Self-stabilizing Distributed Data Structures

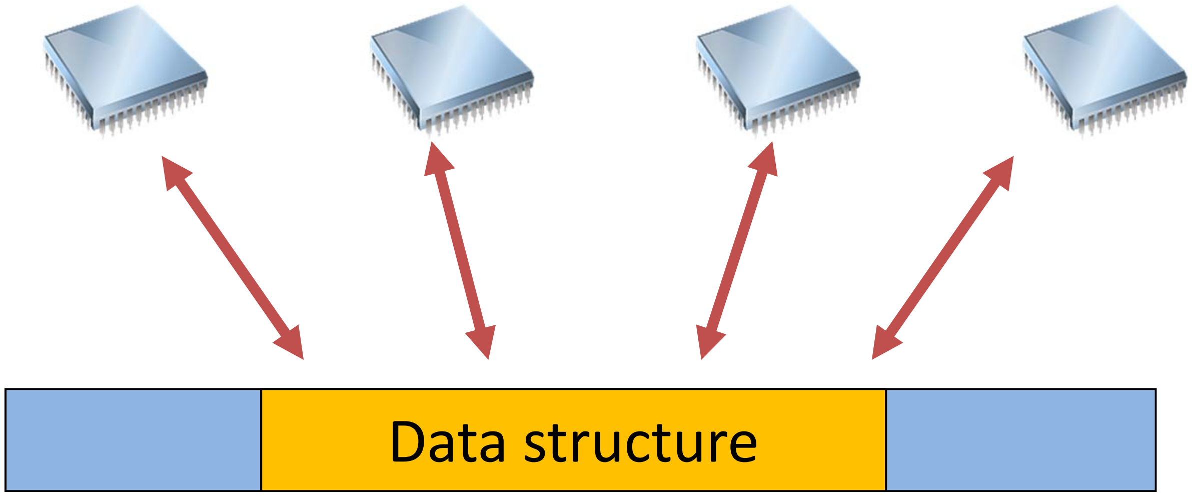
Christian Scheideler
Dept. of Computer Science
University of Paderborn

Structure of the Talk

- Motivation
- Basic model and notation
- Self-stabilizing sorted list
- Monotonically self-stabilizing sorted list
- Conclusion

Motivation

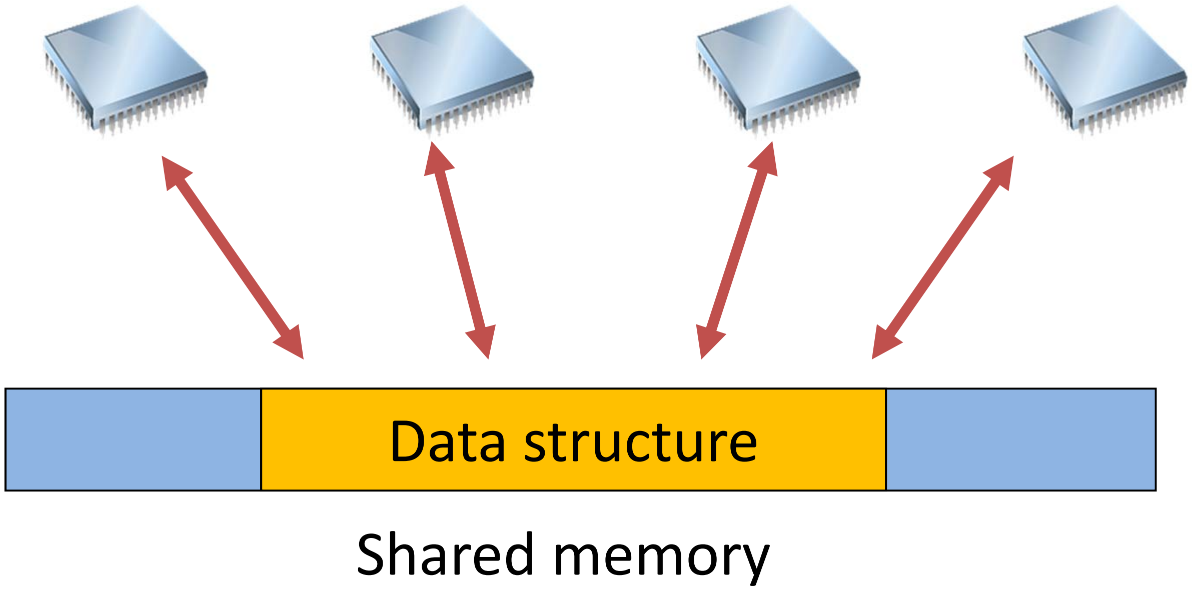
- Long history of concurrent data structures
- Most of them based on shared memory



Shared memory

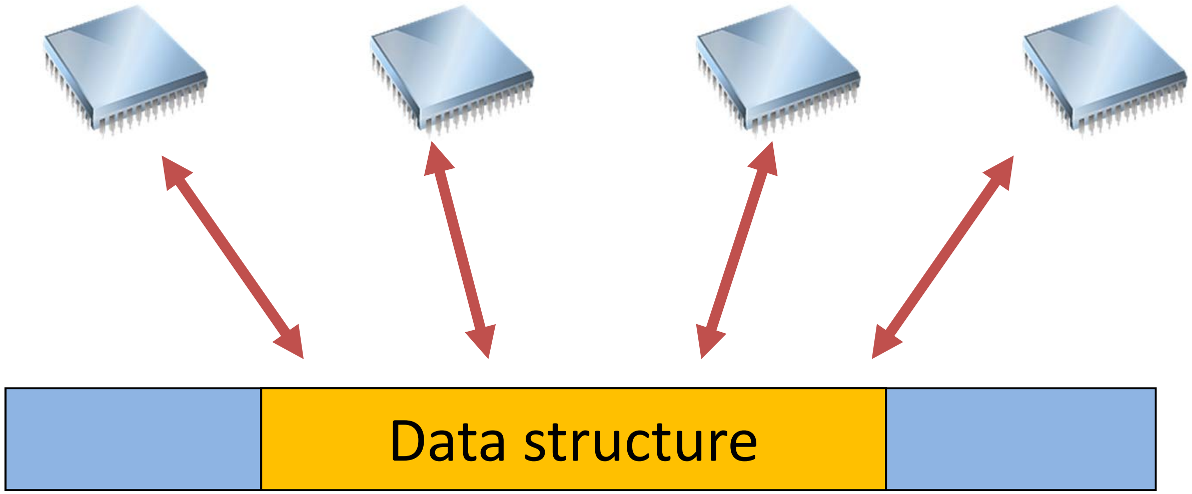
Motivation

- Hardware (processors and memory) is reliable, so no need for DS to be fault-tolerant. But order in which system executes access primitives is unpredictable.



