Computer Networks

EDA387/DIT663

Fault-tolerant Algorithms for Computer Networks

Convergence in the Presence of Faults (Ch. 6)

Today

- We focus on the integration of self stabilization and other fault models, such as crash failures, that can occur during the system execution and not only before the first configuration.

Chapter 6: roadmap

6.1 Digital Clock Synchronization

6.2 Stabilization in Spite of Napping

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

upon a pulse
  forall $P_j \in N(i)$ **do send** $(j, clock_i)$
  $max := clock_i$
  forall $P_j \in N(i)$ **do**
    receive$(clock_j)$
    if $clock_j > max$ **then** $max := clock_j$
  od
  $clock_i := max + 1$

- 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)

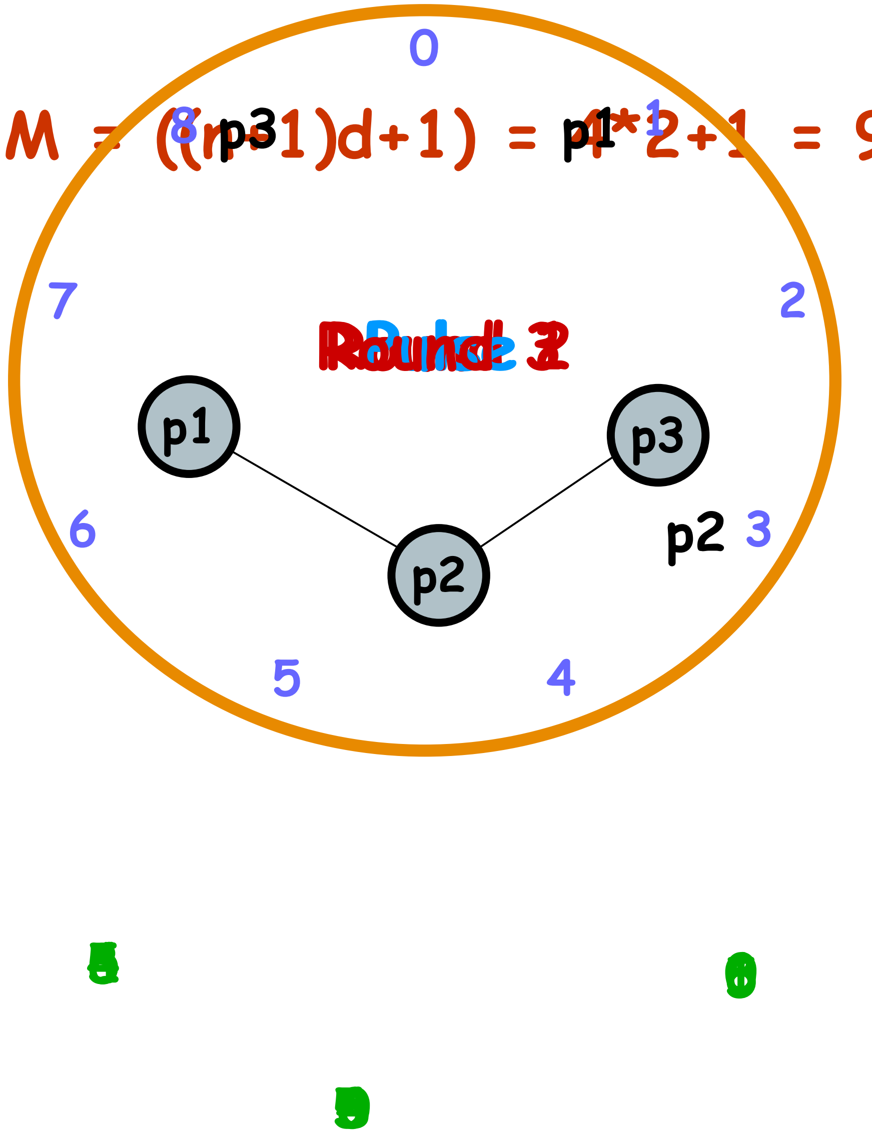
<u>Converge-to-the-max</u>

upon a pulse
    forall $P_j \in N(i)$ **do send** $(j, clock_i)$
    $max := clock_i$
    forall $P_j \in N(i)$ **do**
        receive$(clock_j)$
        if $clock_j > max$ **then** $max := clock_j$
    od
    $\boxed{clock_i := (max + 1) \bmod ((n+1)d+1)}$

