Transactions in Distributed Systems

CS 614 Spring 2002

ANDRÉ ALLAVENA

andre@cs.cornell.edu

Cornell University

Distributed Systems

Why Distributed Systems

- Generalisation of a local system
- Not everything can be done in a local system

Common Problems

- State of system is difficult to define
- Especially with partial crashes

Hint: a database is a “local distributed” system

What is a transaction

A transaction is a collection of operation that represents a unit of consistency and recovery

A transaction starts by initialising things, then reads and/or modifies objects. At the end, either

Commit Changes are saved, resources are released; state is consistent

Abort For an outsider, nothing happened

Concept of distributed transaction

- The only difference is in the word Distributed.
- Some problems are the same as in databases
  - atomicity
  - concurrency (serialisation)
  - recovery
- The solutions to those are conceptually the same
- Add network communication failures
- And external process failures

Variety of Problems and Solutions

- Recoverable Virtual Memory (RVM)
  Memory that survives crashes
- Programming Language (Argus)
  To ease the development of distributed applications (think of an object oriented language, the objects being access by transactions only)
- Distributed Operating System (QuickSilver)
  Transactions are used for all resources management

Concurrency Solution

- Use of read and write locks to synchronise the access / modification of system resources
- A two-phase lock mechanism to allow full seriability. Locks are kept with the object.
- But different policies for different kind of objects. Two-phase are not necessary needed everywhere.
- Programmer should guard against deadlocks

Inconsistent state

- Having atomicity on operations solve the problem of inconsistent distributed state
- An operation can either commit or abort (failures are not tolerated everywhere)
- Nested transactions are trickier, not implemented everywhere, need two-phase commit

Recovery Solution

- Don’t save (on a stable storage) before being requested to commit, and then do save on a stable storage
- Keep a log on a stable storage of the changes you did to your data
- Find a way to recover (consistent) state after a failure / crash and / or abort (cleanly) leaving to the outer transaction to handle the rest of story.

