Building reliable distributed systems

Lecture 1

Roberto Baldoni, Carlo Marchetti, PhD
carlo.marchetti [AT] dis.uniroma1.it
Course overview (ADS)

• This set of lectures is framed within the ADS course, which is composed by 3 sets of seminars:
  – Monitoring large scale complex systems through Complex Event Processing Technologies (Lodi, Montanari, Aniello)
  – Building reliable distributed systems (Baldoni, Lodi, Platania)
  – Byzantine Fault Tolerance (Querzoni, Quema)

• For each of the 3 set of lectures, to access exams, every student must deliver a 5 pgs summary to the course’s principal instructor

• The exam can be given on one of the three areas
Course overview (BRDS)

• Building reliable distributed systems is an ambitious (frequently underestimated) goal, entailing a complex set of issues over several areas
  – hardware and network selection/design, test, maintenance
  – OS design, verification, upgrade
  – software engineering (e.g., testing)
  – ... ... ... ... ... (more and more)

• there’s no unique model/methodology

• Our focus will be on
  – increasing the responsiveness of a distributed service
  – deployed on cheap COTS hardware, network, operative system, application server
  – by studying how to exploit redundancy and coordination among replicas using
    • appropriate models
    • communication techniques and software architectures
Let’s drill down...

• Suppose one wanted to build an application that
  – Has some sort of “dynamic” state (receives updates)
  – Load-balances queries
  – Is fault-tolerant

• How would we do this?
Today’s prevailing solution

Clients  Middle tier runs business logic

Back-end shared database system
Concerns?

- Potentially slow (especially during failures)
  - Recoverability is not high availability

- Doesn’t work well for applications that don’t split cleanly between “persistent” state (that can be stored in the database) and “business logic” (which has no persistent state)
Can we do better?

- What about some form of in-memory database
  - Could be a true database
  - Or it could be any other form of storage “local” to the business logic tier
- This eliminates the back-end database
- But how can we build such a thing?
Lectures Calendar (BRDS)

- **April 29th, May 6th, 2011**
  - Lecturer: Roberto Baldoni
  - Covered topics:
    - course introduction
    - failure and consistency models
    - software replication techniques
    - group communications
Lectures Calendar (BRDS)

- **May 13th, 2011**
  - **Lecturer:** Marco Platania
  - **Covered topics:**
    - introduction to group communication systems
      - brief history
      - architectures and services offered
    - The JGroups toolkit
      - architecture
      - main API
      - protocol stack
    - Examples
Lectures Calendar (BRDS)

• May 20th 2011
  – Lecturer: Giorgia Lodi
  – Covered topics:
    • Brief introduction to JBoss application server: basic concepts
    • JBoss Clustering Service: what is good for and how it works
    • The clustering framework
      – Automatic discovery of clustered nodes
      – Fail-over and load balancing
      – Distributed cluster-wide hot deployment
    • How JBoss clustering framework uses JGroups for group communication
Our problem

• To offer a software service to clients over a network
  – service providers (S) and clients (C) are processes of a distributed systems running over COTS hw, os, network

• The service must be dependable, i.e., available and reliable, despite “failures”
  – we mainly focus on availability
Concistency models

- Concurrent clients update the local internal state of distinct replicas of a service
  - what about reply correctness?
- Suppose to replicate as-is a flight reservation service or a bank account management system
  - deliver inconsistent reply to concurrent clients
- how to define consistency?
  - objects and their operations have input/output semantics
  - consistency criteria are usually given using data-oriented operations (reads and writes on shared and replicated state)
The general organization of a logical data store, physically distributed and replicated across multiple processes.
# Consistency criteria

<table>
<thead>
<tr>
<th>Consistency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strict</td>
<td>Absolute time ordering of all shared accesses matters.</td>
</tr>
<tr>
<td>Linearizability</td>
<td>All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a (nonunique) global timestamp</td>
</tr>
<tr>
<td>Sequential</td>
<td>All processes see all shared accesses in the same order. Accesses are not ordered in time</td>
</tr>
<tr>
<td>Causal</td>
<td>All processes see causally-related shared accesses in the same order.</td>
</tr>
<tr>
<td>FIFO</td>
<td>All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order</td>
</tr>
</tbody>
</table>

(a)

<table>
<thead>
<tr>
<th>Consistency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weak</td>
<td>Shared data can be counted on to be consistent only after a synchronization is done</td>
</tr>
<tr>
<td>Release</td>
<td>Shared data are made consistent when a critical region is exited</td>
</tr>
<tr>
<td>Entry</td>
<td>Shared data pertaining to a critical region are made consistent when a critical region is entered.</td>
</tr>
</tbody>
</table>

(b)

a) Consistency models not using synchronization operations.

b) Models with synchronization operations.
Introducing linearizability (1)

- Operations take time to complete
- They can overlap if executed by concurrent (and possibly distributed) sequential processes
- Two operations are said
  - Concurrent if they overlap
  - Sequential if they don’t
    - $O < O'$
Linearizability In a Shared Memory System

