NTNU – Trondheim
Norwegian University of
Science and Technology

# TTK4145 – Real-time Programming

Lecture 10 – Transaction Fundamentals

# Example exam questions

- From 2014:

  "Transactions" are almost the same as "Atomic Actions" in the Burns & Wellings book, and are relevant when making fault tolerant computer systems. How do transactions contribute to fault tolerance.

- From 2015:

  Many techniques that can contribute to fault tolerance have been presented in this course. For each of those under: Describe shortly how they work/what they entail, and how they contribute to fault tolerance.

  - Acceptance Tests
  - Merging of Error Modes
  - Static Redundancy
  - Dynamic Redundancy
  - Recovery Points
  - Backward Error Recovery
  - Forward Error Recovery
  - N-version Programming
  - Recovery Blocks
  - Atomic Actions
  - **Transactions**
  - Process Pairs
  - Error Injection
  - Fail Fast
  - Checkpoint/Restart

# Learning goals: Transaction Fundamentals

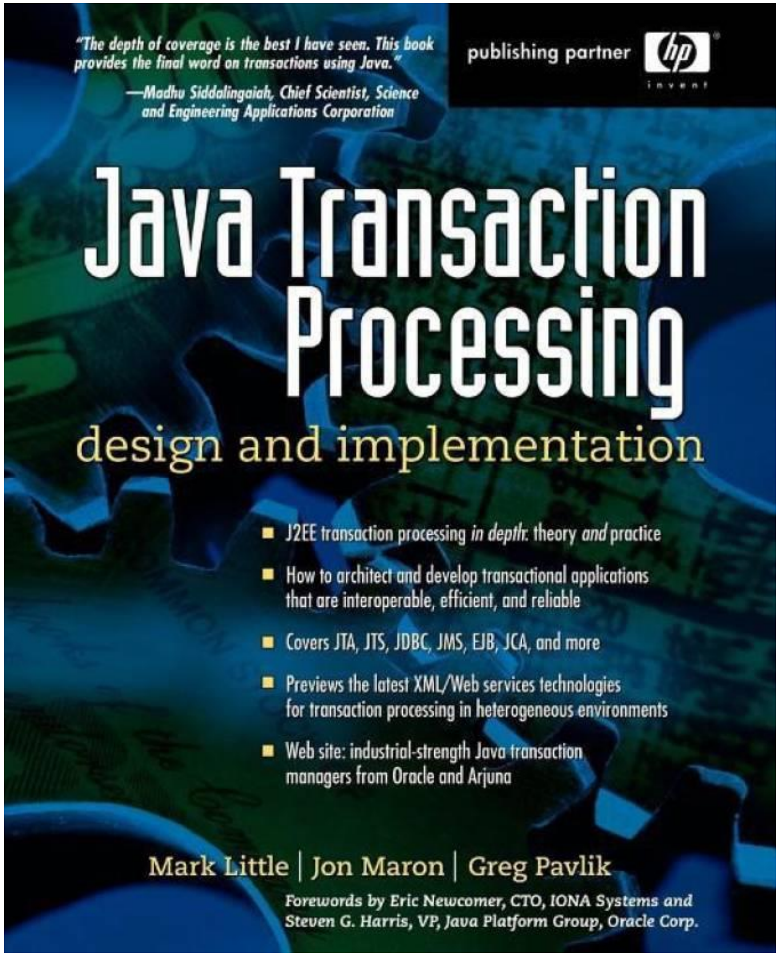
- Knowledge of eight “design patterns”
  - Locking
  - Two-Phase Commit
  - Transaction Manager
  - Resource Manager
  - Log
  - Checkpoints
  - Log Manager
  - Lock Manager,

  how they work and which problems they solve. Ability to utilize these patterns in high-level design.

- Comprehension of terms: Optimistic Concurrency Control, Two-phase commit optimizations, Heuristic Transactions, Interposition.

