

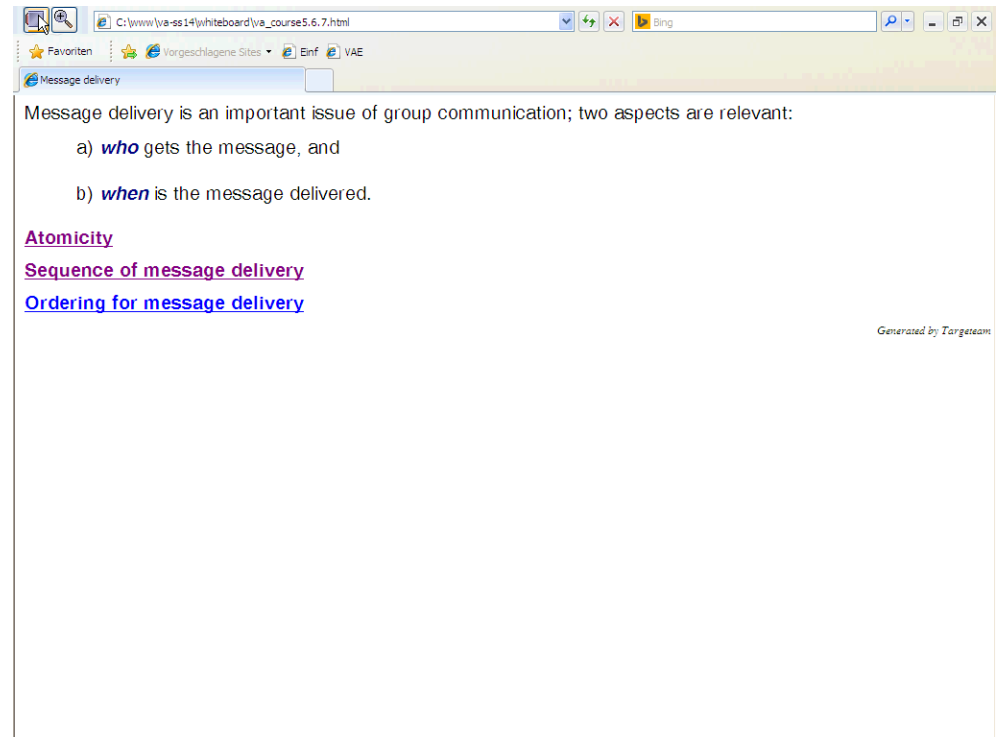
## Script generated by TTT

Title: Distributed\_Applications (03.06.2014)

Date: Tue Jun 03 14:32:42 CEST 2014

Duration: 87:56 min

Pages: 27



Message delivery is an important issue of group communication; two aspects are relevant:

- who* gets the message, and
- when* is the message delivered.

**Atomicity**  
**Sequence of message delivery**  
**Ordering for message delivery**

Generated by Targeteam



### Ordering for message delivery



Delivery of messages without delay in the same sequence is not possible in a distributed system  $\Rightarrow$  ordering methods for message delivery.

synchronously, i.e. there is a system-wide global time ordering.

loosely synchronous, i.e. consistent time ordering, but no system-wide global (absolute) time.

[Total ordering by sequencer](#)

[Virtually synchronous ordering](#)

[sync-ordering](#)



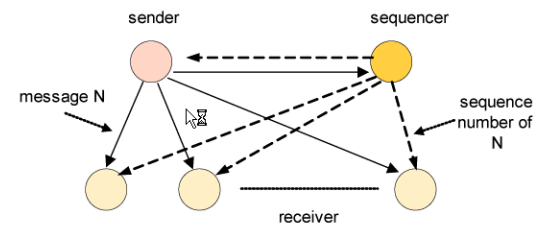
Generated by Targeteam



### Total ordering by sequencer



A selected group member serializes all the messages sent to the group.



1st step: the sender distributes the message N to **all** group members;

2nd step: sequencer (serializer, coordinator) determines a sequence number for N and distributes it to all group members; delivery of N to the application processes takes place according to this number.

