

Models of Distributed Fault Tolerant Computing

Prof. Dave Bakken

Cpt. S/EE 562 Lecture

Chapter 8 from Text (except 8.6)

April 2, 2002

Administrative Notes

- Today
 - Discuss presentations and topics
 - Discuss final projects
- Goal of Chapter 8
 - Show how paradigms of Chapter 7 can be used and combined to achieve fault tolerance in an application-oriented way
 - Skipping 8.6, on transactions

Classes of Failure Semantics (8.1)

- Failures can be classified many ways
 - Main classes: arbitrary, crash, omissive, crash-recovery
 - Can refine in many ways
- Assume arbitrary failures (no assumptions)?
 - Safest assumption!
 - Still need to quantify how many
 - Still need some environmental assumptions: clock drift, latency, ...
 - But lowest price/performance
- When to choose arbitrary?
 - When failures can be catastrophic: lives or mega-bucks
 - When will be under threat of malicious attack

Classes of Failure Semantics (cont.)

- Assume Fail Silence (crash)?
 - Very widely used
 - Must have no externally visible problems before crashing
 - E.g., may trash its stack, read other memory, just then it crashes
 - Greatly simplifies design
 - But not huge coverage of assumption in many environments
 - So don't assume too much reliability from system, unless build on top of these components by self-checking, wrapping, etc.
- Assume Omissive (Weak Fail Silence)?
 - Either crashes, or suffers up to k omission failures
 - Easy to implement, not much worse than fail silence
 - Often have a mix of these and fail silence

Classes of Failure Semantics (cont.)

- Crash-Recovery
 - Most systems today assume components eventually recover
 - Difference from crash: crash assumes total amnesia, crash-recovery assumes some memory (of last checkpoint)
 - Advantage over crash: no state transfers
 - Disadvantage from crash: failover time long till recovery
- Synchronous and Asynchronous Models
 - Timing assumptions key: related to accurate failure detect.
 - Main issue: coverage
 - Asynchronous assumptions very expensive
 - Design can fail not because wrong kind or # of fault, but synchrony assumptions violated
 - Misuse of timeouts in fully asynchronous
 - Neglecting timing failures in synchronous model
 - Partial synchrony: a middle ground (chap 13, not covered)

Basic Fault Tolerance Frameworks (8.2)

- Framework: a partial design solution
- FT frameworks provide building blocks for architect to construct FT systems with
- Hardware FT
 - Detecting HW faults using HW mechanisms
 - Usually low-level (**why?**)
 - Detects and recovers or masks
 - Example: parity bits, memory address range checks, etc.

Hardware FT (cont.)

- Failure detected in HW FT
 - Stop component
 - If have spares, can have **supervisor** perform **switchover** to spare
 - Cold standby: start spare only after failure
 - Hot standby: spare started before failure
- Next evolution: have multiple component replicas run in parallel
 - E.g., 3 replicas and comparator/voter can mask a single error without any service interruption
 - Called **Triple-Modular-Redundancy** (TMR)
 - Extension: **N-Modular Redundancy** (NMR)

Software-Based Hardware Fault Tolerance

- Goal: tolerate HW faults using SW techniques
 - AKA “**Fault-Tolerant Software**” (*not* Software Fault Tol.)
 - Basis for modular FT
 - Many previous techniques studied are applicable
- Detecting a component failure
 - Can test periodically: will at least bound bad duration
 - Can force output to go through correctness-testing filter
 - Can run in parallel and check results (active replication)

Software Fault Tolerance

- Tolerates software faults; 3 main categories
 1. **Heisenbugs**: fail due to particular sequence of events
 - E.g., race condition
 - Difficult to reproduce
 - Solution: execute more than once and compare
 2. **Bohrbugs**: fail in exact same place of computation each time
 - So redundancy in time or space not helpful
 - Solution: design diversity (only way)
 3. Faults specific to a peculiar HW or config param
 - E.g., OS or Pentium floating point chip
 - Solution: heterogeneity
 - Note: not a very common case

Fault-Tolerant Communication

- Communication fundamental
 - Generally has its own framework and assumptions from other components
- Classes of failures in communications networks
 - Assertive: use value redundancy (CRCs, etc)
 - Omissive errors: common, use temporal redundancy

Fault Tolerance Strategies (8.3)

- Many factors affect choice of strategy
 - Classes of failures
 - Cost of failure
 - Performance/price ratio of delivered system
 - Available technology
- Design path
 - Define a strategy
 - Design with frameworks discussed previously

Fault Tolerance Strategies (cont.)