1. LINEARIZZABILE

\[ \text{x write(a)} \ p_1 \ 	ext{ok()} \quad \text{x read()} \ p_1 \ 	ext{ok(b)} \]

\[ \text{x write(b)} \ p_2 \ 	ext{ok()} \]

2. NON LINEARIZZABILE

\[ \text{x write(a)} \ p_1 \ 	ext{ok()} \quad \text{x read()} \ p_1 \ 	ext{ok(a)} \]

\[ \text{x write(b)} \ p_2 \ 	ext{ok()} \]
Linearizability

- **Liveness**: every invocation has a matching response
- **Safety**: there exists a permutation $P$ of all operations on $O$ such that
  - $P$ is legal wrt $O$ (respects the semantic of the object)
  - If the response of $O$ occurs before $O'$ in the run, then $O$ precedes $O'$ in $P$ (respects the real-time ordering of operations)
Fifo queue example

• Assume a FIFO queue q is implemented by two distinct processes
Fifo queue example

If p1 returns “a” then
S=E1

Otherwise (p1 returns “b”) S=E2

Enq() ops are concurrent.
Both “OK(a)” and “OK(b)” are valid outputs
Fifo queue example

- If we assume q starts empty in E (p1 and p2), then there exists S=o₂o₁o₄o₃ s.t. linearizability is satisfied by E
Fifo queue example

- Is E linearizable?
- ...
- Why?
- ...

\[ \begin{array}{c}
\text{p1} & \text{ENQ(a)} & \text{OK()}
\end{array} \quad \begin{array}{c}
\text{DEQ()} & \text{OK(b)}
\end{array} \]

\[ \begin{array}{c}
1 & o_1 & 2
\end{array} \quad \begin{array}{c}
5 & o_3 & 6
\end{array} \]

\[ \begin{array}{c}
\text{p2} & \text{ENQ(b)} & \text{OK()}
\end{array} \]

\[ \begin{array}{c}
3 & o_2 & 4
\end{array} \]
Suppose each process hosts a replica of an object $O$, providing service $S$ to external clients (not represented in figure).
Implementing linearizability among distributed object replicas

- Sufficient conditions for implementing linearizable executions of a replicated object implemented by n sequential processes called replicas are:
  - **Uniform Atomicity**: if a replica completes operation O then eventually all correct replicas complete O
  - **Uniform Order**: if two replicas both execute two operation O and O’, then they execute O and O’ in the same order

- Note that
  - real times are referred to executions on replicas and not on clients (operations can be reordered wrt to client invocations real time order)
  - Uniform properties iff non-malicious fault models!
Important observation

• Uniform atomicity and order are *sufficient* conditions for guaranteeing linearizability
  – we will see that implementing these sufficient conditions implies using expensive communication primitives
  – knowledge of the systems in terms of
    • semantics
    • state composition and effect of object operations on it
    • real shared and replicated state
    • client invocation patterns
    • etc
  – permits the design and use of more efficient and optimized primitives and protocols to maintain the necessary consistency level
SW replication techniques

- Two main techniques
  - Active replication
    - Deterministic replicas, failure masking
  - Passive replication (or primary-backup)
    - Nondeterministic replicas, no failure masking

- System model: asynchronous distributed system
  - Processes: clients and replicas (non overlapping)
  - Channels: quasi-reliable (reliable delivery, no creation, no duplication)

- Crash fault model for processes
Passive replication

- The primary replica manages all the request processing
- Backup replicas “follow” the primary state
Replica update

- Consistent replication is “fighting possible inconsistencies due to non-determinism”, therefore
  - The primary replica executes operations and updates backups with:
    - `invId`: unique request identifier
    - `res`: response to be sent to client (retransmissions)
    - `state-update`: state to be set by backups
  - Primary waits for all backup acks before proceeding
Failure scenarios

• Let’s note
  – faulty backups are not a problem

• Let’s assume
  – in case of a primary failure, another (unique) primary is eventually elected among backups
  – Clients uniquely identify their requests and know the set of possible primaries to implement re-invocations

• In the following slides, by analysing primary failure scenarios, we identify the main inconsistency sources in order to spot the necessary mechanisms to avoid them
Primary failure 1

- The failure occurs after the client received the reply
  - No problem: a new primary will be elected (see assumptions)
Primary failure 2

• If the primary fails before updating any backup, no reply is sent back to clients
  – No problem, again
  – We assume clients to eventually connect to the new primary
Primary failure 3

- The failure occurs before receiving all acks by bcks.
  - Problem: some bcks updated, some others not
    - Update atomicity violation
    - A new primary could have lost some information and propagate erroneous updates
Enforcing order and atomicity

• As long there’s at most a primary in the replica group, order is guaranteed:
  – Need: enforcing a consistent view of replica group composition by all alive replicas

• As long as backups receive the same set of updates from primaries, responses returned to clients are consistent
  – Need: enforcing atomicity of update delivery among backups

• Both needs can be solved using a communication primitive named view synchronous multicast
  – This is our first GROUP communication primitive
Service Architecture