# Java Transaction Processing

- The lecture's topics are for the most part covered by Chapter 1 (available on Blackboard)
- Chapter is written for users of an transactional infrastructure that needs to know how it works
- Not that useful for embedded systems, we must learn to build these systems
- For our purposes the book chapter is too complex and detailed
- Assumed implementation is Remote Procedure Calls (RPC), but principles are the same regardless
- Focus on the Learning Goals, seek other references if needed ("Google it")



# Transactions

All or nothing

# Context

- Have simplified error handling using acceptance tests

- The general approach
  - Identify failure modes
  - Merging of error modes
  - Make system failfast with acceptance test
    - Failed acceptance test leads to failure and restart

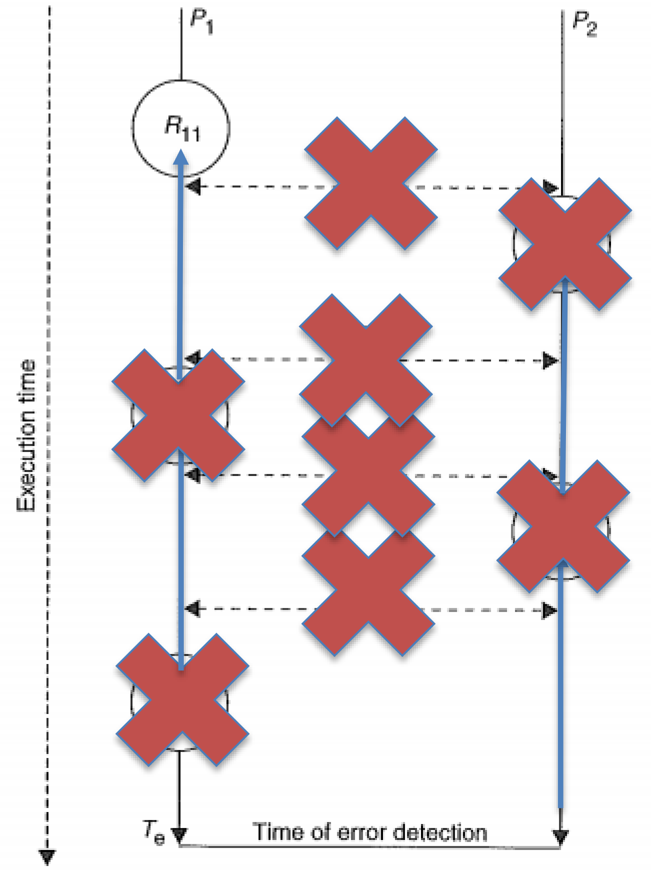
- Benefit: this handles unexpected errors
- Disadvantage: the information about what went wrong is lost

- How do we fix the error if we don't know where it occurred?
  - Backwards error recovery with recovery point

- What if the error affects multiple participants?

# Repetition: The domino effect

Error occurring in Process $P_2$ at time $T_e$ results in domino effect due to Inter-Process Communication between recovery points



# Confinement

- The steps of software dynamic redundancy
  - Detect
  - **Confine**
  - Recover
  - Treat

- In the case of multithreaded participants, we do not know which recovery point we need to go back to

- We might not even know which participants need to take part in fixing the error

- We need better confinement, especially since cooperating participants might not only be modules
  - Processes, threads, procedures, distributed systems

# Laying a foundation

- We are looking for a way to implement multithreaded fault tolerance in a concise way

- The theme of this lecture is to lay the foundation for a framework that
  - Keeps track of participants in the action
  - Defines a consistent “starting point” for the action
  - Enforces borders for the action
  - Ensures that the action is consistent when completed

- This framework turns actions into transactions

# A detour on ACID

Formally, transactions have the following properties:

- **Atomicity**

  The transaction completes successfully (commits) or if it fails (abort) *all* of its effects are undone (rolled back)

- **Consistency**

  Transactions produce consistent results and preserve application-specific invariants

- **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 (except by a catastrophic failure)

# Building the framework

Starting easy, then adding features

# Create memberships

- Not good enough with static confinement in the form of modules
  - Modules still a great idea, but we're shifting perspective

- Want dynamic confinement in the form of participants being explicit members of an transaction
  - Keeping this vague for now

- Participants that are members in the transaction only communicate to other members
  - Possible errors are confined, non-members are not affected

- Synchronizing orders between elevators can be an transaction, the participants are the different elevator's order modules
  - The membership list is static, the participants are always known

# Locking

- Have already talked about locking in shared variable setting
  - Semaphore, mutex
  - More in-depth after Easter
- In transaction … locking means that you can only access a variable, or part of memory, if it’s owned by a transaction I’m a member of
- This can be done with synchronization, semaphores, non-member will be blocked
- Alternatively, simply abort if trying to access memory that are not part of our transaction membership
- Most important, members of a transaction can only interact with variables that the transaction owns