Motivation

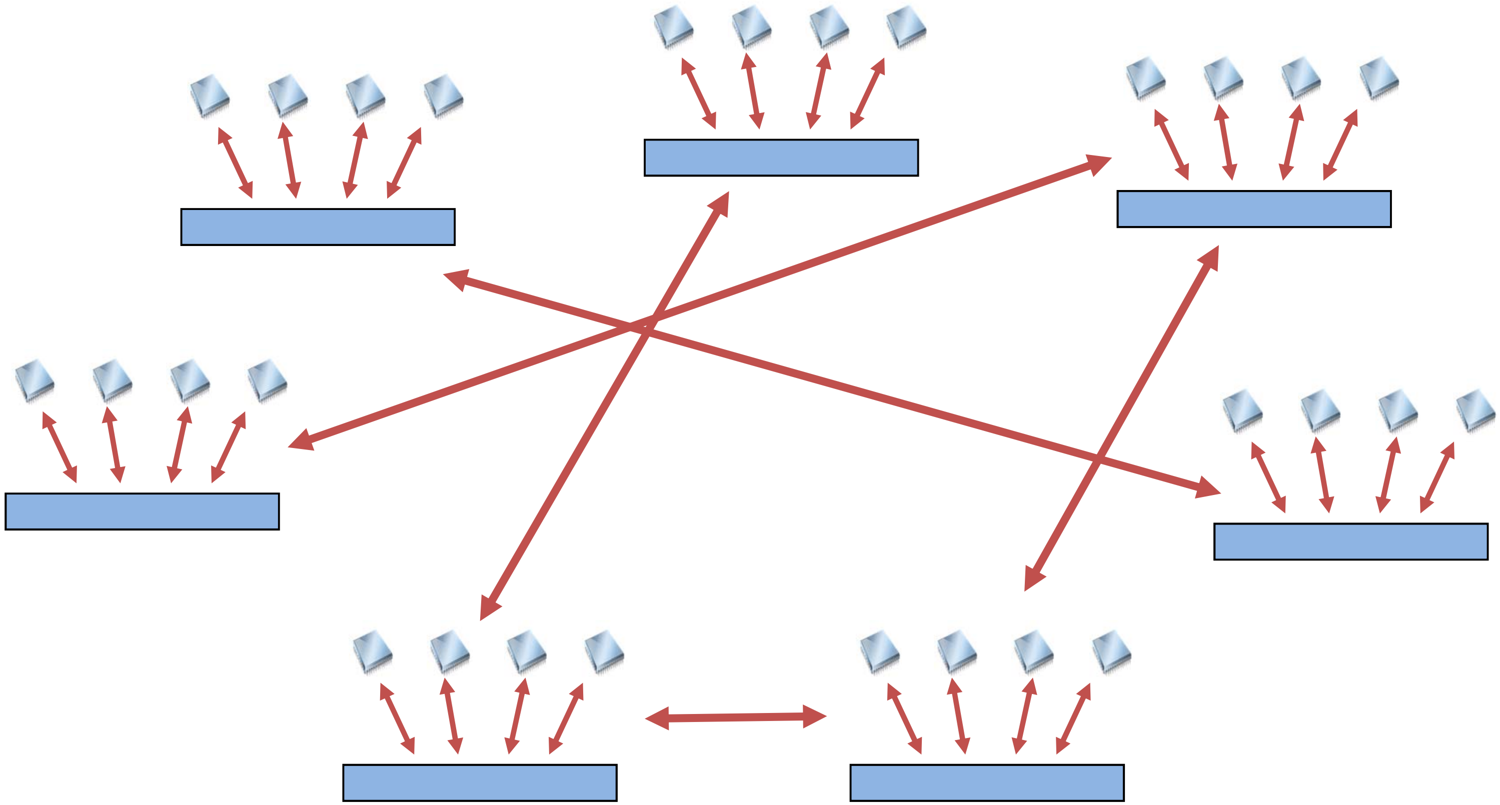
Challenge: avoid illegal states



Shared memory

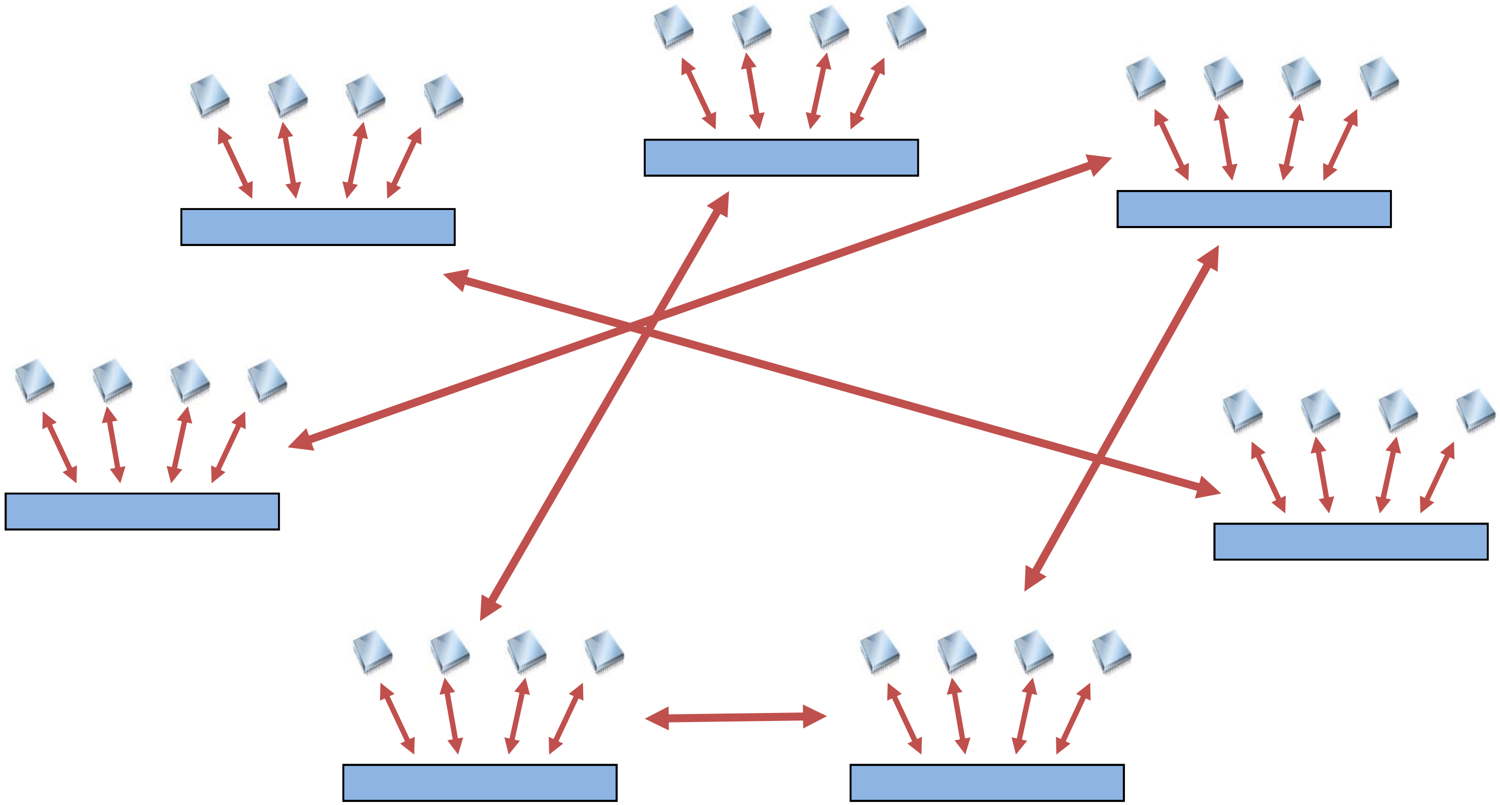
Motivation

Situation different for large distributed systems:



