Self-Stabilizing Distributed Data Structures

Christian Scheideler

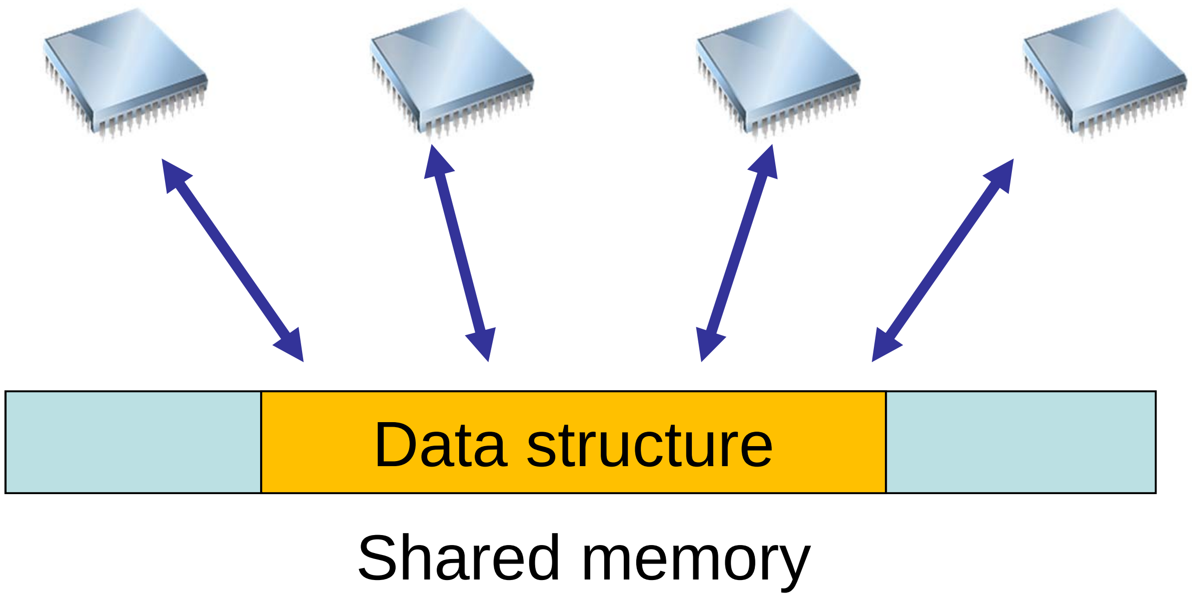
Dept. of Computer Science

University of Paderborn

Joint work with Riko Jacob, Mikhail Nesterenko,
Andrea Richa, Stefan Schmid, and many others