Application processes

B
A
C
D

join
leave
join

A seems to have failed

membership views

{A}
{A,B,D}
{A,D}
{A,D,C}
{D,C}

GMS processes

X
Y
Z

GMS
Reminder: Group Communication

- Terminology: group create, view, join with state transfer, multicast, client-to-group communication
- This is the “dynamic” membership model: processes come & go
VSM: definition

• View: tuple containing an integer (Vid) and an ordered list of processes ids (processes that “belong” to the view)

\[ v_0 = (0, \{x^1, x^2, x^3\}) \]
\[ v_1 = (1, \{x^1, x^3\}) \]
\[ v_2 = (2, \{x^1, x^2, x^3\}) \]

• Views are agreed upon among processes according to a specification implemented by a “group membership service”
  – A process is said to “install” a view upon delivering it
**VSM: definition**

- **Some membership service properties:**
  - If $p \in g_x$ crashes, a new view $v$ has to be defined and $p$ not belongs to $v$
  - Let $v_i(g_x)$ be a view s.t. $p \in v_i(g_x)$, then either $p$ installs $v_i(g_x)$ or $\exists k > 0$ t.c. $p \notin v_{i+k}(g_x)$.
  - $\forall p,q \in g_x$, if $p$ and $q$ both install $v_i(g_x)$ and $v_j(g_x)$ and $i \neq j$, then $p$ and $q$ install them in the same order

- **NOTE:** processes can be removed by views even if not crashed (unreliable failure detection)
VSM: definition

- We just introduced views and their properties wrt processes (the group membership service)
- This is not sufficient for enforcing consistency
VSM: definition

- VSM is a multicast primitive based on a group membership service
- The properties of VSM basically relate message delivery to the views in which messages are delivered
- This permits to state atomicity of delivery within a view, formally “view synchrony”:
  - “if $\exists p \in v_i$ that delivers message $m$ in $v_i$ and installs $v_{i+1}$, then $\forall q \in v_i$ that delivers $v_{i+1}$, $q$ delivers $m$ before delivering $v_{i+1}$”
VSM: definition

- Scenario C is not strictly a problem for passive replication. Why?
VSM and passive replication

• GM+VSM “solves” passive replication
  – Primary can be elected using views and the order of members in a view
    • Order is then guaranteed by the primary
  – The multicast primitive permits to update either all or none backups in a view
    • Update atomicity is guaranteed by VSM

• Problem: we have not dealt with liveness conditions
Active replication

**Basics**
- All replicas are equal (no primary) and deterministic
  - The reply only depends on the sequence of executions
- When a client issues a request,
  - All correct replicas receive the request and return a reply to the client
  - Client only waits for the first reply
Active replication

- All correct replicas execute requests and return replies to clients
Consistency

• In order to get consistency there must hold:
  – **Uniform atomicity**: if a replica executes a request req, then eventually all correct replicas execute req
  – **Uniform order**: if two replicas execute the same two requests, they execute these requests in the same order

• Channels are perfect
  – No consistency can be guaranteed without building a proper communication layer
Total order multicast
TO specifications

- **TO(UA,SUTO)**
  - The strongest TO spec.

**TO(NUA,SUTO)**
TO specifications (2)

• TO(UA,WUTO)

  p₁
  m₁  m₂  m₃  m₄

  p₂
  m₁  m₂  m₃  m₄

  p₃
  m₁  m₂  m₃  m₄

• TO(NUA,WUTO)

  p₁
  m₁  m₂  m₃  m₄  m₅  m₆

  p₂
  m₁  m₂  m₃  m₄  m₅  m₆

  p₃
  m₁  m₂  m₃  m₄  m₅  m₆
TO specifications (3)

- **TO(UA, WNUTO)**

- **TO(NUA, WNUTO)**

- **TO(UA, SUTO)** (Strongest total order)

- **TO(UA, WUTO)**

- **TO(NUA, WUTO)**

- **TO(NUA, WNUTO)**
Associating Implementations to Specifications

**Definition.** Let $\mathcal{I}$ be a TO implementation and let $R_{\mathcal{I}}$ be the set of runs that $\mathcal{I}$ can generate. $\mathcal{I}$ enforces a TO specification $S$ iff:

1. $R_{\mathcal{I}} \subseteq R_S$, and
2. $\forall S' \; S' \rightarrow S \Rightarrow R_{\mathcal{I}} \not\subseteq R_{S'}$.

Methodology for the association

1. verify that $\mathcal{I}$ is a TO implementation, i.e. $R_{\mathcal{I}} \subseteq R_{TO(NUA,WNUTO)}$. If affirmative, go to step 2, otherwise $\mathcal{I}$ is not a TO implementation;

2. set the current specification $S$ to the one on the top of the hierarchy, i.e. $TO(UA, SUTO)$;

3. while there exists an unchecked specification $S'$ such that $S \rightarrow S'$ belongs to the transitive reduction of $\rightarrow$ depicted in Figure 5, check if there exists a run $r \in R_{\mathcal{I}}$ satisfying the associated predicate $\neg P$. If affirmative, repeat this step setting current the specification $S$ to $S'$. Otherwise go to step 4;

4. $S$ represents the specification implemented by $\mathcal{I}$. 
TOM implementation based on VSC

Point-to-point primitives specification

**VSC1.** If a correct process $p$ R-sends a message $m$ to a correct process $q$, then $q$ eventually R-receives $m$.

**VSC2.** For each message $m$, a process $p$ R-receives $m$ at most once, and only if $m$ was R-sent to $p$ by some process $q$. 
TOM implementation based on VSC

Broadcast primitives specification

**VSC3.** If a correct process \(p\) [U]Rcasts a message \(m\), then it eventually [U]Rdelivers \(m\).

**VSC4.** For each message \(m\), each process [U]Rdelivers \(m\) at most once, and only if \(m\) was [U]Rcast by some process.