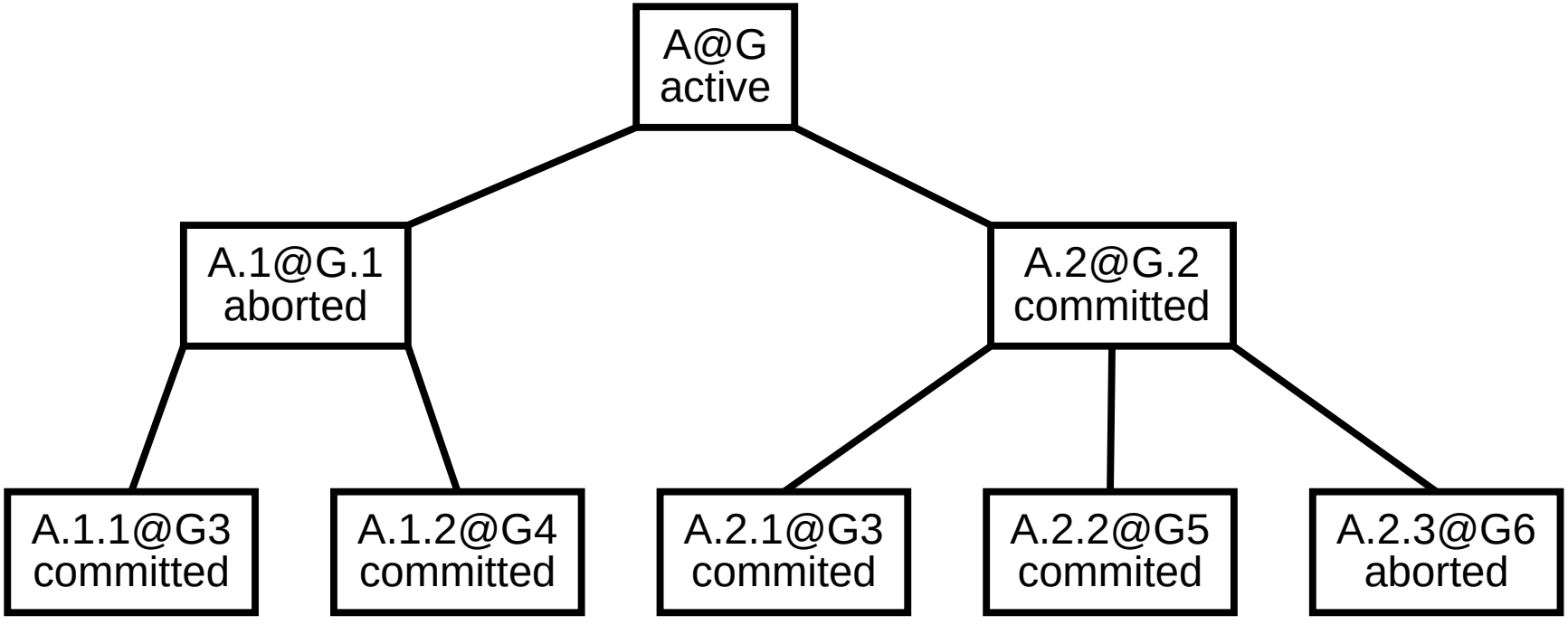
Commit / Abort

Rule of everything or nothing

Everything means local as well as remote

- When there are nested transactions, commit and abort must propagate all the way down. There are two sorts of commits, the commit to the top level transaction (which has to go to stable storage in some way), and the commit to an outer transaction which could be aborted one day.
- System has to stay consistent!

Two-Phase Commit Protocol



Difference between parent and top-level commit

Two-Phase Commit Protocol

- Sub actions commit to the parent, in a way that can be undone (such as not saved on stable storage)
- Top level commits by sending a request to its sub-committed tree
- They write in their log `prepare` and the object, and release the read locks
- Upon reception of all `prepare OK` Top level logs “Committed” and notify the subtree or send an abort
- Children now log the commit, and store on stable storage their object
  or else log abort and discard the object (undo), and release their locks

Argus

- A programming language and System for Distributed Computing
- Intended for programs that keep online data for long periods of time
- Guardians provide encapsulation of objects and resources
- Actions allow atomicity of processes

Assumptions

- A failed node doesn’t send messages
- Messages are always delivered, in order (retransmissions at higher level)
- Corruption of packets can be detected