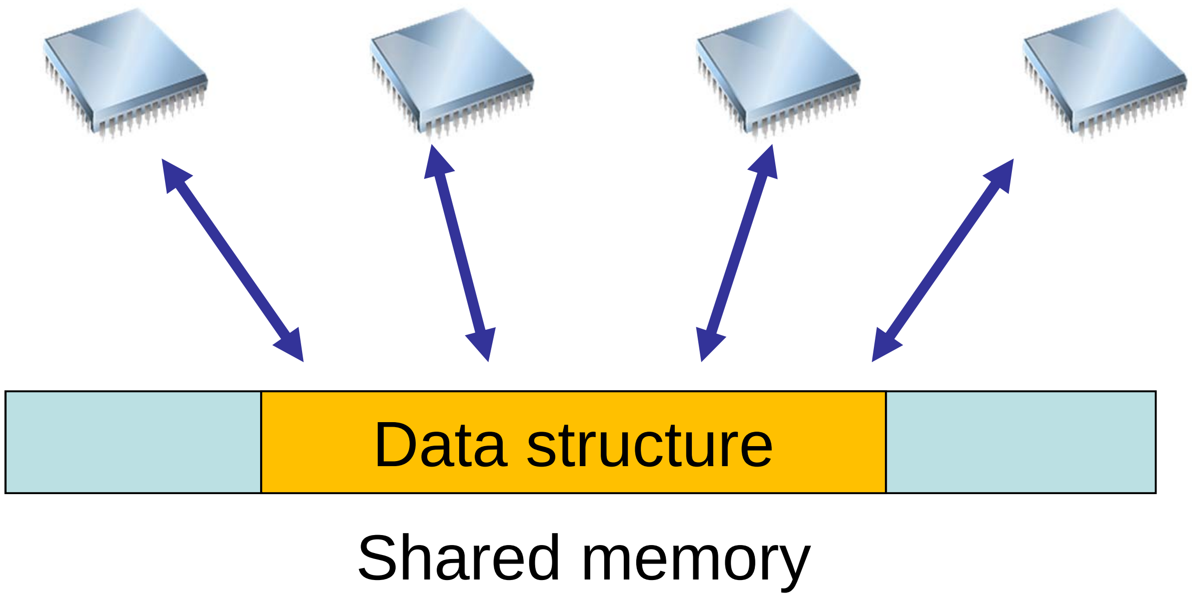
Motivation

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



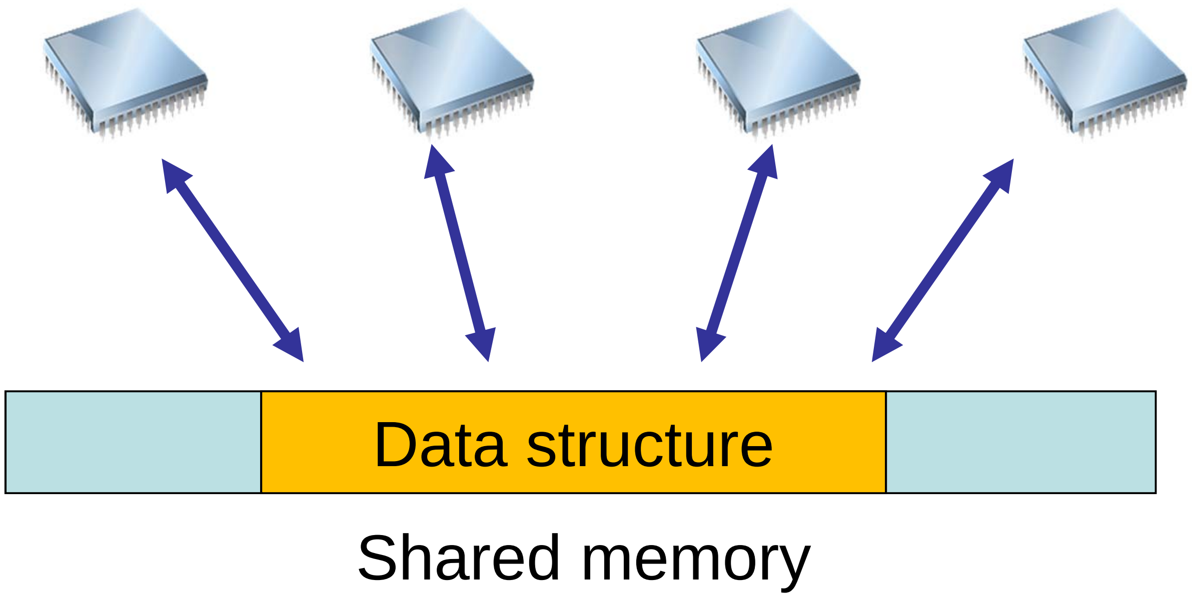
Motivation

- Shared 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

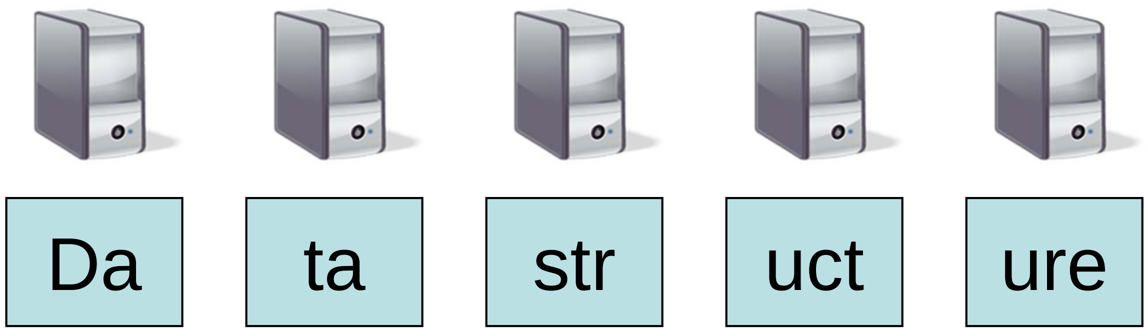


Motivation

Situation different for large distributed systems:

- no (hardware-supported) shared memory available
- continuous change in membership and faults
- adversarial behavior

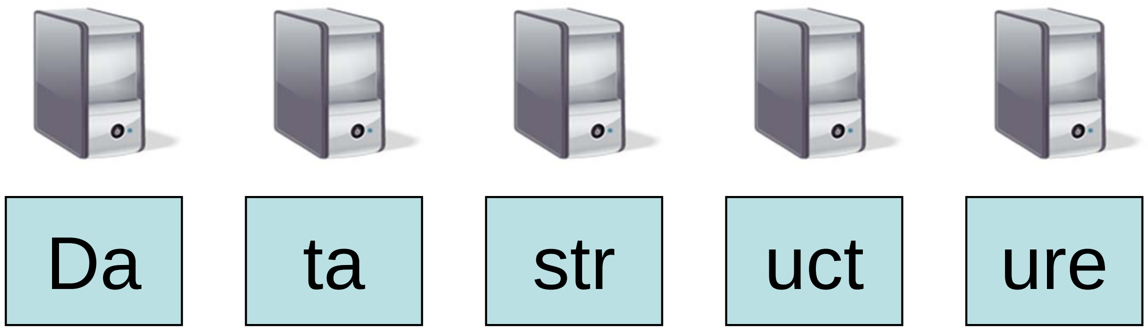
→ Illegal states cannot be avoided.



Motivation

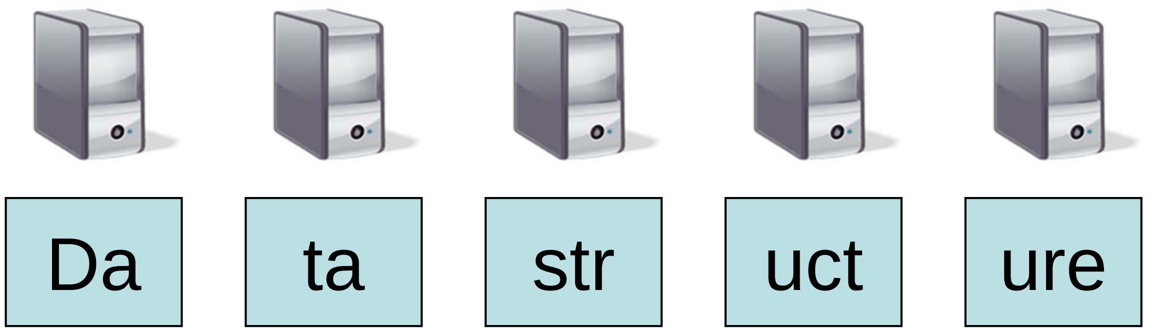
How to best manage a distributed data structure?

- Emulate a reliable shared memory layer
  pro: only data plane   con: can be expensive!
- Directly implement DS on top of system
  pro: more efficient   con: needs to take care of dynamics, faults, and adversarial behavior by itself!



Topic of this Talk

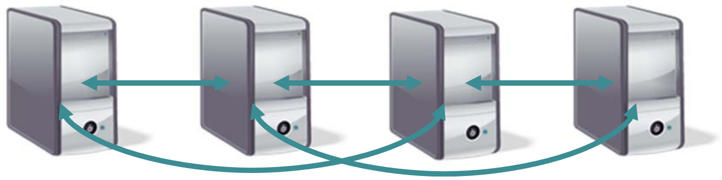
Rigorous framework for study of efficient and robust direct implementations of distributed data structures



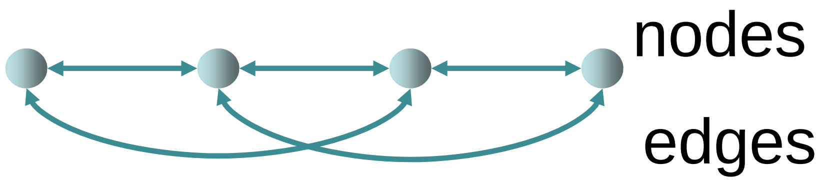
Model

We will model data structures as directed graphs.

