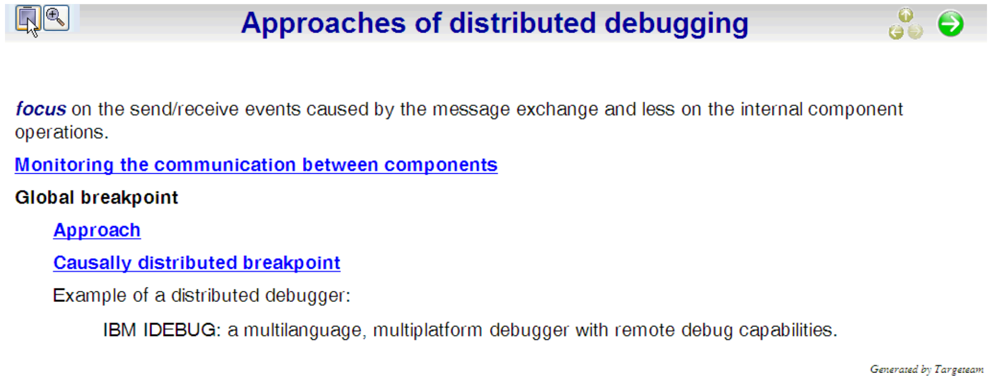
**Script** **generated by TTT**

| | |
|---|---|
| Title: | Distributed_Applications (11.06.2012) |
| Date: | Mon Jun 11 09:15:27 CEST 2012 |
| Duration: | 45:55 min |
| Pages: | 15 |



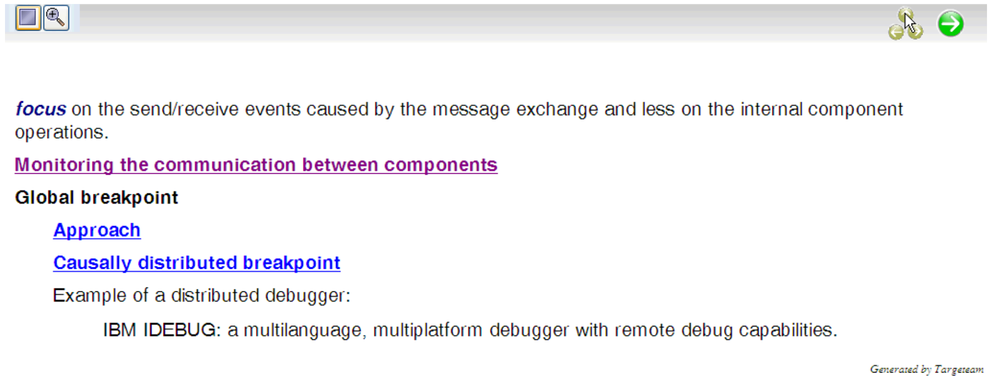
## Approaches of distributed debugging

***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.

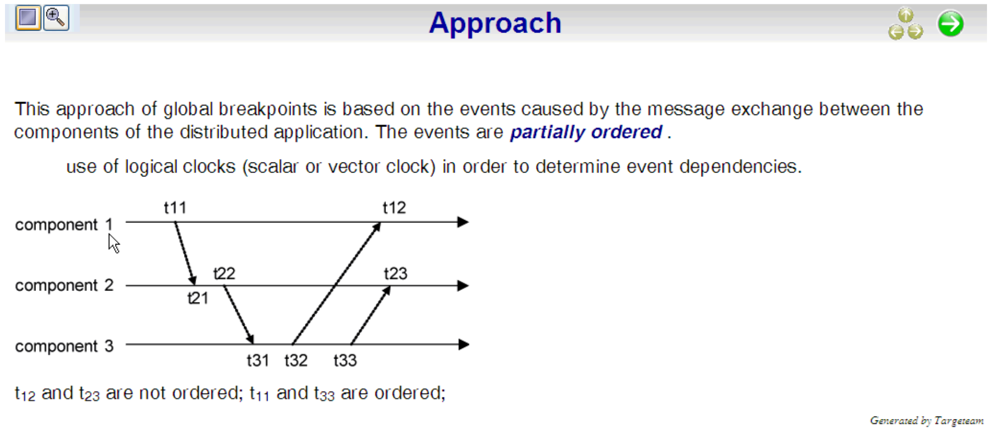


***focus*** on the send/receive events caused by the message exchange and less on the internal component operations.

Monitoring the communication between components

**Global breakpoint**

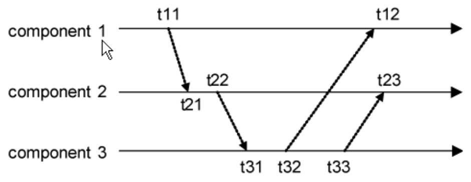
- Approach
- Causally distributed breakpoint
- Example of a distributed debugger:
  - IBM IDEBUG: a multilanguage, multiplatform debugger with remote debug capabilities.