# Backwards Error Recovery with Recovery Points

- As a participant we now have a defined membership in a transaction and access to that transaction's locked variables
  - This ensures Isolation
- Another important property of a transaction is that it can only terminate in one of two ways: *committed* or *aborted* (rolled back)
  - This ensure atomicity
- Achieve this with an acceptance test and then make a decision based on result
  - Commit: Store the result and perform side effects
  - Abort: Undo everything, go back to consistent state we were in
    - No consequences of what we attempted to do should be in the system
    - Pretend button was not pushed, not light the lamp
- Not ideal in an embedded setting, we have deadlines to reach
  - This is a starting point, in the future we'll think about forward error recovery to solve the issue
- On the other hand, backwards error recovery is simple and valid for all software
  - Forward error recovery is difficult, application depended

## Pseudocode: Backwards Error Recovery with Recovery Points

```
Allocate locks
Store variable values (recovery points)
Do work, jump to end: if problems
label end:
if(error){
   set variables back to recovery point
   status = FAIL;
}else{
   status = OK;
}
Release locks
return status;
```

## Backwards Error Recovery for several participants

```
status = OK;
if(doWorkX(...) == FAIL) status = FAIL;
if(doWorkY(...) == FAIL) status = FAIL;
if(doWorkZ(...) == FAIL) status = FAIL;
if(status == OK){
   commitWorkX(); commitWorkY(); commitWorkZ();
}else{
   abortWorkX(); abortWorkY(); abortWorkZ();
}
```

- Commit unlocks the lock, everything is fine
- Abort goes back to recovery point for that participant
- Each participant stores recovery point when they receive membership
- No explicit synchronization of recovery points needed
- Synchronization occurs with the commit decision

# The Transaction Manager

- Desirable to create a reusable module that handles transactions
- Clients only need to communicate with the transaction manager to start and end transactions, not the participants
- Each participant does work based on a transaction id
- After work is complete, ask each participant if they failed, then distribute collective decision about commit or abort

```
TransactionId tid = tm_beginWork();
doWorkX(tid,...);
doWorkY(tid,...);
doWorkZ(tid,...);
result = tm_endWork(tid);
```

# Transaction Manager interface

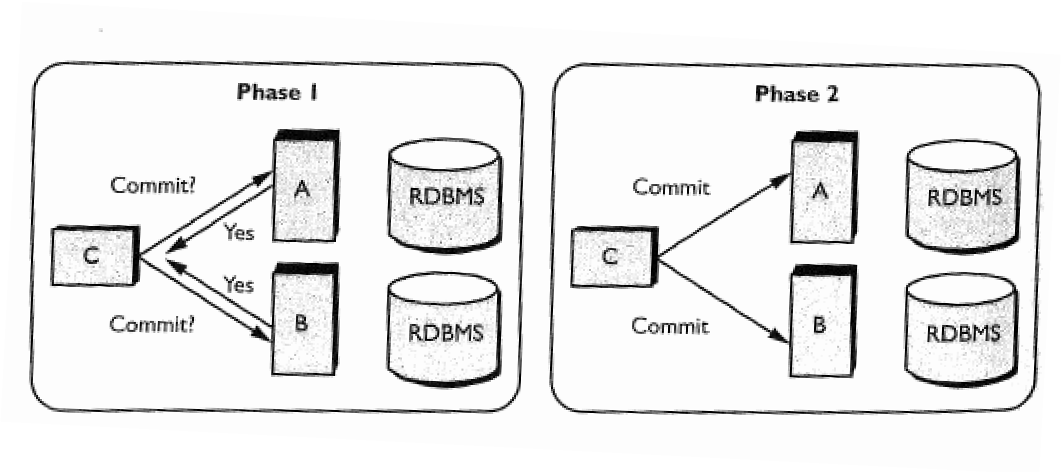
- `tm_beginWork()`
  - Creates a transaction; generates unique ids, keeps track of members.
- `tm_joinTransaction(participant)`
  - Adds participant to members
- `tm_endWork(tid)`
  - Asks all participants for status (`Status prepareToCommit(tid)`), counts votes, `commits(tid)` or `aborts(tid)` and returns status.