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

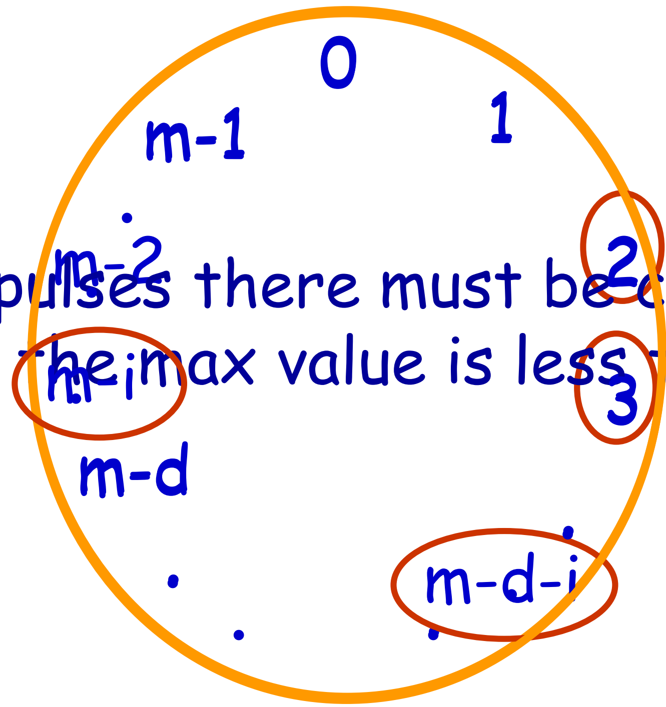
For Example:

$M = ((n-1)d+1) = 4*2+1 = 9$



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 i steps, no clock value that is in $(x+i, y+i)$

  – After M-y+1 pulses the system reaches a 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

```
upon a pulse
        forall P_j ∈ N(i) do send (j,clock_i)
        min := clock_i
        forall P_j ∈ N(i) do
                receive(clock_j)
                if clock_j < min then min := clock_j
        od
        clock_i := (min + 1) mod (2d +1)
```

Digital clocks with a constant number of states are impossible

Consider only deterministic algorithm:

There is no uniform digital clock-synchronization algorithm that uses only a constant number of states per processor.