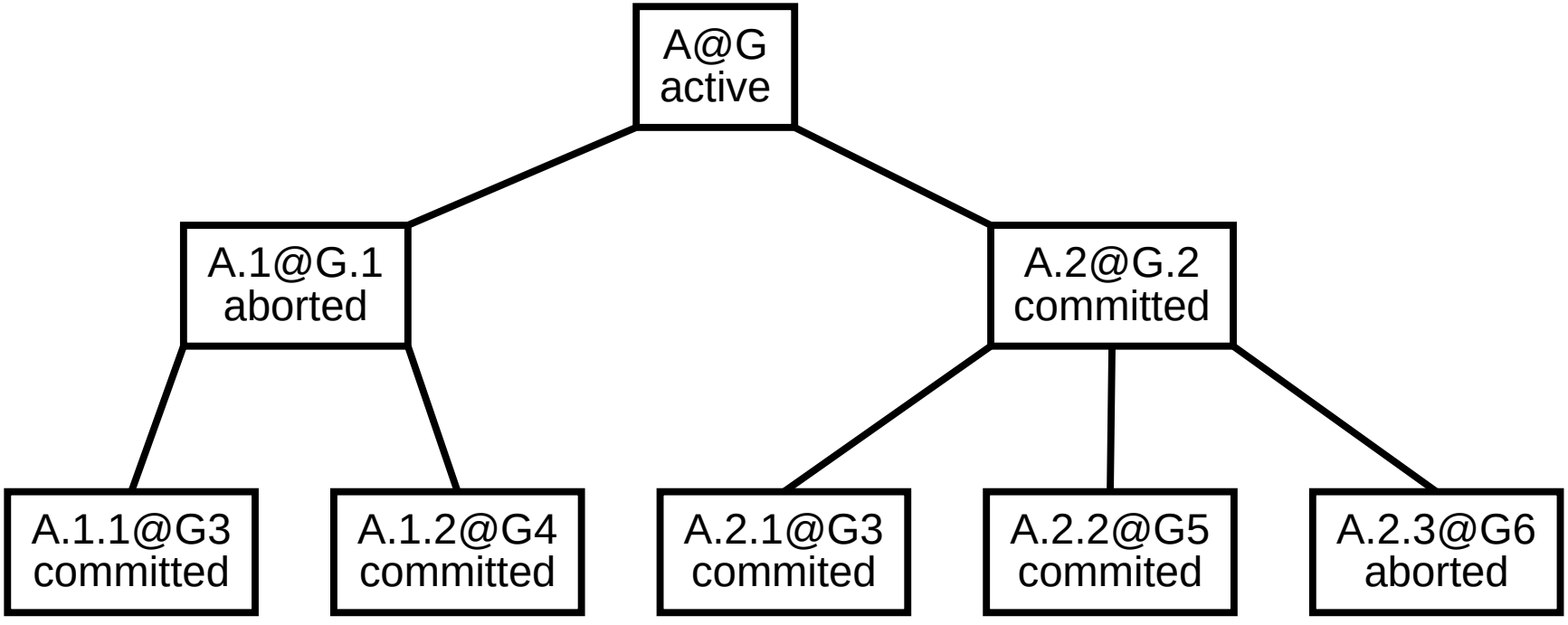
Guardians

- Guardians are objects that encapsulated resources
- No other way of accessing the resource than using the dedicated handler
- They resides in a single node, but could be moved from one node to another
- Nodes can have 0, 1 or more guardians
- Resources are only accessed through handlers
- Guardians can create other guardians
- Guardians have stable and volatile resources

Actions

- Actions are total, atomic. They either abort or commit, but don’t leave an inconsistent state
- They can be nested
- Actions work on copies of their object, and keep version number
- When an action commits, it propagates its locks and local version of the guardian to the parent action
- As well as a list of participating guardians which committed
- Strict two-phase locking (and locks held until a father aborts or top-level commits) (ensures seriabilility)

Action Tree


A@G
active
A.1@G.1
aborted
A.2@G.2
committed
A.1.1@G3
committed
A.1.2@G4
committed
A.2.1@G3
commited
A.2.2@G5
commited
A.2.3@G6
aborted

Locks and nested transactions

- Synchronisation access to resources is done via locks
- An action can acquire a *read* lock if and only if all holders of write locks are ancestors
- An action can acquire a *write* lock if and only if all holders of read or write locks are ancestors

Implementation

- There is a list of committed children which lies along, as well as an abort list.
- Only commits and prepare are actually resent until getting answer. Release of locks for example are not guaranteed to be received.
- Crashes and orphans processes are taken care by a mysterious *orphan destruction algorithm*
- Each node has a special privileged guardian: the guardian manager, all other guardians are his children

Argus Conclusion

- Nested queries do not induce a high overhead
- Communications are expensive
- Atomic types of object are difficult