**VSC5.** If a process (respectively, a correct process) \(p\) URdelivers (respectively, Rdelivers) \(m\) in view \(v\), then all processes which are either correct or deliver a view change event in \(v\) URdeliver (resp. Rdeliver) \(m\).

**VSC6.** If a process \(p\) [U]Rdelivers a message \(m\) in view \(v\) and a process \(q\) [U]Rcasts \(m\) in view \(v'\) then \(v = v'\).
Static vs dynamic group communication

- To check the enforced specification done by a protocol we have to look at all the possible subruns
Fixed Sequencer-based TO implementations

Figure 11: Communication pattern of fixed sequencer protocols (borrowed from [16])

- **Broadcast-Broadcast (BB)**. The sender broadcasts message \( m \) to all members. Upon receiving \( m \), the sequencer assigns a sequence number \( seq(m) \), denoted \( seq(m) \), and then broadcasts \( seq(m) \) to all members (see Figure 11(a)). As an example, the Ensemble system [20] implements this pattern;

- **Send-Broadcast (SB)**. The sender sends message \( m \) to the sequencer, which assigns a sequence number \( seq(m) \) and then broadcasts the pair \( \langle m, seq(m) \rangle \) to all members (see Figure 11(b)). This pattern is implemented, for example, by the Ensemble system [20];

- **Ask-Broadcast (AB)**. The sender first gets a sequence number to assign to \( m \) from the sequencer via a simple rendezvous, then it broadcasts the pair \( \langle m, seq(m) \rangle \) to all members (see Figure 11(c)). JavaGroups [6] is an example of a system implementing this pattern.
A generic TO protocol

