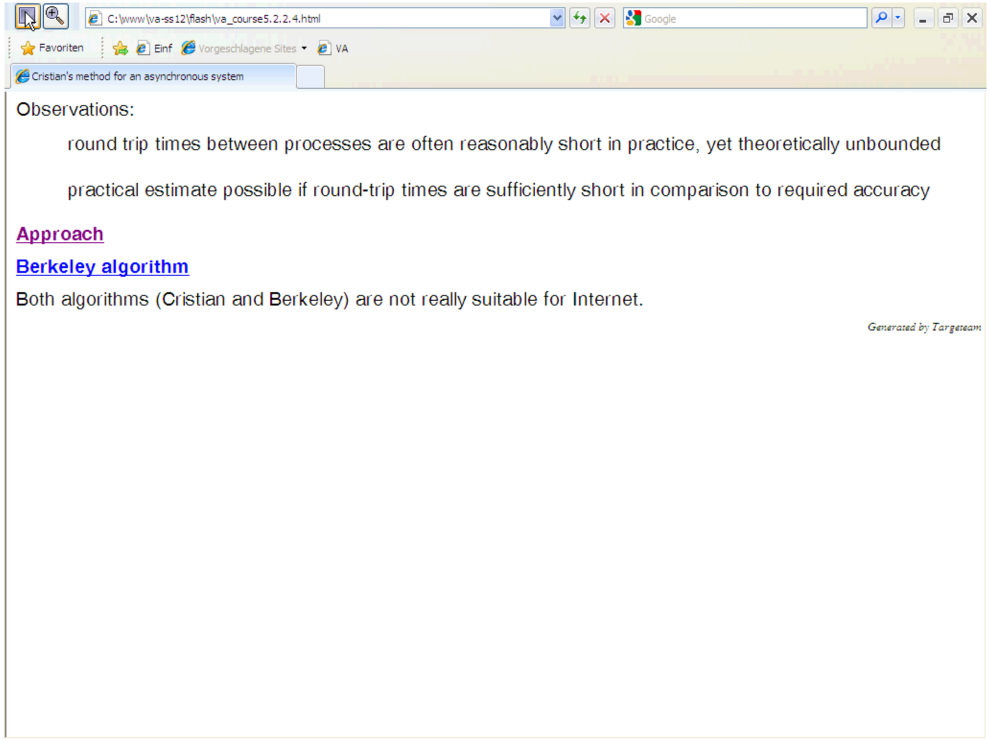
Script **generated by TTT**

Title: Distributed_Applications (05.06.2012)

Date: Tue Jun 05 14:33:03 CEST 2012

Duration: 89:01 min

Pages: 25



Observations:

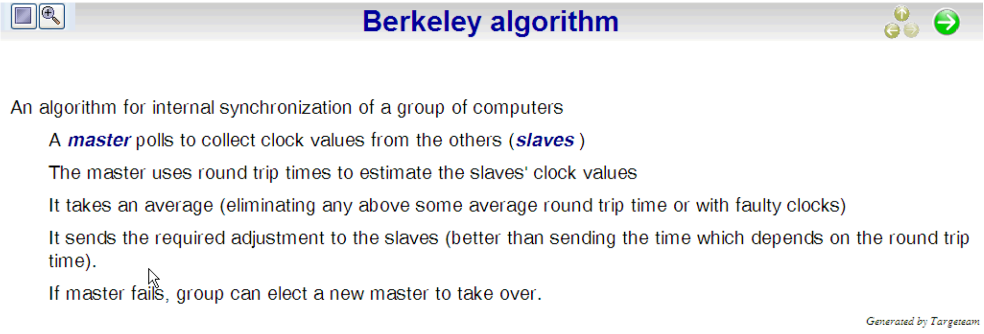
- round trip times between processes are often reasonably short in practice, yet theoretically unbounded
- practical estimate possible if round-trip times are sufficiently short in comparison to required accuracy

Approach

Berkeley algorithm

Both algorithms (Cristian and Berkeley) are not really suitable for Internet.

