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 Torus.

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,
- `bcicast`: causally ordered multicast.

**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.

`bcicast` 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 $g$.

$$T_{S,next}(N) = \max\{T_g(N) + |G|, j|G|\}$$

with $j$ being a unique identifier of sender S.

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

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

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.

**Algorithm of the cbcast protocol**

Let $n$ be the number of group members of $G$. Each $g \in G$ has a unique number of $\{1, \ldots, n\}$ 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 $z_j = (z_{j1}, \ldots, z_{jn})$ specifies the number of messages received in sequence from group members.

Example: $z_{jk} = k$; $k$ 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.

$z_{jk} := z_{jk} + 1$; 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 $j$:

1. $(C1)$: $z_j = z_i + 1$.
2. $(C2)$: $\forall k \neq i. z_{ik} \leq z_{jk}$.

**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**

**JGroups**

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.

```
protocolStack:
  application
  Sequencer
  Total ordering of messages
  GMS
  group membership layer
  Frag
  fragmentation layer
  UDP
  network
```
JGroups

management of group membership.

Programming Interface of JGroups

Programs are identified via channels:

- channel.connect("MyGroup");
- a channel is connected to a protocol stack specifying its properties.

- Application
- Sequencer: Total ordering of messages using a coordinator
- GMS: group membership layer
- Frag: fragmentation layer
- UDP
- Network

Code Example

Group communication

Introduction

Group communication facilities 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 IS19

JGroups

Group communication

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

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 amongst all correct processes, $p$, 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, $p$ sets decision variable $d$ and enters the decided state after which the value of $d$ remains unchanged.

Properties

**Algorithm**

The Byzantine Generals Problem

Interactive Consistency Problem

Relationship between these Problems

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.

Algorithm to solve consensus in a failure-free environment:

Each process reliably multicasts proposed values.

After receiving response, solves consensus function $\text{majority}(v_1, \ldots, v_n)$, 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.

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.
The Byzantine Generals Problem

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:
  termination: eventually each correct process sets its decision variable.
  agreement: the decision value of all correct processes is the same.

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.

Assume that the previous problems could be solved, yielding the following decision variables

  \( \text{Consensus} : C, (v_1, \ldots, v_n) \) returns the decision value of \( p_i \).
  \( \text{Byzantine Generals} : BG, (k, v) \) returns the decision value of \( p_i \), where \( p_k \) is the commander which proposes the value \( v \).
  \( \text{Interactive Consistency} : IC, (v_1, \ldots, v_n) \) returns the k-th value in the decision vector of \( p_i \), where \( v_1, \ldots, v_n \) are the values that the processes proposed.

Possibilities to derive solutions out of the solutions to other problems

  solution to IC from BG
  run BG n times, once with each \( p_i \) acting as commander.
  \( IC, (v_1, \ldots, v_n) \) returns \( BG, (k, v) \) with \( (i, k = 1, \ldots, n) \).

  solution to C from IC
  run IC to produce a vector of values at each process.
  apply an appropriate function on the vector's values to derive a single value.
  \( C, (v_1, \ldots, v_n) = \text{majority}(IC, (v_1, \ldots, v_n)[1], \ldots, IC, (v_1, \ldots, v_n)[n]) \).

  solution to BG from C
  commander \( p_k \) sends its proposed value \( v \) to itself and each of the remaining processes.
  all processes run C with the values \( v_1, \ldots, v_n \) that they receive.
Relationship between these Problems

**Interactive Consistency**: IC(\(v_1, ..., v_n, s\)) returns the \(k\)-th value in the decision vector of \(p\), where \(v_1, ..., v_n\) are the values that the processes proposed.

Possibilities to derive solutions out of the solutions to other problems:

- **solution to IC from BG**
  - run \(BG\) \(n\) times, once with each \(p\) acting as commander.
  - \(IC(v_1, ..., v_n, s) = BG(k, v_n)\) with \(i = 1, ..., n\)

- **solution to C from IC**
  - run IC to produce a vector of values at each process.
  - apply an appropriate function on the vector's values to derive a single value.
  - \(C_i(v_1, ..., v_n, s) = \text{majority}(IC(v_1, ..., v_n, s), ..., IC(v_1, ..., v_n, s)[n])\)

- **solution to BG from C**
  - commander \(p_0\) sends its proposed value \(v\) to itself and each of the remaining processes.
  - all processes run \(C\) with the values \(v_1, ..., v_n\) that they receive.
  - derive \(BG(k, v) = C_i(v_1, ..., v_n)\) with \(i = 1, ..., n\)

Termination, agreement and integrity preserved in each case.

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

The algorithm proceeds in \(f+1\) rounds in order to reach consensus.

- at the end of \(f+1\) rounds, all surviving processes are in position to agree.

Algorithm for process \(p \in \text{consensus group } G\)

- **On initialization**
  - \(\text{values}_i(0) = \{\}\)

- **in round** \(r\) \((1 \leq r \leq f+1)\)
  - \(BG\)-multicast(g, values) \(\{r\} = \text{values}\) \(\{r-1\}\)
    - send only values that have not been sent.
  - \(\text{values}_i(r+1) = \text{values}_i(r)\)
  - while (in round \(r\)) {
    - on \(BG\)-deliver \(v_j\) from some \(p_j\)
      - \(\text{values}_i[r+1] = \text{values}_i[r+1] \cup v_j\)
  - }

After \((f+1)\) rounds

- assign \(d_i = \text{minimum}(\text{values}_i[f+1])\)

**Distributed Consensus**

A 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, 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.

**Consensus Problem**

- Consensus in synchronous Networks

**Authentication service Kerberos**

**Definition**: Authentication means verifying the identities of the communicating partners to one another in a secure manner.

Kerberos has been developed at the MIT as a part of the distributed framework Athena. Kerberos is part of a variety of authentication components. The Kerberos authentication protocol is based on the protocol by Neeman and Schröder.

**Introduction**

- **This course provides only a short introduction to Kerberos** (for further information, consult the Kerberos Web-Site)

**Motivation**

**Security objects of Kerberos**

**Authentication process scenario**

- **Authentication process scenario**

- **Authentication process scenario**

- **Authentication process scenario**

- **Authentication process scenario**
Kerberos assumes the following components:

- Client C
- Server S
- Key distribution center KDC
- Ticket granting service TGS

**Goal of Kerberos**

A client C requests the service of the server S. KDC and TGS are supposed to guarantee the secrecy and authenticity requirements:

1. **KDC manages the secret keys of the registered components.**
2. **Within a session TGS provides the client C with tickets for authentication with servers of the distributed system.**

**Authentication process scenario**

**Graphical representation**

**Description of exchanged messages**

**Problems with Kerberos**

Manipulation of local computer clocks to circumvent the validity time of tickets, i.e., synchronization of clocks in distributed systems must be authorized and authenticated.

**Example: user login with Kerberos**
1. login program of the workstation W sends user name N to KDC.
2. if the user is known, then KDC sends a session key $K_{U}$ encrypted with the user password, as well as a TGS ticket.
3. login program requests the password from the user and decrypts the session key $K_{U}$ using the password; if the password was correct, then the decrypted session key $K_{U}$ and the session key $K_{a}$ within the TGS ticket are identical.
4. the TGS ticket can be removed from the main memory because for further communication, only $K_{a}$ and the TGS ticket are used; both are used to authenticate the user at TGS if the user requests a server S.
5. establish a user login session on workstation W.