- Data structure established by computers / processes:



- Graph representation:

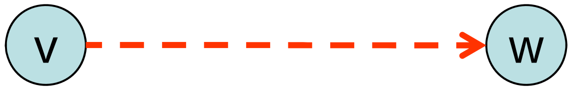


- Edge $A \rightarrow B$ means: A knows B

Model

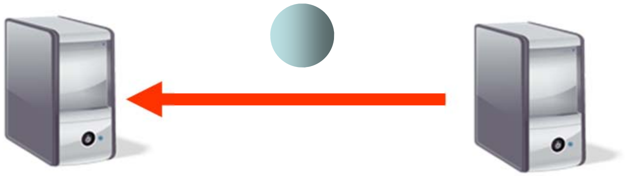
- Edge set E_L: set of pairs (v,w) over all nodes v and w, with the property that v has a link to w (explicit connections).

- Edge set E_M: set of pairs (v,w) with the property that there is a link request in v containing a reference to w (implicit connections).



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

Model

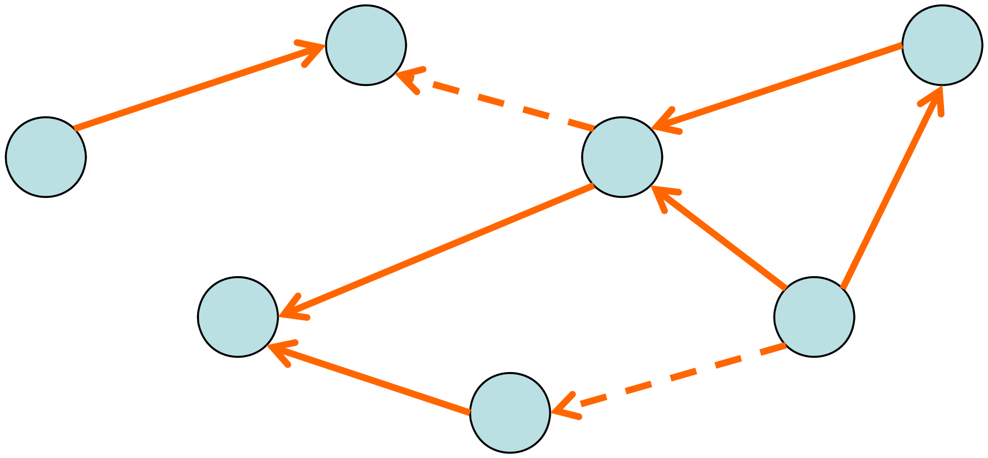


Assumptions:

- nodes can only communicate via explicit connections
- the requests are forwarded in FIFO order along an explicit connection (FIFO: first-in-first-out)
- for simplicity: no corrupted references or references to failed nodes (so here no need for failure detectors)

Distributed Data Structures

Fundamental goal: topology of data structure (i.e., G) is kept weakly connected at all times



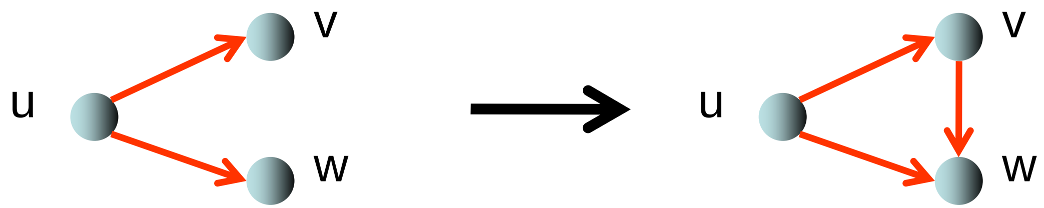
Fundamental rule: never just „throw away“ a reference!

Distributed Data Structures

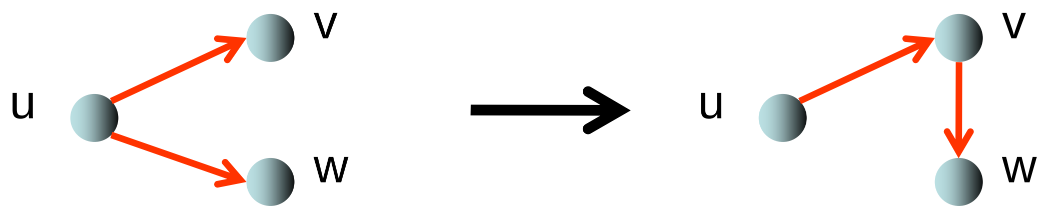
Admissible rules for distributed data structures:

- The following network changes are admissible for a node u so that there is no danger of losing connectivity:

Introduction:



Delegation:

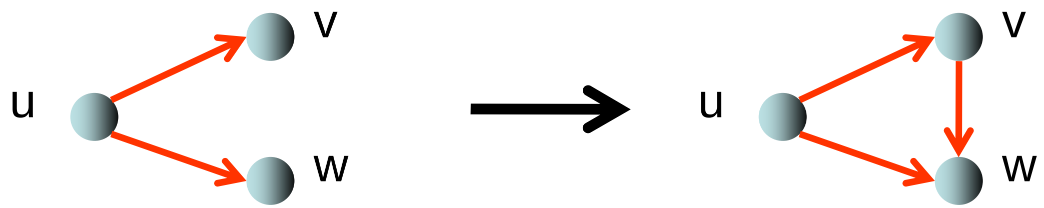


Fusion:

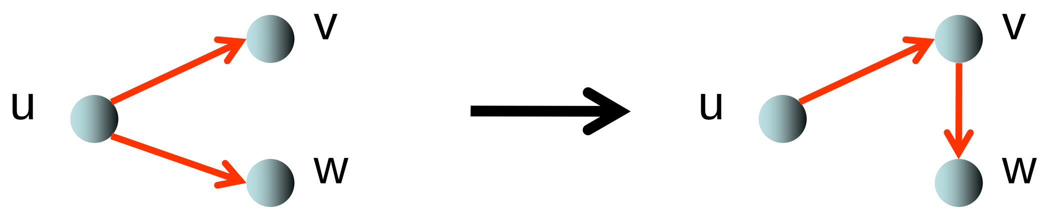
Distributed Data Structures

Theorem 1: These rules are universal in a sense that one can get from any weakly connected graph G=(V,E) to any strongly connected graph G´=(V,E´) via these rules.

Introduction:



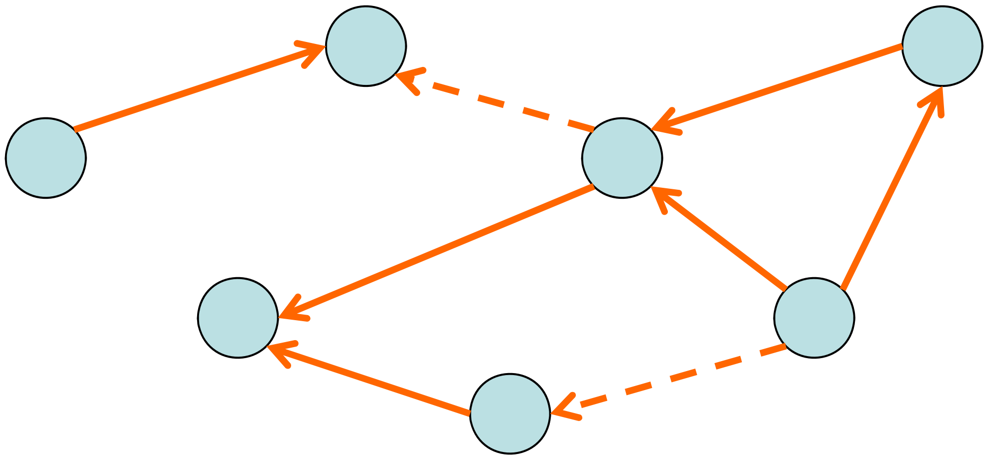
Delegation:



Fusion:

Distributed Data Structures

So introduction, delegation, and fusion allow a DS, in principle, to recover from any illegal state.



Ideally: DS recovers monotonically from illegal state

Distributed Data Structures

Condition for reachability:

- Monotonic reachability: If there is a directed path from u to v in G at time t, then also at any time t´>t under the condition that no node leaves the system or becomes faulty.

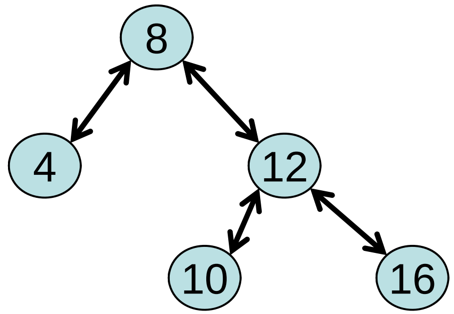
Remark: The introduction, delegation, and fusion rules satisfy this condition.

