Introduction to Self-stabilization

Franck Petit
(*with some materials from M. Herlihy & S. Tixeuil*)

Sorbonne University
`Firstname.Lastname@lip6.fr`

Outline

Example

- $U_0 = a$
- $U_{n+1} = \frac{U_n}{2}$ if U_n is even
- $U_{n+1} = \frac{3U_n+1}{2}$ if U_n is odd

Example

- $U_0 = a$
- $U_{n+1} = \frac{U_n}{2}$ if U_n is even
- $U_{n+1} = \frac{3U_n+1}{2}$ if U_n is odd

n	0	1	2	3	4	5	6	7	8	9	10	11
U_n	7	11	17	26	13	20	10	5	8	4	2	1

Example

- $U_0 = a$
- $U_{n+1} = \frac{U_n}{2}$ if U_n is even
- $U_{n+1} = \frac{3U_n+1}{2}$ if U_n is odd

27	41	62	31	47	71	107	161	242
121	182	91	137	206	103	155	233	350
175	263	395	593	890	445	668	334	167
251	377	566	283	425	638	319	479	719
1079	1619	2429	3644	1822	911	1367	2051	3077
4616	2308	1154	577	866	433	650	325	488
244	122	61	92	46	23	35	53	80
40	20	10	5	8	4	2	1	...

Example

- $U_0 = a$
- $U_{n+1} = \frac{U_n}{2}$ if U_n is even
- $U_{n+1} = \frac{3U_n+1}{2}$ if U_n is odd
- Converges towards a “correct” behavior
  - 12121212121212121212121212…
  - Independent from the initial value

Example

- Enumerator of Even Numbers

```
unsigned char x = 0;
...
for (;;)
{
    ...
    printf ("%d ",x);
    x = x + 2;
    ...
}
```

Example

- Self-Stabilizing Enumerator of Even Numbers (Overflow Control)

```
unsigned char x;
...
for (;;)
{
    ...
    printf ("%d ",x);
    x = ((x = x + 2) > 254 ) ? 0: x + 2;
    ...
}
```

Example

- Self-Stabilizing Enumerator of Even Numbers (Parity Check)

```
unsigned char x;
...
for (;;)
{
    ...
    printf ("%d ",x);
    x = (x % 2) ? x + 1 : x + 2;
    ...
}
```

Example

- Self-Stabilizing Enumerator of Even Numbers (Parity Check—Reset)

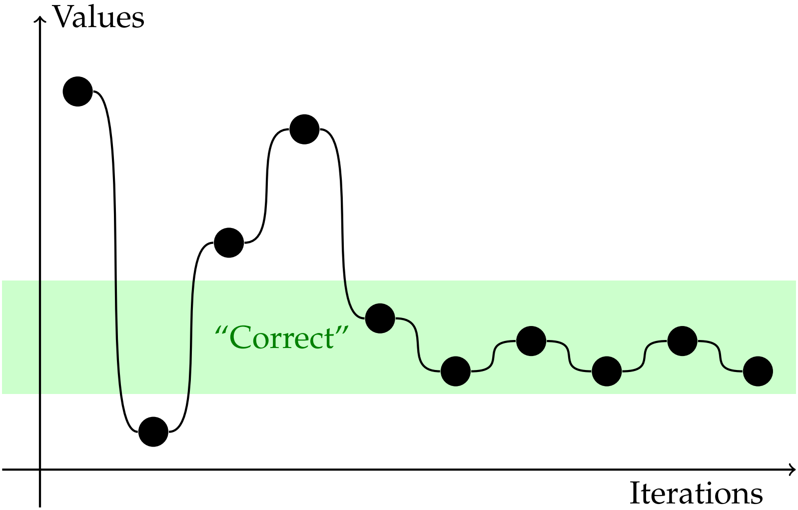
```
unsigned char x;
...
for (;;)
{
    ...
    printf ("%d ",x);
    x = (x % 2) ? 0 : x + 2;
    ...
}
```

Example

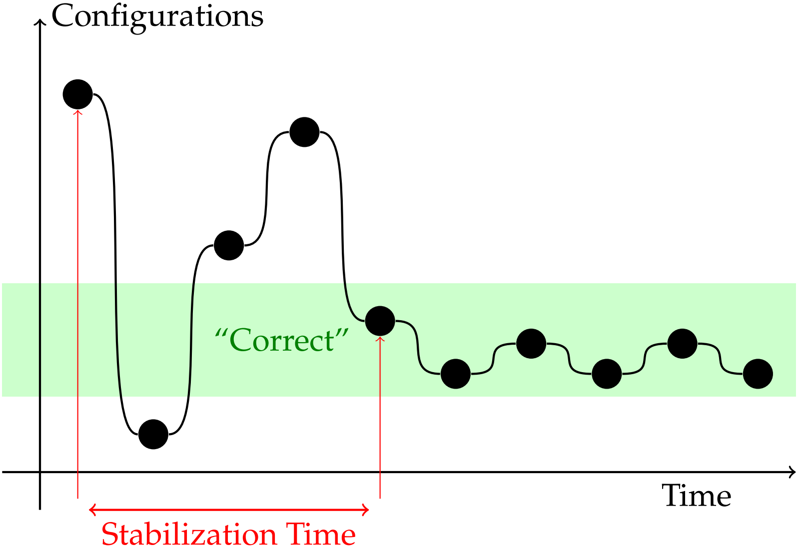
- Self-Stabilizing Enumerator of Even Numbers (Left Shift)

```
unsigned char x;
...
for (;;)
{
    ...
    printf ("%d ",x<<1);
    x++;
    ...
}
```

Example



Self-stabilization



Memory Corruption

- Example of a sequential program:

```
int x = 0;
...
if( x == 0 ) {
  // code assuming x equals 0
}
else {
  // code assuming x does not equal 0
}
```

Distributed Systems

- Locality of information

Distributed Systems

- Locality of information
- Locality of time

Distributed Systems

- Locality of information
- Locality of time
- $\Rightarrow$ non-determinism

Distributed Systems

- Locality of information
- Locality of time
- $\Rightarrow$ non-determinism

Definition (Configuration)

Product of the local states of the system components.

Definition (Execution)

Interleaving of the local executions of the system components.

Distributed Systems

Definition (Classical System, *a.k.a.* Non stabilizing)

Starting from a particular initial configuration, the system immediately exhibits correct behavior.

Definition (Self-stabilizing System)

Starting from any initial configuration, the system eventually reaches a configuration from with its behavior is correct.

Self-stabilization

Definition (Self-stabilizing System)

Starting from any initial configuration, the system eventually reaches a configuration from with its behavior is correct.

- Defined by Dijkstra in 1974

Self-stabilization

Definition (Self-stabilizing System)

Starting from any initial configuration, the system eventually reaches a configuration from with its behavior is correct.

- Defined by Dijkstra in 1974
- Advocated by Lamport in 1984 to address fault-tolerant issues

Fault Tolerance

Definition ((Distributed) Task)

A task is **specified** in terms of:

Fault Tolerance

Definition ((Distributed) Task)

A task is **specified** in terms of:

- **Safety**: *Bad actions*, which should not happen
  - At the intersection, traffic lights are green on two different axes.
  - Processes enter the critical section simultaneously.
  - *Windows* crashes.

Fault Tolerance

Definition ((Distributed) Task)

A task is **specified** in terms of:

- **Safety**: *Bad actions*, which should not happen
- **Liveness**: *Good actions*, which should (eventually) happen
  - At the intersection, if one of the traffic lights is red then, it eventually becomes green.
  - Every process eventually enter the critical section.
  - *Windows* eventually reboots.

Fault Tolerance

Definition (Fault)

A fault is an action that corrupts the task specification by changing the correct functioning of a system component.

Fault Tolerance

Definition (Fault)

A fault is an action that corrupts the task specification by changing the correct functioning of a system component.

- At the intersection, traffic lights are off.
- A process requesting the critical section is stuck.
- *Windows* boot loops on a blue screen with white markings.

Fault Tolerance

Definition (Fault)

A fault is an action that corrupts the task specification by changing the correct functioning of a system component.

- Type $\rightarrow$ fail-stop, crash, omission, Byzantine, …
- Duration
- Detection Rate
- Correction Rate
- Frequency

Fault Tolerance

Definition (Fault)

A fault is an action that corrupts the task specification by changing the correct functioning of a system component.

