CSE 515 — Winter 2004

Dependable Distributed Systems

Class 9
Two Top-level Topics:

1. Taxonomy
   - Terminology
   - Mapping the space of dependability

2. Paradigms for distributed fault tolerance
   - A high-level view of the ways that we can build fault-tolerance into a distributed system.
What’s the connection between...

... fault tolerance and distribution?

• Distribution needs fault tolerance

• Fault tolerance needs distribution
Taxonomy

Why bother?

1.

2.
Faults, Errors and Failures

• Fault
  – An event (presumably, undesired)

• Error
  – A state (presumably bad) internal to the (sub-) system

• Failure
  – externally observable behavior of (sub-)system no longer meets its specification
  – requires the existence of a specification!
Fault Models

Why do we need a fault model?

- There is always some catastrophe too serious to be tolerated
- Dependability is not free

When building a distributed system:

- We need a way of describing the faults despite which we must be dependable
- We focus on interaction faults
Omissive Faults

• Omission: some component does not engage in a particular interaction (ever)

• Crash: some component does not engage in an interaction, nor in any of the subsequent interactions. Also known as “fail stop”

• Timing: some component does not engage in a particular interaction at the right time
  – All omissive faults are in the time dimension
Assertive faults

• The data communicated in an interaction are wrong
  – *Syntactically* wrong, e.g., packet format is out of conformance to protocol
  – *Semantically* wrong, e.g., packet format is OK, but data does not conform to reality
Consistency Faults

• If a component is specified as interacting with other components in multiple ways, we can also get consistency faults, e.g.,
  – a multicast message might be sent to some peers but not to others — inconsistent omission
  – it might not be sent at all — consistent omission
  – the “copies” of the messages might have different contents — inconsistent assertive
Coverage

We might be asked: how likely is it that this system will be dependable?

• To answer such a question, we must first ask: in the face of what eventualities?
  – Environmental assumption: probability that the environment will behave as we have assumed
    - temperature in given range, not more than assumed number of faults of the assumed kind
  – Operational assumptions: probability that the programs will do what we have assumed
How do computers fail?

• Gray (1986) study:
  – 42%: incorrect system administration
  – 25%: buggy software
  – 18%: hardware
  – 14%: environmental
    - (9% power failures > 4 hours)
  – 3%: other

• Some categories more under-reported than others.
Strategies for Dependability

1. Avoid or mask all of the faults that you can

2. Tolerate the rest
   - prevent the fault causing an error, or
   - prevent the error from causing a failure

3. Provide for recovery if a failure does occur
   - Not always possible, e.g., with aeroplane flight control
Fault Tolerance

Fault tolerance comes through *redundancy* in space, time and value

- **space** redundancy: several copies of the same component, *e.g.*, disks, servers
- **time** redundancy: repeat the action, *e.g.*, send multiple copies of message, restart failed computation (after a Heisenbug)
- **value** redundancy: add extra data, *e.g.*, error correcting codes, signatures
Error processing

1. Detect the error
   - time-outs
   - value redundancy

2. Recover from it
   - backward error recovery, e.g., retransmit lost message, restore from checkpoint
   - forward error recovery, i.e., continue on, correcting effects of the error

3. Mask the error
   - in a lower level component, e.g. process-pair.
Modularity

• Modularity is the key to fault tolerance
  – allows for independence of hardware and software components
  – allows for replication of components
  – allows a component to be replaced by a sub-system of higher dependability
  – allows graceful degradation to a lower level of service
Modularity and Publish-Subscribe
–Conceptual (V&R Fig 3.21)

Fault-tolerant (V&R Fig 6.9(b))
Distributed Fault-tolerance: How to get it

1. Failure Detection
2. Membership
3. Communication
4. Replication management
5. Resilience
6. Recovery
Failure Detection

• To recover from a failure, you have to detect it first

• Even if you can mask the failure, you still need to detect it
  – Why?

• Failure detectors can fail!

• A detector is
  – accurate, if correct processes are not labeled “failed”
  – complete: failed processes are eventually reported
Local Failure Detectors

• Assume perfect channel between detector and target
  – Watch-dog components
  – self checking routines or boards
• Timeliness may still be a problem
\[ n \geq 2f + 1 \]

- In (a), it is impossible to tell which node is faulty.
- In (b) if we know that \( f = 1 \) (at most 1 node is faulty), it must be node B.
Distributed Failure Detection

• Perfect failure detectors: (*strong accuracy* & *strong completeness*) possible if
  – failures are crashes
  – system is synchronous
  – channel is perfect, or omissions are bounded

• Normally, failure detectors are imperfect:
  – no bounds on channel failure
  – no bounds on delay
FLP Incompleteness

Fischer, Lynch & Paterson 1985

• In an asynchronous system with one faulty processor, it’s impossible to guarantee consensus.

• An *eventually weak* failure detector (p199) would enable one to reach consensus.

• So:
  – deduce that it’s impossible to build even an eventually weak failure detector in an asynchronous system.