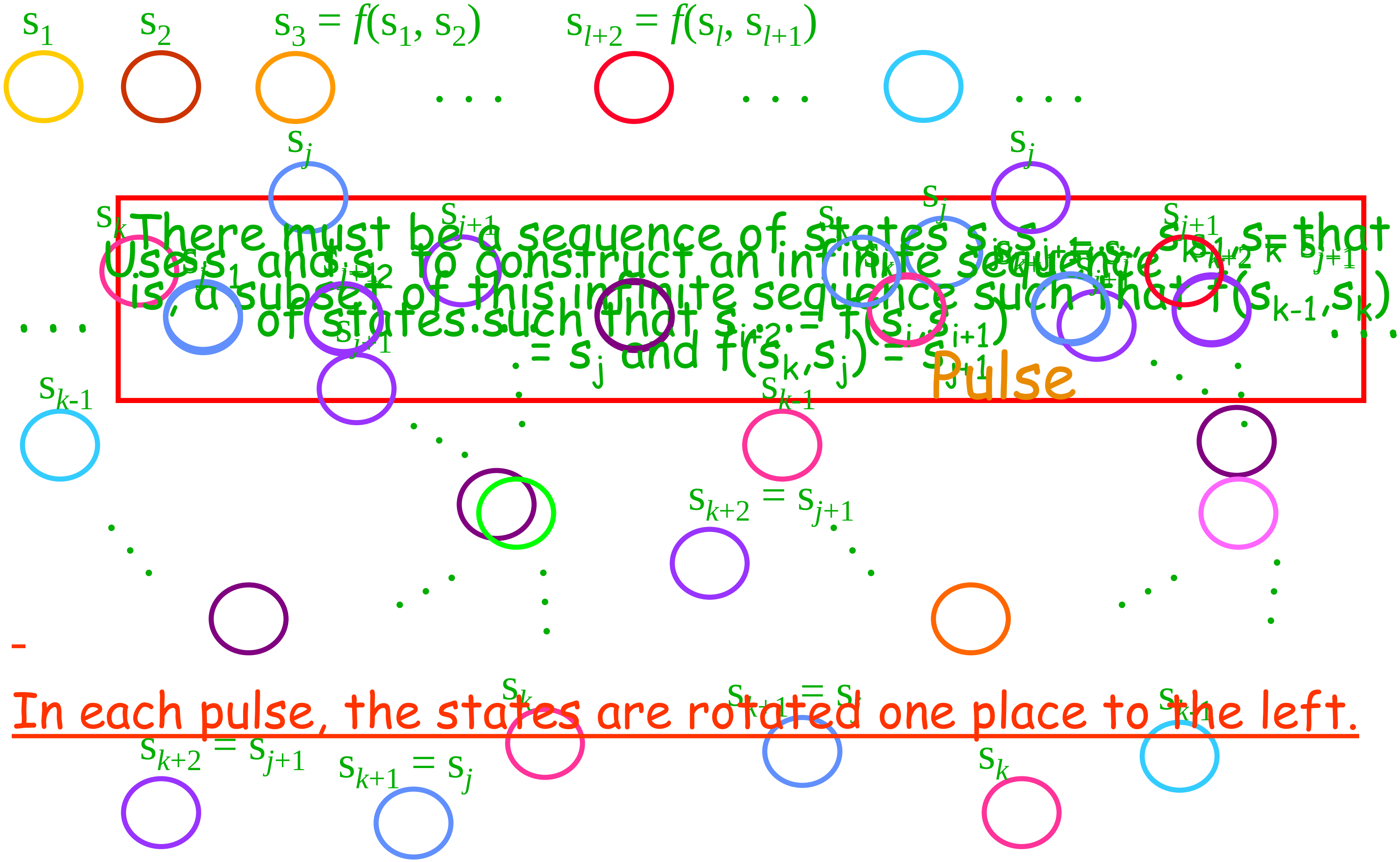
Digital clocks with a constant number of states are impossible

- 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

Digital clocks with a constant number of states are impossible

- The idea is to choose a sufficiently large ring for which the given algorithm will never stabilize.
- The proof shows that a configuration exists for a sufficiently large ring such that the states of the processors rotate:
  - 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



Uses s_1 and s_2 to construct an infinite sequence of states such that $s_{l+2} = f(s_l, s_{l+1})$

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

Pulse

In each pulse, the states are rotated one place to the left.

Digital clocks with a constant number of states are impossible

- Since the states of the processors encodes the clock values, and the set of states just rotates around the ring,
  - It has to be the case in which all the states encode the same clock value.
- On the other hand, the clock value must be increments in every pulse.

  Contradiction.

Do we have to assume that the ring is unidirectional?

Digital clocks with a constant number of states are impossible

- Let s_1 and s_2 be two states in S;
  - e.g., the first two states according to some arbitrary state ordering
- Use s_1 and s_2 to construct an infinite sequence of states such that $s_{l+2} = f(s_l, s_{l+1})$.
- There must be a sequence of states $s_j, s_{j+i}, \ldots, s_{k-1}, s_k$
  - that is, a subset of the above infinite sequence
    - such that $f(s_{k-1}, s_k) = s_j$ and $f(s_k, s_j) = s_{j+1}$ and $k > j + 2$; or, equivalently, $s_{k+1} = s_j$ and $s_{k+2} = s_{j+1}$.