### Zookeeper

a distributed open-source coordination service for distributed applications

keeps all the servers in sync.

guarantees a total order of messages.

use of majority quorum for coordinator selection .

Generated by Targeteam



Delivery of messages without delay in the same sequence is not possible in a distributed system  $\Rightarrow$  ordering methods for message delivery.

synchronously, i.e. there is a system-wide global time ordering.

loosely synchronous, i.e. consistent time ordering, but no system-wide global (absolute) time.

[Total ordering by sequencer](#)

[Virtually synchronous ordering](#)

[sync-ordering](#)

Generated by Targeteam



Delivery of messages without delay in the same sequence is not possible in a distributed system  $\Rightarrow$  ordering methods for message delivery.

synchronously, i.e. there is a system-wide global time ordering.

loosely synchronous, i.e. consistent time ordering, but no system-wide global (absolute) time.

[Total ordering by sequencer](#)

[Virtually synchronous ordering](#)

[sync-ordering](#)

Generated by Targeteam



Delivery of messages without delay in the same sequence is not possible in a distributed system  $\Rightarrow$  ordering methods for message delivery.

synchronously, i.e. there is a system-wide global time ordering.

loosely synchronous, i.e. consistent time ordering, but no system-wide global (absolute) time.

[Total ordering by sequencer](#)

[Virtually synchronous ordering](#)

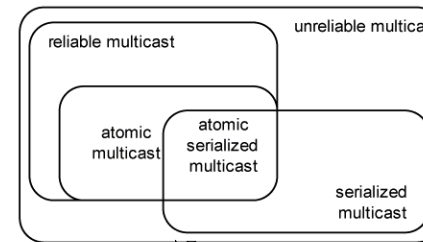
[sync-ordering](#)

Generated by Targeteam



**Multicast messages** for constructing distributed systems based on group communication;

different multicast communication semantics



[Multicast classes](#)

Multicasting can be realized by using IP multicast which is built on top of the Internet protocol IP.

Java API provides a datagram interface to IP multicast through the class `MulticastSocket`.

Generated by Targeteam



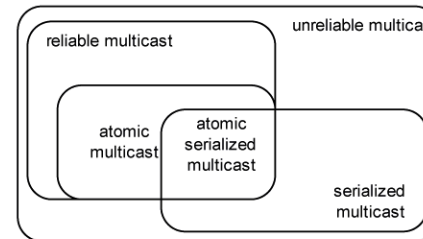
Depending on the message delivery guarantee, five classes of multicast services can be distinguished.

1. **unreliable multicast** : an attempt is made to transmit the message to all members without acknowledgement; at-most-once semantics with respect to available members; message ordering is not guaranteed.
2. **reliable multicast** : the system transmits the messages according to "best-effort", i.e. the "at-least-once" semantics is applied.
  - B-multicast primitive: guarantees that a correct process will eventually deliver the message as long as the multicaster does not crash.
  - B-deliver primitive: corresponding primitive when a message is received.
3. **serialized multicast** : consistent sequence for message delivery; distinction between totally ordered and causally ordered (i.e. virtually synchronous)
4. **atomic multicast** : a reliable multicast which guarantees that either all operational group members receive a message, or none of them do.
5. **atomic, serialized multicast** : atomic message delivery with consistent delivery sequence

Generated by Targeteam



**Multicast messages** for constructing distributed systems based on group communication; different multicast communication semantics



**Multicast classes**

Multicasting can be realized by using IP multicast which is built on top of the Internet protocol IP.

Java API provides a datagram interface to IP multicast through the class `MulticastSocket`.

Generated by Targeteam



**Introduction**

Group communication facilitates the interaction between groups of processes.

Motivation

Important issues

Conventional approaches

Groups of components

Management of groups

Message dissemination

Message delivery

Taxonomy of multicast

Group communication in ISIS

JGroups

Generated by Targeteam



The ISIS system developed at Cornell University is a framework for reliable distributed computing based upon process groups. It specifically supports group communication. Successor of ISIS was **Horus**.