**Generic fixed sequencer protocol**(initial_view)

1. `integer seqnum_p ← 0;`
2. `integer last_delivered_p ← 0;`
3. `set received_p ← ∅;`
4. `set received_seqsp ← ∅;`
5. `set pending_p ← ∅;`
6. `view current_view_p ← initial_view;`
7. `procedure Deliver()`
   8. `while ∃m: m ∈ received_p ∧ (id(m), seq(m)) ∈ received_seqsp ∧ seq(m) = last_delivered_p + 1 do`
      9. `TOdeliver(m);`
      10. `last_delivered_p ← last_delivered_p + 1;`
      11. `received_p ← received_p \ {m};`
      12. `received_seqsp ← received_seqsp \ {(id(m), seq(m))};`
    13. when (view_change(new_view)) do
        14. `while received_p ≠ ∅ do`
            15. `MESSAGE m ← Φ(received_p, received_seqsp);`
            16. `TOdeliver(m);`
            17. `received_p ← received_p \ {m};`
            18. `received_seqsp ← received_seqsp \ {(id(m), seq(m))};`
      19. `seqnum ← 0;`
      20. `last_delivered_p ← 0;`
      21. `received_seqsp ← ∅;`
      22. `current_view_p ← new_view;`
      23. `x-RECOVER();`
8. `procedure TOCast(m)`
   9. `pending_p ← pending_p \ {m};`
10. `x-TOCAST(m);`
11. when (||Redeliver((m, CI)) do
12. `if (CI.view = current_view) then`
    13. `x-DELIVER((m, CI))`
14. when (Receive((m, CI)) from q) do
15. `if (CI.view = current_view) then`
    16. `x-RECEIVE((m, CI), q)_RSA_
Broadcast Based seq protocol

```
GENERIC FIXED SEQUENCER PROTOCOL(initial_view)
1. INTEGER seqnum_p ← 0;
2. INTEGER last_delivered_p ← 0;
3. SET received_p ← ∅;
4. SET received_seqs_p ← ∅;
5. SET pending_p ← ∅;
6. VIEW current_view_p ← initial_view;
7. procedure Deliver()
8. while ∃m : m ∈ received_p ∧ (id(m), seq(m)) ∈ received_seqs_p ∧ seq(m) = last_delivered_p + 1 do
9.   TOdeliver(m);
10.  last_delivered_p ← last_delivered_p + 1;
11.  received_p ← received_p \ {m};
12.  received_seqs_p ← received_seqs_p \ {(id(m), seq(m))};
13. when (view_change(new_view)) do
14.   while received_p ≠ ∅ do
15.     MESSAGE m ← Φ(received_p, received_seqs_p);
16.     TOdeliver(m);
17.     received_p ← received_p \ {m};
18.     received_seqs_p ← received_seqs_p \ {(id(m), seq(m))};
19.  seqnum_p ← 0;
20.  last_delivered_p ← 0;
21.  received_seqs_p ← ∅;
22.  current_view_p ← new_view;
23.  procedure TOCAST(m)
24.     pending_p ← pending_p \ {m};
25.     x-TOCAST(m);
26. when ([U]Rcast((m, CI))) do
27.     if (CI.type = ApplMsg) then
28.       received_p ← received_p \ {m};
29.       pending_p ← pending_p \ {m};
30.     if (Sequencer?(current_view)) then
31.       seqnum_p ← seqnum_p + 1;
32.     [U]Rcast((⊥,[SeqNum,id(m), seqnum,current_view]));
33.   case CI.type = SeqNum :
34.     received_seqs_p ← received_seqs_p \ {(CI.id, CI.seqnum)};
35.     procedure BB-DELIVER((m, CI))
36.     switch
37.       case CI.type = ApplMsg:
38.         received_p ← received_p \ {m};
39.         pending_p ← pending_p \ {m};
40.       if (Sequencer?(current_view)) then
41.         seqnum_p ← seqnum_p + 1;
42.         [U]Rcast((⊥,[SeqNum,id(m), seqnum,current_view]));
43.     case CI.type = SeqNum :
44.         received_seqs_p ← received_seqs_p \ {(CI.id, CI.seqnum)};
45.         DELIVER();
46.         procedure BB-RECOVER()
47.         for each m ∈ pending_p
48.           [U]Rcast((m, ApplMsg, ⊥, ⊥, current_view));
```
Send Broadcast seq protocol

```
GENERIC FIXED SEQUENCER PROTOCOL(initial_view)
1  INTEGER seqnum_p ← 0;
2  INTEGER last_delivered_p ← 0;
3  SET received_p ← Ø;
4  SET received_seqsp ← Ø;
5  SET pending_p ← Ø;
6  VIEW current_view_p ← initial_view;
7  procedure Deliver()
8     while ∃m : m ∈ received_p ∧ (id(m), seq(m)) ∈ received_seqsp ∧ seq(m) = last_delivered_p + 1 do
9         TOdeliver(m);
10        last_delivered_p ← last_delivered_p + 1;
11        received_p ← received_p \ {m};
12        received_seqsp ← received_seqsp \ {(id(m), seq(m))};
13        when (view_change(new_view)) do
14            while received_p ≠ Ø do
15                MESSAGE m ← Φ(received_p, received_seqsp);
16                TOdeliver(m);
17                received_p ← received_p \ {m};
18                received_seqsp ← received_seqsp \ {(id(m), seq(m))};
19                seqnum ← 0;
20                last_delivered_p ← 0;
21                received_seqsp ← Ø;
22                current_view_p ← new_view;
23                x-RECOVER();
24                procedure TOCAST(m)
25                    pending_p ← pending_p \ {m};
26                    x-TOCAST(m);
27                    when ([U]Reliver((m, CI))) do
28                        if (CI.view = current_view) then
29                            x-DELIVER((m, CI));
30                    when (RReceive((m, CI)) from q) do
31                        if (CI.view = current_view) then
32                            x-RECEIVE((m, CI), q)
33
34
35
36
37
38
39
40
41
42
43
44
45
46
SB FIXED SEQUENCER PROTOCOL()

procedure SB-TOCAST(m)
Rsend((m, [1, 1, 1, current_view])) to sequencer(current_view);

procedure SB-DELIVER((m, CI))
pending_p ← pending_p \ {m};
received_p ← received_p \ {m};
received_seqsp ← received_seqsp \ {(CI.id, CI.seqnum)};
DELIVER();

procedure SB-RECEIVE((m, CI), q)
If (Sequencer?(current_view)) then
seqnum ← seqnum + 1;
[U]Rcast((m, [1, id(m), seqnum, current_view]));

procedure SB-RECOVER()
for each m ∈ pending_p
Rsend((m, [1, 1, 1, current_view])) to sequencer(current_view);
```
Ask Broadcast seq protocol

```
GENERIC FIXED SEQUENCER PROTOCOL(initial_view)
    1 INTEGER seqnum_p ← 0;
    2 INTEGER last_delivered_p ← 0;
    3 SET received_p ← Ø;
    4 SET received_seqs_p ← Ø;
    5 SET pending_p ← Ø;
    6 VIEW current_view_p ← initial_view;
    7 procedure Deliver()
        8 WHILE ∃m : m ∈ received_p ∧ (id(m), seq(m)) ∈ received_seqs_p ∧ seq(m) = last_delivered_p + 1 DO
           9 TOdeliver(m);
           10 last_delivered_p ← last_delivered_p + 1;
           11 received_p ← received_p \ {m};
           12 received_seqs_p ← received_seqs_p \ {(id(m), seq(m))};
        13 WHEN (view_change(new_view)) DO
           14 WHILE received_p ≠ Ø DO
              15 MESSAGE m ← Φ(received_p, received_seqs_p);
              16 TOdeliver(m);
              17 received_p ← received_p \ {m};
              18 received_seqs_p ← received_seqs_p \ {(id(m), seq(m))};
              19 seqnum ← 0;
              20 last_delivered_p ← 0;
              21 received_seqs_p ← Ø;
              22 current_view_p ← new_view;
              23 x-Recover();
        24 procedure TOCAST(m)
           25 pending_p ← pending_p \ {m};
           26 x-TOCAST(m);
        27 WHEN ([U]Rdeliver((m, CI))) DO
           28 IF (CI.view = current_view) THEN
              29 x-Deliver((m, CI));
        30 WHEN (Receive((m, CI)) from q) DO
           31 IF (CI.view = current_view) THEN
              32 x-Receive((m, CI), q)
```

```
AB FIXED SEQUENCER PROTOCOL()
    33 procedure AB-TOCAST(m)
       34 Rsend((⊥, [GetSeq,id(m), ⊥, current_view])) to sequencer(current_view);
    35 procedure AB-DELIVER((m, CI))
       36 pending_p ← pending_p \ {m};
       37 received_p ← received_p ∪ {m};
       38 received_seqs_p ← received_seqs_p \ {(id(m), seq(m))};
       39 DELIVER();
    40 procedure AB-RECEIVE((m, CI), q)
       41 switch
           42 case CI.type = GetSeq:
              43 IF (Sequencer?(current_view)) THEN
                 44 seqnum ← seqnum + 1;
                 45 Rsend((⊥, [SeqNum, CI.id, seqnum, current_view])) to q;
           46 case CI.type = SeqNum:
              47 MESSAGE m' ← m'' : m'' ∈ pending ∧ id(m'') = CI.id;
              48 [U]Rcast((m', [⊥, CI.id, CI.seqnum, current_view]));
    49 procedure SB-RECOVER()
       50 FOR each m ∈ pending_p
           51 Rsend((⊥, [GetSeq, id(m), ⊥, current_view])) to sequencer(current_view);
```
### Specification enforced by a TO protocol