Is reachability sufficient for a data structure?

Distributed Data Structures

Is reachability sufficient for a data structure?

No, because the operations of a data structure only work if the data structure has the desired form (e.g., a binary search tree).



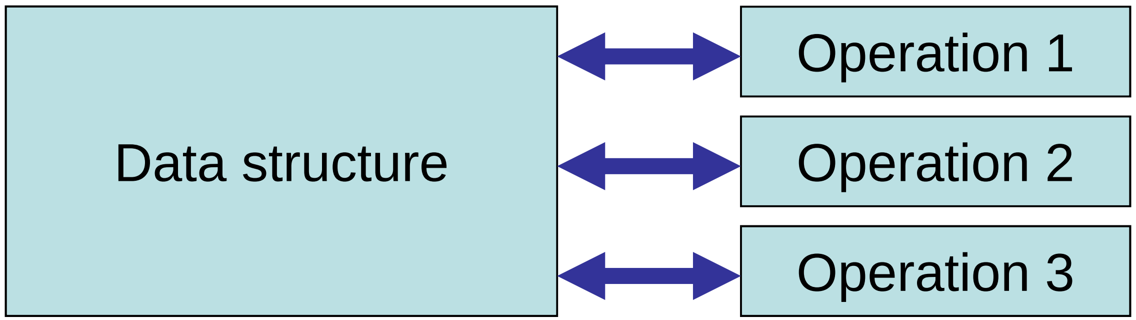
Therefore, we demand monotonic DS-reachability:

If v is reachable for DS-operations from u at time t, then also at any time t´>t under the condition that no node leaves the system or becomes faulty.

How can we stabilize a data structure DS while preserving monotonic DS-reachability?

Self-Stabilizing Data Structures

Recall the fundamental concept of a data structure

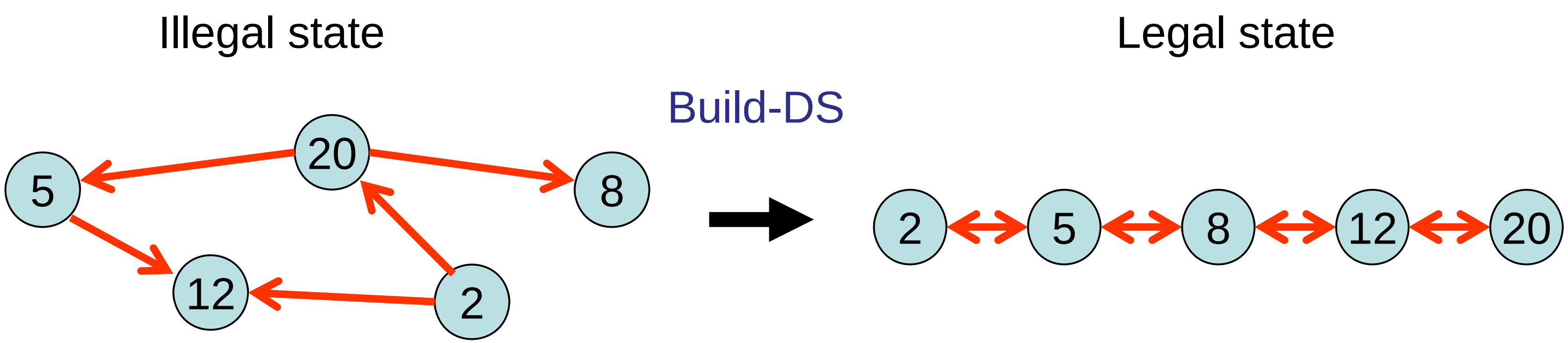


Standard operation in sequential case:
Build-DS(S): given a set of elements S, construct data structure DS for S

Distributed dynamic case:
Build-DS: distributed protocol that can stabilize DS from an arbitrary weakly connected state and that can also guarantee monotonic DS-reachability.