- Balancing Fault Tol. (FT) & Fault Avoidance (FA)
 - Key factor: number and severity of faults
 - Balance
 - Length of mission with
 - Expected reliability of components for a
 - Given amount of redundancy
 - Don't add redundancy beyond point of diminishing returns
- Key factor is severity of faults
 1. Define architecture and components as above
 2. Evaluate coverage of assumptions, and adjust above
 3. Try to use COTS (Commercial Off-The-Shelf)
 - Very cost effective, but not great confidence
 - So wrap or self-check or harden or ...

Fault Tolerance Strategies (cont.)

- Tolerating design faults
 - Most design faults in mature systems are transient!
 - So temporal or spatial redundancy is just fine
 - Fine for all but highly-critical systems
- Highly-critical systems
 - Cannot assume only transient faults
 - Must handle design faults, too (non-transient)
 - Use design diversity, N-Version programming
- Perfect non-stop operation
 - Achievable??? ... a qualified “yes”
 - Goal: mask errors so users never detect effects
 - Expensive; for loss of life or mega-bucks
 - Failover time “short enough”; for web unnoticeable in latency!
 - Cannot achieve non-stop in partitionable or asynchronous...
 - So if true, need more redundant network HW and RT mechanisms

Fault Tolerance Strategies (cont.)

- Reconfigurable operation
 - Active replication is expensive
 - Cheaper alternative: use error recovery
 - Glitch in service can be visible: lost update etc
 - Techniques: semi-active or passive
 - Reconfiguration causes temp. unavailability and possible glitch
- Recoverable operation
 - Use components that crash and later recover; assume
 - Duration of recovery known and bounded and short enough for app
 - Crash does not cause incorrect computations
 - Techniques: checkpoint to stable store
 - Can have long failover
 - Best for long-running applications
 - Scientific computations, etc.

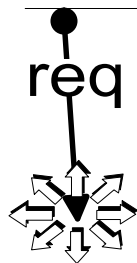
Fault Tolerance Strategies (cont.)

- Fail-Safe or Fail-Operational
 - Sometimes need emergency action when redundancy not enough for assumption coverage
 - Rather than catastrophic failure, do a shutdown
 - Called **fail-safe** behavior
 - Very important to safety-critical systems
- Sometimes shutting down impossible
 - Airplane has to keep flying even if engine has problems
 - Must have contingency plans until safe stop possible
 - Called **fail-operational**

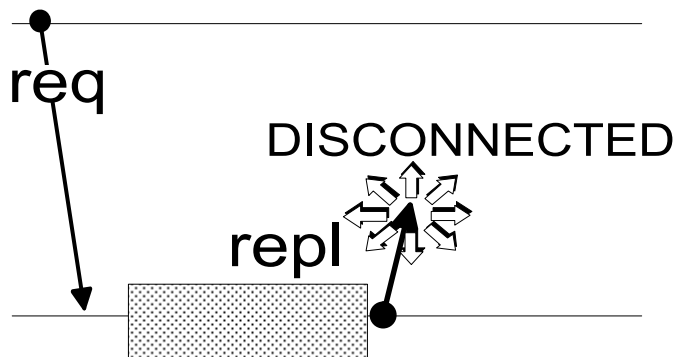
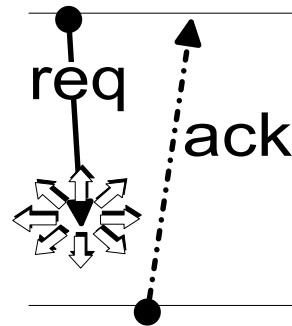
Fault-Tolerant Remote Operations (8.4)

- RPCs and RMIs are building blocks
- Main components of interest
 - Client
 - Network
 - ServerEach can fail
- Assume benign (and common) situation
 - Clients and servers fail by crashing
 - Networks crash or lose messages
- Goal: server executes requests once, and only once
 - Called **exactly-once** semantics
 - Not generally achievable....
- Network omission may drop request or reply....