<table>
<thead>
<tr>
<th>Ordering protocol</th>
<th>Communication primitive</th>
<th>TO specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Broadcast-broadcast sequencer</td>
<td>Rcast/Rcast</td>
<td>TO(NUA, WNUTO)</td>
</tr>
<tr>
<td></td>
<td>URcast/URcast</td>
<td>TO(UA, SUTO)</td>
</tr>
<tr>
<td></td>
<td>Rcast/URcast</td>
<td>TO(NUA, WUTO)</td>
</tr>
<tr>
<td></td>
<td>URcast/Rcast</td>
<td>TO(UA, WNUTO)</td>
</tr>
<tr>
<td>Send-broadcast sequencer</td>
<td>Rcast</td>
<td>TO(NUA, WNUTO)</td>
</tr>
<tr>
<td></td>
<td>URcast</td>
<td>TO(UA, SUTO)</td>
</tr>
<tr>
<td>Ask-broadcast sequencer</td>
<td>Rcast</td>
<td>TO(NUA, WUTO)</td>
</tr>
<tr>
<td></td>
<td>URcast</td>
<td>TO(UA, SUTO)</td>
</tr>
</tbody>
</table>

**Table 5: TO specification enforced by each ordering protocol**

- **Basic Results**
  - All protocols ensure at least TO(NUA, WNUTO)
  - All protocols employing URCAST ensure TO(UA, SUTO)
Specification enforced by a TO protocol

<table>
<thead>
<tr>
<th>Ordering protocol</th>
<th>Communication primitive</th>
<th>TO specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Broadcast-broadcast sequence</td>
<td>Rcast/Rcast</td>
<td>TO(NUA, WNUTO)</td>
</tr>
<tr>
<td></td>
<td>U Rcast/U Rcast</td>
<td>TO(UA, SUTO)</td>
</tr>
<tr>
<td></td>
<td>Rcast/U Rcast</td>
<td>TO(NUA, WUTO)</td>
</tr>
<tr>
<td></td>
<td>U Rcast/Rcast</td>
<td>TO(UA, WNUTO)</td>
</tr>
<tr>
<td>Send-broadcast sequencer</td>
<td>Rcast</td>
<td>TO(NUA, WNUTO)</td>
</tr>
<tr>
<td></td>
<td>U Rcast</td>
<td>TO(UA, SUTO)</td>
</tr>
<tr>
<td>Ask-broadcast sequencer</td>
<td>Rcast</td>
<td>TO(NUA, WUTO)</td>
</tr>
<tr>
<td></td>
<td>U Rcast</td>
<td>TO(UA, SUTO)</td>
</tr>
</tbody>
</table>

(a) A run satisfying \(\neg WUTO \land \neg SUTO \land \neg UA\) generated by a BB fixed sequencer protocol using only \(Rcast\)

(b) A run satisfying \(\neg WUTO \land \neg SUTO \land \neg UA\) generated by a SB fixed sequencer protocol using only \(Rcast\)

Figure 16: Runs generated by BB and SB fixed sequencer protocols using only \(Rcast\)
Specification enforced by a TO protocol