Digital clocks with a constant number of states are impossible

- Any sequence of $|S|^2+1$ such pairs has at least one pair (s_j, s_{j+1}) that appears more than once.
- Thus, any segment of $2(|S|^2+1)$ states in the infinite sequence can be used in our proof.
- Now, we are convinced that there is a sequence s_j, s_{j+i}, …, s_{k-1}, s_k in which the combination $\mathrm{s}_{\mathrm{k+i}}$, $\mathrm{s}_{\mathrm{k+2}}$ and the combination s_{j}, $\mathrm{s}_{\mathrm{j+i}}$ are identical. Therefore, it holds that $f(s_{k-i}, s_k)$=sj and $f(s_k, s_j)=s_{j+y}$.

Digital clocks with a constant number of states are impossible

- Now construct a unidirected ring of processors using the sequence s_j, s_{j+i}, …, s_{k-1}, s_k, where the processor in state s_k is the left neighbor of the processor in state s_j.
  - Each processor uses its own state and the state of its left neighbor to compute the next state;
    - in accordance with our construction, the state s_{j+i} is changed to s_{j+i+1} for every $0 \le i < k-j$, and the state s_k is changed to s_j.
- We conclude that, in each pulse, the states are rotated one place to the left.
- Note that the above is true in an infinite execution starting in the configuration defined above.

Digital clocks with a constant number of states are impossible

- Is it possible that such an infinite execution will converge?
- Since the states of the processors encodes the clock value and the set of states is not changed during an infinite execution (it just rotates around the ring), we must assume that all the states encode the same clock value.
- On the other hand, the clock value must be incremented in every pulse.

Digital clocks with a constant number of states are impossible

- This is impossible, since the set of states is not changed during the infinite execution.
- In the more complicated case of bidirectional rings, the lower-bound proof uses a similar approach.

Chapter 6: roadmap

6.1 Digital Clock Synchronization

6.2 Stabilization in Spite of Napping

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
- Given 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_i's 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

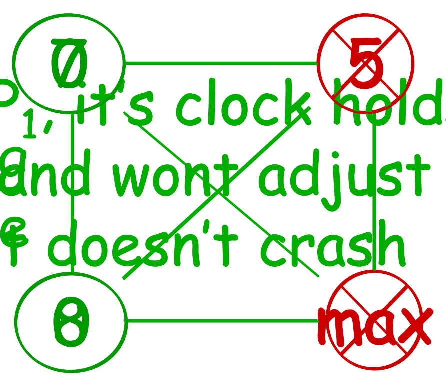
- Simple example for *k =1*, using the unbounded clocks

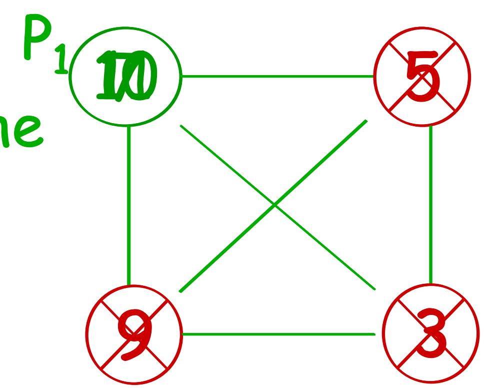
  In every step – each processor reads the clock values of the other processors, and chooses the maximal value (denote by *x*) and assigns *x*+1 to its clock

Note that this approach wont work using bounded clock values

The clock value never changes until the napping processor with max value starts to work

After an execution of P_1, it's clock holds the maximal clock value, and wont adjust its clock as long as it doesn't crash