ISIS is a toolkit whose basic functions include process group management and ordered multicast primitives for communication with the members of the process group.

abcast: totally ordered multicast.

cbcast: causally ordered multicast.

abcast protocol

cbcast protocol

Generated by Targeteam



**atomic broadcast** supports a total ordering for message delivery, i.e. all messages to the group  $G$  are delivered to all group members of  $G$  in the same sequence.

abcast realizes a serialized multicast

abcast is based on a **2-phase commit** protocol; message serialization is supported by a distributed algorithm and logical timestamps.

#### Phase 1

Sender  $S$  sends the message  $N$  with logical timestamp  $T_S(N)$  to all group members of  $G$  (e.g. by multicast).

Each  $g \in G$  determines a new logical timestamp  $T_g(N)$  for the received message  $N$  and returns it to  $S$ .

#### Phase 2

$S$  determines a new logical timestamp for  $N$ ; it is derived from all proposed timestamps  $T_g(N)$  of the group members  $g$ .

$$T_{S,new}(N) = \max(T_g(N)) + j/|G|, \text{ with } j \text{ being a unique identifier of sender } S.$$

$S$  sends a commit to all  $g \in G$  with  $T_{S,new}(N)$ .

Each  $g \in G$  delivers the message according to the logical timestamp to its associated application process.

Generated by Targeteam

The ISIS system developed at Cornell University is a framework for reliable distributed computing based upon process groups. It specifically supports group communication. Successor of ISIS was **Horus**.

ISIS is a toolkit whose basic functions include process group management and ordered multicast primitives for communication with the members of the process group.

abcast: totally ordered multicast.

cbcast: causally ordered multicast.

[abcast protocol](#)

[cbcast protocol](#)

Generated by Targeteam



Let  $n$  be the number of group members of  $G$ . Each  $g \in G$  has a unique number of  $\{1, \dots, n\}$  and a state vector  $z$  which stores information about the received group messages.

The state vector represents a [vector clock](#).

Each message  $N$  of sender  $S$  has a unique number; message numbers are linearly ordered with increasing numbers.

Let  $j$  be a group member of the group  $G$ .

the state vector  $z_j = (z_{ji})_{i \in \{1, \dots, n\}}$  specifies the number of messages received in sequence from group member  $i$ .

Example:  $z_{ji} = k$ ;  $k$  is the number of the last message sent by member  $i \in G$  and received in correct sequence by the group member  $j$ .

at group initialization all state vectors are reset (all components are 0).

**Sending** a message  $N$ ;  $j \in G$  sends a message to all other group members.

$z_{ji} := z_{ji} + 1$ ; the current state vector is appended to  $N$  and sent to all group members.

**Receiving** a message  $N$  sent by member  $i \in G$ .

Message  $N$  contains state vector  $z_i$ . There are two conditions for delivery of  $N$  to the application process of  $j$

(C 1):  $z_{ji} = z_{ii} - 1$ .

(C 2):  $\forall k \neq i: z_{ik} \leq z_{jk}$ .

Generated by Targeteam

**causal broadcast** guarantees the correct sequence of message delivery for causally related messages.

Concurrent messages can be delivered in any sequence; this approach minimizes message delay.

[Introduction](#)

[Algorithm of the cbcast protocol](#)

Generated by Targeteam



**JGroups** is a reliable group communication toolkit written in Java. It is based on IP multicast and extends it with

reliability, especially ordering of messages and atomicity.

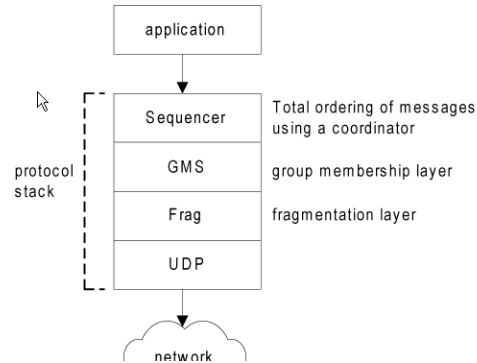
management of group membership.