Generated by Targeteam

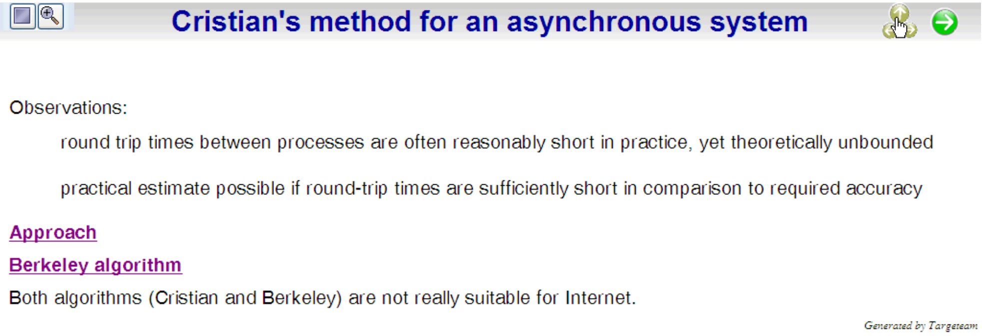


Berkeley algorithm

An algorithm for internal synchronization of a group of computers

- A ***master*** polls to collect clock values from the others (***slaves***)
- The master uses round trip times to estimate the slaves' clock values
- It takes an average (eliminating any above some average round trip time or with faulty clocks)
- It sends the required adjustment to the slaves (better than sending the time which depends on the round trip time).
- If master fails, group can elect a new master to take over.

Generated by Targeteam



Cristian's method for an asynchronous system

Observations:

- round trip times between processes are often reasonably short in practice, yet theoretically unbounded
- practical estimate possible if round-trip times are sufficiently short in comparison to required accuracy

Approach

Berkeley algorithm

Both algorithms (Cristian and Berkeley) are not really suitable for Internet.

Generated by Targeteam

The synchronization subnet can reconfigure if failures occur, e.g.

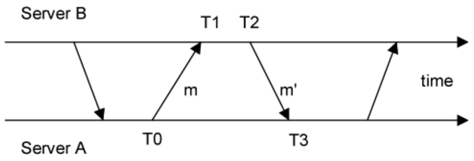
- a primary that loses its UTC source can become a secondary
- a secondary that loses its primary can use another primary

Modes of synchronization

- Multicast:
  - A server within a high speed LAN multicasts time to others which set clocks assuming some delay (not very accurate)
- Procedure call:
  - A server accepts requests from other computers (like Cristian's algorithm). Higher accuracy. Useful if no hardware multicast.
- Symmetric:
  - Pairs of servers exchange messages containing time information
  - Used where very high accuracies are needed (e.g. for higher levels)

Messages between a pair of NTP peers

All modes use UDP transport protocol for the message exchange



- Each message bears timestamps of recent events:
  - Local times of Send and Receive of previous message
  - Local time of Send of current message
- Recipient (Server A) notes the time of receipt T3 (we have T0, T1, T2, T3).
- In symmetric mode there can be a non-negligible delay between messages

Accuracy of NTP

For each pair of messages between two servers, NTP estimates an offset o, between the two clocks and a delay d_i (total transmission time for the two messages m and m', which take t and t')

$T_{i-2} = T_{i-3} + t + o$ and $T_i = T_{i-1} + t' - o$

This gives us the delay (by adding the equations)

$d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1}$

Also the offset (by subtracting the equations)

$o = o_i + (t' - t)/2$, where $o_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2$

Estimate of offset

- Using the fact that t, t' > 0 it can be shown that
  - $o_i - d_i/2 \leq o \leq o_i + d_i/2$.
  - Thus o_i is an estimate of the offset and d_i is a measure of the accuracy
- NTP servers filter pairs $<o_i , d_i>$
  - retains the 8 most recent pairs
  - estimates the offset o
- NTP applies peer-selection to identify peer for reliability estimate.
- Accuracy
  - over Internet: tens of ms
  - over a LAN: 1 ms

NTP - synchronization of servers

The synchronization subnet can reconfigure if failures occur, e.g.