Self-Stabilizing Data Structures

Example: sorted list



Definition 2: Build-DS stabilizes the data structure DS if

1. when starting from an arbitrary weakly connected state, Build-DS can get DS back into a legal state in finite time (convergence) and
2. when starting from an arbitrary legal state, Build-DS maintains a legal state for DS (closure),

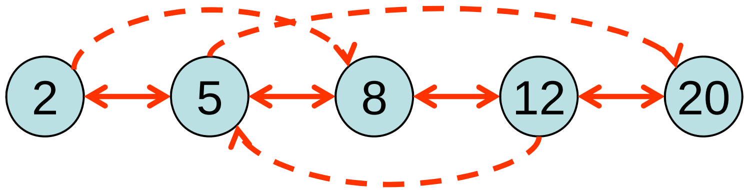
as long as no operations are executed in DS and no node leaves the system or becomes faulty.

Self-Stabilizing Data Structures

What exactly do we mean by a „legal" state?

Our approach: We call the state of a data structure DS legal if DS is legal without considering the implicit connections.

Example: for a sorted list the following topology would be legal

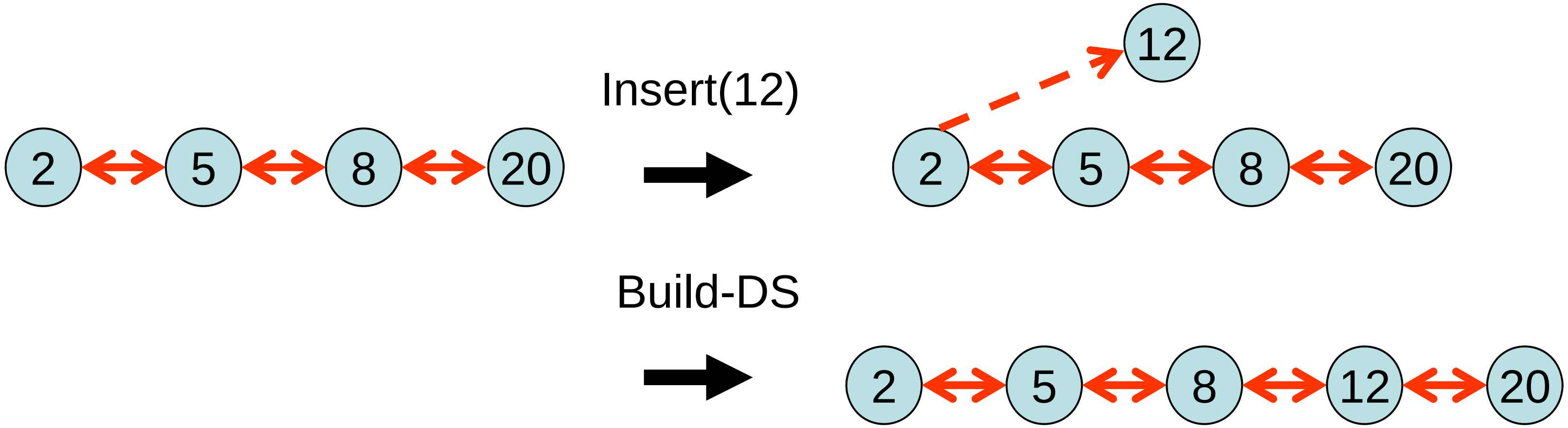


Definition 3: Build-DS monotonically stabilizes the data structure DS if Build-DS stabilizes DS (see Def. 2) and also ensures monotonic DS-reachability.

Self-Stabilizing Data Structures

Observation: If Build-DS stabilizes a data structure, then Build-DS could also be used to stabilize an operation.

