Script generated by TTT

Title: Distributed_Applications (19.06.2012)

Date: Tue Jun 19 14:30:08 CEST 2012

Duration: 92:42 min

Pages: 31

Group communication

Introduction

Group communication facilities the interaction between groups of processes.

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

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| , with j 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.

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

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, ..., 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 $z_j = (z_{ji})_{i \in \{1,...,n\}}$ 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.

$z_{jj} := z_{jj} + 1$; 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): $\forall\ k \neq i$: $z_{ik} \leq z_{jk}$.

cbcast protocol

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.

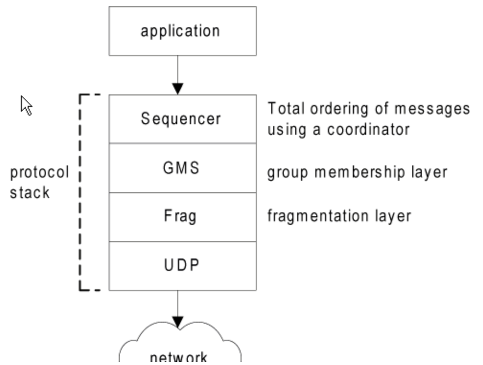
management of group membership.

Programming Interface of JGroups

groups are identified via channels.

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

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



JGroups

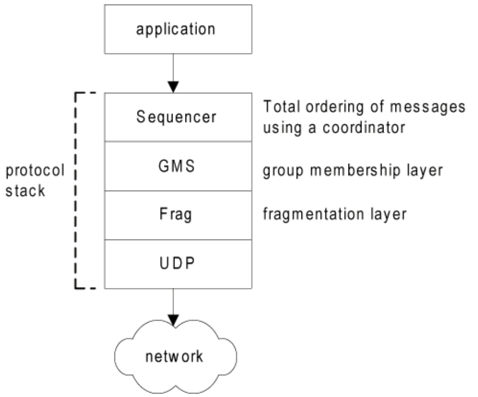
management of group membership.

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 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();
```


Group communication

Introduction

Group communication facilities the interaction between groups of processes.

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

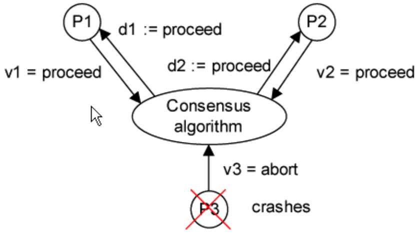
Consensus Problem

Consensus in synchronous Networks

Consensus Problem

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

- p_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](#)

[Relationship between these Problems](#)

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

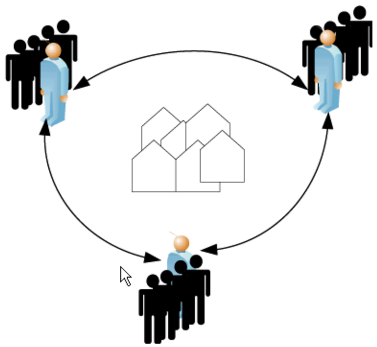
- ***termination*** : eventually, each correct process sets its decision variable
- ***agreement*** : the decision variable of all correct processes is the same in the decided state.
- ***integrity*** : if the correct processes all proposed the same value, then any correct process has chosen that value 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 `majority(v1 , ..., vn )` ,
  - 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

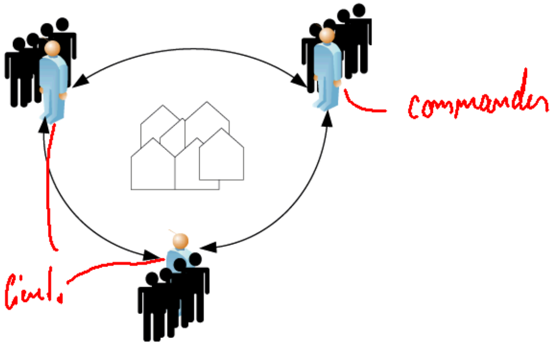


difference to consensus problem: one process supplies a value that others have to agree on

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



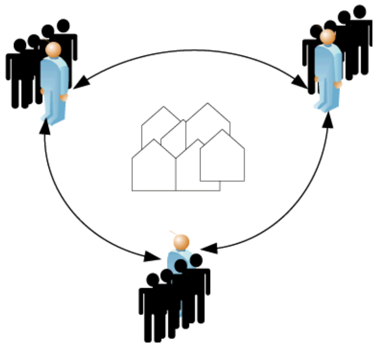
difference to consensus problem: one process supplies a value that others have to agree on

properties:

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

The Byzantine Generals Problem

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.
- integrity: if the commander is correct, then all processes decide on the value that the commander proposes.

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

- ***Consensus*** : $C_i(v_1, \dots, v_n)$ returns the decision value of p_i
- ***Byzantine Generals*** : $BG_i(k, v)$ returns the decision value of p_i where p_k is the commander which proposes the value v
- ***Interactive Consistency*** : $IC_i(v_1, \dots, v_n)[k]$ returns the k-th value in the decision vector of p_i where $v_1, \dots, v_n$ 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
- $IC_i(v_1, \dots, v_n)[k] = BG_i(k, v_k)$ with $(i, k = 1, \dots, n)$

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
- $C_i(v_1, \dots, v_n) = \text{majority}(IC_i(v_1, \dots, v_n)[1], \dots, IC_i(v_1, \dots, v_n)[n])$

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 $v_1, \dots, v_n$ that they receive

Relationship between these Problems

proposes the value v

Interactive Consistency : $IC_i (v_1, \dots, v_n)[k]$ returns the k-th value in the decision vector of p_i where $v_1, \dots, v_n$ 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
  - $IC_i (v_1, \dots, v_n)[k] = BG_i (k, v_k)$ with (i, k = 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
  - $C_i (v_1, \dots, v_n) = majority(IC_i (v_1, \dots, v_n)[1], \dots, IC_i (v_1, \dots, v_n)[n])$
- 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 $v_1, \dots, v_n$ that they receive
  - derive $BG_i (k, v) = C_i (v_1, \dots, v_n)$ with i = 1, .., n
- termination, agreement and integrity preserved in each case.

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

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 $p_i \in$ concensus group g