<table>
<thead>
<tr>
<th>Ordering protocol</th>
<th>Communication primitive</th>
<th>TO specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Broadcast-broadcast sequencer</td>
<td>Rcast/Rcast</td>
<td>TO(NUA, WNUTO)</td>
</tr>
<tr>
<td></td>
<td>URcast/URcast</td>
<td>TO(UA, SUTO)</td>
</tr>
<tr>
<td></td>
<td>Rcast/URcast</td>
<td>TO(NUA, WUTO)</td>
</tr>
<tr>
<td></td>
<td>URcast/Rcast</td>
<td>TO(UA, WNUTO)</td>
</tr>
<tr>
<td>Send-broadcast sequencer</td>
<td>Rcast</td>
<td>TO(NUA, WNUTO)</td>
</tr>
<tr>
<td></td>
<td>URcast</td>
<td>TO(UA, SUTO)</td>
</tr>
<tr>
<td>Ask-broadcast sequencer</td>
<td>Rcast</td>
<td>TO(NUA, WUTO)</td>
</tr>
<tr>
<td></td>
<td>URcast</td>
<td>TO(UA, SUTO)</td>
</tr>
</tbody>
</table>

(a) A run satisfying $WUTO \land \neg SUTO \land \neg UA$ generated by $BB(Rcast, URcast)$

(b) A run satisfying $\neg WUTO \land \neg SUTO \land UA$ generated by $BB(URcast, Rcast)$

Figure 17: Runs generated by BB fixed sequencer protocols using both $Rcast$ and $URcast$
Specification enforced by a TO protocol

<table>
<thead>
<tr>
<th>Ordering protocol</th>
<th>Communication primitive</th>
<th>TO specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Broadcast-broadcast sequencer</td>
<td>Rcast/Rcast</td>
<td>TO(NUA,WNUTO)</td>
</tr>
<tr>
<td></td>
<td>U Rcast/U Rcast</td>
<td>TO(UA,SUTO)</td>
</tr>
<tr>
<td></td>
<td>Rcast/U Rcast</td>
<td>TO(NUA,WUTO)</td>
</tr>
<tr>
<td></td>
<td>U Rcast/Rcast</td>
<td>TO(UA,WNUTO)</td>
</tr>
<tr>
<td>Send-broadcast sequencer</td>
<td>Rcast</td>
<td>TO(NUA,WNUTO)</td>
</tr>
<tr>
<td></td>
<td>U Rcast</td>
<td>TO(UA,SUTO)</td>
</tr>
<tr>
<td>Ask-broadcast sequencer</td>
<td>Rcast</td>
<td>TO(NUA,WUTO)</td>
</tr>
<tr>
<td></td>
<td>U Rcast</td>
<td>TO(UA,SUTO)</td>
</tr>
</tbody>
</table>

Figure 18: A run satisfying $WUTO \land \neg SUTO \land \neg UA$ generated by an AB fixed sequencer protocol
Privilege-based protocols In privilege-based protocols, a single logical token circulates among processes and grants to its holder the privilege to send messages. Each message is sent along with a sequence number derived from a value carried by the token which is increased after each message sent. Receiver processes deliver messages according to their sequence numbers. As only one token may circulate, and only the token holder may send messages, messages are delivered in a total order. Totem [17] and Spread [18] are examples of systems implementing this protocol.

In several privilege-based protocols, e.g. [17, 18, 9], processes are organized in a logical ring, and a process passes the token to the next process upon the occurrence of the first of the following internal events: (i) no more messages to send, or (ii) maximum use of some resources achieved (e.g. maximum token-holding interval, maximum number of messages sent by the process). These kind of protocols usually can be configured to implement URCast at the Total Order layer, augmenting Rcast with additional mechanisms thanks to the token passing. An example of such protocols is the one implemented by Spread [18].

<table>
<thead>
<tr>
<th>Ordering protocol</th>
<th>Communication primitive</th>
<th>TO specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Broadcast-broadcast sequencer</td>
<td>Rcast/Rcast</td>
<td>TO(NUA, WNUTO)</td>
</tr>
<tr>
<td></td>
<td>UCast/URcast</td>
<td>TO(NUA, SUTO)</td>
</tr>
<tr>
<td></td>
<td>Rcast/URcast</td>
<td>TO(NUA, WNUTO)</td>
</tr>
<tr>
<td></td>
<td>UCast/UCast</td>
<td>TO(NUA, SUTO)</td>
</tr>
<tr>
<td>Send-broadcast sequencer</td>
<td>Rcast</td>
<td>TO(NUA, WNUTO)</td>
</tr>
<tr>
<td>Ask-broadcast sequencer</td>
<td>Rcast</td>
<td>TO(NUA, WNUTO)</td>
</tr>
<tr>
<td>Ask-broadcast sequencer</td>
<td>UCast</td>
<td>TO(NUA, SUTO)</td>
</tr>
<tr>
<td>Privilege-based</td>
<td>Rcast</td>
<td>TO(NUA, WNUTO)</td>
</tr>
<tr>
<td></td>
<td>UCast</td>
<td>TO(NUA, SUTO)</td>
</tr>
</tbody>
</table>

Table 3. TO specification enforced by each ordering protocol
Sw architecture for replication