- a primary that loses its UTC source can become a secondary
- a secondary that loses its primary can use another primary

Modes of synchronization

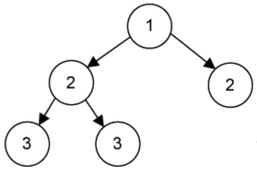
- Multicast:
  - A server within a high speed LAN multicasts time to others which set clocks assuming some delay (not very accurate)
- Procedure call:
  - A server accepts requests from other computers (like Cristian's algorithm). Higher accuracy. Useful if no hardware multicast.
- Symmetric:
  - Pairs of servers exchange messages containing time information
  - Used where very high accuracies are needed (e.g. for higher levels)

Network Time Protocol (NTP)

Cristian and Berkeley algorithm are intended for the Intranet.

NTP defines an architecture for a time service and a protocol to distribute time information over the Internet.

NTP synchronizes clients to UTC



Primary servers are connected to UTC sources

Secondary servers are synchronized to primary servers

Synchronization subnet - lowest level servers in users' computers

NTP - synchronization of servers

Messages between a pair of NTP peers

Accuracy of NTP

Basic mechanisms for distributed applications

Issues

The following section discusses several important basic issues of distributed applications.

Data representation in heterogeneous environments.

Discussion of an execution model for distributed applications.

What is the appropriate error handling?

What are the characteristics of distributed transactions?

What are the basic aspects of group communication (e.g. algorithms used by ISIS) ?

How are messages propagated and delivered within a process group in order to maintain a consistent state?

External data representation

Time

Distributed execution model

Failure handling in distributed applications

Distributed transactions

Group communication

Distributed Consensus

Authentication service Kerberos

Components of a distributed application communicate through messages causing events in the components.

The component execution is characterized by three classes of events:

internal events (e.g. the execution of an operation).

message sending.

message receiption.

in some cases distinction between message reception and message delivery to application as separate events.

The execution of a component TK creates a sequence of events e_1 , ..., e_n ,...

The execution of the component TK_i is defined by (E_i , $\rightarrow_i$) with:

E_i is the set of events created by TK_i execution

$\rightarrow_i$ defines a total order of the events of TK_i

The relation $\rightarrow_{msg}$ defines a causal relationship for the message exchange:

send(m) $\rightarrow_{msg}$ receive(m), i.e. sending of the message m must take place prior to receiving m.

There are the following interpretations

a → b, i.e. a before b; b causally depends on a.

a || b, i.e. a and b are concurrent events.

Rules for "happened-before" after Lamport

In order to guarantee consistent states among the communicating components, the messages must be delivered in the correct order. The happened-before relation after Lamport may help to determine a message sequence for a distributed application.

The following rules apply:

Events within a component are ordered with respect to the before-relation , i.e. $a \rightarrow b$

if "a" is a send event of component TK1, and "b" the respective receive event of component TK2, then $a \rightarrow b$;

if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$;

if $\neg (a \rightarrow b)$ and $\neg (b \rightarrow a)$, then $a \| b$; i.e. a and b are concurrent, i.e. they are not ordered.

Utilization of logical clocks to determine the event sequence.

Let

T: a set of timestamps

C: $E \rightarrow T$ a mapping which assigns a timestamp to each event

$a \rightarrow b \Rightarrow C(a) < C(b)$

If the reverse deduction is valid, too (⇔), then the clock is called strictly consistent.

Ordering by logical clocks

Each component manages the following information:

its local logical clock lc; lc determines the local progress with respect to occuring events.

its view on the global logical clock gc; the value of the local clock is determined according to the value of the global clock.

There exist functions for updating logical clocks in order to maintain consistency; the following two rules apply.

Rules

- Rule R1 specifies the update of the local clock lc when events occur.
- Rule R2 specifies the update of the global clock gc.
  1. ***Sending event*** : determine the current value of the local clock and attach it to the message.
  2. ***Receiving event*** : the received clock value (attached to the message) is used to update the view on the global clock.

