Script generated by TTT

Title: Distributed_Applications (04.06.2013)

Date: Tue Jun 04 14:30:48 CEST 2013

Duration: 89:10 min

Pages: 27

Synchronizing physical clocks

physical clocks are used to compute the current time in order to timestamp events, such as

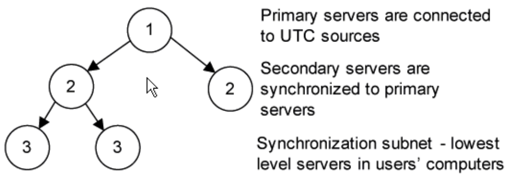
- modification date of a file
- time of an e-commerce transaction for auditing purposes

External - internal synchronization
Clock correctness
Synchronization in a synchronous system
Cristian's method for an asynchronous system
Network Time Protocol (NTP)
Precision Time Protocol (PTP)

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.
- use of 64 bit timestamps.
- NTP synchronizes clients to UTC.



NTP - synchronization of servers
Messages between a pair of NTP peers
Accuracy of NTP

Synchronizing physical clocks

physical clocks are used to compute the current time in order to timestamp events, such as

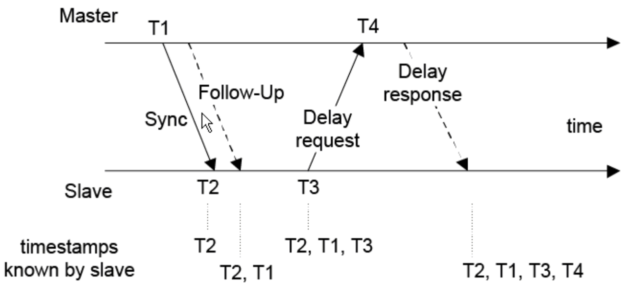
- modification date of a file
- time of an e-commerce transaction for auditing purposes

External - internal synchronization
Clock correctness
Synchronization in a synchronous system
Cristian's method for an asynchronous system
Network Time Protocol (NTP)
Precision Time Protocol (PTP)

Generated by Targeteam

Synchronization Message Exchange

PTP defines a master-slave hierarchy.



- master periodically transmits a Sync message using UDP multicast.
- the follow-up message includes the actual time the Sync message left the master.
- slave initiates exchange with master to determine round-trip delay: delay-request and delay-response messages.
- if d is the transit time for the Sync message and o the constant offset between master and slave clocks
  - $T2 - T1 = o + d$ and $T4 - T3 = -o + d$
  - $o = (T2 - T1 - T4 + T3)/2$

Precision Time Protocol (PTP)

PTP is a protocol to synchronize clocks throughout a computer network

NTP is typically used over the Internet handling large amounts of nondeterministic delays; accuracy in the ms-range.

PTP is designed for LANs achieving clock accuracy in the sub-microsecond range.

Synchronization Message Exchange

- PTP supports an algorithm to perform a distributed selection of the best candidate clock.

Time

Time is an important and interesting issue in distributed systems

- We need to measure time accurately:
  - to know the time an event occurred at a computer
  - to do this we need to synchronize its clock with an authoritative external clock
- Algorithms for clock synchronization useful for
  - concurrency control based on timestamp ordering
  - authenticity of requests e.g. in Kerberos

Three notions of time:

- time seen by an external observer ⇒ global clock of perfect accuracy.
  - However, there is ***no global clock in a distributed system***
- time seen on clocks of individual processes.
- logical notion of time: event a occurs before event b.

Introduction

Synchronizing physical 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

Classes of events

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 reception.
  - 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

- $a \rightarrow b$, 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 ($\Leftrightarrow$), 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.

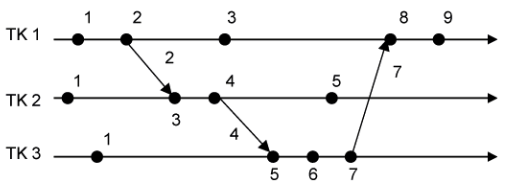
There exist functions for updating logical clocks in order to maintain consistency; the following two rules apply.

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

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: $C(a) < C(b) \Leftrightarrow a \rightarrow b$

Description

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

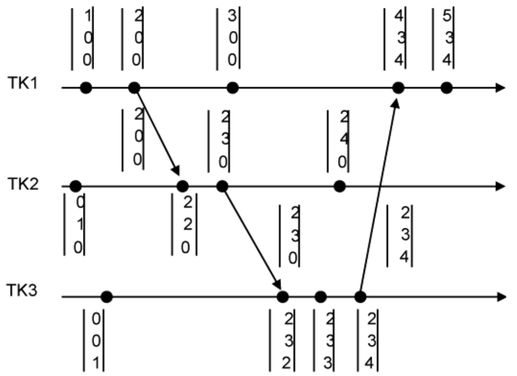
After receiving a message with vector vt from another component, the following actions are performed at the component TK_i

update the logical global time: $1 \le k \le n$: $vt_i [k] := \max (vt_i [k], vt[k])$.

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

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

Description

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: $1 \le k \le n$: $vt_i [k] := \max (vt_i [k], vt[k])$.

execute R1

deliver message to the application process of component TK_i

Generated by Targeteam

Events

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

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?

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.

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.

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.

[Motivation](#)
[Steps for testing a distributed application](#)
[Debugging of distributed applications](#)
[Approaches of distributed debugging](#)

Approaches of distributed debugging

focus on the send/receive events caused by the message exchange and less on the internal component operations.

Monitoring the communication between components

Global breakpoint

- **Approach**
- **Causally distributed breakpoint**
- Example of a distributed debugger:
  - IBM IDEBUG: a multilanguage, multiplatform debugger with remote debug capabilities.

Approaches of distributed debugging

focus on the send/receive events caused by the message exchange and less on the internal component operations.

Monitoring the communication between components

Global breakpoint

- **Approach**
- **Causally distributed breakpoint**
- Example of a distributed debugger:
  - IBM IDEBUG: a multilanguage, multiplatform debugger with remote debug capabilities.

Generated by Targeteam

Distributed transactions

Distributed transactions are an important paradigm for designing reliable and fault tolerant distributed applications; particularly those distributed applications which access shared data concurrently.

General observations

Isolation

Atomicity and persistence

Two-phase commit protocol (2PC)

Distributed Deadlock

General observations

Several requests to remote servers (e.g. RPC calls) may be bundled into a transaction.

```
begin-transaction
    callrpc (OP1 , ...., )
    .....
    callrpc (OPn , ...., )
end-transaction
```

A distributed transaction involves activities on multiple servers, i.e. within a transaction, services of several servers are utilized.

Transactions satisfy the ***ACID*** property: Atomicity, Consistency, Isolation, Durability.

1. ***atomicity*** : either all operations or no operation of the transaction is executed, i.e. the transaction is a success (commit) or else has no consequence (abort).
2. ***durability*** : the results of the transaction are persistent, even if afterwards a system failure occurs.
3. ***isolation*** : a not yet completed transaction does not influence other transactions; the effect of several concurrent transactions looks like as if they have been executed in sequence.
4. ***consistency*** : a transaction transfers the system from a consistent state to a new consistent state.