- Typical two-tier sw replication architecture for client/server applications
- Client use retrasmission and dynaming name resolution mechanisms
- Replication handlers implement consistency mechanisms in replicas using group communication toolkits
  - Request identification, numbering, forwarding
  - Request and response buffering
  - ...

```
client

R. H.

Replication Handler
replica

Replication Handler
replica

Replication Handler
replica

Server

Server

Server
```
Horus/JGroups/Ensemble protocol stacks

Application belongs to process group

- total
  - fc
  - mbrshp
  - frag
  - nak
  - comm
- merge
  - mbrshp
- total
  - parcld
    - frag
    - mbrshp
    - nak
    - frag
    - comm
    - nak
    - comm
Group communications toolkits

JavaGroups. JavaGroups is a Java group communication system based on the concept of micro-protocols (as Ensemble). As for Spread, JavaGroups does not exactly comply with our reference architecture, as it does not provide a primary component membership service. However, this can be implemented by coding a simple specific micro-protocol [19]. JavaGroups offers two micro-protocols implementing the Total Order layer, namely TOTAL, which embeds an AB fixed sequencer protocol using Rcast, and TOTAL_TOKEN, which embeds a privilege-based protocol enabled to implement URcast on top of Rcast. As proven in [4], these protocols enforce \( TO(NUA, WUTO) \) and \( TO(UA, SUTO) \), respectively, if JavaGroups is provided with the primary component membership service micro-protocol.
Ensemble. Ensemble provides fine-grained control over its functionality, which can be selected simply layering micro-protocols, i.e. well-defined stackable components implementing simple and specific functions. In particular, Ensemble can be configured to implement virtual synchrony and a primary component membership service. A TO primitive is obtained layering a micro-protocol resembling the Total Order layer into a virtually synchronous stack. In the following we consider the micro-protocols named Seqbb and Sequencer, which correspond to BB and SB fixed sequencer protocols using Rcast, respectively (see Section 3.2). As shown in [4], layering one of these protocols in a virtually synchronous stack allows us to enforce TO(NUA, WNUTO).
Group communications toolkits

Spread. Spread is a toolkit designed for large scale networks based on a client-daemon architecture. It offers several communication abstraction, enabled by selecting the so-called “service type”. Spread implements a partitionable membership service based on the extended virtual synchrony model [19], which extends virtual synchrony to partitionable environments. To comply with the reference architecture of Figure 2, it is thus necessary to assume either absence of network partitioning or the presence of a software filter implementing a primary component membership service and virtual synchrony on top of extended virtual synchrony [19]. In these cases, the privilege-based protocol embedded by Spread (enabled by selecting the Agreed service type) implements $TO(NUA, WUTO)$. In contrast, selecting the Safe service type, the protocol implements $URcast$ on top of $Rcast$ (see Section 3.2) and thus the implemented TO specification is $TO(UA, SUTO)$. 
<table>
<thead>
<tr>
<th>Toolkit</th>
<th>TO implementation</th>
<th>Protocol type</th>
<th>TO specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spread</td>
<td>Safe</td>
<td>PB(Ucast)</td>
<td>TO(UA, SUTO)</td>
</tr>
<tr>
<td></td>
<td>Agreed</td>
<td>PB(Rcast)</td>
<td>TO(NUA, WUTO)</td>
</tr>
<tr>
<td>Ensemble</td>
<td>Seqbb</td>
<td>BB(Rcast/Rcast)</td>
<td>TO(NUA, WNUTO)</td>
</tr>
<tr>
<td></td>
<td>Sequencer</td>
<td>SB(Rcast)</td>
<td>TO(NUA, WNUTO)</td>
</tr>
<tr>
<td>JavaGroups</td>
<td>TOTAL_TOKEN</td>
<td>PB(Ucast)</td>
<td>TO(UA, SUTO)</td>
</tr>
<tr>
<td></td>
<td>TOTAL</td>
<td>AB(Rcast)</td>
<td>TO(NUA, WUTO)</td>
</tr>
</tbody>
</table>

*Table 4. Main characteristics of the group toolkits with respect to their TO implementations*
**Fig. 4.** Average message latency

<table>
<thead>
<tr>
<th>Toolkit</th>
<th>Configuration</th>
<th>Additional mechanisms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spread (Safe)</td>
<td>Safe service type</td>
<td>VS + PC GMS filters</td>
</tr>
<tr>
<td>Spread (Agreed)</td>
<td>Agreed service type</td>
<td>VS + PC GMS filters</td>
</tr>
<tr>
<td>Ensemble (BB)</td>
<td>VS + PC GMS + Sequencer</td>
<td>-</td>
</tr>
<tr>
<td>Ensemble (SB)</td>
<td>VS + PC GMS + Sequencer</td>
<td>-</td>
</tr>
<tr>
<td>JavaGroups [TR]</td>
<td>VS + TOTAL + TOKEN</td>
<td>PC GMS filter</td>
</tr>
<tr>
<td>JavaGroups [AB]</td>
<td>VS + TOTAL</td>
<td>PC GMS filter</td>
</tr>
</tbody>
</table>

Table 5. Configurations and additional mechanisms necessary to achieve TO specifications supported by each of the examined group toolkits

**Fig. 5.** Overall system throughput