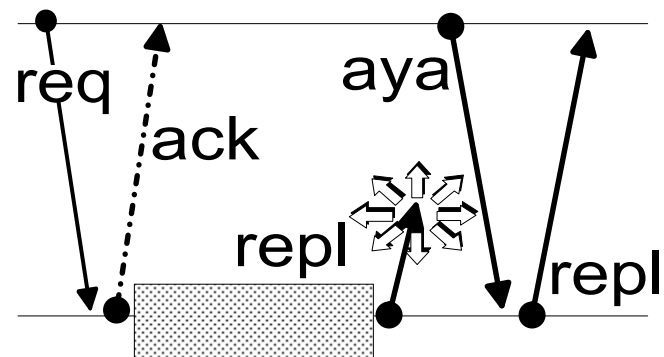
Fault-Tolerant Remote Operations (cont.)



(1)



(2)

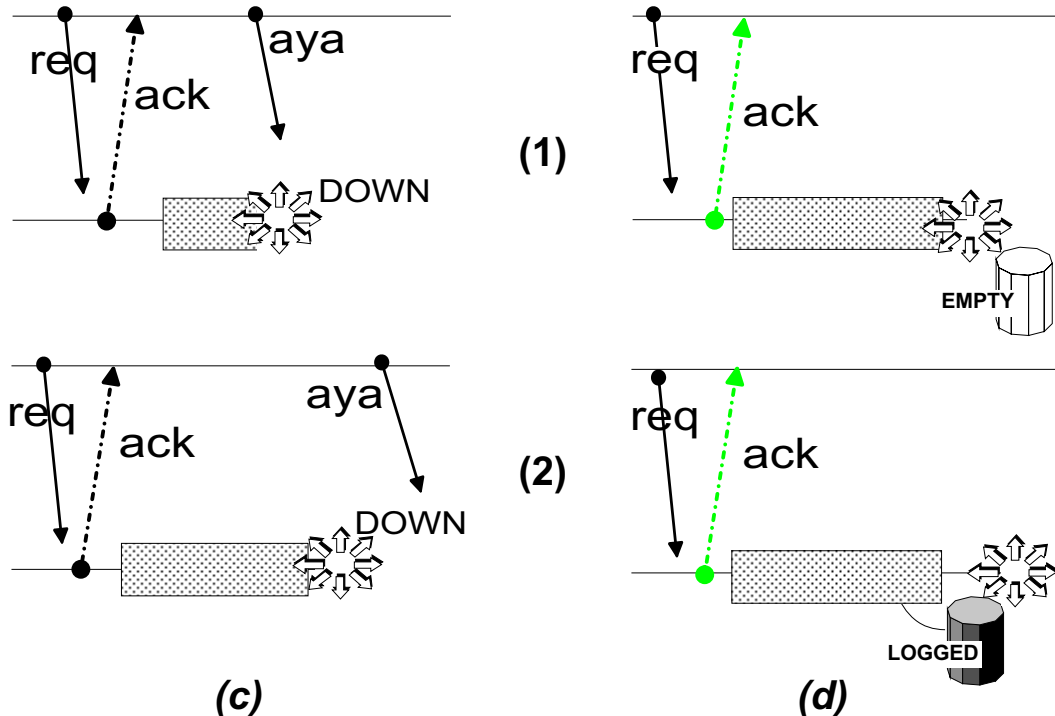


(a)

(b)

- Either request (top) or reply (bot.) can be dropped (a)
 - Both are indistinguishable from client point of view (**why?**)
- Solution (8.1(b))
 - Server ACKs request to client knows gotten; times out
 - If reply timeout reached, client sends “are you alive”

Fault-Tolerant Remote Operations (cont.)



- But server can fail, too.... (c)
 - Can fail before executing operations (top)
 - Can fail after executing operation (bottom)
 - Both indistinguishable by client
- Logging server state and replies can reduce (not eliminate) window of vulnerability (8.1(d))
 - **At-most-once** semantics

Fault-Tolerant Remote Operations (cont.)

- Stateful servers + at-most-once == poor performance!
 - Avoid when re-executing request does not hurt
 - Examples??
 - Such servers/services are **idempotent**
 - RPC semantics are called **at-least-once** (0 or 1 times)
 - Servers can be stateless
 - Faster
 - Very quick failover time
- Client failure
 - Not a data consistency problem, generally
 - Problem when has a lock or object reference count
 - Solution: leases: mutex for a while, must renew or lose
- **Q: any case where replicated clients really needed?**
 - Hint: think data consistency and one kind of failure semantics

