Fault-Tolerance in Distributed Systems
Fault-Tolerance

- So far, assumed the system has been reliable

- Fault-tolerant computing by sequential algorithms and single-processors is limited

- Distributed systems have the \textit{partial-failure} property
  - As the number of components (e.g. processes) are increased:
    - It is \textit{extremely} likely that failure will occur in some component
    - It is \textit{extremely} unlikely that failure will occur in all components
Example

- The PASS (Primary Avionics Software System) developed by IBM in 1981, was used in a space shuttle
  - Could have been done on one computer
  - But 4 separate processes were used for fault-tolerance
    - Voting on the outcome
Space Shuttle DS hardware
Two main approaches to Fault-tolerance

- The two main approaches to fault-tolerance in distributed systems are
  - Robust Algorithms
  - Stabilizing Algorithms (sometimes Self-stabilizing)
Fault-tolerance: Robust Algorithms

- Robust algorithms
  - Correct processes should continue behaving correctly in spite of failures
  - Tolerate failures by using replication and voting
  - Never wait for all processes (Tree algorithm, Echo algorithm) because processes could fail
  - Usually deal with permanent faults
Fault-tolerance: Stabilizing Algorithms

- Stabilizing algorithms
  - Correct processes might be affected by failure, but will eventually become correct.
  - The system can start in any state (possibly faulty), but should eventually resume correct behavior.
  - Usually deal with *transitory faults*
Failure models and their hierarchies

- Initially dead processes (*benign fault*)
  - A subset of the processors never ever start

- Crash model (*benign fault*)
  - A process functions properly according to its local algorithm until a certain point where it stops indefinitely
  - Never restarts again

- Byzantine behavior (*malign fault*)
  - The algorithm may execute any possible local algorithm
  - May send and receive arbitrary messages
Failure hierarchy

- **Dead process** special case of **crashed process**
  - Case when the crashed process crashes before it starts executing

- **Crashed process** special case of **Byzantine process**
  - Case when the Byzantine process crashes, and then keeps staying in that state for all future transitions
How good is a robust algorithm?

- Byzantine-robust implies Crash-robust

- Crash-robust implies Initially-dead-robust
Typical Guarantees

- Turns out, robust algorithms can typically tolerate:
  - $N/2$ benign failures for $N$ processes
  - $N/3$ malign failures for $N$ processes
Decision Problems

- Robust algorithms typically try to solve some decision problem
  - Each correct process irreversibly “decides”
Requirement

Three requirements on decision problems:

- **Termination**
  - All correct processes *eventually* decide

- **Consistency**
  - *Constraint* on different processes decisions:
    - Consensus problem: every decide should be equal
    - Election: Every decide except one should be the same

- **Non-triviality**
  - Fixed trivial outputs (e.g. always decide “yes”) excluded
  - Processes *should need to communicate* to be able to solve the problem
Consensus Problems

- Consensus problems very important in DS
  - Distributed Databases
    - All processes must agree whether to *commit* or *abort* a transaction
    - If any process says abort, all processes should abort
Fundamental Result: FLP85

- Reliable Broadcast
  - All correct processes deliver the same set of messages
  - The set only contains messages from correct processes

- Atomic Broadcast (reliable)
  - A reliable broadcast where it is guaranteed that every process receives its messages in the same order as all the other processes
Atomic Broadcast $\rightarrow$ Consensus

- Given a reliable atomic broadcast
  - We can implement a consensus algorithm

- Let every node broadcast either 0 or 1
  - Decide on the first number that is received
  - Since every correct process will receive the messages in the same order, they will all decide on the same value

- Solving Reliable Atomic Broadcast is equivalent to solving consensus
Fischer, Lynch, Paterson 1983/85

- Consensus cannot be solved in asynchronous model
  - With possibility of one process crashing

- http://www.sics.se/~ali/flp85.pdf

- Most influential paper award PODC 2001
Reminder, order of events

- **Intuition**
  - The order in which two *applicable* events are executed is not important!

- **Order Theorem**
  - Let $e_p$ and $e_q$ be two events on two different processors $p$ and $q$ which are both applicable in configuration $\gamma$. Then $e_p$ can be applied to $e_q(\gamma)$, and $e_q$ can be applied to $e_p(\gamma)$.
  - Moreover, $e_p(e_q(\gamma)) = e_q(e_p(\gamma))$. 
Definitions

- A sequence of events $\sigma = (e_1, e_2, \ldots, e_k)$ is applicable in configuration $\gamma$ if $e_1$ is applicable in $\gamma$, $e_2$ applicable in $e_1(\gamma)$...

- If the resulting configuration is $\delta$ we write $\sigma(\gamma) = \delta$ or $\gamma \approx_{\sigma} \delta$

- If $\sigma$ only contains events of a subset of the processes $P$, we write $\gamma \approx_P \delta$
Order of sequences

- Diamond Theorem
  - Let sequences $\sigma_1$ and $\sigma_2$ be applicable in configuration $\gamma$, and let no process participate in both $\sigma_1$ and $\sigma_2$. Then $\sigma_2$ is applicable in $\sigma_1(\gamma)$, $\sigma_2$ is applicable in $\sigma_2(\gamma)$, and $\sigma_1(\sigma_2(\gamma)) = \sigma_2(\sigma_1(\gamma))$

- Proof
  - By induction using the order theorem
Illustration of the Diamond Theorem

\[ \delta = \sigma_2(\sigma_1(\gamma)) = \sigma_1(\sigma_2(\gamma)) \]
Summary

- Distributed Systems attractive when it comes to fault-tolerance
- Two main methods to achieve fault-tolerance
  - Robust algorithms
  - Stabilizing algorithms
- Different failure models
  - Initially dead
  - Crashed
  - Byzantine
- Consensus important
  - Transactions
  - Atomic Broadcast
- Consensus impossible in asynchronous systems with one possible fault