# Transactions and Fault Tolerance

Availability and Reliability are the big concerns. Reliability is defined to be a lower probability of a fault happening; it implies correctness.

What are we referring to:

- arbitrary process
- data
- hw

Mechanisms:

- redundancy of everything
- make actions atomic
- checking software actions to make hw appear more reliable

1.) What are the problems that can occur?
2.) What are the techniques?
    - overhead
    - success/usefulness

<u>What are the problems that can occur?</u>

Processes:

- they can terminate early because of programmer error or hw problem
- kernel (or other privileged processes) can clobber data of others or can start sending load msgs

Q: How do you finish the functionality originally asked for?

HW:

- hw breaks

Q: How does part of system continue running?

Data:

- server can go down on which data is located
- if stored in volatile memory, can be lost if hw fails
- if stored on disk, you could have a head crash
- multiple processes referencing same data can cause consistency and correctness problems

Q: How do you make the data as logically correct as possible?

<u>What are the techniques?</u>

Redundancy:

**Processes (Fault tolerance in UNIX, Non-stop kernel)**

- process pair
- checkpoint to your back-up periodically
- when back-up comes up, you need to make sure it does everything since checkpoint in same order primary process did things
    - make sure you don't redo actions

Fault Tolerant in UNIX

- msgs sent to both primary and backup (totally ordered atomic msging)
- backup keeps count of # of msgs sent since last synch
- quarterback, halfback, fullback
- do you back up the backup?
- Sync -> storing addrspace (diry pages) to page server and backup logs some kernel state

Non-Stop Kernel

- primary checkpoints to backup
- some non-idempotent requests, servers track those using some assigned sequence number. They store result so it can just be returned.

Overhead

- use of hw to store another process limits total # of processes in system

- traffic from sending msgs. Special type of msging that is more expensive.
- Servers may have to keep some state.

Success/Usefulness

- don't have to redo all the work that's been done before
- don't use as much hw resources as by having redundant copies running simultaneously
- would be good for long running processes
- good for server processes that need to be constantly available

**HW (Non-Stop kernel)**

- duplicate buses
- duplicate disks
- duplicate power supplies

Benefits of doing this?

- msgs are fault tolerant which allows them to implement process redundancy
- disks permit data mirroring. But also probably ensure there is always some place to write
- hw less likely to go down -> less likely to go down in middle of something important

➔ less often that you have process failures and therefore need the backup

- this solution is fast and easy, but expensive

**Data (Recovery Techniques)**

- incremental dumping
- audit trails
- logs
- replication
- backup copies
- differential files

Overhead – based on frequency and amount of checkpoints

- audit trail – time, result stored in log. Very high overhead so justified only if very high reliability needed.
- Salvation program – low overhead. Cost incurred only at crashes
- Multiple copies – very high overhead.
- Incremental dumping and backup – tolerable overhead

Success/Usefulness

- if small losses are ok, backup + incremental dumping + salvation programs ok.
- Careful replacement and multiple copies can avoid all data loss.
- Incremental dumping and backup good for big damage recovery.

Atomic actions (Quicksilver, Guardians, DB, Transaction Concept):

Transaction – transformation of state which has properties of atomicity, durability (effects survive failures), and consistency.

2 Techniques proposed. Said to be very similar:

1.) Time Domain Addressing – objects never altered, rather an object is considered to have a time history and object address becomes <name, time>
2.) Logging and Locking – clusters current state of all objects together and relegates old versions to the log. (most people do this.)

- read and write locks used
- grab lock on data you want
- log your modifications (undo and redo logs)
- only when you commit do those changes take effect
- for commit/abort decisions, one centralized decider
- problems:
    - nested transactions
    - transactions may be long lived
    - transactions not unified with programming languages

Guardians

- unify transactions with programming language
- encapsulate and control accesses to resources
- users access resources through handlers
    - location independent

    - commits or fails
- contains data objects and processes (which run handlers)
- whenever top-level action completes, data written to storage
- whenever guardian brought up, always have most up to date consistent storage
- availability not improved except that whenever machine comes up, guardian comes up

Overhead

- requires additionally at-most-once RPC semantics

Quicksilver

- uses atomic transactions as unified failure recovery mechanism for client-server structured distributed system
- exposes basic commit protocol and log recovery primitives.  Clients and servers get to tailor.
- atomic 2 phase message protocol used
- IPC tagged with Tid
- IPC keeps track of all servers receiving mgs belonging to transaction
- When trying to commit, coordinating TM sends vote requests to all participants (recursively propagates)
- Servers prepare when they commit vote.  Everything saved so that if there is failure before receives commit msg, server can ask TM how it all ended and appropriately update their state.
- Coordinator tells everybody to commit or not commit.
- Servers implement their own recoverable storage, choose their own log recovery algorithms, and drive their own recovery.
- Programs don’t need to know about mechanism and it is still used.  IPC automatically does it.
- Move to your server processes all functionality that you need to be recoverable. (microkernel)
- Log transactions done by LM.

Overhead

- IPC
- Having TM/LM replicated is expensive
- Need to communicate between them all
- TM and servers need to keep track of state

DB

- build transaction stuff on top of OS.  Can have conflicts w/OS and replication of functionality
- OS doesn’t provide functionality required.  OS handles these inefficiently or incorrectly according to DB.
    - Buffer pool mgmt.
    - Forcing to disk in order (intentions list and commit flags)
    - scheduling of processes difficult because of sharing of data
    - DB wants record accesses not block accesses (creates additional tree structure)
        - Locking granularity wanted at tuple/record rather than page

➔ microkernel would be useful

➔ atomic actions difficult to implement on top of OS, so we need to define some special type of system.

Checking software to make HW appear more reliable (RIO):

Rio

- memory is fast and can allow you to forgo going to slow disk to ensure persistence and reliability of data
- memory interface is problem – easy to do bad stuff if privileged
- make sure all memory accesses checked by going through TLB
- “warm reboot”
- write memory after reboot to disk and then perform clean up on that dump

Overhead

- every access goes through TLB (might cause some conflicts or pollution)
- every single access you have to change the permissions to a page
- requires correct HW to allow “warm reboot”