Distributed algorithms for fault-tolerance

PhD Course, Fall 2003

Simin Nadjm-Tehrani

www.ida.liu.se/~snt

Course planning

- Course page under construction

  `www.ida.liu.se/~snt/teaching/Distalg/`

  – goals, literature & web resources
- Time plan: Intensive, November 3-4th and 24-25th, 2003
- Examiner: Simin Nadjm-Tehrani
- 1-2 Guest lectures

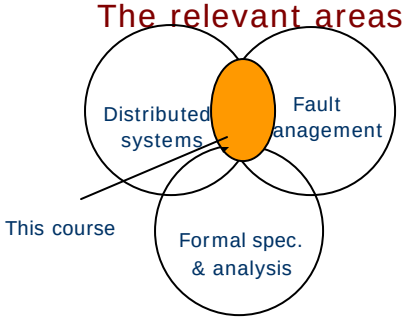
Course idea

- A short introduction of basic notions and models for distributed systems
- Review of fundamentals for fault-tolerance and replication in distributed systems
- Your expectations and background?

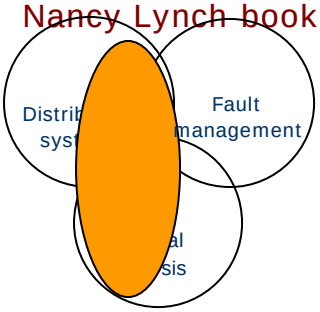
Organisation & Examination

- During day 1 there will time for group discussions and individual questions
- After day 2 each participant formulates three questions they will study in more depth
- Day 3 will start with a 15 min. presentation on studied material by each participant
- Examination:
  - term paper (topic to be agreed) or written assignments

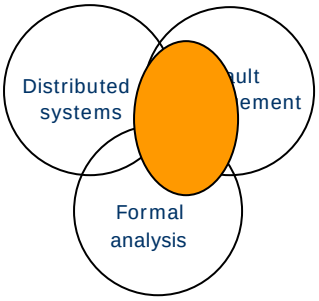
The relevant areas



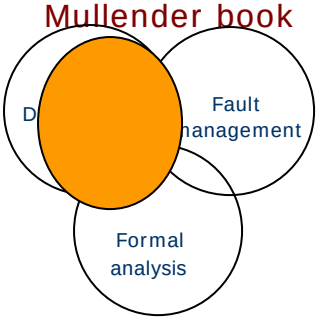
Nancy Lynch book



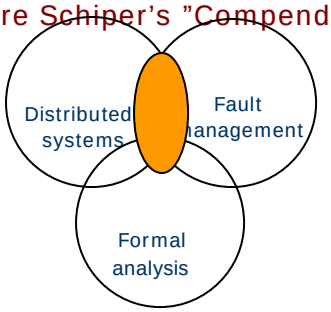
Part III of Tel´s book



Mullender book



Andre Schiper's "Compendium"



Errors, faults & Failures

- **Fault**: a defect within the system or a situation that can lead to failure
- **Error**: manifestation (symptom) of the fault - an unexpected behaviour
- **Failure**: system not perfoming its intended function

Examples

- Year 2000 bug
- Bit flips in hardware due to cosmic radiation in space
- Loose wire
- Air craft retracting its landing gear while on ground

Effects in time:
Permanent/ transient/ intermittent

Fault ⇒ Error ⇒ Failure

- Goal of system verification and validation is to ”remove” faults
- Goal of hazard analysis and FTA is to focus on important faults, those that lead to catastrophic failures
- Goal of fault-tolerance methods is to reduce effects of errors if they appear - *eliminate or* delay *failures*

On-line fault-management

- Fault-detection
  - By program or its environment
- Fault-tolerance (containment) using redundancy
  - software
  - hardware
  - data

From article in Edinburgh Review, 1824: D. Lardner

"The most certain and effectual check upon errors which arise in the process of computation is to cause the same computations to be made by separate and independent computers; and this check is rendered still more decisive if their computations are carried out by different methods."*

* *people who compute*

Static Redundancy

To be used all the time (whether errors showed up or not), just in case...

- SW: Active replication of servers
- HW: Voting and masking
- Data: Parity bits, checksums

Limitations

- ”N-version” programming, a word of caution:
- Main problem is to get the replicas to do differently in test cases that may lead to failures
- The Night/Leveson experiment:
  - The erroneous behaviours were to be found by pre-determined test cases. Some errors missed by all the 28 partners in the experiment!