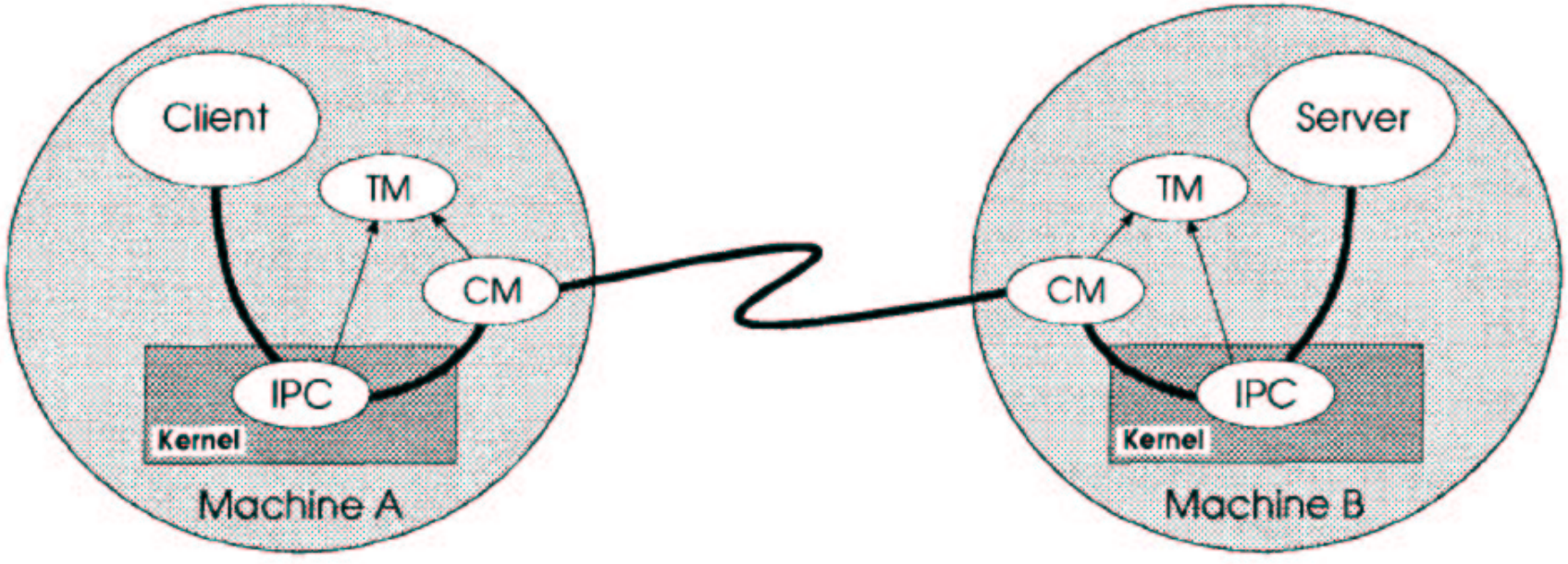
QuickSilver

- QuickSilver is a general purpose distributed operating system supporting transactions.
- Transactions are used for all the resource management in the system
- The same mechanism for accessing any resource, local or remote
- Every program runs in the context of transactions (ex shell script)
- Constraint: Unix applications should be easy to port to QS

QuickSilver

- QS supports atomicity, recoverability and concurrency of transactions.
- Each server has its own concurrency policy
- Commits are one-phase or two-phase commit depending on the server (ex, file system)

Overview



Remote IPC in QuickSilver

Transaction Management

There are 3 parts in the transaction management

1. Transaction Manager
2. Transactional IPC (Inter Process Communication)
3. Log Manager

Transaction Manager

- The TM starts a transaction when it receives an IPC from a process, as so it handles the start and finish of the transaction
- The TM assigns globally unique TID and registers it with the kernel
- The TM coordinate the decision to commit or abort along the participants of the transactions

The servers are responsible for implementing the recovery, cleaning after an abort, and saving modification upon a commit

Transactional IPC

- IPC are done on behalf of a transaction
- Remote requests are handled by the local Communication Manager
- There are *participation classes* (specifies protocol)
  - no-state (no notification of termination)
  - stateless (1-phase commit, to clean up state)
  - recoverable state (2-phase commit)

Log Manager

- Records are appended to the end of the log file
- The log is used to recover, but also as checkpoint in long running applications

Weak Seriability: DFS

Distributed File System (weak seriability)

- writes locks only for renaming/creating a directory
- read locks are not required when reading a directory
- read locks on files are released when closing that file
- write locks are kept until transaction commits or aborts