## Approach

This approach of global breakpoints is based on the events caused by the message exchange between the components of the distributed application. The events are ***partially ordered*** .

use of logical clocks (scalar or vector clock) in order to determine event dependencies.



$t_{12}$ and $t_{23}$ are not ordered; $t_{11}$ and $t_{33}$ are ordered;

*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.

## Failure handling in distributed applications

Motivation

Steps for testing a distributed application

Debugging of distributed applications

Approaches of distributed debugging

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

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.

Isolation refers to the serializability of transactions. All involved servers are responsible for the serialization of distributed transactions. Example:

let U, T be distributed transactions accessing shared data on the two servers R and S.

if the transactions at server R are successfully executed in the sequence U before T, then the same commit sequence must apply to server S.

Timestamp ordering

Locking

**Optimistic concurrency control**

if conflicts are rare, optimistic concurrency control may be useful: no additional coordination necessary during transaction execution.

The check for access conflicts occurs when transactions are ready to "commit";

In a single server transaction, the server issues a unique timestamp to each transaction when it starts.

In a distributed transaction each server is able to issue globally unique timestamps.

for distributed transactions, the timestamp is the pair

(local timestamp, server-ID)

The local timestamp refers to the first server which issued the transaction timestamp.

Assume: timestamp(trans) = $t_{trans}$ and timestamp(obj) = $t_{obj}$

transaction trans accesses object obj

```
if (t_trans < t_obj ) then abort(trans) else access obj;
```

## Isolation

Isolation refers to the serializability of transactions. All involved servers are responsible for the serialization of distributed transactions. Example:

let U, T be distributed transactions accessing shared data on the two servers R and S.

if the transactions at server R are successfully executed in the sequence U before T, then the same commit sequence must apply to server S.

Timestamp ordering

Locking

**Optimistic concurrency control**

if conflicts are rare, optimistic concurrency control may be useful: no additional coordination necessary during transaction execution.

The check for access conflicts occurs when transactions are ready to "commit";

In a single server transaction, the server issues a unique timestamp to each transaction when it starts.

In a distributed transaction each server is able to issue globally unique timestamps.

for distributed transactions, the timestamp is the pair

(local timestamp, server-ID)

The local timestamp refers to the first server which issued the transaction timestamp.

Assume: timestamp(trans) = $t_{trans}$ and timestamp(obj) = $t_{obj}$

transaction trans accesses object obj

```
if (t_trans < t_obj ) then abort(trans) else access obj;
```

Each server maintains locks for its own data items. Transaction trans requests lock (e.g. read, write lock) before access.

A transaction trans is well-formed if:

- trans locks an object obj before accessing it.
- trans does not lock an object obj which has already been locked by another transaction; except if the locks can coexist, e.g. two read locks.
- prior to termination, trans removes all object locks.

A transaction is called a ***2-phase*** transaction if no additional locks are requested after the release of objects ("2-phase locking").

## Isolation

Isolation refers to the serializability of transactions. All involved servers are responsible for the serialization of distributed transactions. Example:

- let U, T be distributed transactions accessing shared data on the two servers R and S.
- if the transactions at server R are successfully executed in the sequence U before T, then the same commit sequence must apply to server S.

Timestamp ordering

Locking

**Optimistic concurrency control**

if conflicts are rare, optimistic concurrency control may be useful: no additional coordination necessary during transaction execution.

- The check for access conflicts occurs when transactions are ready to "commit";

These aspects of distributed transactions may be realized by one of the following approaches. Let trans be a transaction.

**Intention list**

all object modifications performed by trans are entered into the intention list (log file).

- When trans commits successfully, each server S performs all the modifications specified in $AL_S$ (trans) in order to update the local objects; the intention list $AL_S$ (trans) is deleted.

**New version**

When trans accesses the object obj, the server S creates the new version $obj_{trans}$ ; the new version is only visible to trans.

- When trans commits successfully, $obj_{trans}$ becomes the new, commonly visible version of obj.
- It trans aborts, $obj_{trans}$ is deleted.

This protocol supports the communication between all involved servers of the distributed transaction in order to jointly decide if the transaction should commit or abort.

We can distinguish between two phases

- ***Voting phase*** : the servers submit their vote whether they are prepared to commit their part of the distributed transaction or they abort it.
- ***Completion phase*** : it is decided whether the transaction can be successfully committed or it has to be aborted; all servers must carry out this decision.

Steps of the two-phase commit protocol

Operations

Communication in the two-phase commit protocol

Problems