group Communication (3)

- **Guarantees:**
  - **integrity**: if p delivers m, then
    - it will not deliver m again,
    - surely **p** ∈ **group(m)** ,
    - the process that sent m
      - is in the view in which p delivers m
  - **validity**:
    - correct processes always deliver messages they send
    - if system fails to deliver a message to any process q
      - immediately notifies surviving processes by delivering view that excludes q
      - hence, if for the next view **q** ∉ **view'(g)**, then p knows that q has failed

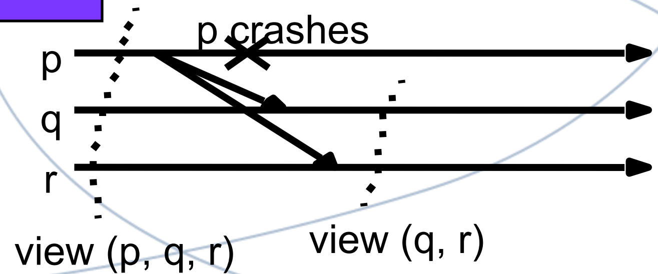
View Synchronous Group Communication (4)

- **example: p sends m while in view {p, q, r}, p crashes soon after sending m.**
  - [redacted]
  - [redacted]
  - [redacted]
  - [redacted]

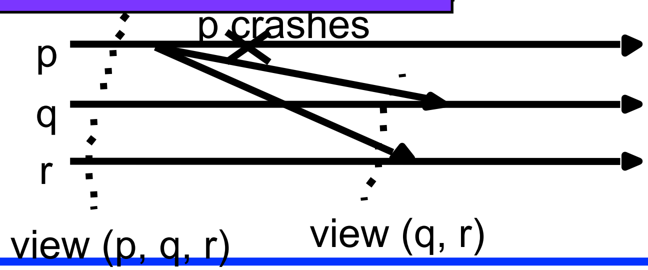
a [redacted]



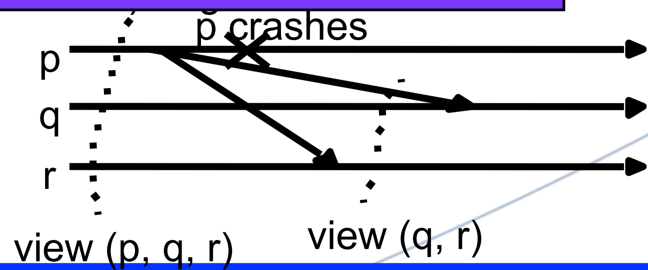
b [redacted]



c [redacted]



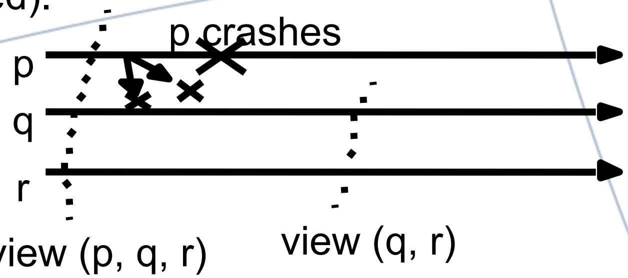
d [redacted]



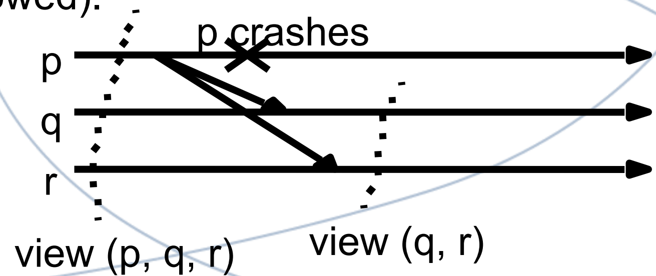
View Synchronous Group Communication (4)

- **example: p sends m while in view {p, q, r}, p crashes soon after sending m.**
  - a) If p crashes before m reaches any of q and r, then q and r each deliver new view {q, r} and neither delivers m
  - b) m has reached at least one of q and r before p crashes, then q and r deliver first m, and then view {q, r}
  - c) not allowed for q and r to first deliver view {q, r} and then m, since this would mean delivering a message from a failed process
  - d) nor can the two deliver the message and then the new view in opposite orders

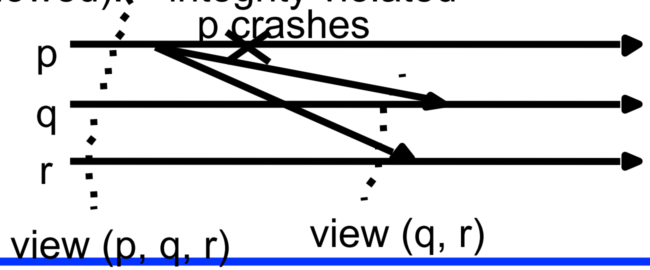
a (allowed).



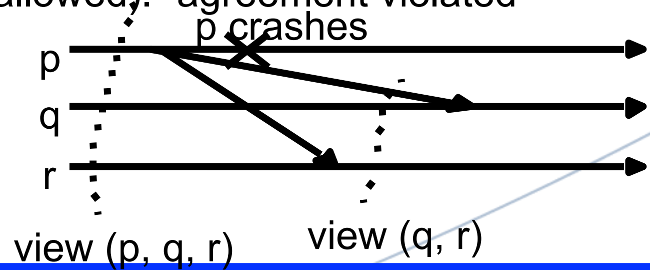
b (allowed).



c (disallowed)..– integrity violated



d (disallowed).- agreement violated



View Synchronous Group Communication (5)

- **In a view-synchronous system:**
  - the delivery of a new view draws a conceptual line across the system, and
  - every message is consistently delivered on one side or the other of that line
- **This enables the programmer to:**
  - draw useful conclusions about the set of messages that other correct processes have delivered when it delivers a new view,
  - based only on the local ordering of message delivery and view delivery events

View Synchronous Group Communication (6)

- **E.g., initialize state of new replicas (state-transfer)**
  - How to initialize the state of a newly joined replica?
  - Problem:
    - updates being performed concurrently
      - with new process joining and state capture
    - new replica
      - must not miss any update not reflected in the acquired state
      - cannot reapply updates already reflected in the state (unless idempotent)
    - very hard to achieve without middleware support as group communication

View Synchronous Group Communication (7)

- **E.g., initialize state of new replicas (state-transfer)**
  - How to initialize the state of a newly joined replica?
  - With Group Communication and View Synchrony
    - 1. Upon delivery of first view with a new process included
    - 2. Some (e.g., oldest) process captures its state
      - before executing any other operation (e.g., delivering messages)
    - 3. Sends its captured state to new joining process
      - all processes suspend execution until instructed to resume
      - all members have same state, same set of updates applied and will apply later ones (in same order) in same new view
    - 4. New process gets state and multicasts '*commence*'
      - all processes, including the new one, proceed from now on
      - apply operations/updates as they are delivered by GCS.

Summary

- **Indirect Communication (Introduction)**
- **Introduction to Replication (Review)**
- **Group Communication**
  - Processes and groups
  - Group membership services
  - JGroups toolkit example
  - Views and view properties
  - View synchrony