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.

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
**Synchronization Message Exchange**

PTP defines a master-slave hierarchy.

- **Master**
  - T1
  - Follow-Up
  - Delayed Request
  - Delay Response

- **Slave**
  - T2
  - T3

Timestamps known by slave:
- T2, T1
- T2, T1, T3
- T2, T1, T3, T4

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 \( \delta \) the constant offset between master and slave clocks:

\[
T2 - T1 = \delta + d \quad \text{and} \quad T4 - T3 = -\delta + d
\]

\[
\delta = \frac{(T2 - T1 - T4 + T3)}{2}
\]

\*Generated by Orgpam*
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 **e₁, ..., eₙ, ...**. The execution of the component **TK** is defined by \((E, \rightarrow)\) with:

- \(E\) is the set of events created by **TK**.
- \(\rightarrow\) defines a total order of the events of **TK**.

The relation \(\rightarrow\) defines a causal relationship for the message exchange:

- \(\text{send}(m) \rightarrow \text{receive}(m)\), i.e., sending of the message **m** must take place prior to receiving **m**.

There are the following interpretations:

- \(a \rightarrow b\), i.e., **b** causally depends on **a**.
- \(a \parallel b\), i.e., **a** and **b** are concurrent events.

### Ordering by logical clocks

Each component manages the following information:

- Its local logical clock **lc**; it determines the local progress with respect to occurring 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.
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: \( C(a) < C(b) \Rightarrow a \rightarrow b \)

---

**Example for vector clocks**

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

\( v_i[0] \) is the local logical clock of \( TK_i \).

\( v_i[k] \) 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: \( v_i[k] = y \), i.e., according to the view of \( TK_i \), \( TK_i \) has advanced to the state \( y \), i.e., up to the event \( y \).

the vector \( v_i[1,...,n] \) represents the view of \( TK_i \) on the global time (i.e., the global execution progress for all components).

**Execution of R1**

\( v_i[i] := v_i[i] + d \)

**Execution of R2**

After receiving a message with vector \( v_i \) from another component, the following actions are performed at the component \( TK_i \):

- update the logical global time: \( 1 \leq k \leq n : v_i[k] := \min(x \cdot v_i[k], v_i[k]) \).

- execute R1

- deliver message to the application process of component \( TK_i \)

---

**Example**

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

\( v_i[0] \) is the local logical clock of \( TK_i \).

\( v_i[k] \) 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: \( v_i[k] = y \), i.e., according to the view of \( TK_i \), \( TK_i \) has advanced to the state \( y \), i.e., up to the event \( y \).

the vector \( v_i[1,...,n] \) represents the view of \( TK_i \) on the global time (i.e., the global execution progress for all components).

**Execution of R1**

\( v_i[i] := v_i[i] + d \)

**Execution of R2**

After receiving a message with vector \( v_i \) from another component, the following actions are performed at the component \( TK_i \):

- update the logical global time: \( 1 \leq k \leq n : v_i[k] := \min(x \cdot v_i[k], v_i[k]) \).

- execute R1

- deliver message to the application process of component \( TK_i \)
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
- 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 acknowledgments, etc.)
- Failure-prone RPC-interfaces;
  - bugs in the distributed subsystems themselves.
Motivation

Steps for testing a distributed application

Debugging of distributed applications

Approaches of distributed debugging

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 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.

   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.
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, multiprocessor debugger with remote debug capabilities.

---

Distributed transactions

**General observations**

- **Isolation**
- **Atomicity and persistence**
- **Two-phase commit protocol (2PC)**

**Distributed Deadlock**

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

**Begin transaction**

\[
\begin{align*}
\text{collrpc } & \langle \text{OPs} \rangle, \ldots, \langle \text{OPs} \rangle \\
\end{align*}
\]

**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 (abortion).
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.

---

General observations

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