Possible Usages of Transactions

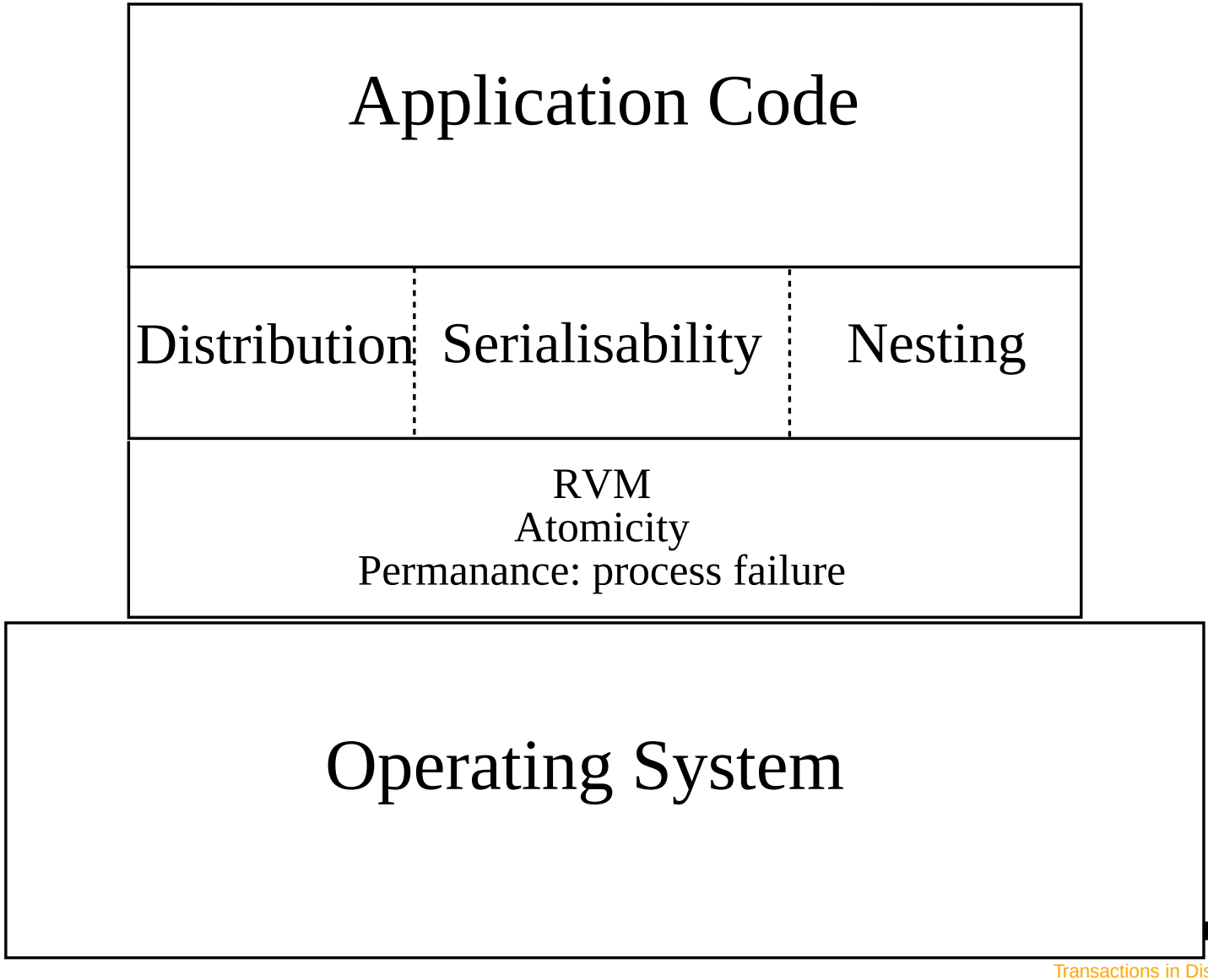
- safe updates / installation of software
- no need of a working copy of a file.
- undo mechanism for applications
- safe “kill” of programs, such as make (no temp files lying around)

But a few difficulties to expect with long running transactions (ex window manager) which end up having a huge state to commit when it will.

Lessons of QuickSilver

- Writing transactional applications is simple, robust distributed ones are much easier than for vanilla Unix
- Writing simple transactional servers is easy, complex ones are difficult but worthwhile
- A flexible concurrency control policy is desirable
- Can live without nested transactions most of the time, and can survive anyway
- Long lasting transactions shouldn’t be a problem, but...
- A strong log system is difficult to implement
- Overhead is not significant (but IPCs said to be slow)

Lightweight Recoverable Virtual M



RVM: a simplified database

RVM is just a simplified database, which

- has only one type of lock
- saves to disk to ensure recoverability
- is used as a library linked to applications
- can be bypassed (`no_abort`)
- is an extra layer between the application and the operating system (portability and simplicity)
- needs to have the programmer (or the compiler) declares the areas he is modifying

Conclusion

- The use of transactions can be generalised from databases to any part of a system.
- Note that there is still the Impossibility of distributed consensus with faulty process lying around.
- Are some of these systems / ideas used today?