TTK4145 – Real-time Programming

Sondre Bø Kongsgård
sondrebk@stud.ntnu.no

Contents

1 Fault Model and Software Fault Masking

This chapter is based on Ch. 3.7 in the book Transaction Processing: Concepts and Techniques (GR).

Fault model The one used in this chapter involves three entities: *processes, messages and storage.*

Unexpected faults Faults that are not tolerated by the design. Two categorizations:

- *Dense faults*: The algorithms will be n-fault tolerant. If there are more than n faults within a repair period, the service may be interrupted (in this case the system should be designed to stop).
- *Byzantine faults*: The fault model postulates certain behavior – for example it may postulate that programs are failfast. Faults in which the system does not conform to the model behavior are called Byzantine.

Underlying progression

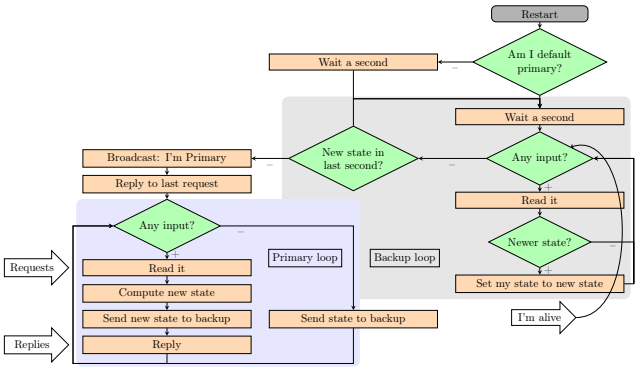
- *Failfast*: They either execute the next step, or they fail and reset to the null state.
- *Available*: Failfast + repairability
- *Reliable*: Continuous operation

Checkpoint-Restart Write state to storage after each acceptance test, but before each event. This approach can be somewhat slow (hours/days).

Process pairs OS generates a backup process for each new primary process. The primary sends *I'm Alive* messages to the backup on a regular basis.

The backup can take over in three different ways:

- *Checkpoint-restart.* The primary records its state on a duplexed storage module. At takeover, the backup starts by reading these duplexed storage pages. (quick repair)
- *Checkpoint message.* The primary sends its state changes as messages to the backup. At takeover, the backup gets its current state from the most recent message. (basic pairs must checkpoint)
- *Persistent.* The backup restarts in the null state and lets the transaction mechanism clean up (undo) any recent uncommitted state changes. (simple)



2 Transaction Fundamentals

This chapter is based on Ch. 1 in the book Java Transaction Processing (LMP).

Atomic operation An operation that is indivisible (it is not possible to decompose it into smaller operations), eg. a money transfer between two bank accounts.

Transaction Refers to the term *atomic transaction*, which provides an "all-or-nothing" property to work that is conducted within its scope. An atomic transaction has the same properties as an atomic action plus the added feature that if a failure happens, the components are returned to their natural state instead of being left possibly inconsistent as in an atomic action.

Controlling a transaction A few definitions:

- *Commit:* When all parts of the transaction commits, the transaction finishes and all its actions are made permanent.
- *Rollback:* An abortion of the transaction. All work done by the transaction is reverted.

ACID properties Properties of an atomic action:

- *Atomicity.* The transaction completes successfully (commits) or if it fails (aborts) all of its effects are undone (rolled back).
- *Consistency.* Transactions produce consistent results and preserve the internal consistency of the data it acts on.
- *Isolation.* Intermediate states produced while a transaction is executing are not visible to others. Furthermore, transactions appear to execute serially, even if they are actually executed concurrently.
- *Durability.* The effects of a committed transaction are never lost.

Termination A transaction can be terminated in two ways: *committed* or *aborted* (rolled back). When a transaction is committed, all changes made within it are made durable (forced onto stable storage). When a transaction is aborted, all of the changes are undone.

Coordinator Responsible for governing the outcome of the transaction. It can be implemented as a separate service or may be co-located with the user for improved performance. It communicates with enlisted participants to inform them of the desired termination requirements.

Transaction manager factory Responsible for managing coordinators for many transactions.

Two-phase commit This protocol is used to ensure consensus between participating members of the transaction (ie. ensure it has an atomic outcome).

In phase 1 the coordinator communicates with all the action participants to determine whether they will commit or abort. An abort reply, or no reply, from any participant acts as a veto, causing the entire action to abort. If the transactions commits, then phase 2 is entered and the coordinator forces the participants to carry out the decision. Each participant making a commit response to the coordinator in phase 1 is blocked until they have received the coordinator's phase 2 message.

Note that the two-phase commit protocol is not client-server based. It simply talks about a coordinator and participants, and makes no assumption about where they are located.

Interference free / Serializable If there are variables that are not shared between programs, some aspects of the programs can be executed concurrently. A partly concurrent execution order is termed serializable if it produces an equivalent result as doing it in serial order. If a program possess this property it is said to be atomic with respect to concurrency.

Concurrency control mechanisms Ensures serializability.

- *Two-phase concurrency control:* All operations on an object is of type "read" or "write". Many computations can hold a "read-lock", which is associated with an object by a computation before reading it. However, if someone wants to change (insert/modify/delete) the object, this requires a "write-lock". This can not be obtained if any of the other operations hold either a read or write lock on that object. This is to make sure that no one changes something others are using in their operations. If you have obtained a write-lock, this will block all other requests for read and write until you release the lock. All computations must follow a two-phase locking policy.

  In phase 1, the growing phase, locks are acquired, but not released. In phase 2, the shrinking phase, locks are released but can't be acquired. To avoid the cascade roll back problem (see under), all locks are released simultaneosly in the shrinking phase.

