Example algorithm that is both stabilizing and robust

- Robustness and stabilization - to approaches to combat system faults
- guarded command language review
- alternating bit protocol ABP - robust but not stabilizing
- stabilizing ABP
- stabilizing ring token token circulation algorithm

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

Guarded Command Language (GCL)

- *[ ...] - 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);

Alternating Bit Protocol

Objective: transmit data reliably from sender to receiver over unreliable channel

Invariant: no more than 2 msgs in system, correct message carries same number as ns

process p

- *[ receive ack(i) ♦
  - if i = ns then
    - ns := \downarrow ns
    - ms := get()
    - send data(ms, ns)

- [] timeout ♦
  - send data(ms, ns)
]

process q

- *[ receive data(mr, i) ♦
  - put(mr)
  - send ack(i)
]
Stabilizing
Alternating Bit Protocol

Invariant: numbers carried by messages (and nr) monotonically decrease and no greater than ns

process p
[*]
  receive ack(i)  ♦
    if i = ns then
      ns := ns + 1
      ms := get()
    send data(ms,ns)
  []
    timeout ♦
    send data(ms,ns)
]*

process q
[*]
  receive data(mr,i)  ♦
    if i < nr then
      put(mr)
      nr := i
    send ack(i)
]*

Dijkstra’s K-State Token Circulation Algorithm

Objective: circulate a single token among processors

• the system consists of a ring of K processors (ids 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

Processor ρ₀
[*]
  s₀ = s₀ - 1 ♦ s₀ := (s₀ + 1) mod K
]*

Processor ρ₁ (0 < i < K)
[*]
  sᵢ ≠ sᵢ - 1 ♦ sᵢ := sᵢ - 1
]*

states
• 0  • 1  • 2
• 3  • 4

simulation