Distributed execution model

Events

- Classes of events
- Rules for "happened-before" after Lamport

Ordering by logical clocks

Logical clocks based on scalar values

- Description
- Example

Logical clocks based on vectors

- Description
- Example for vector clocks
- Characteristics of vector clocks

Description

The clock value is specified by positive integer numbers.

the local clock lc and the view on global clock (gc) are both represented by the counter C.

Execution of R1

prior to event execution, C is updated: C := C + d.

Execution of R2

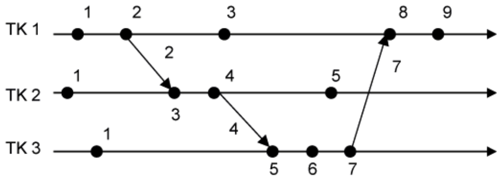
after receiving a message with timestamp C_{msg} (the timestamp is part of the message), the following actions are performed

$C := \max(C, C_{msg})$

execute R1

deliver message to the application component

Example



The scalar clock mechanism defines a partial ordering on the occurring events.

scalar clocks are ***not strictly consistent*** , i.e.

the following is not true: $C(a) < C(b) \Leftrightarrow a \rightarrow b$

Events

- Classes of events
- Rules for "happened-before" after Lamport

Ordering by logical clocks

Logical clocks based on scalar values

- Description
- Example

Logical clocks based on vectors

- Description
- Example for vector clocks
- Characteristics of vector clocks

The time is represented by n-dimensional vectors with positive integers. Each component TK_i manages its own vector vt_i [1....n]. The dimension n is determined by the number of components of the distributed application.

- vt_i [i] is the local logical clock of TK_i .
- vt_i [k] is the view of TK_i on the local logical clock of TK_k ; it determines what TK_i knows about the progress of TK_k
- Example: vt_i [k] = y, i.e. according to the view of TK_i , TK_k has advanced to the state y, i.e. up to the event y.
- the vector vt_i [1....n] represents the view of TK_i on the global time (i.e. the global execution progress for all components).

Execution of R1

$vt_i [i] := vt_i [i] + d$

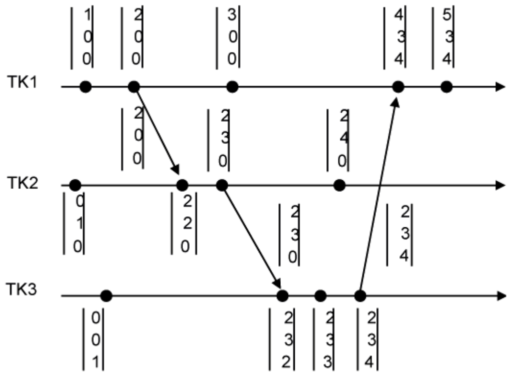
Execution of R2

After receiving a message with vector vt from another component, the following actions are performed at the component TK_i

- update the logical global time: $1 \leq k \leq n$: $vt_i [k] := \max (vt_i [k], vt[k])$.
- execute R1
- deliver message to the application process of component TK_i

---

Example for vector clocks



optimization: omit vector timestamps when sending a burst of multicasts

⇒ missing timestamp means: use values of previous vector timestamp and increment the sender's field only.

Events

- Classes of events
- Rules for "happened-before" after Lamport

Ordering by logical clocks

Logical clocks based on scalar values

- Description
- Example

Logical clocks based on vectors

- Description
- Example for vector clocks
- Characteristics of vector clocks

Distributed execution model

Events

- Classes of events
- Rules for "happened-before" after Lamport

Ordering by logical clocks

Logical clocks based on scalar values

- Description
- Example

Logical clocks based on vectors

- Description
- Example for vector clocks
- Characteristics of vector clocks

Basic mechanisms for distributed applications

Issues

The following section discusses several important basic issues of distributed applications.

- Data representation in heterogeneous environments.
- Discussion of an execution model for distributed applications.
- What is the appropriate error handling?
- What are the characteristics of distributed transactions?
- What are the basic aspects of group communication (e.g. algorithms used by ISIS) ?
- How are messages propagated and delivered within a process group in order to maintain a consistent state?