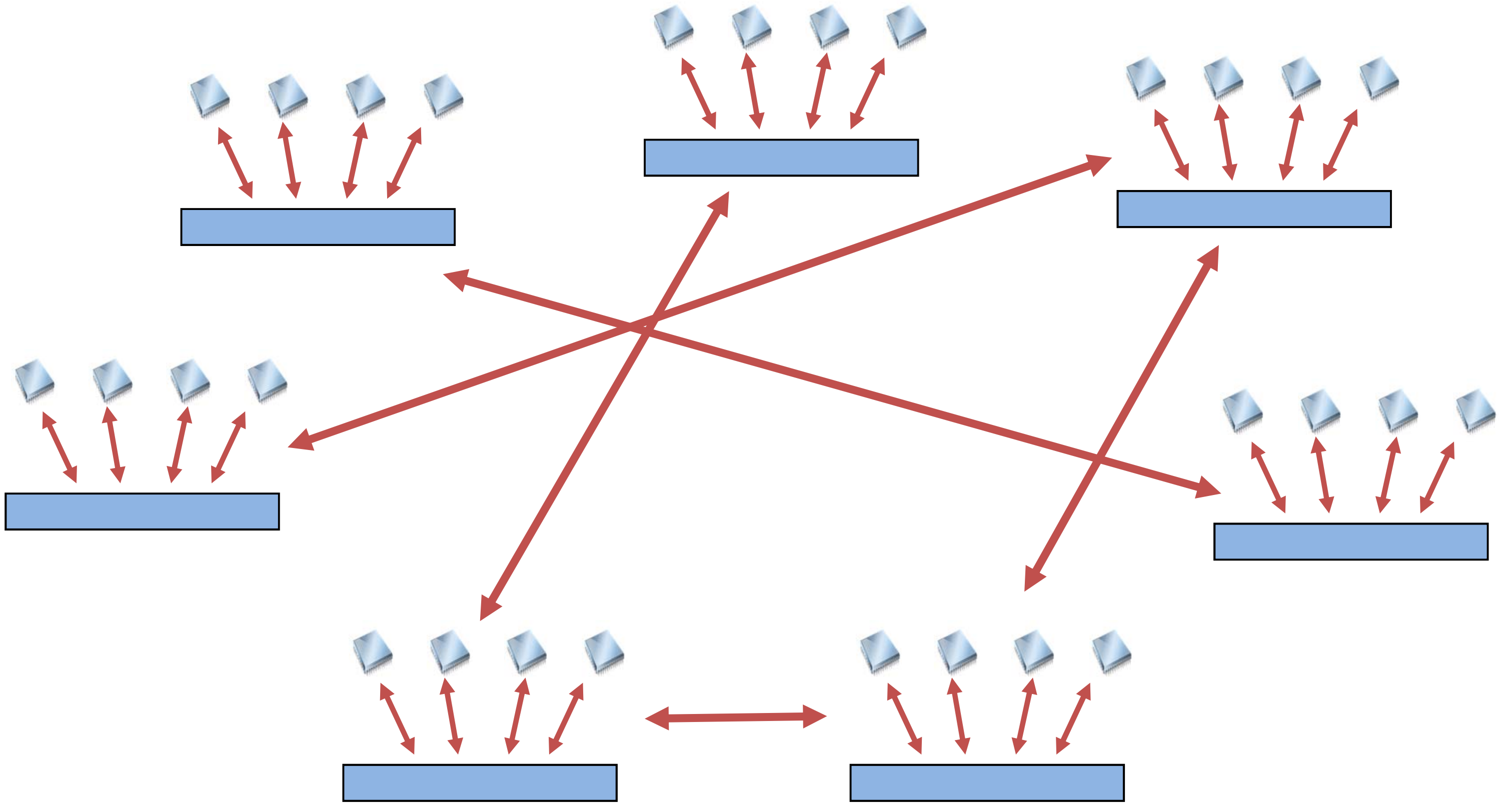
Motivation

Challenges: dynamics, bandwidth



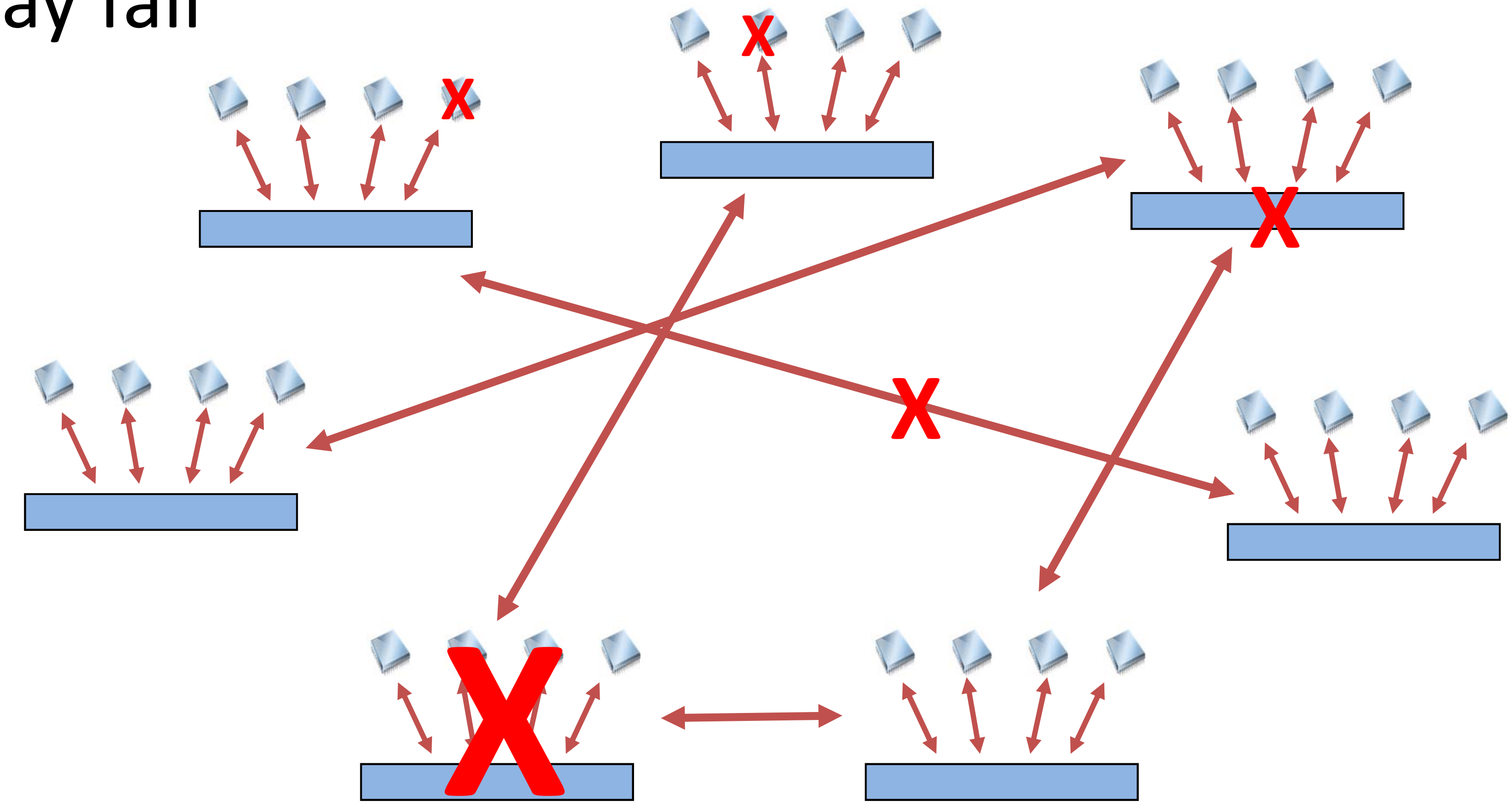
Motivation

Dynamics: cores/machines come and go



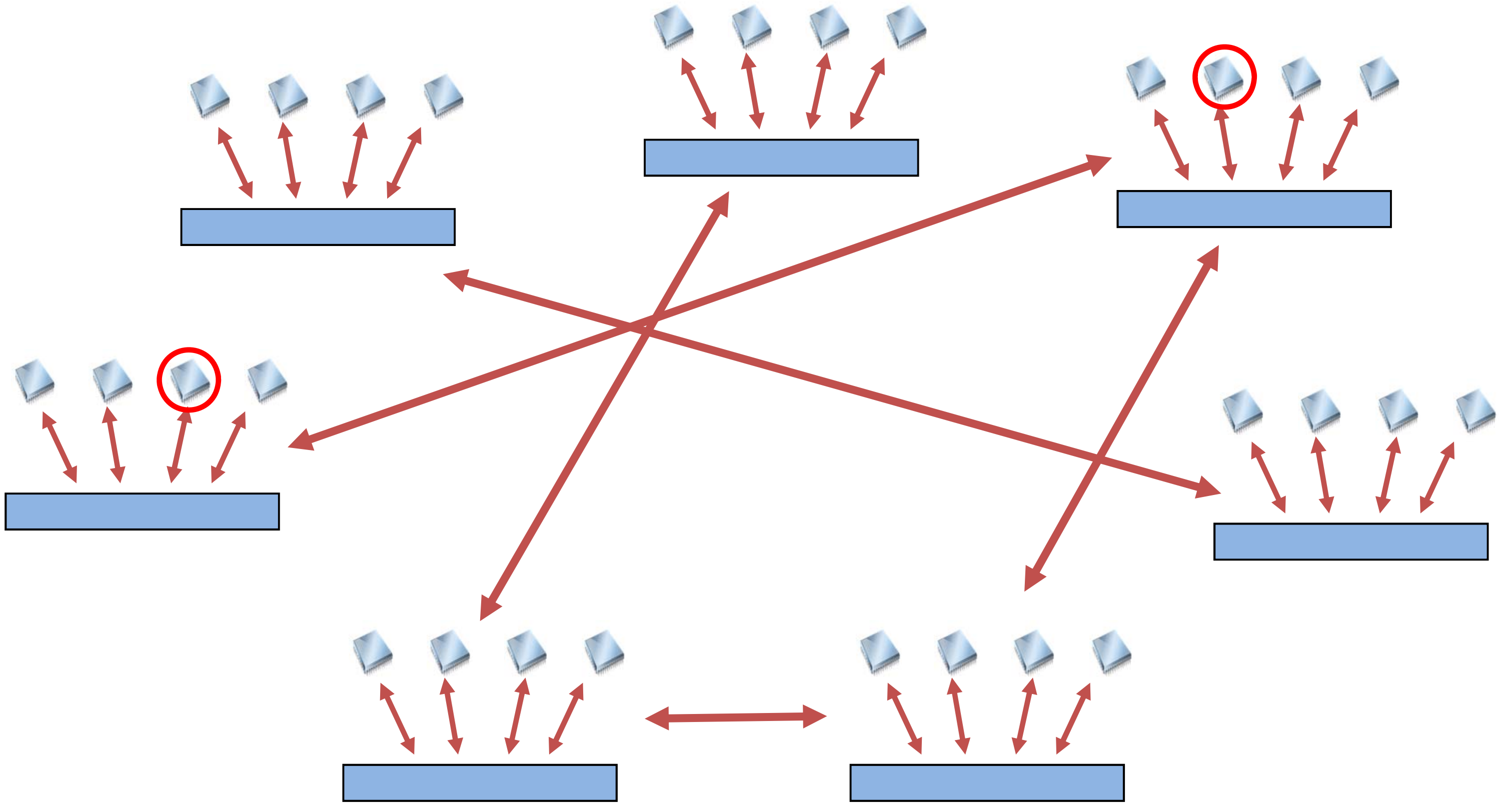
Motivation

Dynamics: cores, memory, machines, or links may fail



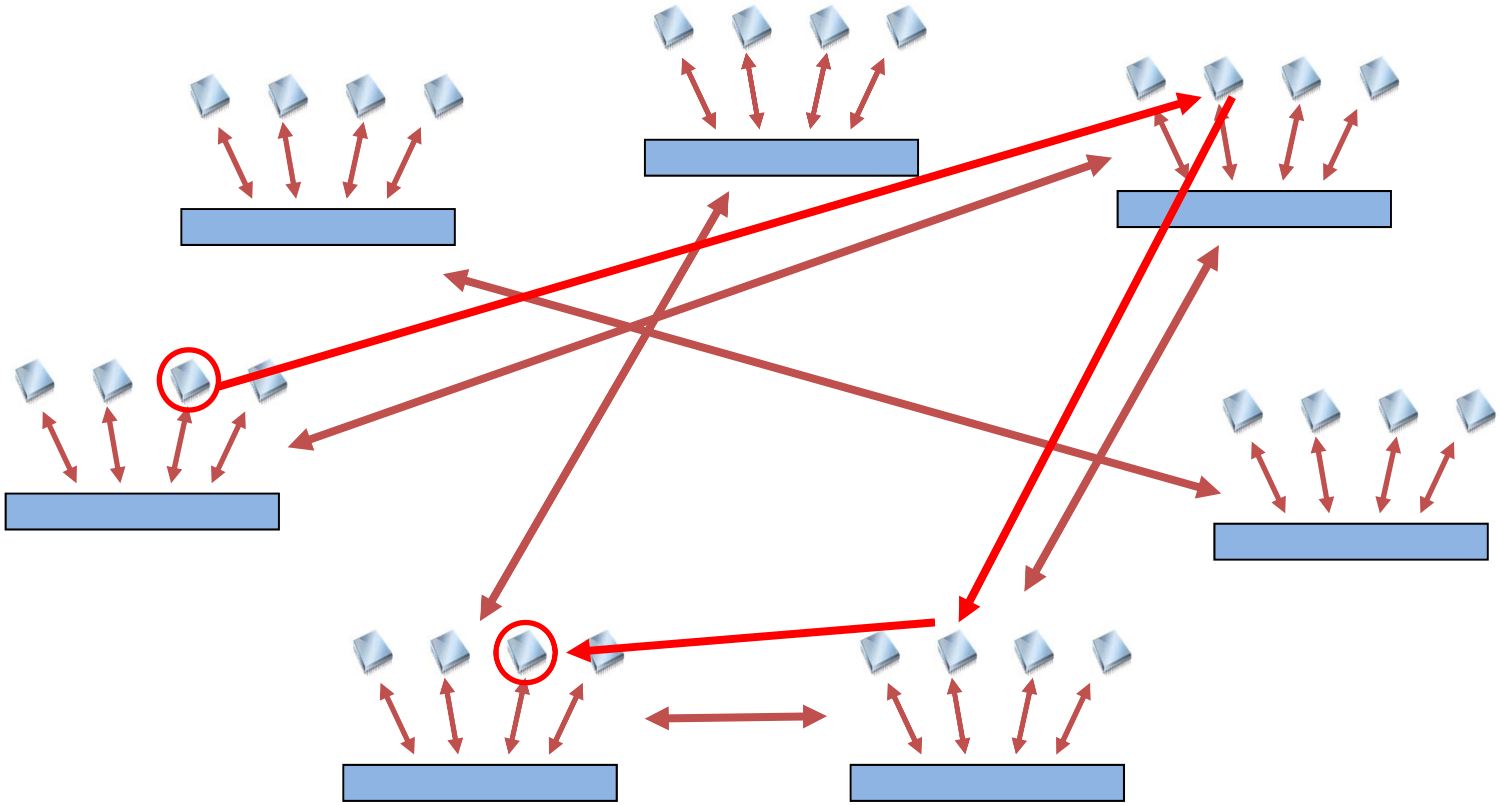
Motivation

Bandwidth: processes in different systems



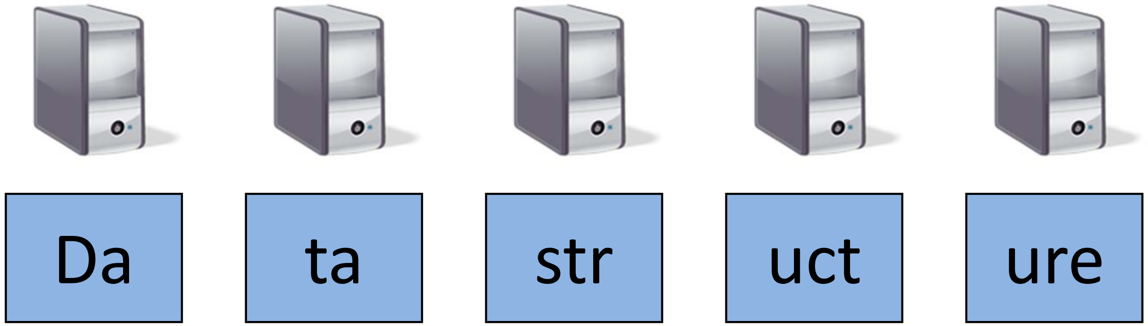
Motivation

Bandwidth: processes do not know everybody



Challenges

- How to map processes to machines?
- How to distribute/migrate data among processes?
- How to interconnect processes?
  (Who should know whom?)
- How to recover from faults?
- …



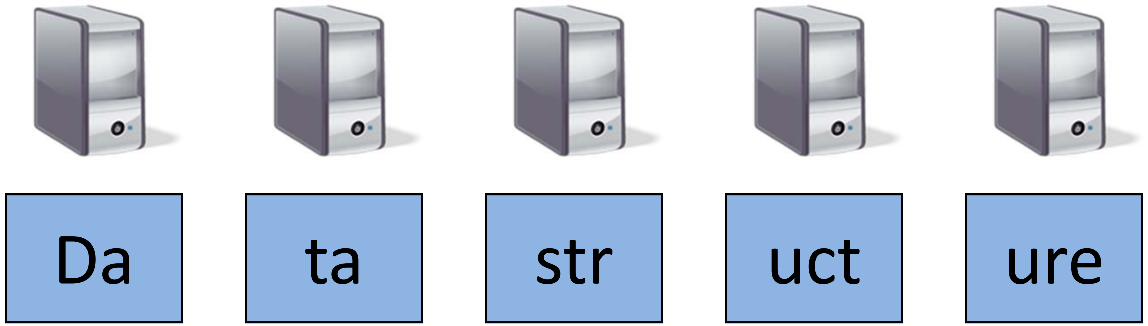
Da ta str uct ure

Motivation

Requirements for distributed data structures:

- Correctness: all requests are served correctly
- Availability: every request is eventually served
- Robustness: can handle any dynamics

→ However, only two of three requirements can be satisfied!