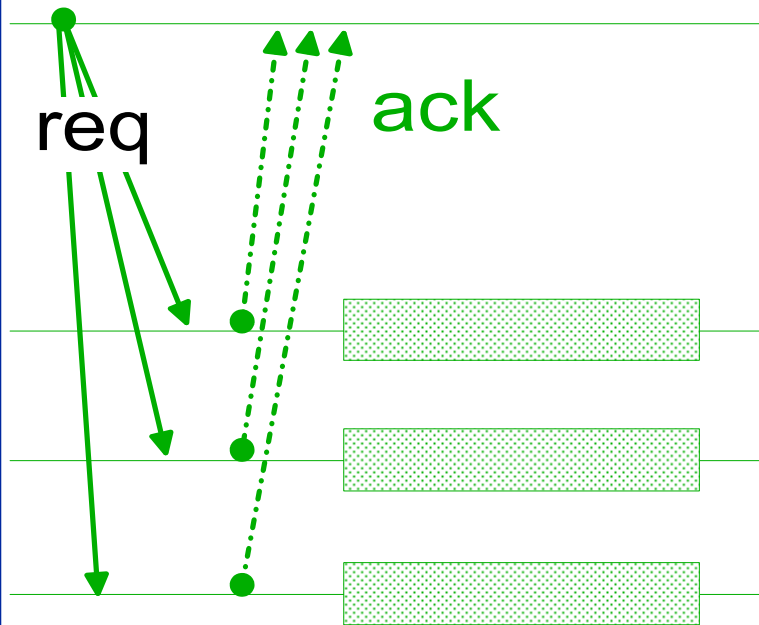
Fault-Tolerant Event Services (8.5)

- Some apps more naturally broken into event notifications than request-reply interactions
- Event-based system components
 - Publishers
 - Subscribers
- Kinds of event channel architectures
 - **Volatile channel sys.**: event consumed/tossed right away
 - **Persistent channel systems**: event kept until consumed

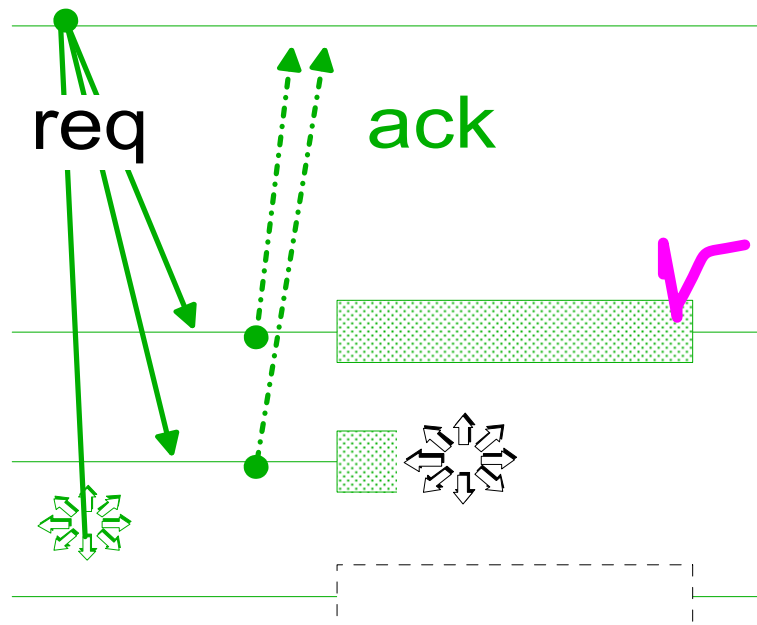
Volatile Channel Architectures

- Often events have to be processed right away
- Here FT means
 1. Ensuring desired event produced
 2. Ensuring event reliably delivered to consumer
 3. Ensuring there is a consumer available to process event
- Techniques
 1. Replication: use multiple sensors
 2. Reliable message delivery (multicast; see #3)
 3. Replicate subscribers, often state machine approach
- Note: can achieve **exactly-once semantics** with state machine approach
 - This is *easier* than RPC, which has round-trip request-reply

Volatile Channel Architectures (cont.)



(a)



(b)

Persistent Channel Architectures

- Similar to volatile channel in many ways
- Differences
 - Temporal decoupling of publishers & subscribers
 - Event channel is not just a communications medium but is a storage medium
- One abstraction
 - Consider channel as a FT server, for pubs and subs
 - “Channel server” then implements a stable storage
 - Lots of solution possibilities, with techniques covered

Persistent Channel Architectures (cont.)