- Type $\rightarrow$ fail-stop, crash, omission, Byzantine, ...
- Duration
- Detection Rate
- Correction Rate
- Frequency

Fault-tolerant algorithm $\Rightarrow$ Tolerates a given class of faults

Fault Tolerance

- **Masking FT**: Both *Safety* and *Liveness* must be guaranteed.

Fault Tolerance

- **Masking FT**: Both *Safety* and *Liveness* must be guaranteed. Unfortunately, [FLP]!

Fault Tolerance

- **Masking FT**: Both *Safety* and *Liveness* must be guaranteed. Unfortunately, [FLP]!
- **Fail-Safe FT**: *Safety* guaranteed but not *Liveness*.

Fault Tolerance

- **Masking FT**: Both *Safety* and *Liveness* must be guaranteed. Unfortunately, [FLP]!
- **Fail-Safe FT**: *Safety* guaranteed but not *Liveness*.
  - Traffic lights are red.
  - Transactions in databases.

Fault Tolerance

- **Masking FT**: Both *Safety* and *Liveness* must be guaranteed. Unfortunately, [FLP]!
- **Fail-Safe FT**: *Safety* guaranteed but not *Liveness*.
- **Non-Masking FT**: *Liveness* guaranteed but not *Safety*.

Fault Tolerance

- **Masking FT**: Both *Safety* and *Liveness* must be guaranteed. Unfortunately, [FLP]!
- **Fail-Safe FT**: *Safety* guaranteed but not *Liveness*.
- **Non-Masking FT**: *Liveness* guaranteed but not *Safety*.
  - Traffic lights are flashing orange.
  - Optimistic algorithm: Data replication mechanisms.

Fault Tolerance

- **Masking FT**: Both *Safety* and *Liveness* must be guaranteed. Unfortunately, [FLP]!
- **Fail-Safe FT**: *Safety* guaranteed but not *Liveness*.
- **Non-Masking FT**: *Liveness* guaranteed but not *Safety*.
  Self-Stabilization: *Safety* eventually guaranteed.

Dijkstra’ self-stabilizing algorithms

- Token-passing on a ring
- Token-passing on a chain with 4 states/process

Atomicity

- An example of a “stabilizing” sequential program

```
int x = 0;
...
while( x == x ) {
  x = 0;
  // code assuming x equals 0
}
```

Atomicity

- An example of a "stabilizing" sequential program

```
int x = 0;
...
while( x == x ) {
  x = 0;
  // code assuming x equals 0
}
```

```
0 iconst_0
1 istore_1
2 goto 7
...
5 iconst_0
6 istore_1
7 iload_1
8 iload_1
9 if_icmpeq 5
```

Atomicity

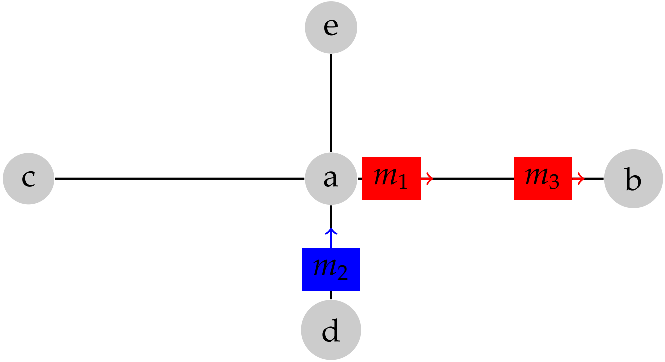
- An example of a “stabilizing” sequential program

```
int x = 0;
...
while( x == x ) {
  x = 0;
  // code assuming x equals 0
}
```

```
0 iconst_0
1 istore_1
2 goto 7
...
5 iconst_0
6 istore_1
7 iload_1
8 iload_1
9 if_icmpeq 5
```

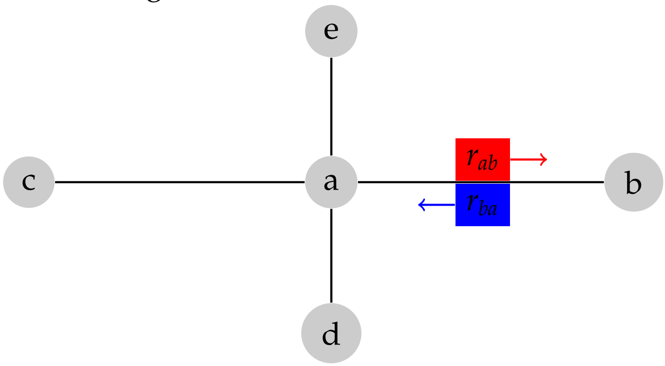
Communications

- Message Passing



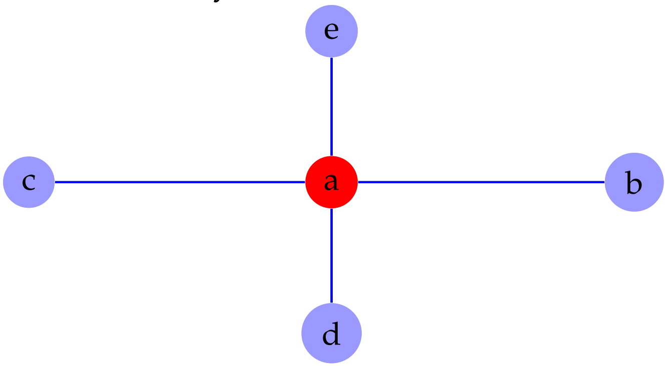
Communications

- Shared Registers



Communications

- Shared Memory



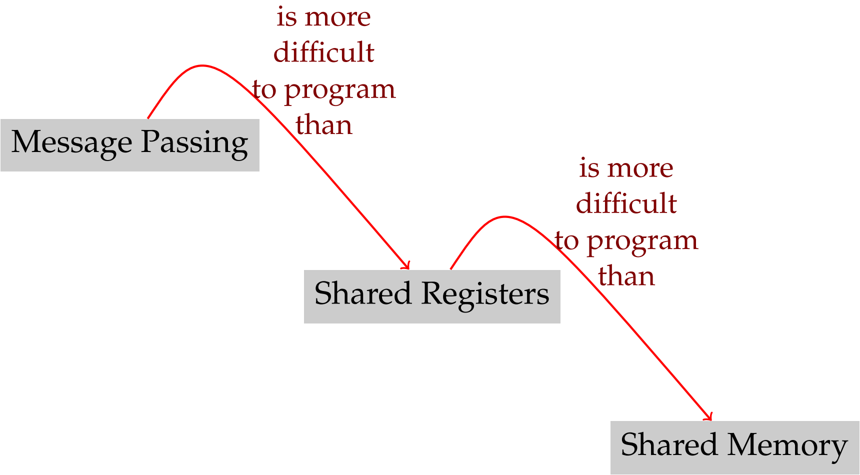
Communications

Message Passing

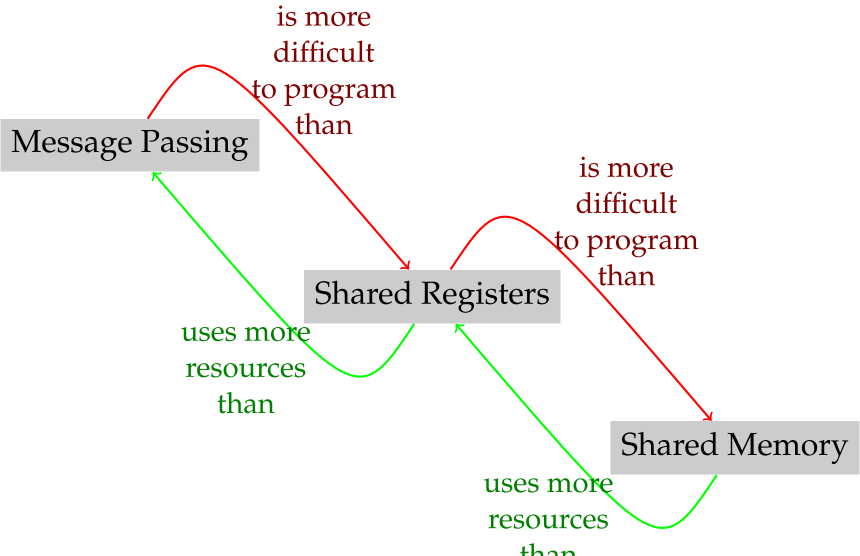
Shared Registers

Shared Memory

Communications



Communications



Example

Definition (Shared Memory)