- *Pessimistic concurrency control:* When a resource is accessed, a lock is obtained on it. The lock will remain held on that resource for the duration of the transaction. Disadvantages with this style:
  - Deadlocks
  - Concurrency may be limited by the granularity of the locks
  - The overhead of acquiring and maintaining concurrency control information in an environment where conflict or data sharing is not high.

- *Optimistic concurrency control:* Locks are only acquired at the end of the transaction when it is about to terminate. When updating a resource, the operation needs to check if this conflicts with updates that may have occurred in the interim, using timestamps. Most transaction systems that offer optimistic concurrency control will roll back if a conflict is detected, which may result in a lot of work being lost.

- *Type-specific concurrency control:* Concurrent read/write or write/write operations are permitted on an object from different transactions provided these operations can be shown to be non-interfering. Object-oriented systems are well suited to this approach, since semantic knowledge about the operations of objects can be exploited to control permissible concurrency within objects.

- *Deadlock detection and prevention:* A transaction is *blocked* if it cannot acquire a lock on a resource it waits for to be released. In some systems, it is possible for transactions to wait for each other – the system is *deadlocked*.

  The only way to resolve a deadlock is for at least one of the transactions to release its locks that are blocking another transaction (it must roll back). Techniques for deadlock detection:
  - *Timeout-based:* A transaction waits a specified period of time, then rolls back assuming there is a deadlock.

– *Graph-based:* Tracks waiting transaction dependencies by constructing a waits-for graph: nodes are waiting transactions and edges are waiting situations. Guaranteed to detect all deadlocks, but may be costly to execute in a distributed environment.

Two-phase commit optimizations There are several variants to the standard two-phase commit:

- *Presumed abort:* If during system recovery from failure no logged evidence for commit of some transaction is found by the recovery procedure, then it assumes that the transaction has been aborted, and acts accordingly. This means that it does not matter if aborts are logged at all, and such logging can be saved under this assumption.
- *One-phase:* If there is only one participant involved in the transaction, there is implicit consensus on whether to commit or not and the coordinator need not drive it through the prepare phase.
- *Read-only:* A participant may indicate to the coordinator if it is responsible for a service that did not do any work during the course of the transaction, and can be left out of the second phase.
- *Last resource commit:* It is possible for a single resource that is one-phase aware (no prepare, only commit or roll-back), to be enlisted in a transaction with two-phase commit aware resources. The coordinator executes the prepare phase on only the latter, and if the decision is to commit, then the one-phase aware resource is informed about this.

Heuristic transactions *Heuristic outcome:* The case in which the coordinator informs the participant that the outcome and the decision are contrary. (May happen when blocking occurs in the two-phase commit protocol (where it is used to ensure atomicity). To break the blocking, participants that are past the prepare phase are allowed to make decisions as to whether commit or rollback.) It has a corresponding *heuristic decision*.

There are two levels of heuristic interaction:

- *Participant-to-coordinator:* If a participant makes an autonomous decision, it must assume that it caused an heuristic and remember the choice. At this point the participant is in a *heuristic state*.

- *Coordinator-to-application:* If the choice is different from the coordinator, then a heuristic outcome must be reported to the application. The transaction will still be marked as committed of rolled back, but it will also be in a heuristic state.

Outcome	Description
Heuristic rollback	The commit operation failed because some or all of the participants unilaterally rolled back the transaction.
Heuristic commit	An attempted rollback operation failed because all of the participants unilaterally committed. One situation where this might happen is if the coordinator is able to successfully prepare the transaction, but then decides to roll it back because its transaction log could not be updated. While the coordinator is making its decision, the participants decides to commit.
Heuristic mixed	Some participants commited, while others were rolled back.
Heuristic hazard	The disposition of some of the updates is unknown. For those which are known, they have either all been committed or all rolled back.

Transaction log Registration of the participants and where they have reached in the protocol. After the coordinator's decision has been conducted, the participant log can be deleted and informed to the coordinator.

Nested transactions transaction!nested Transactions that are contained within the resulting transaction. The enclosing transaction is referred to as a *parent* of a nested (or *child*) transaction → a hierarchical structure (with the root referred to as the *top-level transaction*).

The effect of a nested transaction is provisional upon the commit/rollback of its enclosing transaction (the effects will be recovered if the enclosing transaction aborts).

Subtransactions are useful for two reasons:

- *Fault isolation:* If the subtransaction rolls back then this does not require the enclosing transaction to roll back.
- *Modularity:* If there is already a transaction associated with a call when a new transaction is begun, then the transaction will be nested within it. If the object's methods are invoked without a client transaction, then the object's transaction will be top-level, otherwise they will be nested within the scope of the client's transactions.

Because nested transactions do not make any state changes durable until the enclosing top-level transaction commits, there is no requirement for failure recovery mechanisms for them.

Interposition The technique of using proxy (or subordinate) coordinators. Each machine that imports a transaction context may create a subordinate coordinator that enrolls with the imported coordinator as though it was a participant.

When and why interpositions occurs:

- *Performance:* If a number of participants reside on the same node, or are located physically close, can then improve performance for a remote coordinator to send a single message to a subcoordinator that is co-located with those participants and for that subcoordinator to disseminate the message locally, rather than send the same message to each participants.
- *Security and trust:* A coordinator may not trust indirect participants and vice versa, that makes direct registration impossible.
- *Connectivity:* Some participants may not have direct connectivity with a specific coordinator, requiring a level of indirection.
- *Separation of concerns:* Many domains and services may simply not want to export (possibly sensitive) information about their implementations to the outside world.