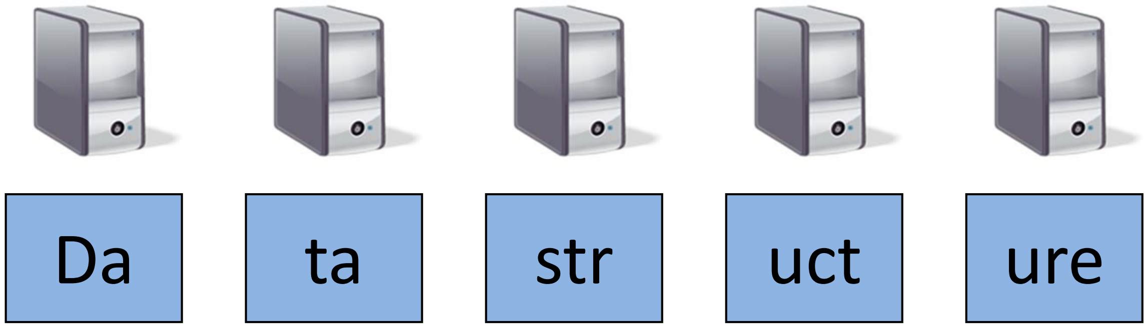
Da ta str uct ure

Motivation

Classical data base:

- Correctness: all requests are served correctly
- Availability: every request is eventually served
- Robustness: can handle any dynamics

→ However, for many information services, availability is **everything** (Google, Ebay, …)!

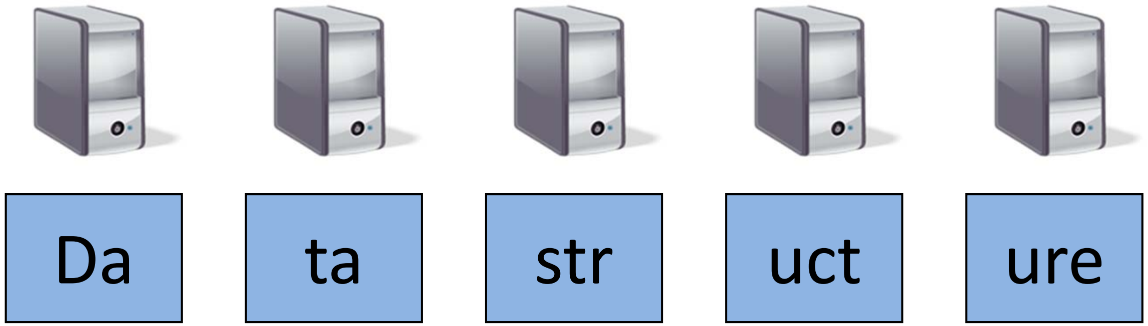


Motivation

Internet-wide information services:

- Correctness: all requests are served correctly
- Availability: every request is eventually served
- Robustness: can handle any dynamics

→ Self-stabilization!



Da ta str uct ure

Motivation

Computational problem P:

Given: initial system state S

Goal: eventually reach legal system state $S' \in L_P(S)$

Definition: A system is self-stabilizing w.r.t. P if the following conditions hold under the assumption that the system does not undergo external changes or faults:

1. Convergence: For all initial system states S, eventually a legal state $S' \in L_P(S)$ is reached.
2. Closure: For all legal states $S \in L_P(S)$, any follow-up state S´ is also legal.

Motivation

Definition: A system is self-stabilizing w.r.t. P if the following conditions hold under the assumption that the system does not undergo external changes or faults:

1. Convergence: For all initial system states S, eventually a legal state $S' \in L_P(S)$ is reached.
2. Closure: For all legal states $S \in L_P(S)$, any follow-up state S´ is also legal.

Not appropriate for data structures since no external changes like injections of operations (insert, delete, search) are allowed while the data structure recovers…

Motivation

Definition: A system is self-stabilizing w.r.t. P if the following conditions hold under the assumption that the system does not undergo external changes or faults:

1. Convergence: For all initial system states S, eventually a legal state $S' \in L_P(S)$ is reached.
2. Closure: For all legal states $S \in L_P(S)$, any follow-up state S´ is also legal.

So adjusted form of self-stabilization needed that we call monotonic self-stabilization. More on that later…

Structure of the Talk

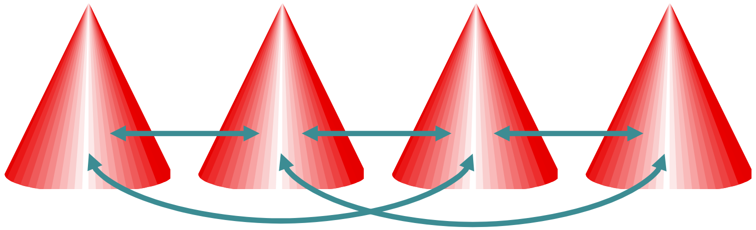
- Motivation
- Basic model and notation
- Self-stabilizing sorted list
- Monotonically self-stabilizing sorted list
- Conclusion

Basic Model

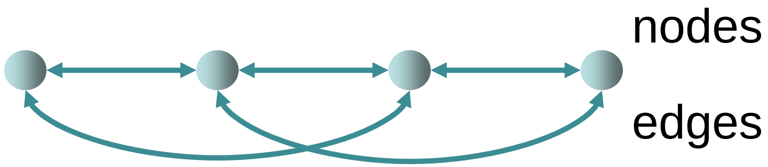
- Distributed data structure managed by a (dynamic) set of n processes. Process A knows B: A → B

We can model knowledge of processes as directed graph.

- Data structure:



- Graph representation:



Basic Model

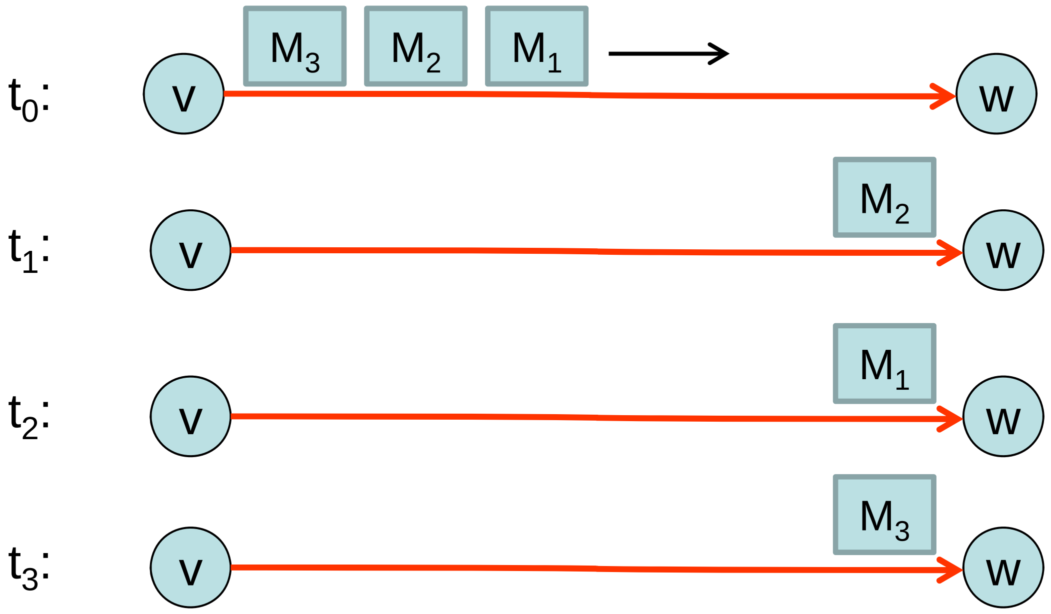
- Edge set E_L: set of pairs (v,w) where v knows w (explicit edges).

- Edge set E_M: set of pairs (v,w) with a message in transit to v containing a reference to w (implicit edges).

- Graph $G=(V,E_L \cup E_M)$: graph of all explicit and implicit edges.

Basic Model

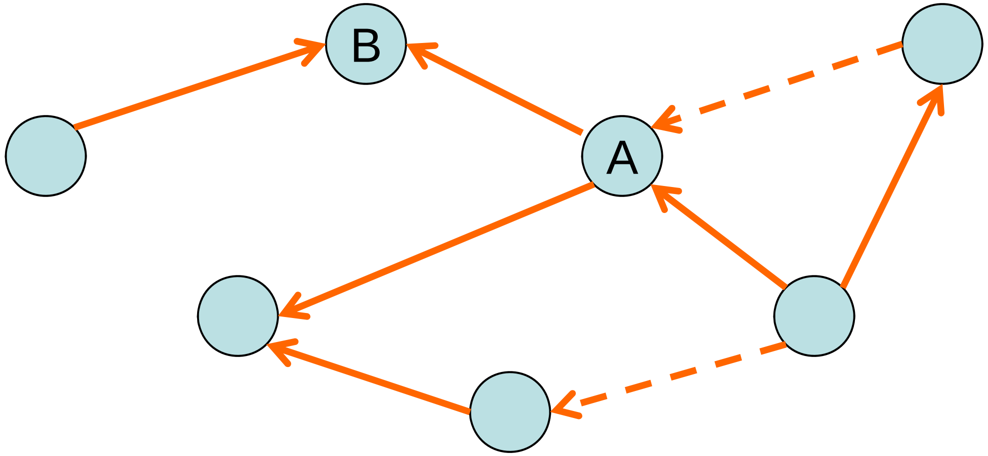
Asynchronous message passing



- all messages are eventually delivered
- but no FIFO delivery guaranteed

Topology Maintenance

Fundamental goal: topology of process graph (i.e., G) is kept weakly connected at any time

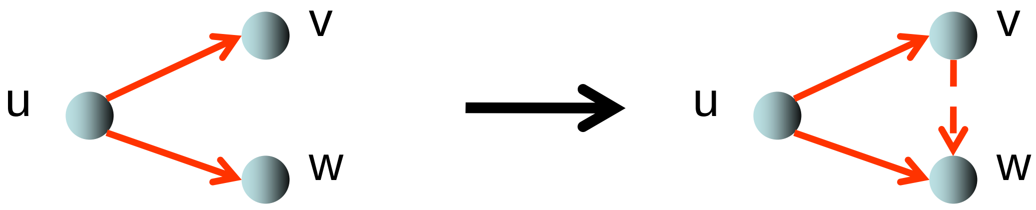


Fundamental rule: never just „throw away“ a reference!