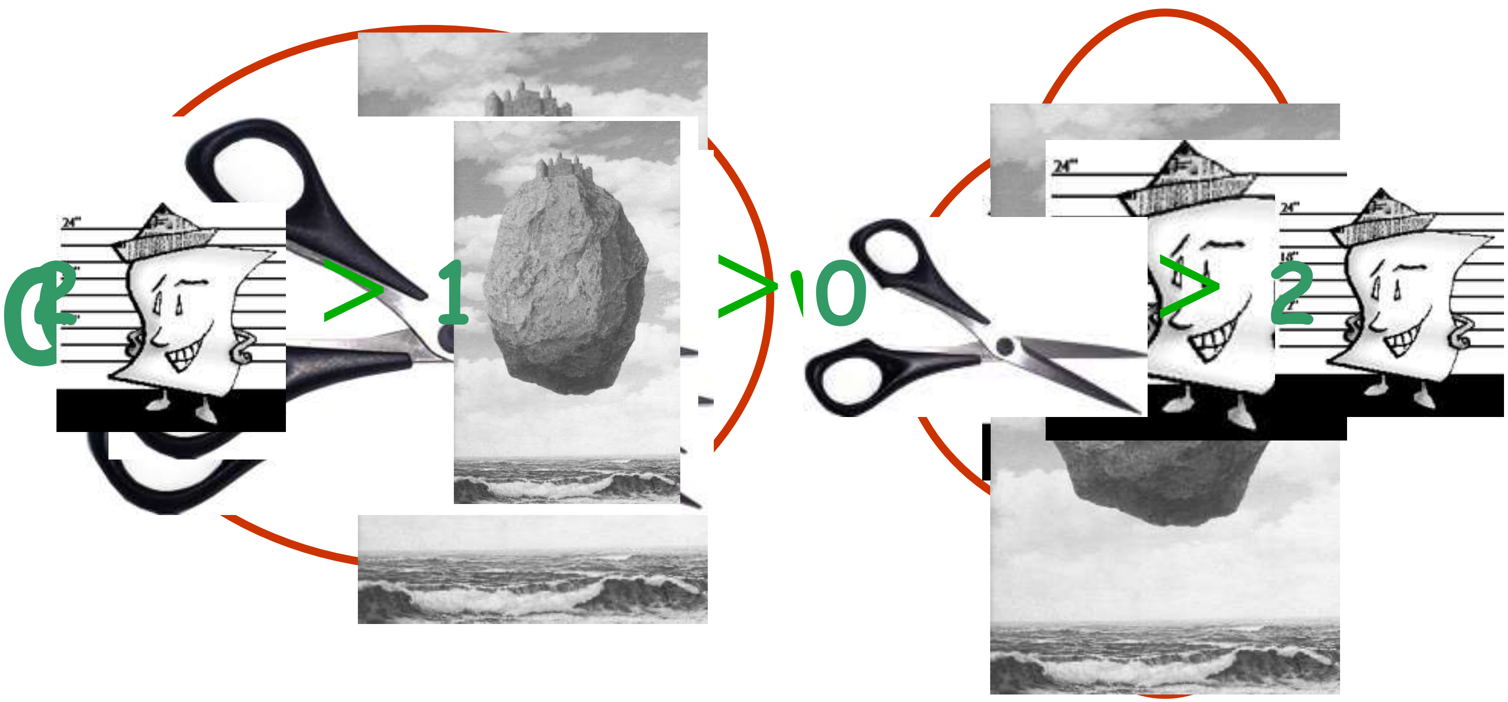
Algorithms that fulfill the adjustment-agreement – bounded clock values

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

P.count[Q]	Q.count[P]
0	1
1	2
2	0

Algorithms that fulfill the adjustment-agreement – bounded solution

- The implementation is based on the concept of the “rock, paper, scissors” children’s game



Algorithms that fulfill the adjustment-agreement – bounded solution

The program for P:

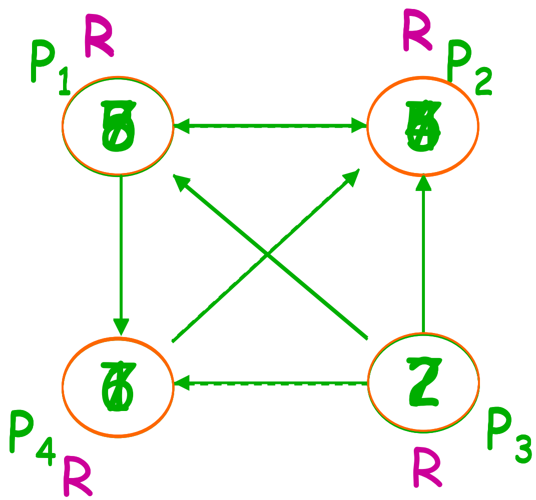
1) Read every count and clock

2) Find the processor set R that are not behind <u>any</u> other processor

3) If $R \neq \emptyset$ then P finds a processor K with the maximal clock value in R and assigns P.clock := K.clock + 1 (mod M)