In one atomic step, read the states of all neighbors and write own state

Definition (Guarded command)

- Guard $\rightarrow$ Action

Example

Definition (Shared Memory)

In one atomic step, read the states of all neighbors and write own state

Definition (Guarded command)

- Guard $\rightarrow$ Action
- Guard: predicate on the states of the neighborhood

Example

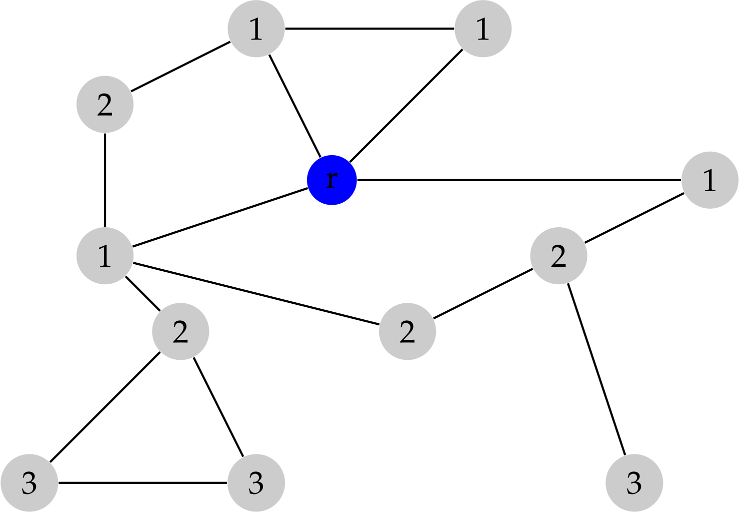
Definition (Shared Memory)

In one atomic step, read the states of all neighbors and write own state

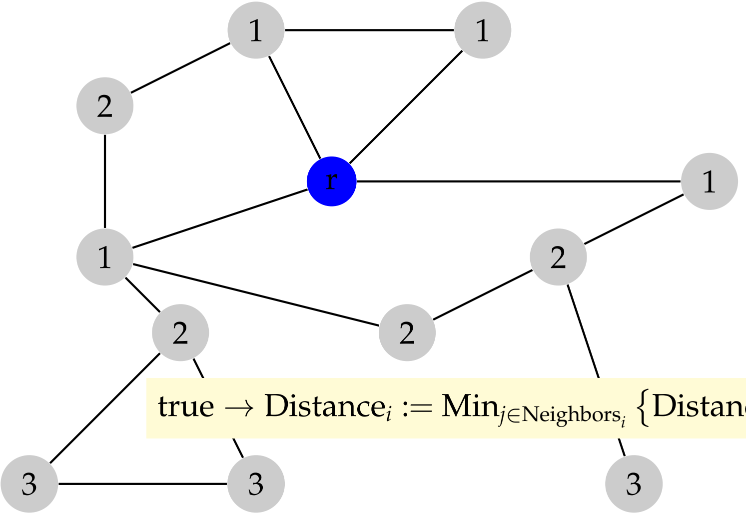
Definition (Guarded command)

- Guard $\rightarrow$ Action
- Guard: predicate on the states of the neighborhood
- Action: executed if *Guard* evaluates to true

Example

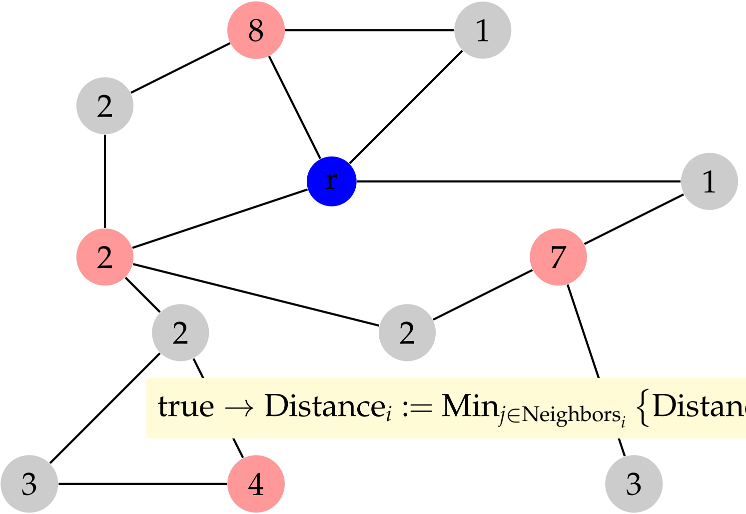


Example



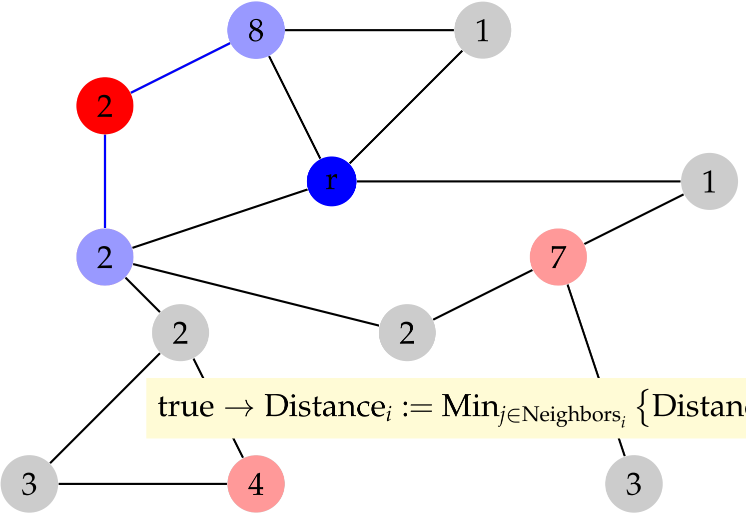
$$\text{true} \rightarrow \text{Distance}_i := \text{Min}_{j \in \text{Neighbors}_i} \{\text{Distance}_j\} + 1$$

Example



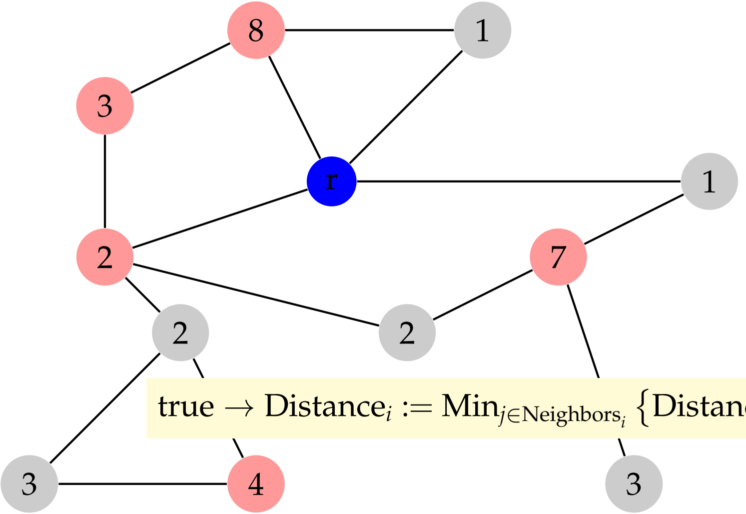
$$\text{true} \rightarrow \text{Distance}_i := \text{Min}_{j \in \text{Neighbors}_i} \{\text{Distance}_j\} + 1$$