# The Resource Manager: The transaction participant

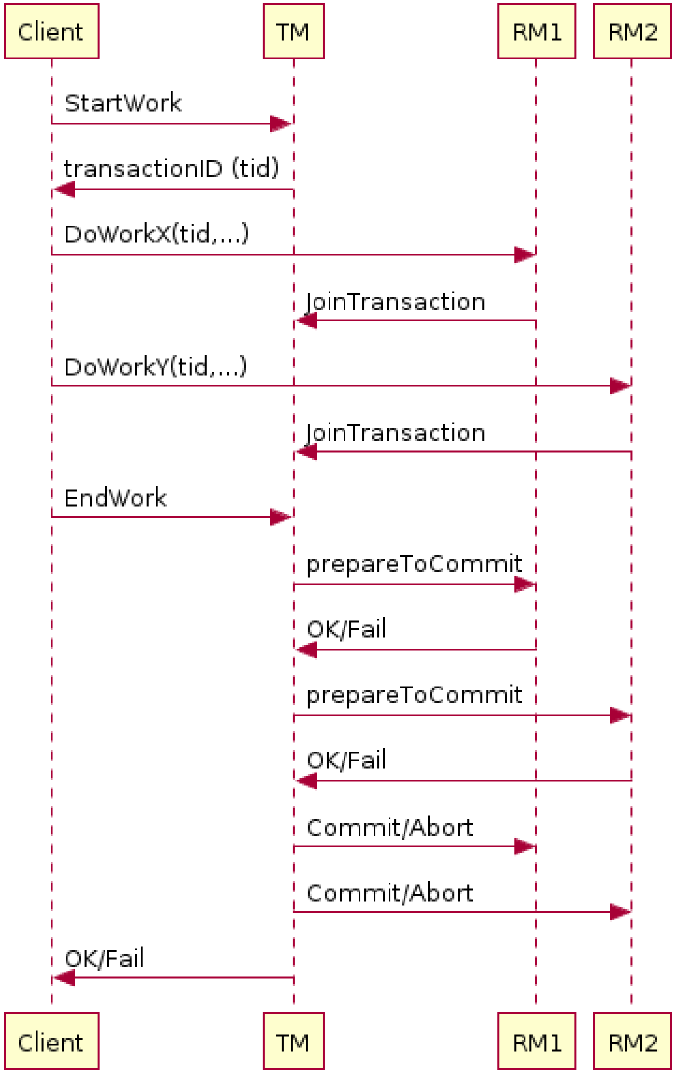
- offers `rm_doWork(tid,...)` functionality
- calls `tm_joinTransaction(me)` if not already a member
- keeps track of locks associated with the transaction
- keeps track of recovery points
- participates in the two-phase commit protocol (`prepareToCommit()`, `commit()`, `abort()`)
- Note: `prepareToCommit(tid)`, `commit(tid)` and `abort(tid)` can be given reusable forms
- The participant is an abstract concept
  - Threads, databases, processes, procedure, servers, distributed systems, etc

# Two-Phase Commit

- Protocol is used to guarantee consensus between participating members of the transaction
- An abort reply, or no reply, from any participant acts as a veto, causing the entire action to abort
- The transaction manager decides to commit or abort based on the responses



# Sequence diagram for a transaction.



# Unresolved issues

- Deadlocks can still happen between transactions
  – Two different transactions might try to acquire each others resource
  – Deadlocks are allowed to happen, but dangerous for real-time systems
  – Standard solution: Give each transaction a deadline, and abort if it does not reach it. Managing of the timers are the responsibility of the transaction manager
  – Abort, then roll back each transaction and try again

- Interaction with the world must be kept outside the transaction framework
  – An effect on the outside world cannot generally be aborted.
  – The solution must be dependent on the application
  – Wait until transaction has committed before doing side effects
  – Develop “intelligent” hardware that can handle abort or even participate in transactions.

# Summary of current framework

- We now handle all failures detected by our acceptance tests
  - as long as the infrastructure (storage, communication, locking) works.
  - and the processes are not restarted.

- After a restart, a participant will forget what work it has already performed in the active transaction
  - Can the recovery points be used for recovering from restarting?