Dynamic Redundancy

Used when error has occured and must be contained

- SW: recovery methods
- HW: switch to back-up modules
- Data: self-correcting codes

Error recovery

Backward:

- roll back the system to a safe state which was reached before the error appeared (when did error appear?)
- restart with alternative module (how is the result affected by earlier module´s side effects?)

Error recovery

Forward:

- ”fix the error” and continue as if nothing happened
- redundancy lies where one fixes the error

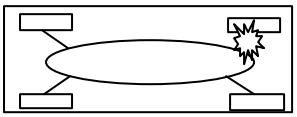
Distributed systems

- Introduce new complications
  - no global clock
  - richer failure models
- Software replication and group mechanisms
  - transparency in treatment of faults

Types of failures

- Node failures
  - Crash
  - Omission
  - Byzantine
- Channel failures
  - Crash (and potential partitions)
  - Message loss
  - Erroneous/arbitrary messages

Brake-by-wire



Adding tolerance

- How to represent a fault-intolerant system?
- What it means to add tolerance, for which type of fault, which type of method?

[Arora & Kulkarni 98, Gärtner 99]

Simple model

- Distributed reactive programs: a set of processes each with a set of variables representing local state
- Each process: a set of actions, specified as guarded commands

  Guard $\rightarrow$ Command
- Program **P**: $P_1 \,||\, P_2 \,||\ldots||\, P_n$

Guarded commands

- If the Boolean condition (the guard) for an action is true, then the action is enabled: it *may* take place

$$\neg ready \wedge y < 10 \rightarrow x := 0; z := 1$$

- Fairness: if a guard is true infinitely often the action will be eventually taken

Computations

- Each computation (run) in the distributed system: a potentially infinite sequence of the (distributed) states
- Based on interleaving of computations of the individual processes

$$g : s_1 s_2 \ldots s_k \ldots$$

- A run is sometimes described in terms of sequence of events leading to a change of state

Desired behaviours

- Behaviours: sets of computations
- Desired properties defined as sets of computations **S**:
  - Safety (what should not happen)
  - Liveness (what should happen)

Correctness

- To show that **P** is correct wrt safety property **S**

show that
set of computations for **P** $\subseteq$ **S**

To add tolerance

- Must decide:
  - What fault classes to tolerate
  - How to detect them
  - What action to take on detection
- Later: ensure that addition of tolerance does not sacrifice correctness

Chosen fault models

- Example: those leading to crash failures
- Extend the program with fault actions, and fault effects based on the chosen fault model

Considering faults

```
Begin
var wait: boolean init false
var up: boolean init true {* to detect error *}
{* normal actions *}
up Ù ¬ wait ® send(m); wait := true
||
up Ù wait Ù rec(a) ® wait:= false
||
{* fault action *}
up ® up := false   {* crash *}
end
```



```
Begin
var wait: boolean init false
var up: boolean init true {* to detect error *}
{* normal actions *}
up Ù ¬ wait ® send(m); wait := true
||
up Ù wait Ù rec(a) ® wait:= false
||
{* fault action *}
up ® up := false   {* crash *}
||
{* protection mechanism *}
¬ up ® up := true  {* recovery *}
end
```


How does FT affect computations?

- Can formalise the effects of fault-tolerance on program behaviour
- Let predicates over state variables denote the set of states in which the predicate holds

$j_1 : x < 10 \wedge y < 1 \qquad j_2 : x < 100 \wedge y < 10$

Formalising fault-tolerance

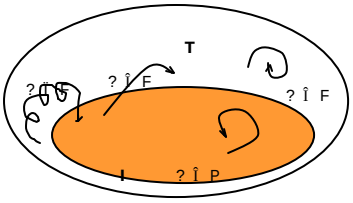
A distributed program **P** *tolerates faults* from a fault class **F** for an invariant **I** iff there exists a predicate **T** such that 3 conditions apply:

- **I** Þ **T**
- **T** is closed in **P** and **F**
- **P** actions in **T** eventually lead to **I**

What does it mean?

- at any state where **I** holds, **T** holds too
- starting from any state in **T**, if any **P** or **F** actions are performed, the resulting state is in **T**
- starting from any **T** state, every sequence of **P** actions alone, eventually reaches a state in **I**

Reachable system states



Two classes of algorithms

- Robust: Correct processes behave correctly even if some processes fail
- Stablising: The behavoius of a correct process may be affected by failures in other processes, but the system is guaranteed to return to a correct configuration

This course

- Treats formal correctness of robust fault-tolerance algorithms
- Briefly covers stablising algorithms
- Mainly treats benign failures and to very little extent malicious failures