Example



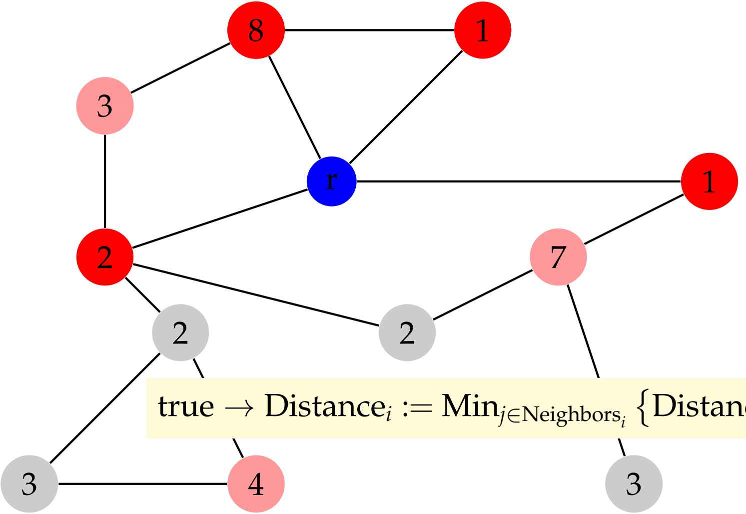
$$\text{true} \rightarrow \text{Distance}_i := \text{Min}_{j \in \text{Neighbors}_i} \{\text{Distance}_j\} + 1$$

Example



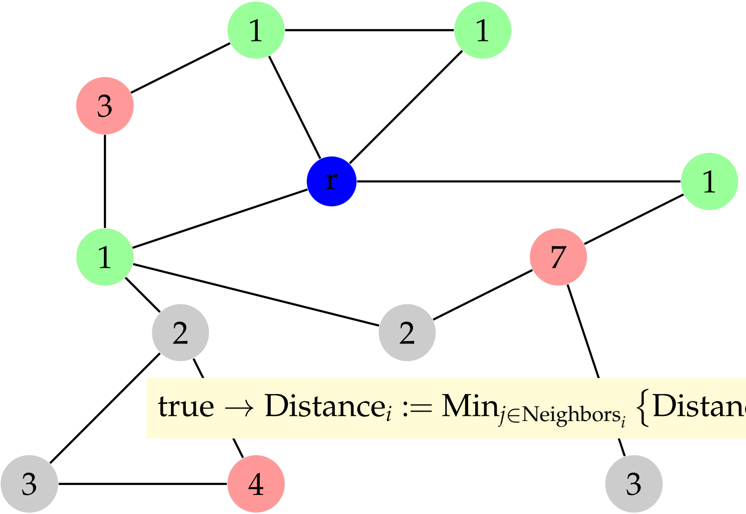
$$\text{true} \rightarrow \text{Distance}_i := \text{Min}_{j \in \text{Neighbors}_i} \{\text{Distance}_j\} + 1$$

Example



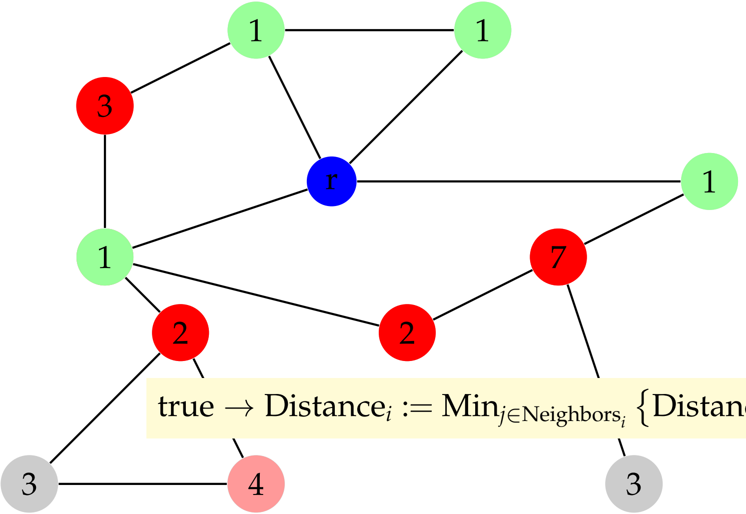
$$\text{true} \rightarrow \text{Distance}_i := \text{Min}_{j \in \text{Neighbors}_i} \{\text{Distance}_j\} + 1$$

Example



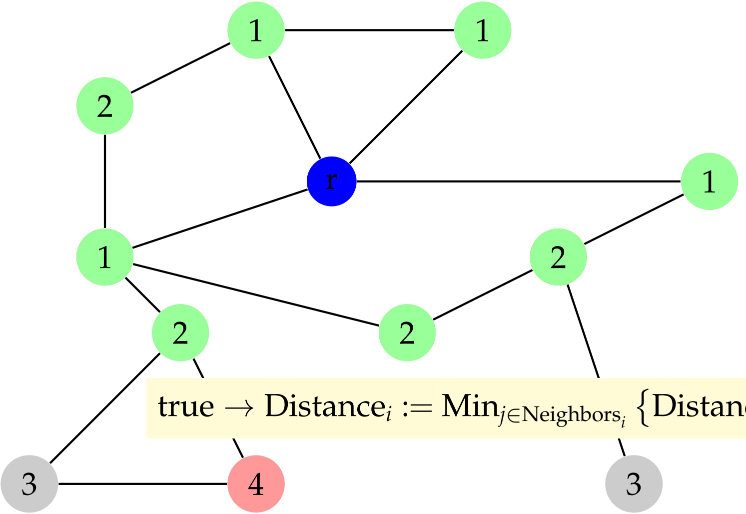
$$\text{true} \rightarrow \text{Distance}_i := \text{Min}_{j \in \text{Neighbors}_i} \{\text{Distance}_j\} + 1$$

Example



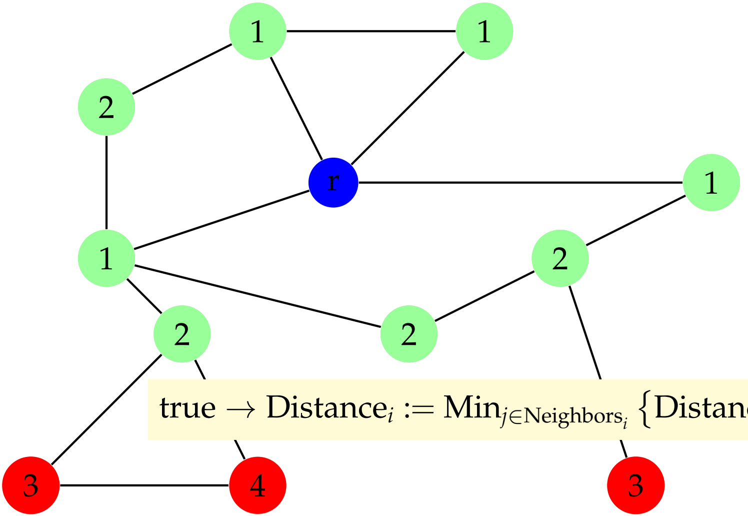
$$\text{true} \rightarrow \text{Distance}_i := \text{Min}_{j \in \text{Neighbors}_i} \{\text{Distance}_j\} + 1$$

Example



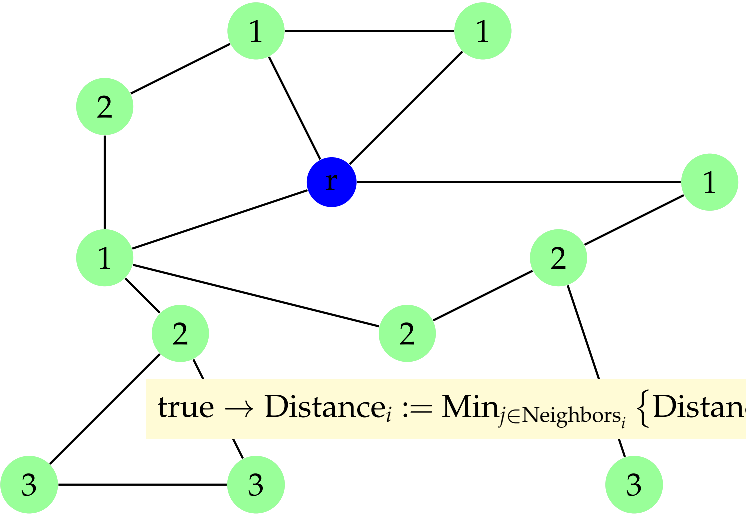
$$\text{true} \rightarrow \text{Distance}_i := \text{Min}_{j \in \text{Neighbors}_i} \{\text{Distance}_j\} + 1$$

Example



$$\text{true} \rightarrow \text{Distance}_i := \text{Min}_{j \in \text{Neighbors}_i} \{\text{Distance}_j\} + 1$$

Example



$$\text{true} \rightarrow \text{Distance}_i := \text{Min}_{j \in \text{Neighbors}_i} \{\text{Distance}_j\} + 1$$

Scheduling

Definition (Scheduler *a.k.a.* Daemon)

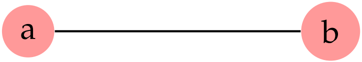
The daemon chooses among activable processors those that will execute their actions.

