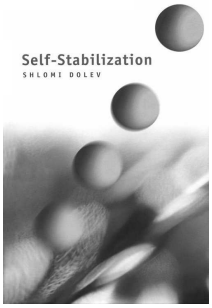
Chapter 6
Self-Stabilization



Self-Stabilization
Shlomi Dolev
MIT Press , 2000

Shlomi Dolev, All Rights Reserved

Chapter 6: Convergence in the Presence of Faults - Motivation

- Processors crash and software (and hardware) may contain flaws
- Byzantine and crash failures are both well studied models
- Algorithms that tolerate failures are of practical interest
- The focus of this presentation is the integration of self-stabilization with other fault models

Byzantine Faults

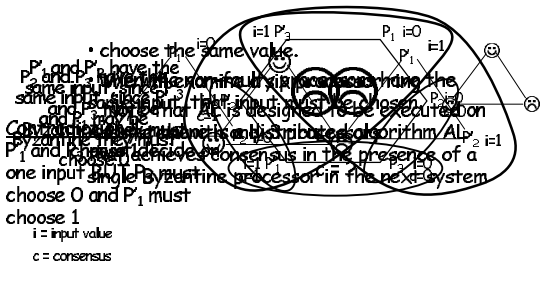
"Byzantine" – permanent faults

- The type of faults is not known in advance
- Processors can exhibit arbitrary "malicious", "two faced" behavior
- Models the possibility of code corruption

Byzantine Fault Model

- A Byzantine processor "fights" against the rest of the processors in order to prevent them from reaching their goal
- If more than 1/3 of the processors are Byzantine it is impossible to achieve basic tasks such as consensus in distributed systems

More than 1/3 of the processors are Byzantine ➔ No convergence



The Use of Self-Stabilization

- What happens if
  - For a short period, more than 1/3 of the processors are faulty or perhaps temporarily crashed?
  - Messages from a non-faulty processor are lost?
- Such temporary violations can be viewed as leaving a system in an arbitrary initial state

Self-Stabilizing algorithms that cope with Byzantine and transient faults and stabilize in spite of these faults are presented, and demonstrate the generality of the self-stabilization concept!

Chapter 6: roadmap

6.1 Digital Clock Synchronization
6.2 Stabilization in Spite of Napping
6.3 Stabilization in Spite of Byzantine Faults
6.4 Stabilization in the Presence of Faults in Asynchronous Systems

Digital Clock Synchronization - Motivation

- Multi processor computers
- Synchronization is needed for coordination – clocks
  - Global clock pulse & global clock value
  - Global clock pulse & individual clock values
  - Individual clock pulse & individual clock values
- Fault tolerant clock synchronization

Digital Clock Synchronization

- In every pulse each processor reads the value of it's neighbors clocks and uses these values to calculate its new clock value .
- The Goal
  (1) identical clock values
  (2) the clock values are incremented by one in every pulse

Digital Clock Sync – Unbounded version

```
01 upon a pulse
02     forall P_j ∈ N(i) do send (j,clock_i)
03     max := clock_i
04     forall P_j ∈ N(i) do
05         receive(clock_j)
06         if clock_i > max then max := clock_i
07     od
08     clock_i := max + 1
```

- A simple induction can prove that this version of the algorithm is correct:
  - If P_m holds the max clock value, by the i'th pulse every processor of distance i from P_m holds the maximal clock value

Digital Clock Synchronization – Bounded version

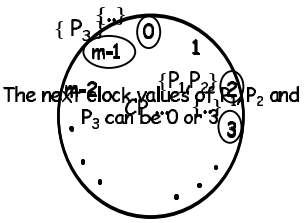
- Unbounded clocks is a drawback in self-stabilizing systems
- The use of 2^{64} possible values does not help creating the illusion of "unbounded":
  - A single transient fault may cause the clock to reach the maximal clock value …

Digital Clock Sync – Bounded version (max)

```
01 upon a pulse
02     forall P_j ∈ N(i) do send (j,clock_i)
03     max := clock_i
04     forall P_j ∈ N(i) do
05         receive(clock_j)
06         if clock_i > max then max := clock_i
07     od
08     clock_i := (max + 1) mod ((n +1)d +1)
```

- The Boundary M = ((n+1)d+1)
- Why is this algorithm correct?
  - The number of clock values can only decrease, and is reduced to a single clock value

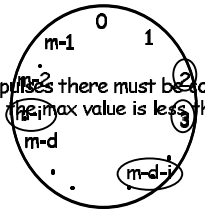
The number of distinct clock values may only decrease



The next clock values of P_1, P_2 and P_3 can be 0 or 3

Digital Clock Sync - Bounded version (max)

- Why is this algorithm correct?
  - If all the clock values are less than M-d we achieve sync before the modulo operation is applied



After d pulses there must be convergence and the max value is less than m

Digital Clock Sync - Bounded version (max)

- ... Why is this algorithm correct?