```
On initialization
    values_i (1) := {v_i }; values_i (0) := {};

in round r (1 ≤ r ≤ f+1)
    B-multicast(g, values_i (r)-values_i (r-1));
       //send only values that have not been sent
    values_i (r+1) := values_i (r)
    while (in round r) {
       On B-deliver(v_j ) from some p_j
          values_i (r+1) := values_i (r+1) ∪ v_j
    }

After (f+1) rounds
    assign d_i = minimum (values_i (f+1))
```


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 part of the distributed framework Athena. Kerberos ist part of a variety authentication components. The Kerberos authentication protocol is based on the protocol by Needham 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

Kerberos assumes the following components

Client C,

Server S,

Key distribution center KDC, and

Ticket granting service TGS.

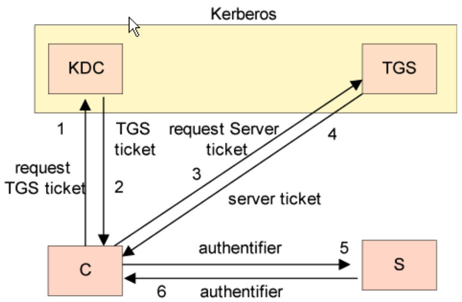
Goal of Kerberos

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.

Kerberos enables authentication through the following three security objects.

1. ***TGS ticket*** : issued by KDC to the client C for presentation at TGS.
2. ***Authentifier*** : generated by client C; it identifies the client and guarantees the validity for the communication with server S.
3. ***Session key*** : generated by Kerberos for the communication between client C and server S.



Animation Kerberos

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

Message 3: C to TGS

C → TGS with information	(C, T_C) $_{K[C,tgs]}$
	ticket(C, TGS) $_{K[tgs]}$
	S

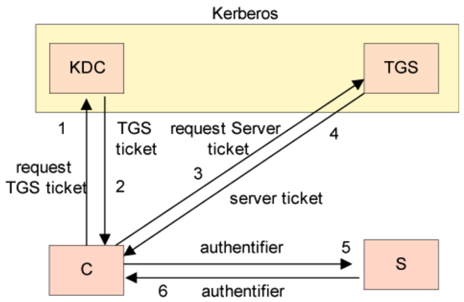
TGS determines a random session key $K_{C,S}$, if

TGS ticket is still valid,

T_C is current, and

field C matches (of the first parameter and of the ticket).

Graphical representation



[Animation 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_N 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_N using the password; if the password was correct, then the decrypted session key K_N and the session key K_N within the TGS ticket are identical.
4. the password can be removed from the main memory because for further communication, only K_N 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.

Distributed Applications - Verteilte Anwendungen

- Prof. J. Schlichter
  - Lehrstuhl für Angewandte Informatik / Kooperative Systeme, Fakultät für Informatik, TU München
  - Boltzmannstr. 3, 85748 Garching

    Email: schlichter@in.tum.de

    Tel.: 089-289 18654

    URL: http://www11.in.tum.de/