- The daemon can be seen as an adversary whose role is to prevent stabilization

Spatial Scheduling

$$\text{true} \rightarrow \text{color}_i := \text{Min}\left\{\Delta \setminus \{\text{color}_j | j \in \text{Neighbors}_i\}\right\}$$

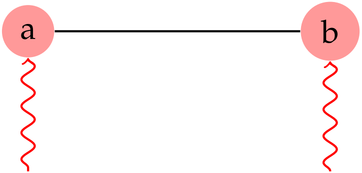
$$\Delta = \{ 0, 1 \}$$



Spatial Scheduling

$$\text{true} \rightarrow \text{color}_i := \text{Min}\left\{\Delta \setminus \{\text{color}_j | j \in \text{Neighbors}_i\}\right\}$$

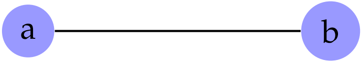
$$\Delta = \{\ 0\ ,\ 1\ \}$$



Spatial Scheduling

$$\text{true} \rightarrow \text{color}_i := \text{Min}\left\{\Delta \setminus \{\text{color}_j | j \in \text{Neighbors}_i\}\right\}$$

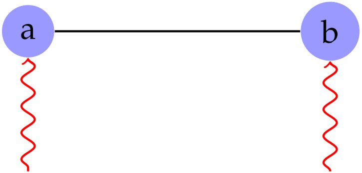
$$\Delta = \{\ 0, 1\ \}$$



Spatial Scheduling

$$\text{true} \rightarrow \text{color}_i := \text{Min}\left\{\Delta \setminus \{\text{color}_j | j \in \text{Neighbors}_i\}\right\}$$

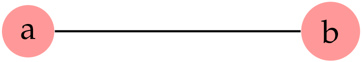
$$\Delta = \{ 0, 1 \}$$



Spatial Scheduling

$$\text{true} \rightarrow \text{color}_i := \text{Min}\left\{\Delta \setminus \{\text{color}_j | j \in \text{Neighbors}_i\}\right\}$$

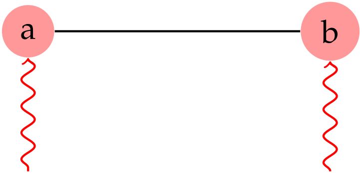
$$\Delta = \{0, 1\}$$



Spatial Scheduling

$$\text{true} \rightarrow \text{color}_i := \text{Min}\left\{\Delta \setminus \{\text{color}_j | j \in \text{Neighbors}_i\}\right\}$$

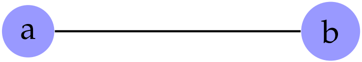
$$\Delta = \{ 0, 1 \}$$



Spatial Scheduling

$$\text{true} \rightarrow \text{color}_i := \text{Min}\left\{\Delta \setminus \{\text{color}_j | j \in \text{Neighbors}_i\}\right\}$$

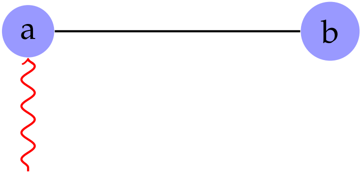
$$\Delta = \{ 0, 1 \}$$



Spatial Scheduling

$$\text{true} \rightarrow \text{color}_i := \text{Min}\left\{\Delta \setminus \{\text{color}_j | j \in \text{Neighbors}_i\}\right\}$$

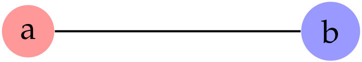
$$\Delta = \{ 0, 1 \}$$



Spatial Scheduling

$$\text{true} \rightarrow \text{color}_i := \text{Min}\left\{\Delta \setminus \{\text{color}_j | j \in \text{Neighbors}_i\}\right\}$$

$$\Delta = \{0, 1\}$$



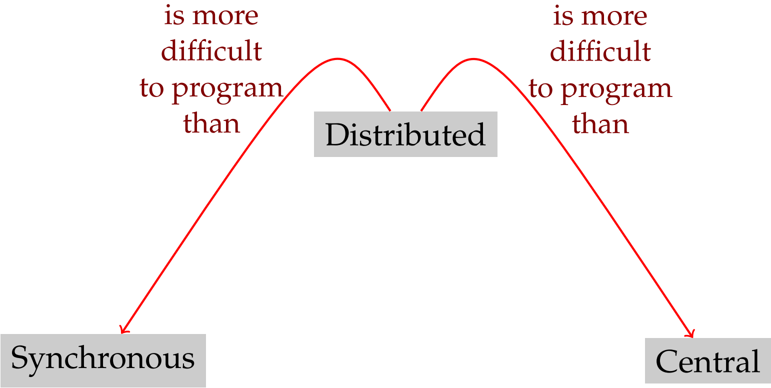
Spatial Scheduling

Distributed

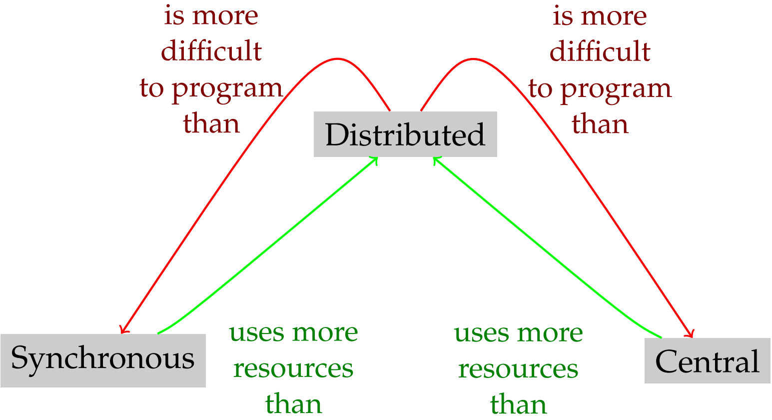
Synchronous

Central

Spatial Scheduling

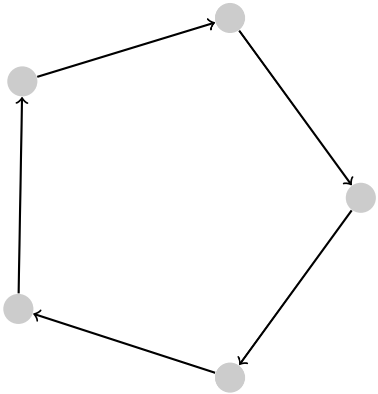


Spatial Scheduling



Temporal Scheduling

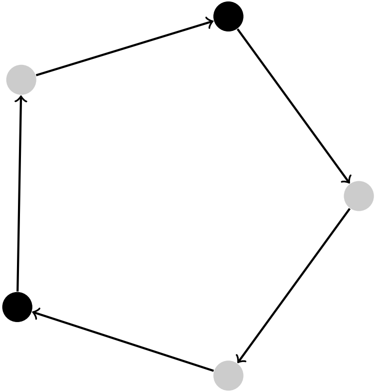
token $\rightarrow$ pass token to left neighbor with probability $\frac{1}{2}$



● = token

Temporal Scheduling

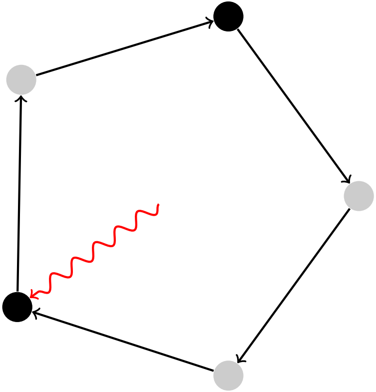
token → pass token to left neighbor with probability $\frac{1}{2}$