  If not all the clock values are less than M-d
  - By the pigeonhole principle, in any configuration there must be 2 clock values x and y such that $y-x \geq d+1$, and there is no other clock value between
  - After M-y+1 pulses the system reaches the configuration in which all clock values are less than M-d

Digital Clock Sync - Bounded version (min)

- The Boundary - M = 2d+1
- Why is this algorithm correct?
  - If no processor assigns 0 during the first d pulses - sync is achieved (can be shown by simple induction)

  Else
  - A processor assigns 0 during the first d pulses, d pulses after this point a configuration c is reached such that there is no clock value greater than d: the first case holds

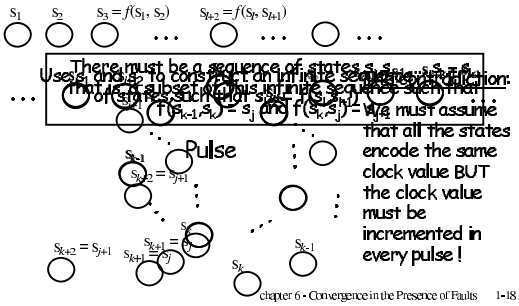
```
01 upon a pulse
02     forall P_j ∈ N(i) do send (j,clock_i)
03     min := clock_i
04     forall P_j ∈ N(i) do
05         receive(clock_j)
06         if clock_i < min then min := clock_i
07     od
08     clock_i := (min + 1) mod (2d +1)
```


Digital clocks with a constant number of states are impossible

- A special case will imply a lower bound for the general case
- A processor can read only the clock of a subset of its neighbors
- In a undirected ring every processor has a left and right neighbor, and can read the state of its left neighbor
- $s_i^{t+1} = f(s_{i-1}^t, s_i^t)$

  s_i^t - state of P_i in time t, f - the transition function
- |S| - the constant number of states of a processor
- The proof shows that in every step, the state of every processor is changed to the state of its right processor

Digital clocks with a constant number of states are impossible



There must be a sequence of states $s_1, s_2, \ldots, s_k, s_{k+1}, \ldots$ that is a subset of this infinite sequence such that $f(s_{k-1}, s_k) = s_j$ and $f(s_k, s_{k+1}) = s_{j+1}$

We must assume that all the states encode the same clock value BUT the clock value must be incremented in every pulse !

Chapter 6: roadmap

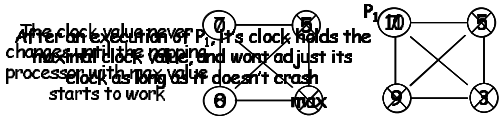
6.1 Digital Clock Synchronization
6.2 Stabilization in Spite of Napping
6.3 Stabilization in Spite of Byzantine Faults
6.4 Stabilization in the Presence of Faults in Asynchronous Systems

Stabilizing in Spite of Napping

- Wait-free self-stabilizing clock-synchronization algorithm is a clock-sync. Algorithm that copes with transient and napping faults
- Each non-faulty operating processor ignores the faulty processors and increments its clock value by one in every pulse
- Requiring the existence of a fixed integer k, once a processor P_i works correctly for at least k time units and continues working correctly, the following properties hold:
  - Adjustment P_i does not adjust its clock
  - Agreement P_is clock agrees with the clock of every other processor that has also been working correctly for at least k time units

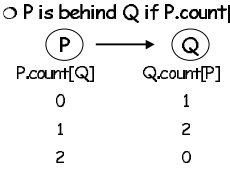
Algorithms that fulfill the adjustment-agreement – unbounded clocks

- Simple example for k=1, using the unbounded clocks
  In every step - each processor reads the clock values of the other processors, and chooses the maximal value (denote by x) and assigns x+1 to its clock
  Note that this approach wont work using bounded clock values



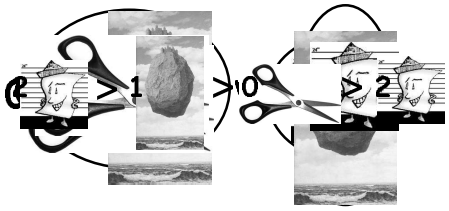
Algorithms that fulfill the adjustment-agreement – bounded clock values

- Using bounded clock values (M)
  - The idea – identifying crashed processors and ignoring their values
- Each processor P has:
  - P.clock ∈ (0... M-1)
  - ∀Q P.count[Q] ∈ (0,1,2)
- P is behind Q if P.count[Q]+1 (mod 3) = Q.count[P]



Algorithms that fulfill the adjustment-agreement – bounded solution

- The implementation is based on the concept of the "scissors, paper, stone" children's game



A