4) For every processor Q, if Q is not behind P then P.count[Q] := P.count[Q] + 1 (mod 3)

Self-stabilizing Wait-free Bounded Solution – Run Sample



Active processor

Simple connection

“behind” connection

K = 2

The algorithm is wait-free and self-stabilizing

- The algorithm presented is a wait-free self-stabilizing clock-synchronization algorithm with k=2 (Th. 6.1)
  - All processors that take a step at the same pulse, see the same view
  - Each processor that executes a single step belongs to R in the next round, in which all the clock values are the same $\Rightarrow$ the agreement requirement holds
  - Every processor chooses the maximal clock value of a processor in R, and increments it by 1 mod M $\Rightarrow$ the adjustment requirement holds
  - The proof assumes an arbitrary start configuration $\Rightarrow$ the algorithm is both wait-free and self-stabilizing

Theorem 6.1

Theorem 6.1: The above algorithm is a wait-free self-stabilizing clock-synchronization algorithm with $k=2$.

Proof

The proof shows that, starting with any values in the order variables and the clock variables, the algorithm meets the adjustment and agreement requirements.

Theorem 6.1 (cont.)

First note that all processors that take a step at the same pulse, see the same view

because they have access to all fields at all registers.

They then compute the same *NB (non-blocking processors)*, which is R is the above code.

We must show that, if any processor p_i executes more than $k=2$ successive steps, then the agreement and adjustment requirements hold following its second step.

Theorem 6.1 (cont.)

Assume p_i executes more than $k=2$ successive steps.

Observe that NB is not empty following p_i 's first step.

Moreover, while p_i continues to execute steps without stopping, it remains in NB.

- The reason is that p_i executes a step in which it increments every order variable $order_{ij}$, such that p_j is not behind p_i.

Algorithms that fulfill the adjustment-agreement – bounded solution

The program for P:

1) Read every count and clock
2) Find the set R that are not behind <u>any</u> other processor
3) If $R \neq \emptyset$ then P finds a processor K with the maximal clock value in R and assigns
   P.clock := K.clock + 1 (mod M)
4) For every processor Q, if Q is not behind P then
   P.count[Q] := P.count[Q] + 1 (mod 3)

Theorem 6.1 (cont.)

Since *NB* is not empty following the first step of p_i, and since all processors that execute a step see the same set *NB*, all the processors that execute a step following the first step of p_i choose the same clock value.

Thus, following the second step of p_i and while p_i does not stop executing steps, the clock values of the processors that belong to NB are the same.

Theorem 6.1 (cont.)

Every processor that executes a single step belongs to *NB*, and the value of the clocks of all processors in *NB* is the same; thus the agreement requirement holds.

Every processor chooses the maximal clock value m of a processor in *NB* and increments m by 1 modulo M; thus, the adjustment requirement holds as well.

Theorem 6.1 (cont.)

The proof of the theorem assumes an arbitrary starting configuration for the execution with any combination of *order* and *clock* values. Thus our algorithm is both wait-free and self-stabilizing. ■

Summary

We have looked into some common fault models and presented how the task of self-stabilizing clock synchronization can consider them.

Review Questions

1. Can a smaller number than $((n + 1)d + 1)$ be used in line 8 of figure 6.2 without changing any other statement in the code? Prove stabilization or present a contradicting example.

Review Questions

2. Consider digital clock-synchronization algorithms for a line of processors P_1, P_2, …, P_n, where P_i , $2 \leq i \leq n-1$, communicates with the processors P_{i-1} and P_{i+1}; similarly, P_1 communicates with P_2 and P_n with P_{n-1}. Assume that processors have no sense of direction: the fact that P_i considers P_{i-1} its left neighbor does not imply that P_{i-1} considers P_i its right neighbor. Will the algorithm presented in figure 6.2 stabilize when the increment operations are modulo 3? Prove your answer or present a contradicting example.

Review Questions

3. Will the unbounded algorithm presented in figure 6.1 stabilizes if the minimal clock value plus one is assigned to $clock_i$? Will it stabilize if the average clock value (counting only the integer part of the average result) is used?