# Log

- Fault model for calculation from previous lecture
  - Calculate
  - Acceptance test
  - Store recovery point
  - Do side effect
- Instead of storing recovery point, we write to the log
- Every participant (including the transaction manager) writes to a log what it plans to do and waits until it is confirmed safe before doing it.
  - Not only side effects; every change of state

# Restarting the resource manager

After restart a participant can get back to the current state by ”executing” all the stored log records in order.

- There are no calculations to be made, all the results are stored in the log already. Simply retracing the steps
- Only execute the commands connected to committed transactions (Commit and abort log records must also be written.)
- Some transactions may lack commit/abort log records. This means the transaction manager has not yet decided, or the decision was made when we were down. Must ask TM if we should roll back
- If asked by the TM about transactions that were active before the crash we should vote for abort.

# Restarting the transaction manager

- All transactions active when restarting are simply aborted

- One exception, if the decision to commit has already been made, the result will be recorded in the log
  - Can then distribute to participants that we commit, and then do side effects

# Adding features to the log

- Also write before-state of variables into log records.
- All log records belonging to this transaction can be undone by going backwards through the log
- No need for recovery points any more!

# Checkpoints

- Since logs are always added to, and never pruned there are two challenges to overcome
  - The medium storing the log file might run out of memory
  - The restarting process might take too long, there is too much log to run through
- The solution is to write Checkpoints to the log
  - The complete state, including a list of all active transactions is written to the log.
  - The checkpoints are written periodically, and might therefore be an inconsistent state
- Log that is older than the last checkpoint that does not contain any active transactions may be deleted.
- Restart from the latest checkpoint where all active transactions are finished, not necessarily the latest checkpoint

# Log Manager

- The added log functionality makes a reusable log manager module attractive

- The manager can queue more log records and optimize disk access.

- If it runs on another machine we may be satisfied with the receipt for received log record rather than waiting for the disk access.

# Lock Manager

- As with the log manager, attractive to make a reusable model that can handle all locking issues
- Can “release all locks associated with transaction X”.
- Can tidy up properly if we are restarted.
- Can handle resources common to more RMs
- Can be extended with deadlock detection or avoidance algorithms.

# Optimistic Concurrency Control

- What is required for a deadlock to happen?
  - Some sort of locking of resources
  - The pessimistic approach, this is unavoidable we need locks

- Optimistic Concurrency Control is the opposite
  - Believe there will be no problem

- Assume non-interference between threads
  - Check afterwards and handle as error

- Example: Two people withdrawing from the same account using an ATM
  - It is unlikely they will both access the account in the same microsecond
  - Assume there will be no issue, no locks on the account
  - Compensate for this, by every night going through accounts and see if we created or vanished money

# Two-phase commit optimizations

- Presumed abort
  - If the TM realizes a transaction will rollback no matter the participants, it can record this locally and inform all the participants as a courtesy
  - The information about the transaction is then simply removed
- One-phase
  - If there is only a single participant in the transaction
  - That participant can decide without interference from the TM
- Read – only
  - A participant that does not modify any data can inform the TM about this
  - It does not need to be informed about the outcome of the transaction and can be omitted from the second phase of the commit
- Last resource commit
  - Might be necessary to enlist participants that aren't two-phase commit aware into a transaction
  - The TM will then first prepare the other participants, and then if it intends to commit passes control to the one-phase aware resource
  - If it commits, the TM logs this, and attempts to commit other resources, can lead to loss of atomicity

# Heuristic transitions

- Some participants cannot tolerate being blocked for an indefinite time while waiting fore failure recovery mechanisms to handle the problem

- Participants that have gotten past the prepare phase of two-phase commit are allowed to make autonomous decisions as to whether they commit or rollback

- When the transaction manager finally informs the participant of the transaction outcome, they compare outcome
  - If they agree, there is no problem
  - If it is contrary, then a possible non-atomic outcome has happened

- The TM can then try to resolve the situation to preserve integrity of the system, this isn’t always possible
  - Heuristic decisions should be avoided unless it’s absolutely necessary

# Interposition

- Describes the transactions manager's ability to be a resource manager in a transaction
- Can construct supermodules from submodules, smaller databases acting as resource in larger transactional systems
- Great way to build modules

# Summary

- This is philosophy - a way to think about error handling and consistency in distributed systems / systems with more threads/processes/participants.

- For us on the real-time / embedded side (that does not have a ready-made transaction infrastructure) the challenge is to adapt this to our use.