Topology Maintenance

Admissible rules for weak connectivity:

- Introduction:



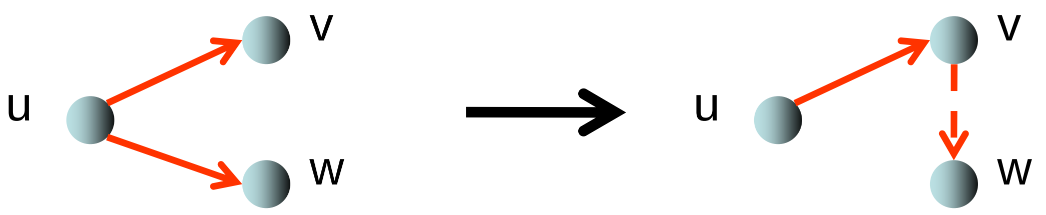
u introduces w to v by sending a message to v containing a reference to w

- special case: u introduces itself to v

Topology Maintenance

Admissible rules for weak connectivity:

- Delegation:



  u delegates its reference to w to v (i.e., afterwards it does not store a reference to w any more)

- Fusion:

Topology Maintenance

Admissible rules for weak connectivity:

- Reversal:

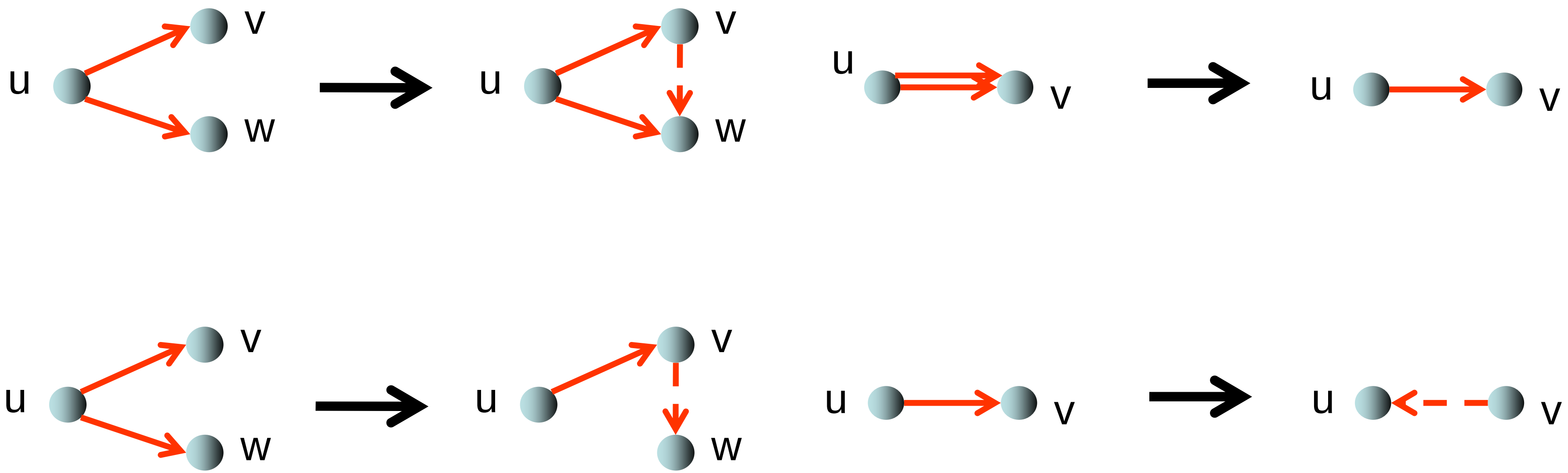
u sends a reference of itself to v and deletes v´s reference

Remarks:

- Advantage: rules can be executed in a local, wait-free manner in arbitrary asynchronous environments
- Introduction, delegation and fusion preserve strong connectivity

Topology Maintenance

Theorem: The 4 rules are universal in a sense that one can get from any weakly connected graph G=(V,E) to any other weakly connected graph G´=(V,E´).



Topology Maintenance

Remark:

- Each of the four rules is necessary to obtain universality.
  - Introduction: only one that generates new edge
  - Fusion: only one that removes edge
  - Delegation: only one that moves edge away
  - Reversal: only one that makes nodes unreachable

- Theorem only shows that in principle it is possible to get from any weakly connected graph to any other weakly connected graph.
- Our goal: algorithms for self-stabilizing distributed data structures

Actions

Processes are controlled by two types of actions:

- Triggered by a local/remote call:
  ⟨name⟩(⟨parameters⟩) → ⟨commands⟩
- Triggered by a local state:
  ⟨name⟩: ⟨predicate⟩ → ⟨commands⟩

All messages are remote action calls.

Example:

```
minimum(x,y) →
        if x<y then m:=x else m:=y
        print(m)                              no return command!
```

Action „minimum“ is executed whenever a request to call minimum(x,y) has been received.

Actions

Processes are controlled by two types of actions:

- Triggered by a local/remote call:
  ⟨name⟩(⟨parameters⟩) → ⟨commands⟩
- Triggered by a local state:
  ⟨name⟩: ⟨predicate⟩ → ⟨commands⟩

All messages are remote action calls.

Example:

```
    timeout: true →
            print(„I am still alive!“)
```

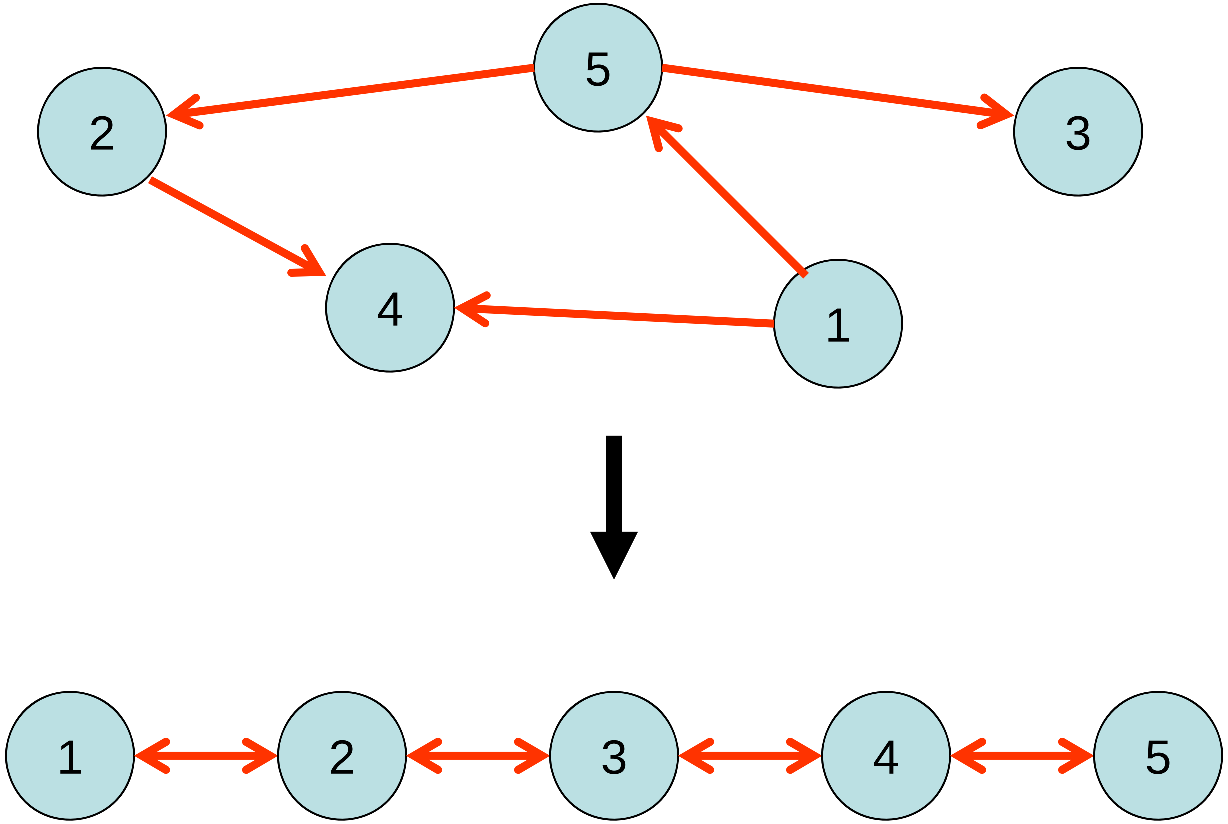
„true“ ensures that action timeout is periodically executed by the given process.

Structure of the Talk

- Motivation
- Basic model and notation
- Self-stabilizing sorted list
- Monotonically self-stabilizing sorted list
- Conclusion

Sorted List

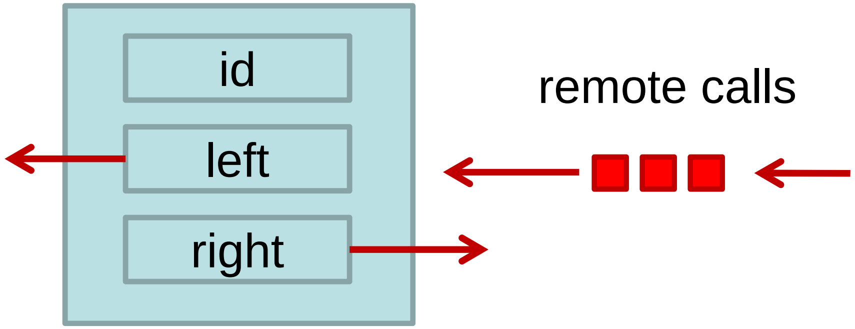
Goal:



Sorted List

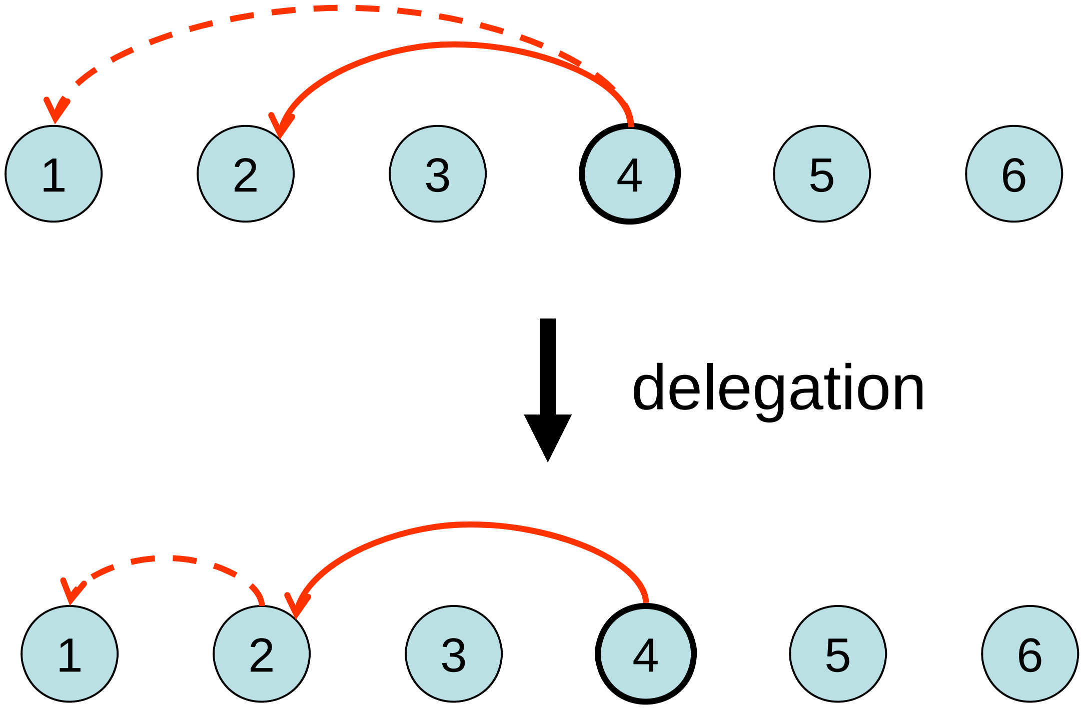
Variables within a node v:

- id: reference of v (we also write id(v))
- left $\in$ V$\cup$\{$\perp$\}: left neighbor of v, i.e., id(left)<id(v) (if id(left) is defined)
- right $\in$ V$\cup$\{$\perp$\}: right neighbor of v, i.e., id(right)>id(v) (if id(right) is defined)



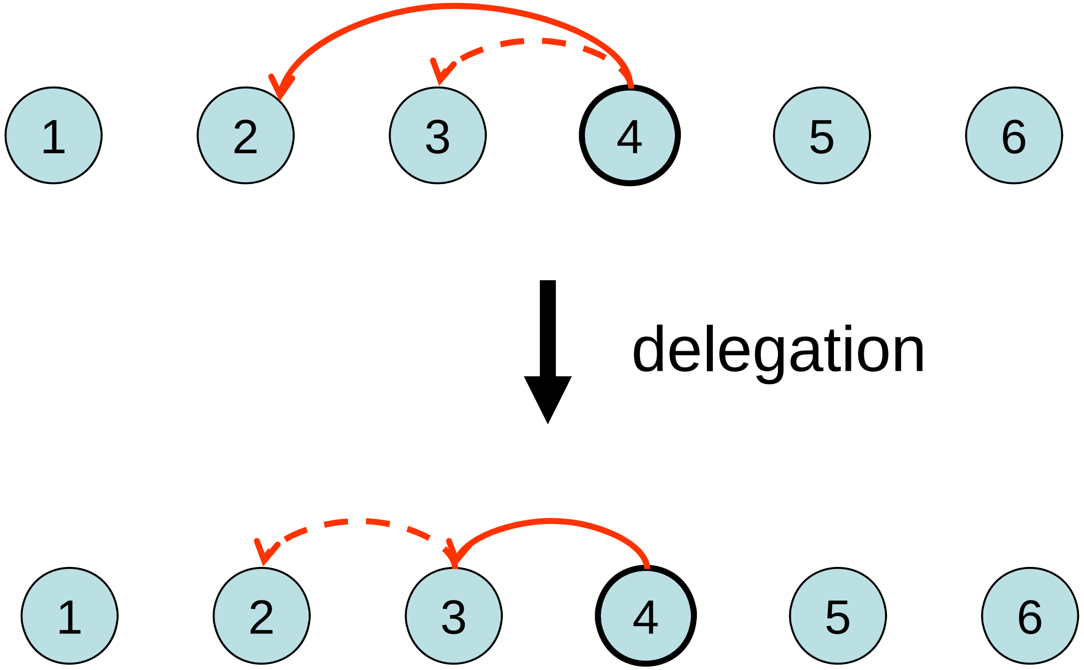
Sorted List

Basic strategy: linearization



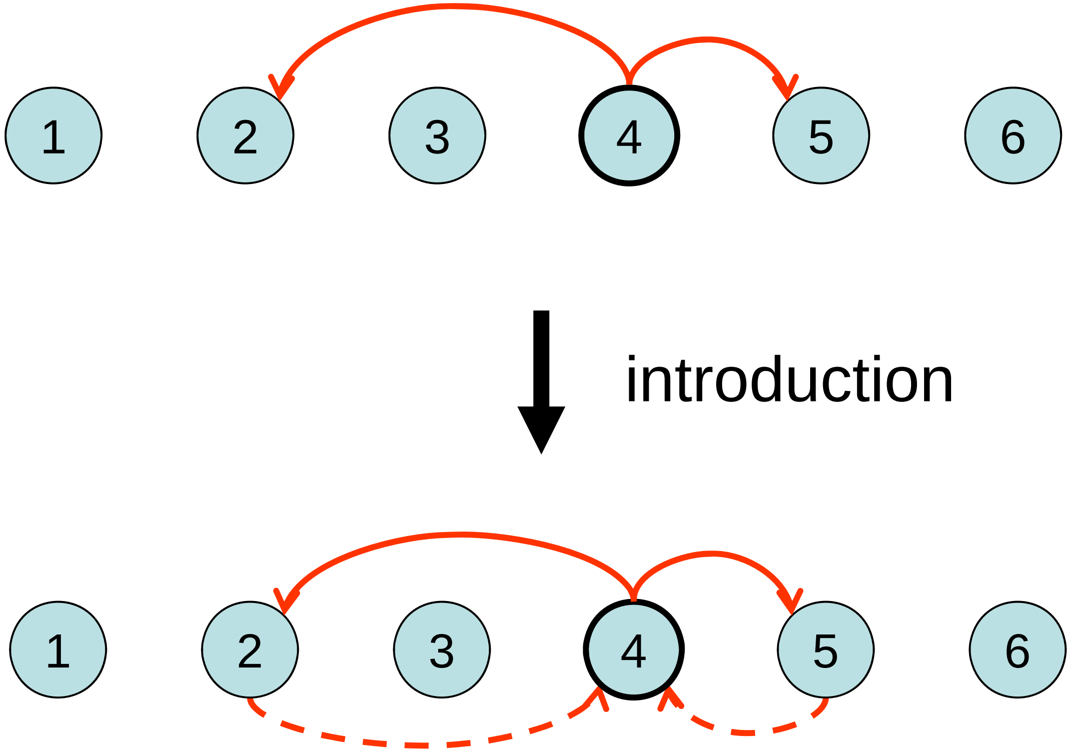
Sorted List

Basic strategy: linearization



Sorted List

Also periodically: self-introduction



Build-List Protocol

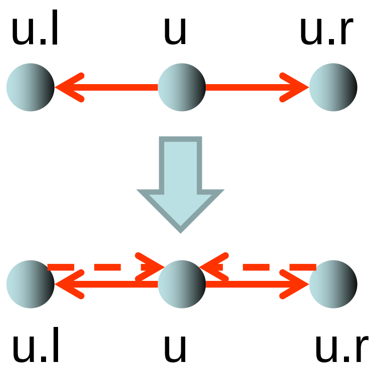
Build-List protocol: handles linearization and self-introduction

Simplifying assumptions:

- Whenever left=⊥, we assume in comparisons that $id(left)=-\infty$.
- Whenever right=⊥, we assume in comp. that $id(right)=+\infty$.
- A remote call $u \leftarrow action(v)$ is only executed if u and v are well defined.

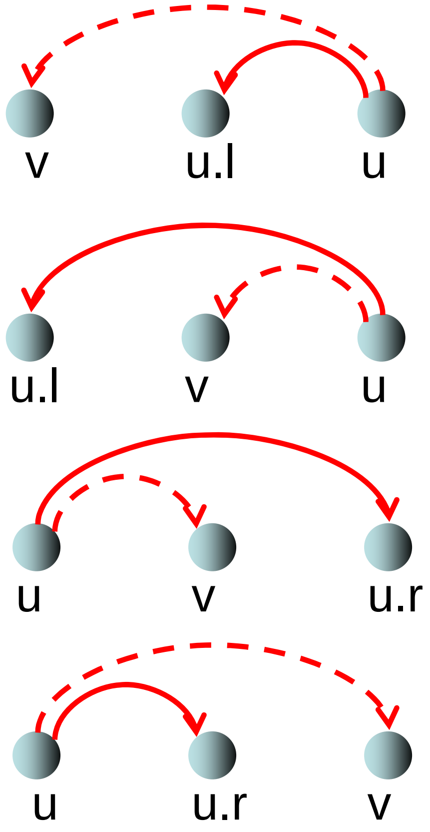
Build-List Protocol

```
timeout: true →
   { executed by node u }
   left←linearize(id)
   right←linearize(id)
```



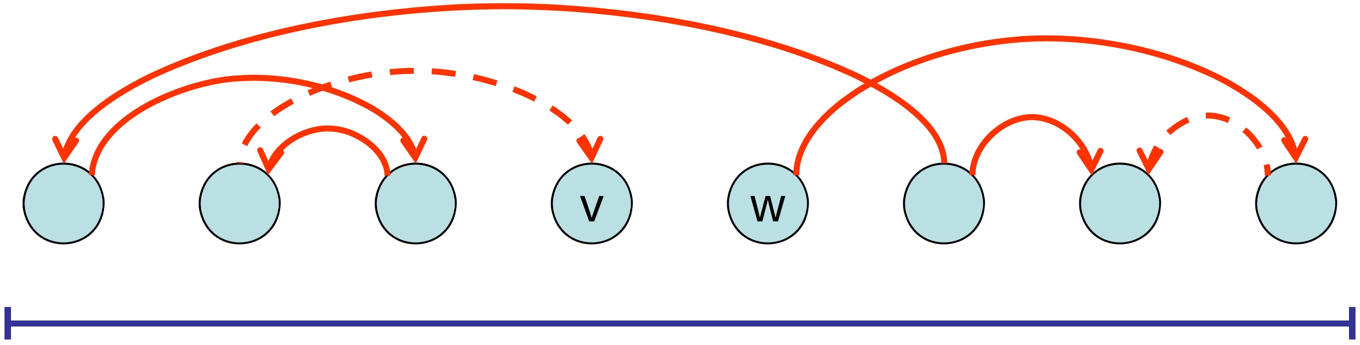
Build-List Protocol

```
linearize(v) →
    { executed by node u }
    if id(v)<id(left) then
        left←linearize(v)
    if id(left)<id(v)<id then
        v←linearize(left)
        left:=v
    if id<id(v)<id(right) then
        v←linearize(right)
        right:=v
    if id(right)<id(v) then
        right←linearize(v)
```



Sorted List

Convergence Proof:



range of path from v to w

Theorem: With Build-List we obtain a self-stabilizing sorted list.

