Fault tolerance in distributed systems

- Motivation
- robust and stabilizing algorithms
- failure models
- robust algorithms
  - decision problems
  - impossibility of consensus in asynchronous networks with crash-failures
  - consensus and agreement with initially-dead processes - knot calculation algorithm
- stabilization
  - Dijkstra's K-state algorithm

Why fault tolerance

- Distributed systems encompass more and more individual devices
- the chance of failure in distributed system can grow arbitrarily large when the number of its components increases
- distributed systems can hardly be restarted after failure
- distributed systems are subjects to *partial failure* property: when one of the components fails the system may still be able to function in a decreased capacity
- as the system grows in size
  - it becomes more likely that one component fail
  - it becomes less likely that the failure occurs in all components
- thus the systems able to deal with failures are attractive

Robust and stabilizing algorithms

- An algorithm is *robust* (*masking*) if the correct operation of the algorithm is ensured even at the presence of specified failures
- the algorithm is stabilizing if it is able to eventually start working correctly regardless of the initial state.
  - stabilizing algorithm does not guarantee correct behavior during recovery
  - stabilizing algorithm is able to recover from faults regardless of their nature (as soon as the influence of the failure stops)
- an algorithm can mask certain kinds of failures and stabilize from others
  - for example: an algorithm may mask message loss and stabilize from topology changes

Failure Models

- Faults form a hierarchy on the basis of the severity of faults
- benign
  - initially dead - a process is initially dead if it does not execute a single step in its algorithm
  - crash model - a process executes steps correctly up to some moment (crash) and stops executing thereafter
- malign - Byzantine - a process executes arbitrary steps (not necessarily in accordance with its local algorithm). In particular Byzantine process sends messages with arbitrary content
- initially dead process is a specially case crashed process which is a special case of Byzantine process
  - if algorithm is Byzantine-robust it can also tolerate crashes and initially dead processes
  - if a problem cannot be solved for initially dead processes, it cannot be solved in the presence of crashes or Byzantine failures
- other fault models can be defined in between

Decision problems

- Study of robust algorithms centers around decision problems
- *decision problem* requires that each (correct) process eventually and irreversibly arrives at a "decision" value
- decision problems requirements:
  - termination - all correct processes decide (cannot indefinitely wait for dead processes)
  - consistency - the decisions of correct processes should be related;
    - *consensus problem* - the decisions are equal
    - *election problem* - only one process arrives at "1" (leader) the others - "0" (non-leaders)
  - non-triviality - different outputs are possible in different executions of the algorithm

Impossibility of consensus

- State is *reachable* if there is a computation that contains it
- Each process has a read-only input variable x_p and write-once output variable y_p initially holding b
- A consensus algorithm is 1-crash robust it it satisfies the following properites:
  - termination - in every 1-crash fair execution all correct processes decide
  - agreement - if, in any reachable state, $y_p \neq b$ and $y_q \neq b$ for correct processes *p* and *q* then, $y_p = y_q$
  - non-triviality - there exist a reachable states such that for some *p*, y_p=1 in one state and y_p=0 in another
- Theorem: there are no asynchronous, deterministic 1-crash robust consensus algorithms
- intuitively this result is explained by the fact that in an asynchronous system it is impossible to distinguish a crashed process from an infinitely slow one

What is possible

- Consensus with initially dead-process fault model is possible
- weaker coordination problems than consensus (such as renaming) are solvable
  - given: a set of processes $p_1,..,p_N$, each process with distinct identity taken from arbitrary large domain. Each process has to decide on a unique new name from smaller domain 1,…,K
- randomized algorithms are possible even for Byzantine failures
- weak termination - termination required only when a given process (general) is correct, the objective is for all processes to learn the general's decision; solvable even in the presence of Byzantine faults
- synchronous systems are significantly more fault tolerant

Consensus with initially dead processes

- If processes are only initially-dead consensus is possible.
- Based on the following knot-computation algorithm
- knot is a strongly connected sub-graph with no outgoing edges
- the objective is for all correct processes to agree on the subset of correct processes
- *L* stands for $\lceil (N+1)/2 \rceil$
- we assume that are at least *L* alive processes
- first phase: each process *p*:
  - sends messages to all processes in the system
  - collects at least *L* messages in set $Succ_p$
- a process is a successor if *p* got a message from it - there is a graph *G* in the system
- thus each correct process has *L* successors
- an initially-dead process does not send any messages. Thus there is a knot in G containing correct processes

Knot calculation algorithm

var $Succ_p$, $Alive_p$, $Rcvd_p$: sets of processes **init** $\varnothing$;

begin shout $\langle$ **name**, $p \rangle$;
    (* that is: **forall** $q \in \mathbb{P}$ **do** send $\langle$ **name**, $p \rangle$ to q *)
    while $\#Succ_p < L$
        do begin receive $\langle$ **name**, $q \rangle$; $Succ_p := Succ_p \cup \{q\}$ **end** ;
    shout $\langle$ **pre**, p, $Succ_p \rangle$;
    $Alive_p := Succ_p$;
    while $Alive_p \not\subseteq Rcvd_p$
        do begin receive $\langle$ **pre**, q, $Succ \rangle$;
            $Alive_p := Alive_p \cup Succ \cup \{q\}$;
            $Rcvd_p := Rcvd_p \cup \{q\}$
        end ;
    Compute a knot in G
end

Knot calculation algorithm (cont.)

- Since each correct process has an outdegree *L* - the knot has at least *L* processes
- since *L* > *N*/2, *G* contains just one knot. Let's call it *K*
- since *p* has *L* successors, one of them is in *K*, thus all nodes in *K* are descendants of *p*
- second phase:
  - each process collects a list of its descendants. Since processes do not fail, no deadlock occurs at this stage
- in the end a process has a set of processes and their descendants which allows it to compute the knot in *G*
- it is possible to do an election and consensus on the basis of knot calculation algorithm
  - election - since processes agree on the knot - they all can agree on the leader by electing the leader - the process with the highest id in K
  - consensus - all processes calculate the input value of the processes of *K* and output the value that occurs most often

Guarded Command Language (GCL)

$$
\begin{array}{l}
*[\\
\quad guard_1 \rightarrow command_1 \\
\quad [\,]guard_2 \rightarrow command_2 \\
\quad \vdots \\
]
\end{array}
$$

- *[…] - execution repeats forever
- $guard_i$ - binary predicate on local vars, received messages, etc.;
- $command_i$ - list of assignment statements;

command is executed when corresponding *guard* is true;
guards are selected nondeterministically,

Advantages:

- GCL allows to easily reason about algorithms and their executions: the program counter position is irrelevant or less important;
- we don't have to consider execution starting in the middle of guard or command (*serializability property*);

Dijkstra's K-State Token Circulation Algorithm

Objective: circulate a single token among processors

Processor ρ^0
*[
$s^0 = s^{k-1} \rightarrow s^0 := (s^0 + 1) \bmod K$
]

Processor ρ^i $(0 < i < K)$
*[
$s^i \neq s^{i-1} \rightarrow s^i := s^{i-1}$
]

- the system consists of a ring of *K* processors (*id*s 0 through *K*-1)
- each processor maintains a state variable *s*; a processor can see the state of it's left (smaller id) neighbor
- guard evaluates to **true** - processor has a privilege (token)
- all processors evaluate their guards, only **one at a time** changes state (C-Daemon)
- after the state change all processors re-evaluate the guards