External data representation

Time

Distributed execution model

Failure handling in distributed applications

Distributed transactions

Group communication

Distributed Consensus

Authentication service Kerberos

Motivation

Steps for testing a distributed application

Debugging of distributed applications

Approaches of distributed debugging

Failures in a local application

- handled through a programmer-defined exception-handling routine.
- no handling.

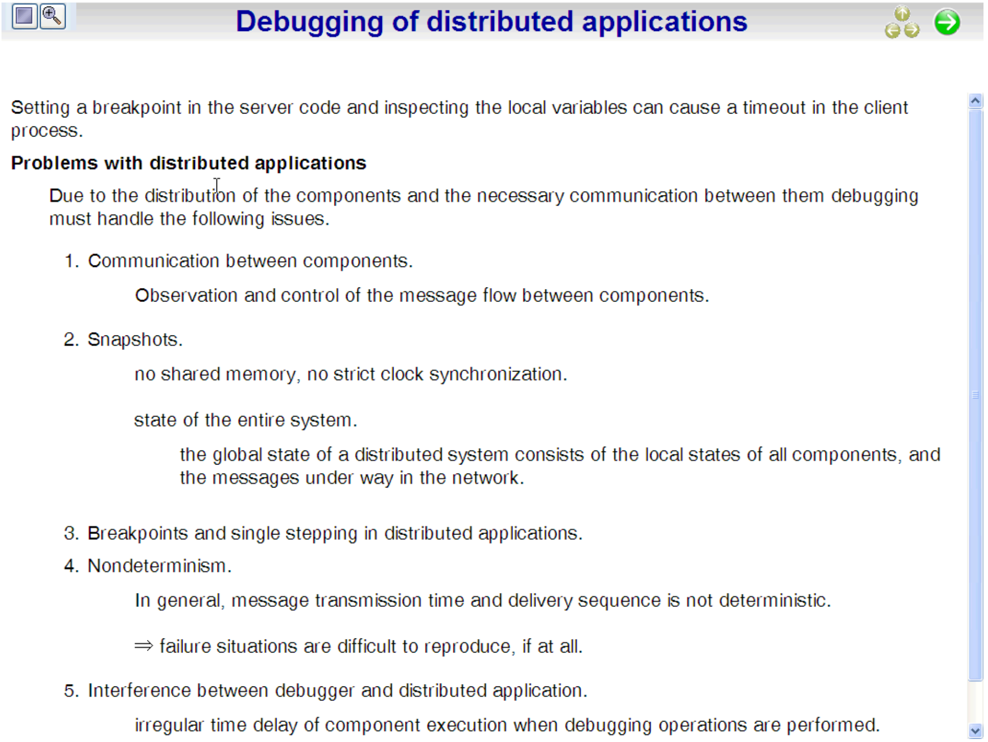
Failures in a distributed application. Failures may be caused by

- communication link failures.
- crashes of machines hosting individual subsystems of the distributed application.
  - The client crashes ⇒ the server waits for RPC calls of the crashed client; server does not free reserved resources.
  - The server crashes ⇒ client cannot connect to the server.
- byzantine failures: processes fail, but may still respond to environment with arbitrary, erratic behavior (e.g., send false acknowledgements, etc.)
- failure-prone RPC-interfaces.
- bugs in the distributed subsystems themselves.

Failure handling in distributed applications

- Motivation
- Steps for testing a distributed application
- Debugging of distributed applications
- Approaches of distributed debugging

Generated by Targeteam



Debugging of distributed applications

Setting a breakpoint in the server code and inspecting the local variables can cause a timeout in the client process.

Problems with distributed applications

Due to the distribution of the components and the necessary communication between them debugging must handle the following issues.

1. Communication between components.

   Observation and control of the message flow between components.

2. Snapshots.

   no shared memory, no strict clock synchronization.

   state of the entire system.

      the global state of a distributed system consists of the local states of all components, and the messages under way in the network.

3. Breakpoints and single stepping in distributed applications.
4. Nondeterminism.

   In general, message transmission time and delivery sequence is not deterministic.

   ⇒ failure situations are difficult to reproduce, if at all.

5. Interference between debugger and distributed application.

   irregular time delay of component execution when debugging operations are performed.