Joining and Leaving

Concurrent join operations:

- Nodes connect to any node in network
- Rest is handled by self-stabilization

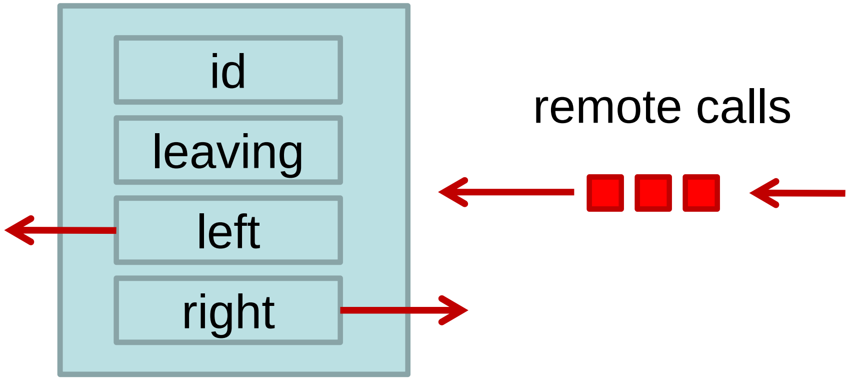
Concurrent leave operations:

- Instant departures fine as long as everything else stays connected

Node Departures

Variables within a node v:

- id: reference of v (we also write id(v))
- leaving∈{true,false}: indicates if v wants to leave
- left ∈ V∪{⊥}: left neighbor of v, i.e., id(left)<id(v) (if id(left) is defined)
- right ∈ V∪{⊥}: right neighbor of v, i.e., id(right)>id(v) (if id(right) is defined)

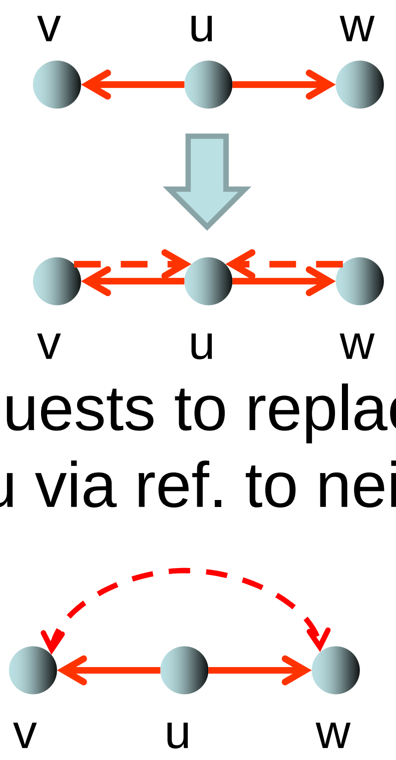


Extended Build-List Protocol

Foreback, Koutsopoulos, Nesterenko, S, Strothmann 14:

```
timeout: true →
    { executed by awake node u }
    if not leaving then
        left←linearize(id)
        right←linearize(id)
    else  { leaving }
        left←replace(RIGHT, right)
        right←replace(LEFT, left)
        sleep
```

only woken up by incoming msg

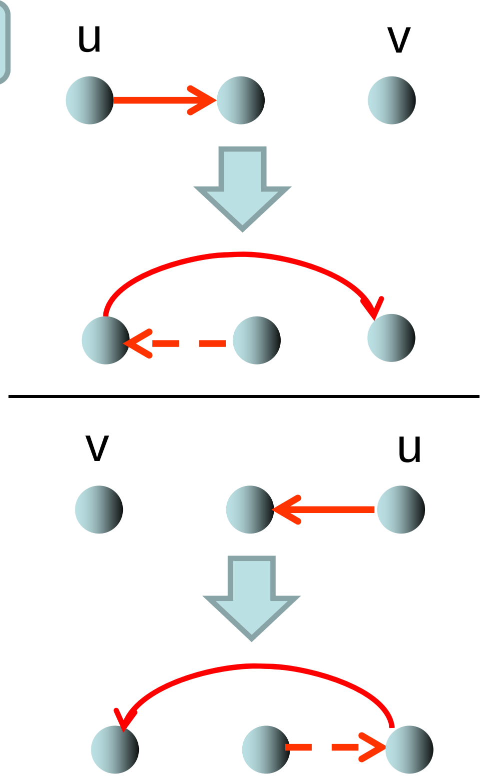


requests to replace ref. to u via ref. to neighbors

Extended Build-List Protocol

```
replace(dir∈{LEFT,RIGHT}, v) →
  { executed by node u }
  if dir=LEFT then
    if not leaving then
      left←linearize(id)
      left:=v
    else
      left←linearize(v)
  else { RIGHT }
    right←linearize(id)
    right:=v
```

breaks symmetry!



Further Results

Self-stabilizing protocols (simpler models & properties):

- Hypertrees [Dolev, Kat 2004]
- Sorted list [Onus, Richa, S 2007]
- Skip lists [Clouser, Nesterenko, S 2008]
- Skip graphs [Jacob, Richa, S, Schmid,Täubig 2009]
- Delaunay graphs [Jacob, Ritscher, S, Schmid 2009]
- De Bruijn graphs [Richa, S, Stevens 2011]
- Chord network [Kniesburges, Koutsopoulos, S 2011]
- Universal [Berns, Ghosh, Pemmeraju 2011]
- …

Structure of the Talk

- Motivation
- Basic model and notation
- Self-stabilizing sorted list
- Monotonically self-stabilizing sorted list
- Conclusion

Monotonic Recovery

- Monotonic reachability:
  Preserved when just using introduction, delegation and fusion
  → satisfied by Build-List Protocol

- Monotonic correctness:
  Those parts of the system that are still functional should remain functional as the repair proceeds
  → non-trivial!

Searching

```
Search(sid) →
    if sid=id then „success“, stop
    if (id(left)<sid<id or id<sid<id(right) then
        „failure“, stop      { guarantees availability }
    if sid<id then left←Search(sid)
    if sid>id then right←Search(sid)
```

Our goal: whenever searching for process B works from process A, it always works afterwards

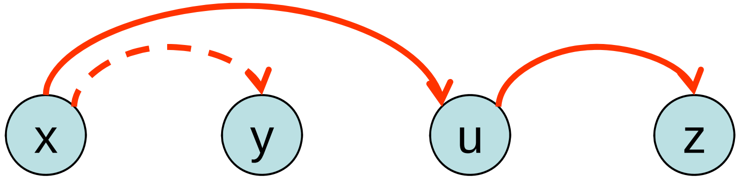
→ monotonic searchability

self-stab. + mon. searchability: mon. self-stabilization

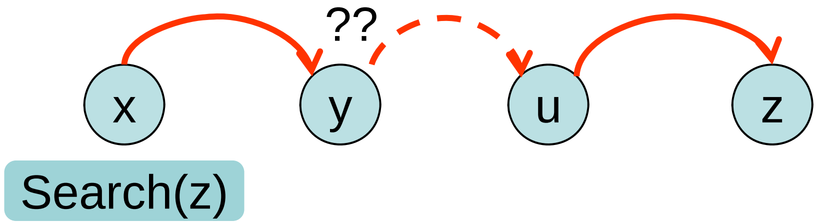
Sorted List

Build-List does not satisfy monotonic searchability.

- Search(z) can get from x to z:

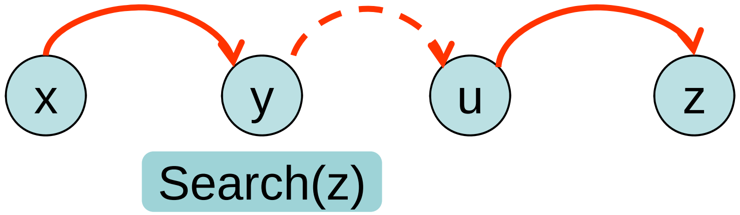


- After linearize(y) this is not the case any more:



Sorted List

- If in this case Search(z) waits at y, availability may not be guaranteed any more.



Why?

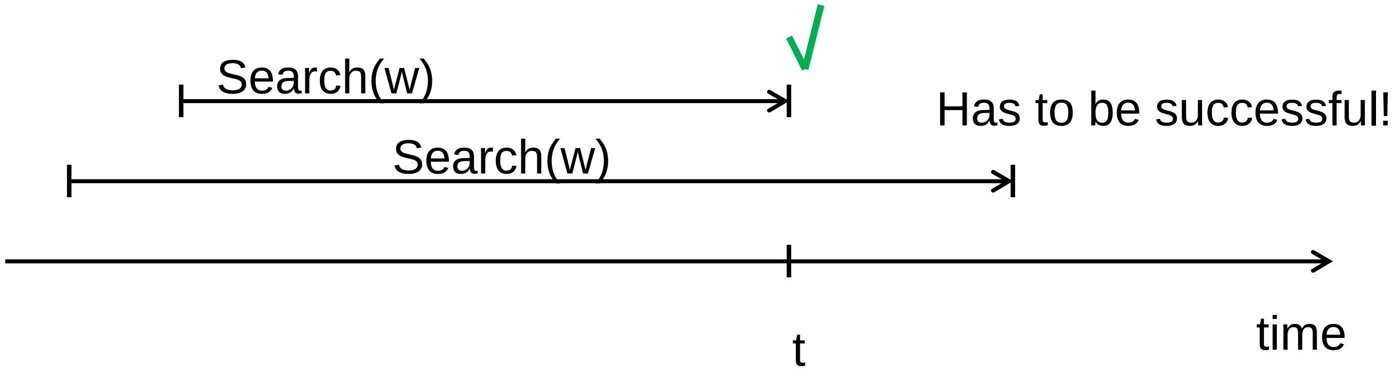
- y may have no clue that there still is some linearize(u) in transit from x.
- Even if y knew that, the information could be wrong (we consider self-stabilizing systems!), so y might wait in vein.

Sorted List

In which way can monotonic searchability be guaranteed?

1. If Search(w) initiated by v reaches process w at time t, then this also holds for all other Search(w) requests initiated by v that have not yet reached w.