### Programming Interface of JGroups

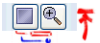
groups are identified via channels.

```
channel.connect("MyGroup");
```

a channel is connected to a protocol stack specifying its properties.



Code Example



```
String props = "UDP:Frag:GMS:causal";
Message send_msg;
Object rcv_msg;
Channel channel = new JChannel(props);
channel.connect("MyGroup");
send_msg = new Message(null, null, "hello World");
channel.send(send_msg);
rcv_msg = (Message) channel.receive(0);
System.out.println("Received " + rcv_msg);
channel.disconnect();
channel.close();
```



reliability, especially ordering of messages and atomicity.  
management of group membership.

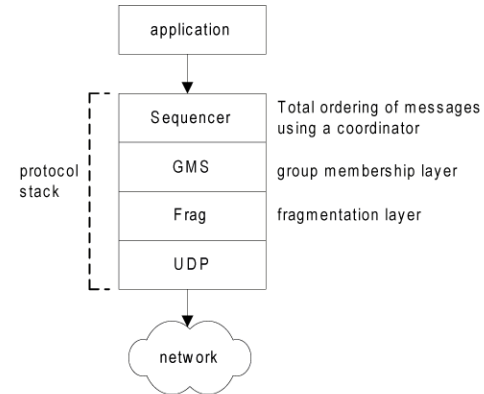
### Programming Interface of JGroups

groups are identified via channels.

```
channel.connect("MyGroup");
```

a channel is connected to a protocol stack specifying its properties.

*and like sockets, asynchronous send/receive*  
*name of group*



Code Example

Distributed Consensus



problem of distributed processes to agree on a value; processes communicate by message passing.

#### Examples

all correct computers controlling a spaceship should decide to proceed with landing, or all of them should decide to abort (after each has proposed one action or the other)

in an electronic money transfer transaction, all involved processes must consistently agree on whether to perform the transaction (debit and credit), or not

desirable: reaching consensus even in the presence of faults

assumption: communication is reliable, but processes may fail

#### Consensus Problem

#### Consensus in synchronous Networks



## Consensus Problem

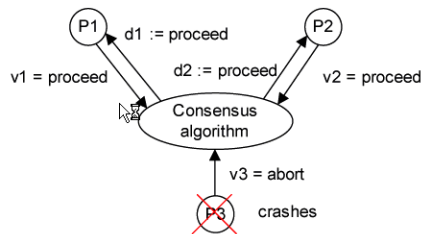


agreement on the value of a **decision variable**  $d_i$  amongst all correct processes

$p_i$  is in state undecided and proposes a single value  $v_i$ , drawn from a set of values.

next, processes communicate with each other to exchange values.

in doing so,  $p_i$  sets decision variable  $d_i$  and enters the decided state after which the value of  $d_i$  remains unchanged



[Properties](#)

[Algorithm](#)

[The Byzantine Generals Problem](#)

[Interactive Consistency Problem](#)

Generated by Targeteam

## Properties



The following conditions should hold for every execution of the algorithm:

**termination**: eventually, each correct process sets its decision variable

**agreement**: the decision variable of all correct processes is the same in the decided state.

**integrity**: if the correct processes all proposed the same value, then any correct process has chosen that value in the decided state.

Generated by Targeteam



## Algorithm



algorithm to solve consensus in a failure-free environment

each process reliably multicasts proposed values

after receiving response, solves consensus function  $\text{majority}(v_1, \dots, v_n)$ , which returns most often proposed value, or undefined if no majority exists.

properties:

termination guaranteed by reliability of multicast.

agreement, integrity: by definition of majority, and the integrity of reliable multicast (all processes solve same function on same data).

when crashes occur

how to detect failure?

will algorithm terminate?

when byzantine failures occur

processes communicate random values.

evaluation of consensus function may be inconsistent.

malevolent processes may deliberately propose false or inconsistent values.