● = token

Temporal Scheduling

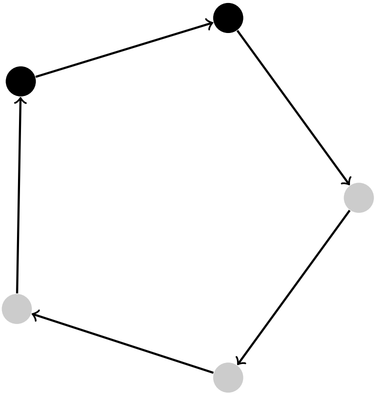
token → pass token to left neighbor with probability $\frac{1}{2}$



● = token

Temporal Scheduling

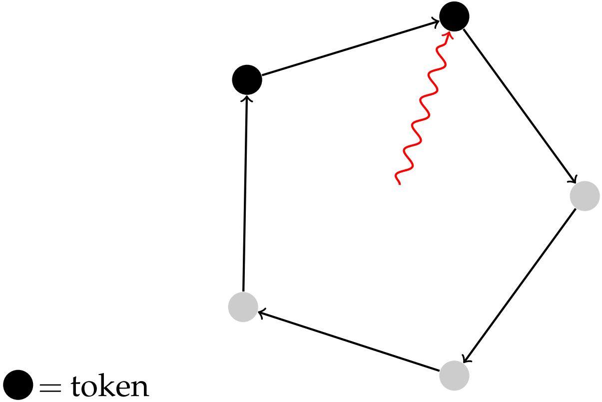
token $\rightarrow$ pass token to left neighbor with probability $\frac{1}{2}$



$\bullet$ = token

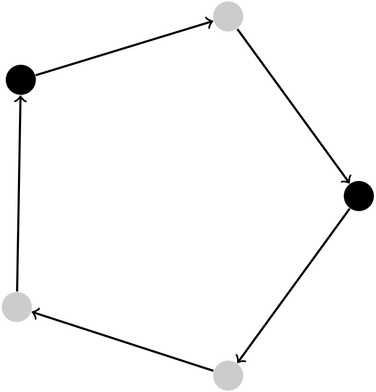
Temporal Scheduling

token $\rightarrow$ pass token to left neighbor with probability $\frac{1}{2}$



Temporal Scheduling

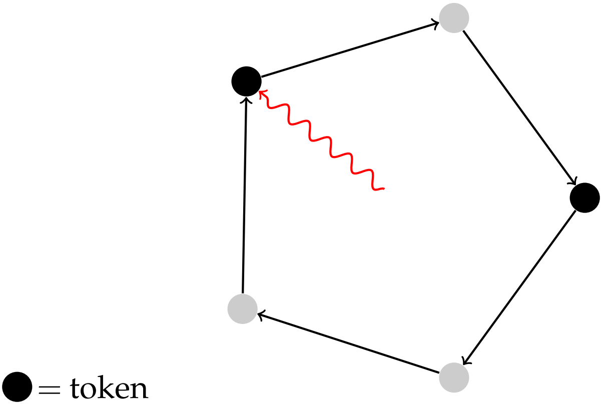
token $\rightarrow$ pass token to left neighbor with probability $\frac{1}{2}$



$\bullet$ = token

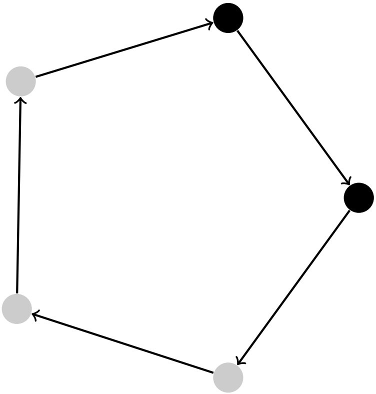
Temporal Scheduling

token $\rightarrow$ pass token to left neighbor with probability $\frac{1}{2}$



Temporal Scheduling

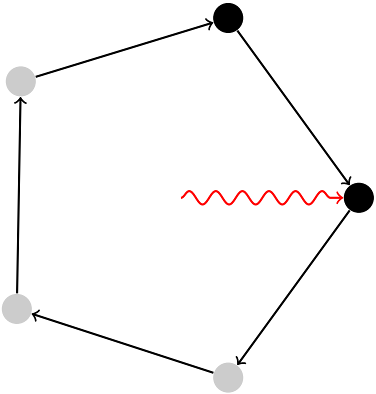
token $\rightarrow$ pass token to left neighbor with probability $\frac{1}{2}$



● = token

Temporal Scheduling

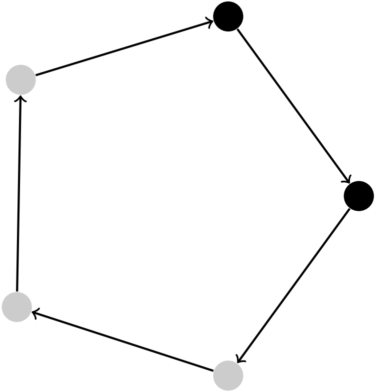
token $\rightarrow$ pass token to left neighbor with probability $\frac{1}{2}$



$\bullet$ = token

Temporal Scheduling

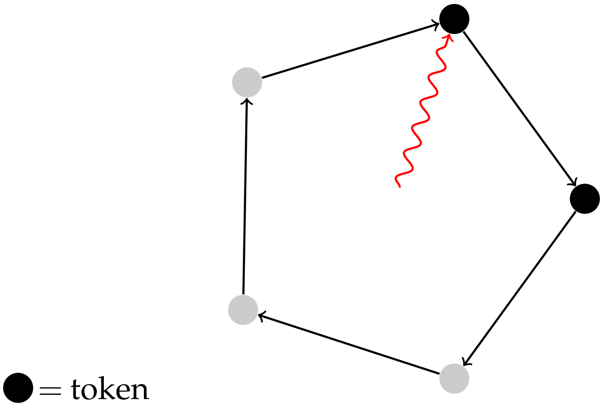
token $\rightarrow$ pass token to left neighbor with probability $\dfrac{1}{2}$



● = token

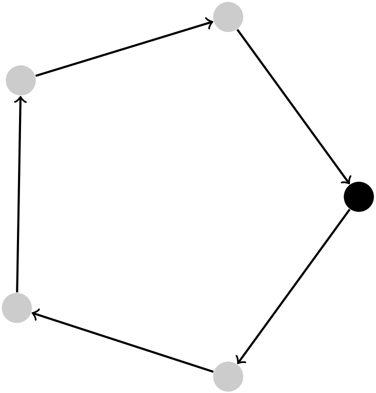
Temporal Scheduling

token $\rightarrow$ pass token to left neighbor with probability $\frac{1}{2}$



Temporal Scheduling

token $\rightarrow$ pass token to left neighbor with probability $\frac{1}{2}$



$\bullet$ = token

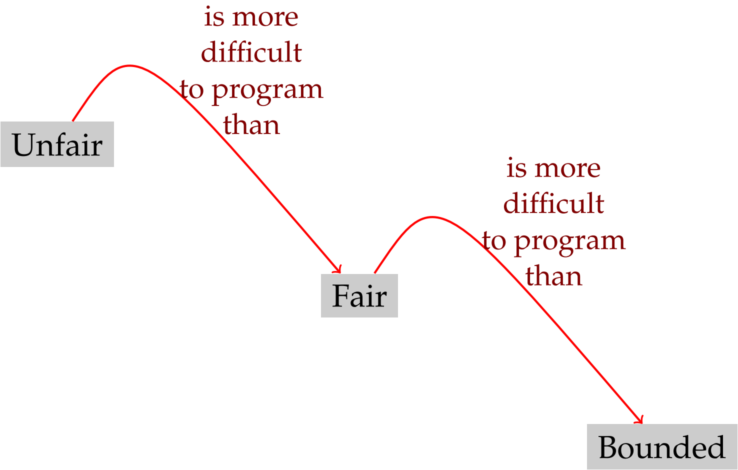
Temporal Scheduling

Unfair

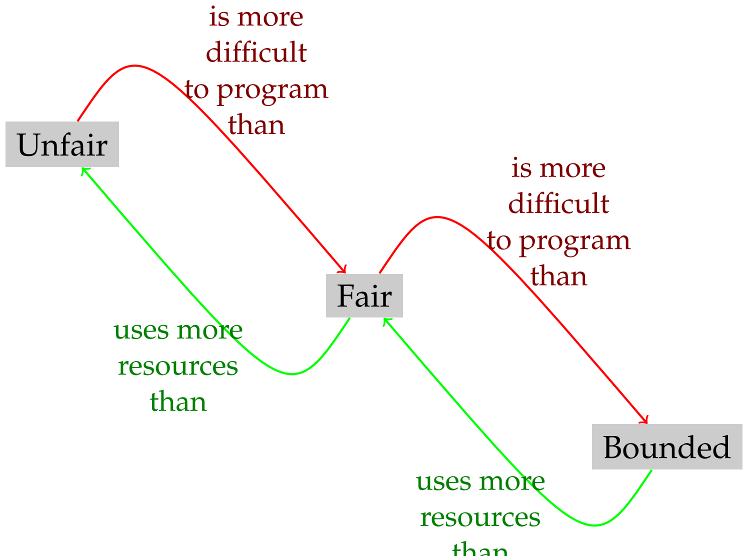
Fair

Bounded

Temporal Scheduling



Temporal Scheduling



Fair Composition

Basic idea

- Compose several self-stabilizing algorithms $Al_1, Al_2, \ldots Al_k$ such that the results of algorithms $Al_1, Al_2, \ldots Al_i$ can be reused by Al_{i+1}
- Al_{i+1} can not detect whether algorithms $Al_1, Al_2, \ldots Al_i$ have stabilized, but behaves as if