Illustration:



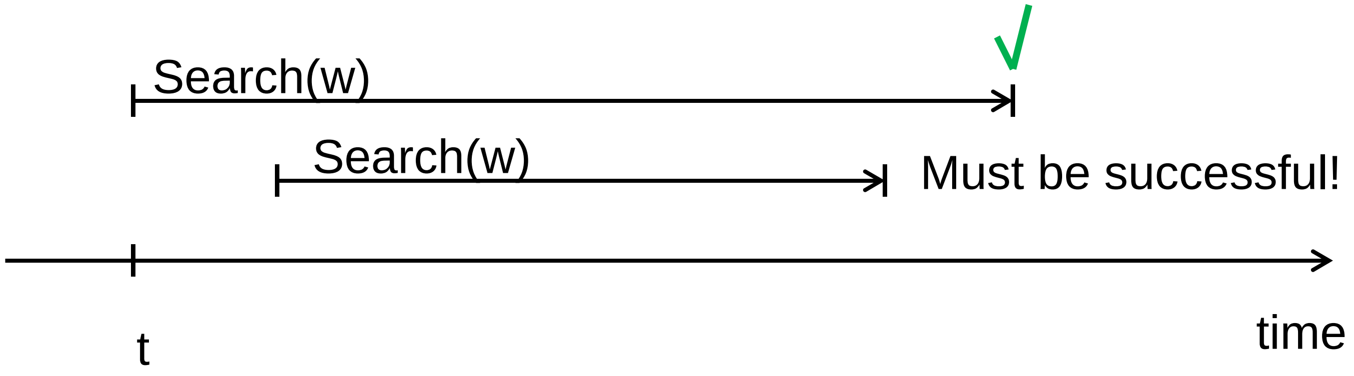
There is a counterexample!

Sorted List

In which way can monotonic searchability be guaranteed?

2. If a Search(w) initiated by v at time t reaches w, then this also holds for all other Search(w) requests initiated by v after time t.

Illustration:



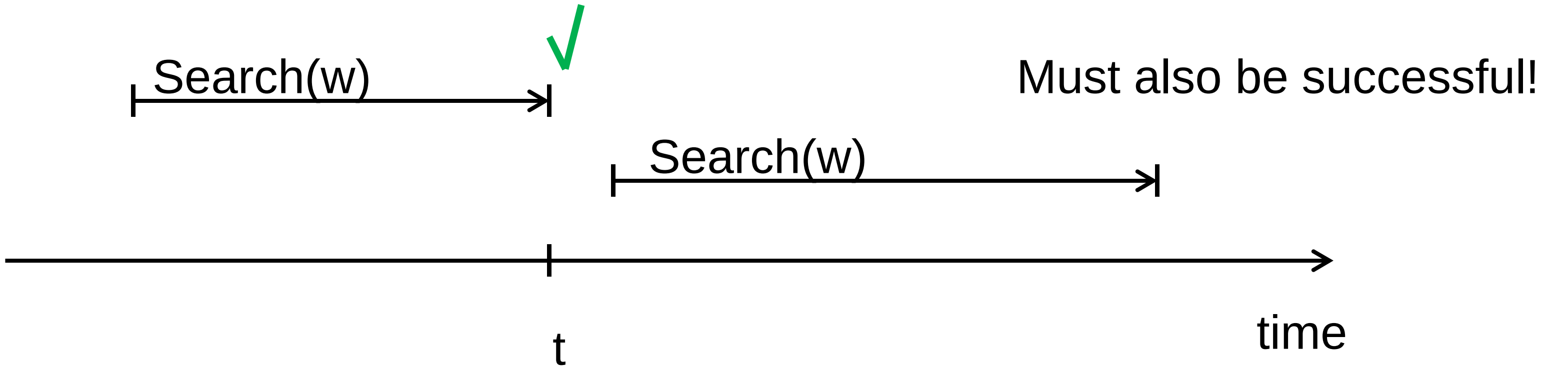
Also here there is a counterexample!

Sorted List

In which way can monotonic searchability be guaranteed?

3. If a Search(w) initiated by v reaches w at time t, then this also holds for all Search(w) injected by v after time t.

Illustration:



Search(w) ✓

Must also be successful!

Search(w)

t

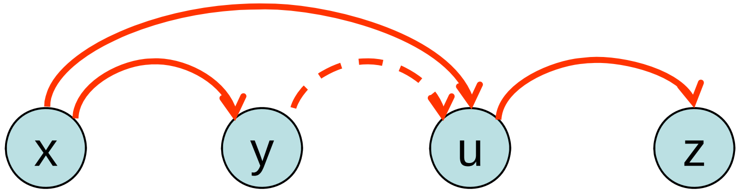
time

This can indeed by satisfied!

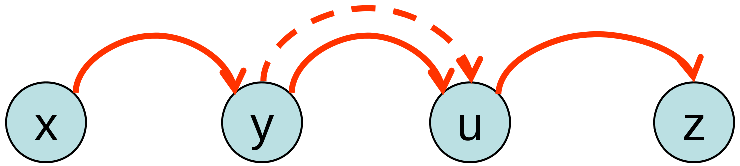
Sorted List

First ensure mon. reachability via explicit edges:

- instead of delegating u, x first introduces u to y:

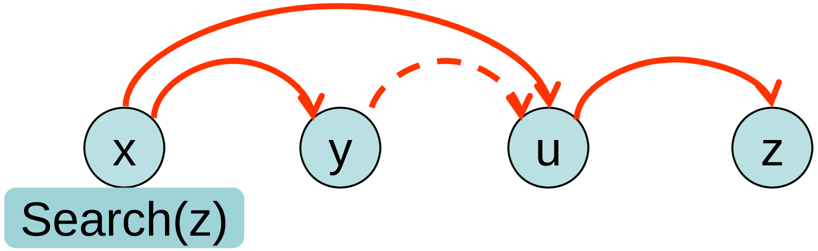


- once y has acknowledged to x the receipt of u, x delegates u to y



Sorted List

Problem: which way does Search(z) have to take, since x may now have several alternatives?

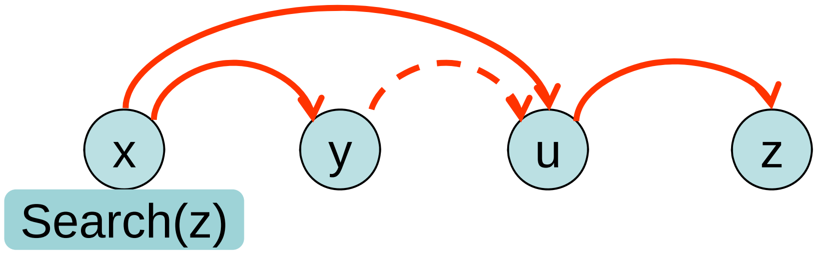


Idee 1: Search(z) waits at x until x only has one right neighbor. This will eventually be the case if the node set is static, but if not, then the search request may wait forever.

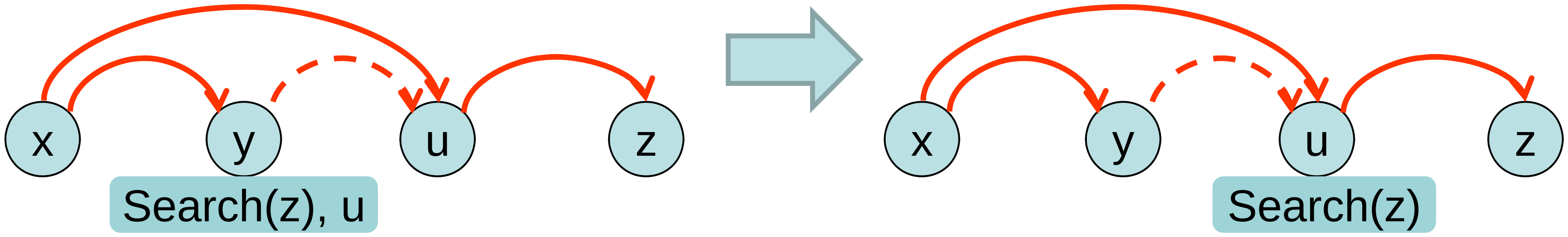
Idee 2: Search(z) is sent along all edges in the direction of z. However, then the number of Search(z) request may exponentially increase over time!

Sorted List

Problem: Which way should Search(z) go, since x may have several alternatives?



Alternative idea: Search(z) is always sent to the closest neighbor in the direction of z, but all other right neighbors will be remembered in the Search(z) request. In this way, Search(z) can get from y to u:



Sorted List

Extended Search protocol:

- Every node v has neighbor sets Left and Right
- Every Search request stores a set Next of nodes that need to be visited on the way to the destination.

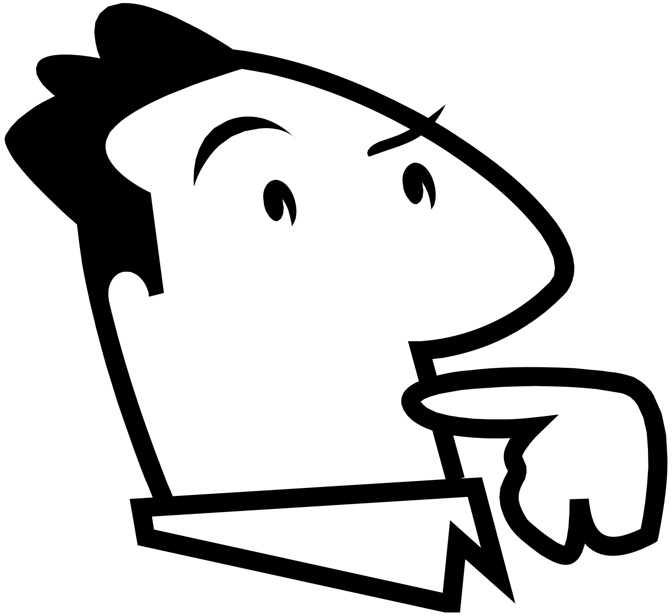
```
Search(sid, Next) $\rightarrow$
    if $sid=id$ then „success“, stop
    if $sid<id$ then
        $Next:=Next \cup Left$
        $v:= \operatorname{argmax}\{ id(w) \mid w \in Next \}$
        if $id(v)<sid<id$ then „failure“, $Left:=Left \cup Next$, stop
                              else $v \leftarrow Search(sid,Next \setminus \{v\})$, $Left:=Left \cup \{v\}$
    else
        $Next:=Next \cup Right$
        $v:= \operatorname{argmin}\{ id(w) \mid w \in Next \}$
        if $id<sid<id(v)$ then „failure“, $Right:=Right \cup Next$, stop
                              else $v \leftarrow Search(sid,Next \setminus \{v\})$, $Right:=Right \cup \{v\}$
```

Important for self-stabilization since some nodes in Next could be critical for connectivity.

Conclusion

This talk: first approach towards designing monotonically self-stabilizing distributed data structures.

Young research area. Runtime and churn not yet well-understood, so much more work needed!



Questions?