Title: DistributedApplications (27.05.2014)
Date: Tue May 27 14:31:15 CEST 2014
Duration: 92:48 min
Pages: 27

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
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.

   - 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.

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.

Components 1, 2, and 3 are not ordered; t11 and t22 are ordered.

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 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.

```plaintext
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 serializability of distributed transactions. Example:

- 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”.

**Examples**
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 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 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”;

**Examples**

The following examples show the concurrency control approaches used by some current systems.

**Dropbox**
- cloud service that provides file backup and enables users to share files and folders, accessing them from anywhere.
- uses optimistic concurrency control; file granularity.

**Wikipedia**
- creating and managing of wiki pages
- uses optimistic concurrency control for editing.

**Google Docs**
- cloud service providing web-based applications (word processor, spreadsheet and presentation) that allow users to collaborate by means of shared documents.
- awareness based concurrency control: if several people edit the same document simultaneously, they will see each other's changes.

**Atomicity and persistence**

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_snew: the new version is only visible to trans.

  When trans commits successfully, obj_snew becomes the new, commonly visible version of obj.

  If trans aborts, obj_snew 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**

---

**Steps of the two-phase commit protocol**

**Communication in the two-phase commit protocol**

---

1. Coordinator $C$ contacts all servers $S_i$ of the distributed transaction $T$ requesting their status for the commit (CanCommit)?
   - if server $S_i$ is not ready, i.e., it votes no, then the transaction part at $S_i$ is aborted;
   - if $S_i$ is ready then trans is aborted; the coordinator sends an abort message to all those servers who have voted ready (i.e., yes).

2. ∀ $i$ with $S_i$ is ready, i.e., commit transaction trans. Coordinator sends a commit message to all servers.

3. Servers send an acknowledgement to the coordinator.

---

Number of messages: $4 \times N$ messages for $N$ servers.
During the 2PC process several failures may occur
one of servers crashes,
the coordinator crashes,
depending on their state, this may result in blocking situations, e.g. the coordinator waits for the commit
acknowledge of a server, or a server waits for the final decision (commit or abort).

Extended 2PC
Three-Phase Commit protocol (3PC) is another approach to overcome blocking of servers until the crashed
coordinator recovers.

Coordinator:
- multicast, ok to commit?
- collect replies
  - all ok =>
  - log commit to outcomes table
  - wait until saved to persistent store
  - send commit
  - else => send abort
  - collect acknowledgments
  - garbage collect data from outcomes table
- After Failure:
  - for each pending protocol in outcomes table
  - send outcome (commit or abort)
  - wait for acknowledgments
  - garbage collect data from outcomes table

Server: first time message (CanCommit) received
- ok to commit =>
- save data to temp area (persistent store)
- reply ok
- commit =>
- make change permanent
- send acknowledgement
- abort => delete temp area
- message is a duplicate (recovering coordinator)
- send acknowledgement
- After Failure:
- for each pending protocol
- contact coordinator to learn outcome

During the 2PC process several failures may occur
one of servers crashes,
the coordinator crashes,
depending on their state, this may result in blocking situations, e.g. the coordinator waits for the commit
acknowledge of a server, or a server waits for the final decision (commit or abort).

Extended 2PC
Three-Phase Commit protocol (3PC) is another approach to overcome blocking of servers until the crashed
coordinator recovers.

Multiple transactions may access objects of multiple servers resulting in a distributed deadlock.
At object access the server lock manager locks the object for the transaction.

Deadlock detection schemes try to find cycles in a wait-for graph.

**Edge Chasing**
distributed approach to deadlock detection
no global wait-for graph is constructed.
each involved server has some knowledge about the edges of the wait-for graph.
servers attempt to find cycles by forwarding messages (called probes).
each distributed transaction T starts at a server ⇒ the coordinator of T.
the coordinator records whether T is active or waiting for a particular object on a server.
lock manager informs coordinator of T when T starts waiting for an object and when T acquires finally the lock.

**Edge Chasing Algorithm**

**Transaction Priorities**

Every transaction involved in a deadlock cycle may cause the initiation of deadlock detection
several servers initiate deadlock detection in parallel
⇒ possible more than one transaction in a cycle is aborted.

Example:
transaction T attempts to access an object A locked by U
transaction W attempts to access an object B locked by V

**Distributed Deadlock**

Multiple transactions may access objects of multiple servers resulting in a distributed deadlock.
at object access the server lock manager locks the object for the transaction.

deadlock detection schemes try to find cycles in a wait-for graph.

theory: construct a global wait-for graph from all local wait-for graphs of the involved servers. Problems:
the central server is a single point of failure.
communication between servers take time.