Fair Composition

Basic idea

- Compose several self-stabilizing algorithms $Al_1, Al_2, \ldots Al_k$ such that the results of algorithms $Al_1, Al_2, \ldots Al_i$ can be reused by Al_{i+1}
- Al_{i+1} can not detect whether algorithms $Al_1, Al_2, \ldots Al_i$ have stabilized, but behaves as if

Example with $k = 2$

- Two simple algorithms server and client are combined to obtain a more complex algorithm
- The server algorithm ensures that some properties (used by the client) will be eventually verified

Example

- Assume the server algorithm Al_1 solves a task defined by a set of legal executions T_1, and the client algorithm Al_2 solves T_2

$$\boxed{A_i}$$

- Let A_i be the set of states of process P_i for Al_1, and let $S_i = A_i \times B_i$ be the set of states of process P_i for Al_2, where anytime P_i executes Al_2, it modifies the B_i part of $A_i \times B_i$

$$\boxed{A_i} - \boxed{B_i}$$

Fair Composition

Definition (Fair composition)

Al is a fair composition of Al_1 and Al_2 if, in Al, every process alternatively executes actions of Al_1 and Al_2

Theorem

If Al_2 is self-stabilizing for T_2 given T_1, and if Al_1 is self-stabilizing for T_1, then the fair composition of Al_1 and Al_2 is self-stabilizing for T_2

$$\boxed{A_i} \rightarrow \boxed{B_i}$$

$$Al_1 \quad Al_2$$

Fair Composition

Definition (Fair composition)

Al is a fair composition of Al_1 and Al_2 if, in Al, every process alternatively executes actions of Al_1 and Al_2

Theorem

If Al_2 is self-stabilizing for T_2 given T_1, and if Al_1 is self-stabilizing for T_1, then the fair composition of Al_1 and Al_2 is self-stabilizing for T_2

$$\boxed{A_i} \rightarrow \boxed{B_i}$$

Al_1 Al_2

Fair Composition

Definition (Fair composition)

Al is a fair composition of Al_1 and Al_2 if, in Al, every process alternatively executes actions of Al_1 and Al_2

Theorem

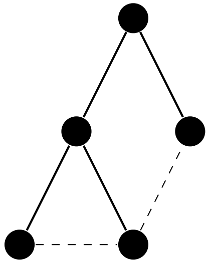
If Al_2 is self-stabilizing for T_2 given T_1, and if Al_1 is self-stabilizing for T_1, then the fair composition of Al_1 and Al_2 is self-stabilizing for T_2

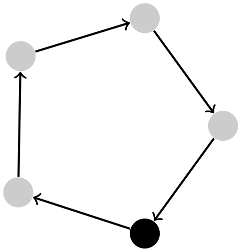
$$A_i \rightarrow B_i$$

Al_1 Al_2

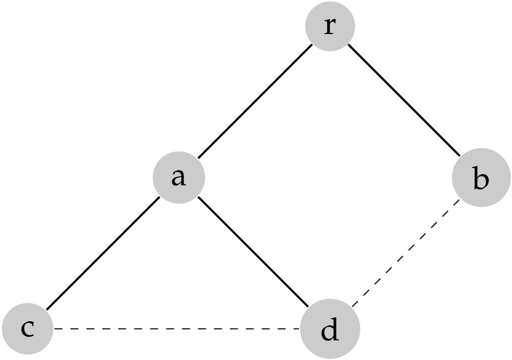
Example

- We are given two self-stabilizing algorithms, one for constructing a tree, one for mutual exclusion on a ring
- We wish to construct a self-stabilizing mutual exclusion algorithm on general graphs