Generated by Targeteam



## The Byzantine Generals Problem



three or more generals are to agree to attack or to retreat.

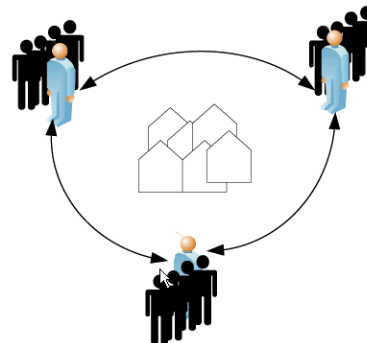
one general, the commander issues order

others (lieutenants to the commander) have to decide to attack or retreat

one of the generals may be treacherous

if commander is treacherous, it proposes attacking to one general and retreating to the other

if lieutenants are treacherous, they tell one of their peers that commander ordered to attack, and others that commander ordered to retreat



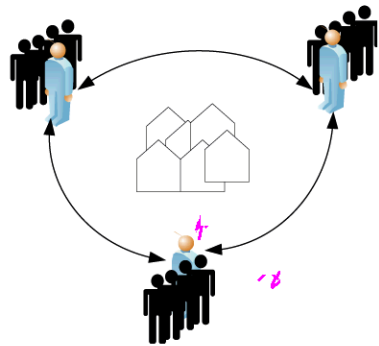
difference to consensus problem: one process supplies a value that others have to agree on

properties:





If messengers are trustworthy, they tell one of their peers that commander ordered to attack, and others that commander ordered to retreat



difference to consensus problem: one process supplies a value that others have to agree on  
properties:

- termination: eventually each correct process sets its decision variable.
- agreement: the decision value of all correct processes is the **same**.
- integrity: if the commander is correct, then all processes decide on the value that the commander proposes.

Generated by Targeteam



**Assumption** : no more than  $f$  of the  $n$  processes crash ( $f < n$ ).

The algorithm proceeds in  $f+1$  rounds in order to reach consensus.

the processes B-multicast values between them.

at the end of  $f+1$  rounds, all surviving processes are in a position to agree.

algorithm for process  $p_i \in$  consensus group  $g$

On initialization

```

valuesi (1) := {vi }; valuesi (0) := {};

```

in round  $r$  ( $1 \leq r \leq f+1$ )

```

B-multicast(g, valuesi (r)-valuesi (r-1));
//send only values that have not been sent
valuesi (r+1) := valuesi (r)
while (in round r) {
  On B-deliver(vj ) from some pj
  valuesi (r+1) := valuesi (r+1) U vj
}

```

After  $(f+1)$  rounds

```

assign di = minimum (valuesi (f+1))

```

Generated by Targeteam



Each process suggests a single value.

**goal** : all correct processes agree on a vector of values ("decision vector"); each component correspond to one processes' agreed value

example: agreement about each processes' local state.

properties:

- termination: eventually each correct process sets its decision vector.
- agreement: the decision vector of all correct processes is the same.
- integrity: if  $p_i$  is correct, then all correct processes decide on  $v_i$  as the  $i$ -th component of their vector.

Generated by Targeteam



- Prof. J. Schlichter
  - Lehrstuhl für Angewandte Informatik / Kooperative Systeme, Fakultät für Informatik, TU München
  - Boltzmannstr. 3, 85748 Garching
  - Email: [schlichter@in.tum.de](mailto:schlichter@in.tum.de)
  - Tel.: 089-289 18654
  - URL: <http://www11.in.tum.de/>

- [Overview](#)
- [Introduction](#)
- [Architecture of distributed systems](#)
- [Remote Invocation \(RPC/RMI\)](#)
- [Basic mechanisms for distributed applications](#)
- [Web Services](#)
- [Design of distributed applications](#)
- [Distributed file service](#)
- [Distributed Shared Memory](#)
- [Object-based Distributed Systems](#)
- [Summary](#)

Generated by Targeteam