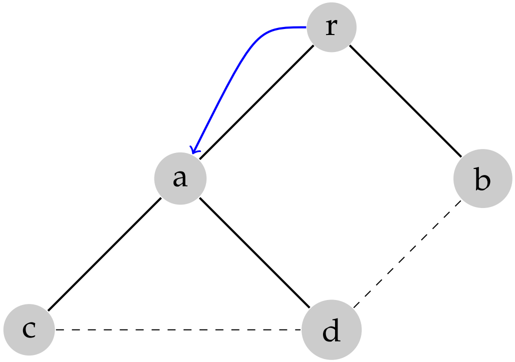




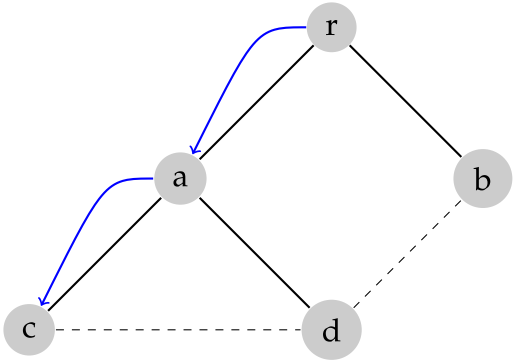
Example



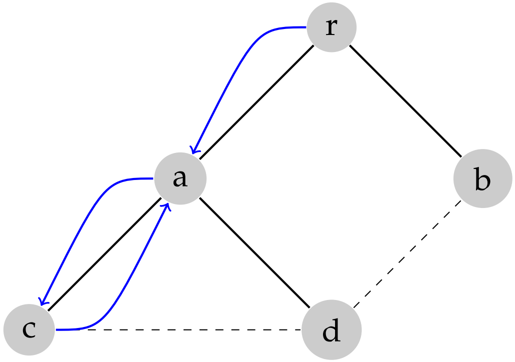
Example



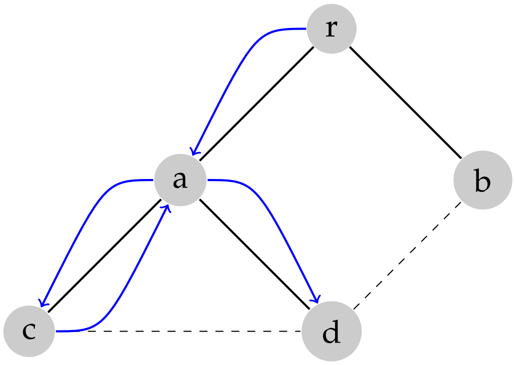
Example



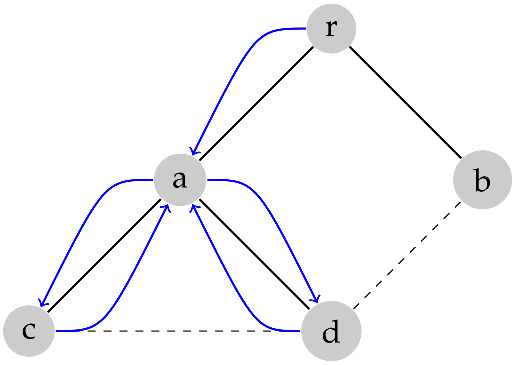
Example



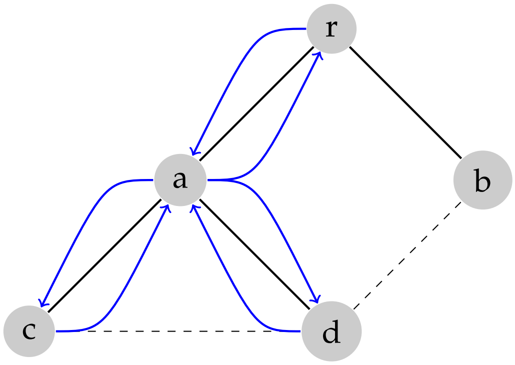
Example



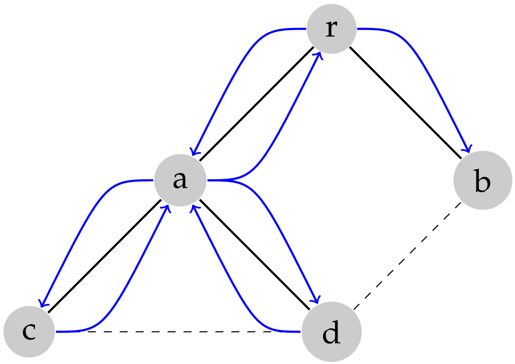
Example



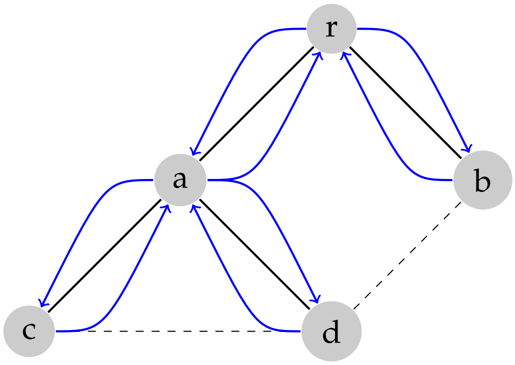
Example



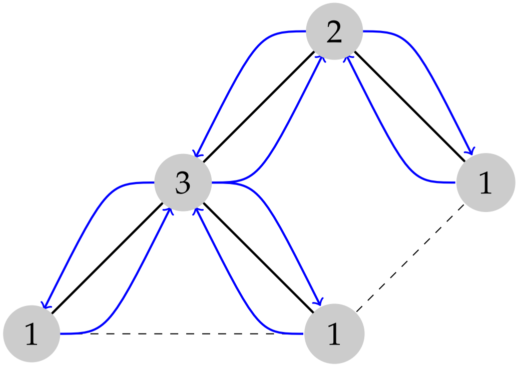
Example



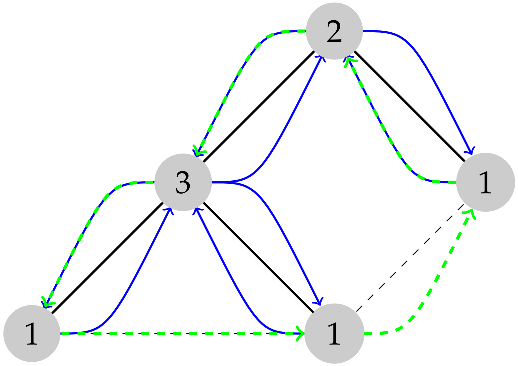
Example



Example



Example



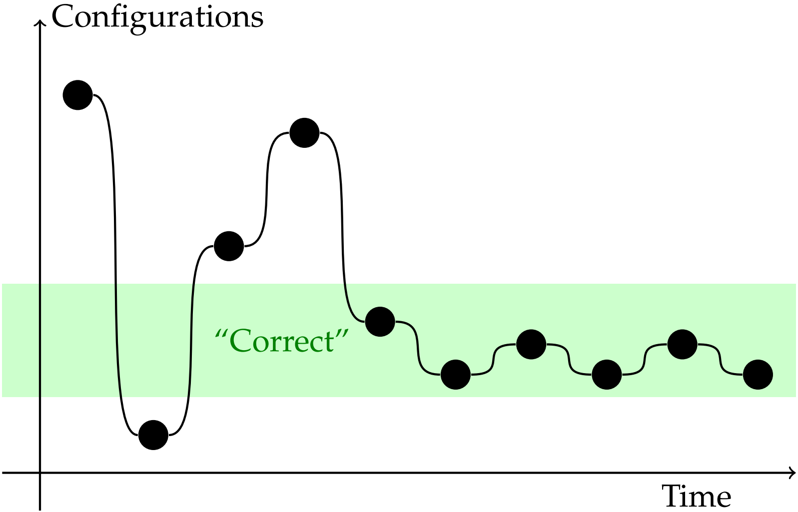
Transfer Function

Basic Idea

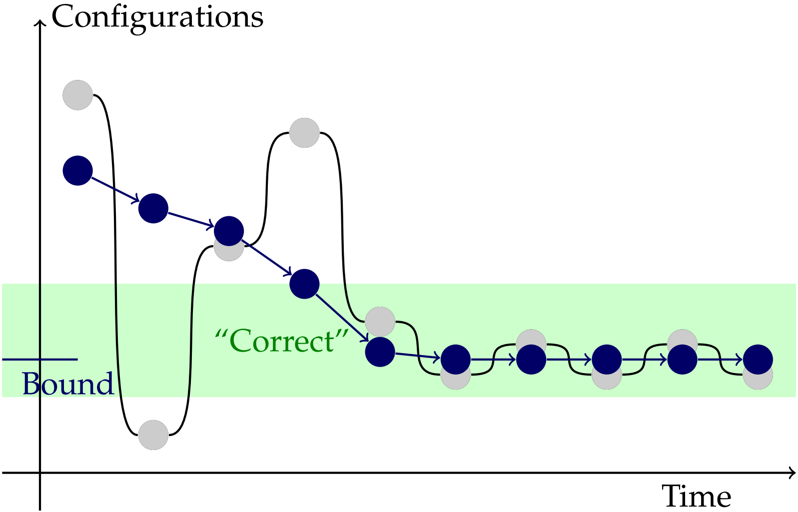
- $c_1 \rightarrow c_2 \rightarrow c_3 \rightarrow c_4 \rightarrow \cdots \rightarrow c_i$
- $FP(c_1) > FP(c_2) > FP(c_3) > \ldots > FP(c_i) = \text{bound}$

- Used to prove convergence
- Can be used to compute the number of steps to reach a legitimate configuration

Transfer Function

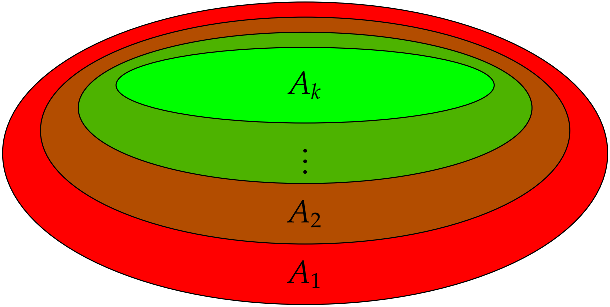


Transfer Function



Convergence stairs

- A_i is a predicate
- A_k is legitimate
- For any i between 1 and k, A_{i+1} is a refinement of A_i



Self-stabilization

Pros

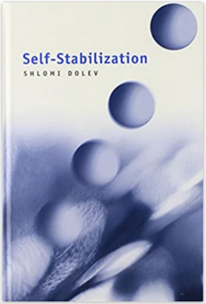
- The network does not need to be initialized
- When a fault is diagnosed, it is sufficient to identify, then remove or restart the faulty components
- The self-stabilization property does not depend on the nature of the fault
- The self-stabilization property does not depend on the extent of the fault

Self-stabilization

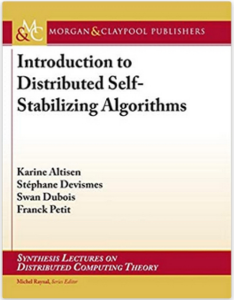
Cons

- *A priori*, “eventually” does not give any bound on the stabilization time
- *A priori*, nodes never know whether the system is stabilized or not
- A single failure may trigger a correcting action at every node in the network
- Faults must be sufficiently rare that they can be considered are transient

References



Self-Stabilization.
S. Dolev
The MIT Press, 2000.



Introduction to Distributed Self-Stabilizing Algorithms.
K. Altisen, S. Devismes, S. Dubois, and F. Petit
Morgan & Claypool Publishers, 2019.