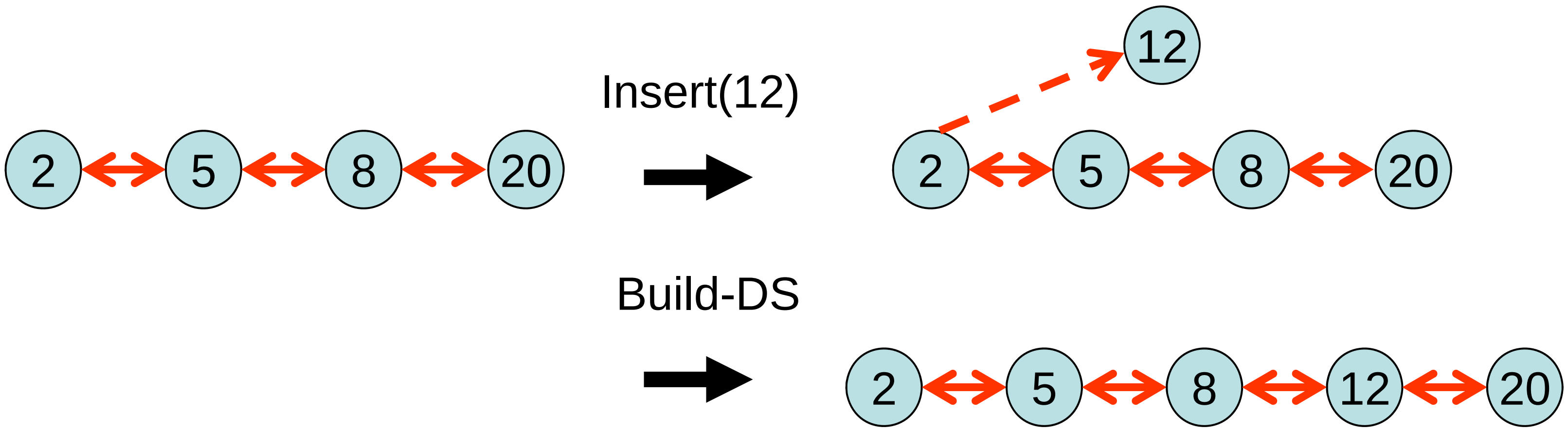
Example: 2 initiates Insert(12) on a sorted list.



Self-Stabilizing Data Structures

Sorted list: The Insert(v) operation has stabilized once v is connected to the current pred(v) and succ(v).

Example: 2 initiates Insert(12) on a sorted list.



Self-Stabilizing Data Structures

How do we want to measure the quality of a distributed data structure DS?

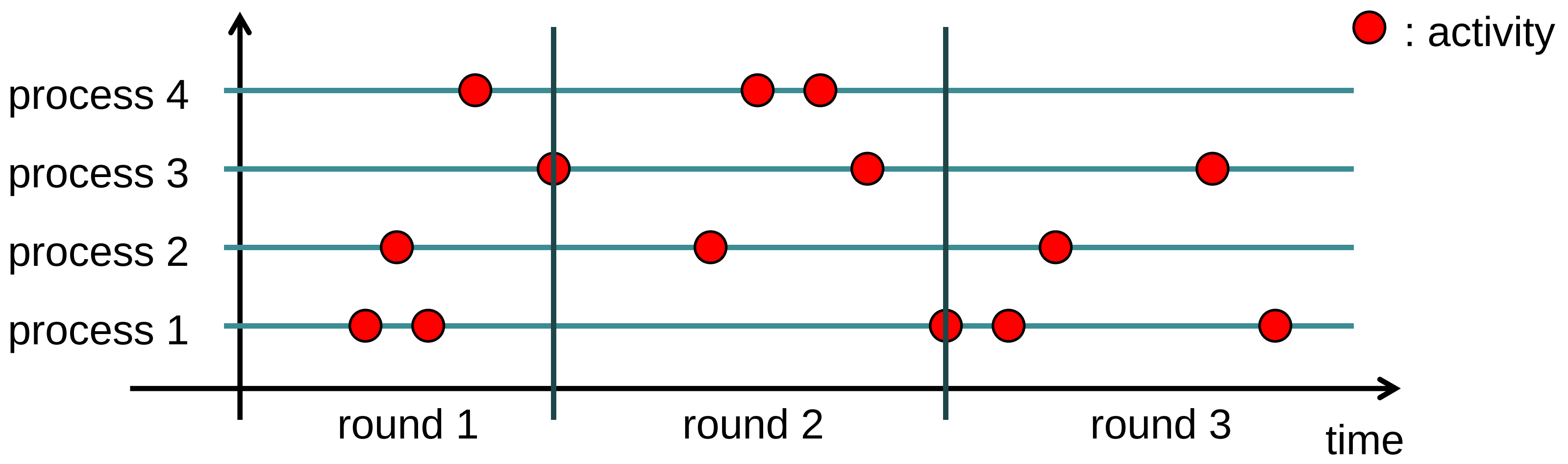
Build-DS Protocol:

- Robustness criteria:
  - Self-stabilization from any weakly connected state
  - Monotonic DS-reachability
- Efficiency criteria:
  - Low worst-case time/work for self-stabilization
  - Low maintenance overhead in stable state
  - Low worst-case time/work for stabilization of a single operation on a stable DS

Self-Stabilizing Data Structures

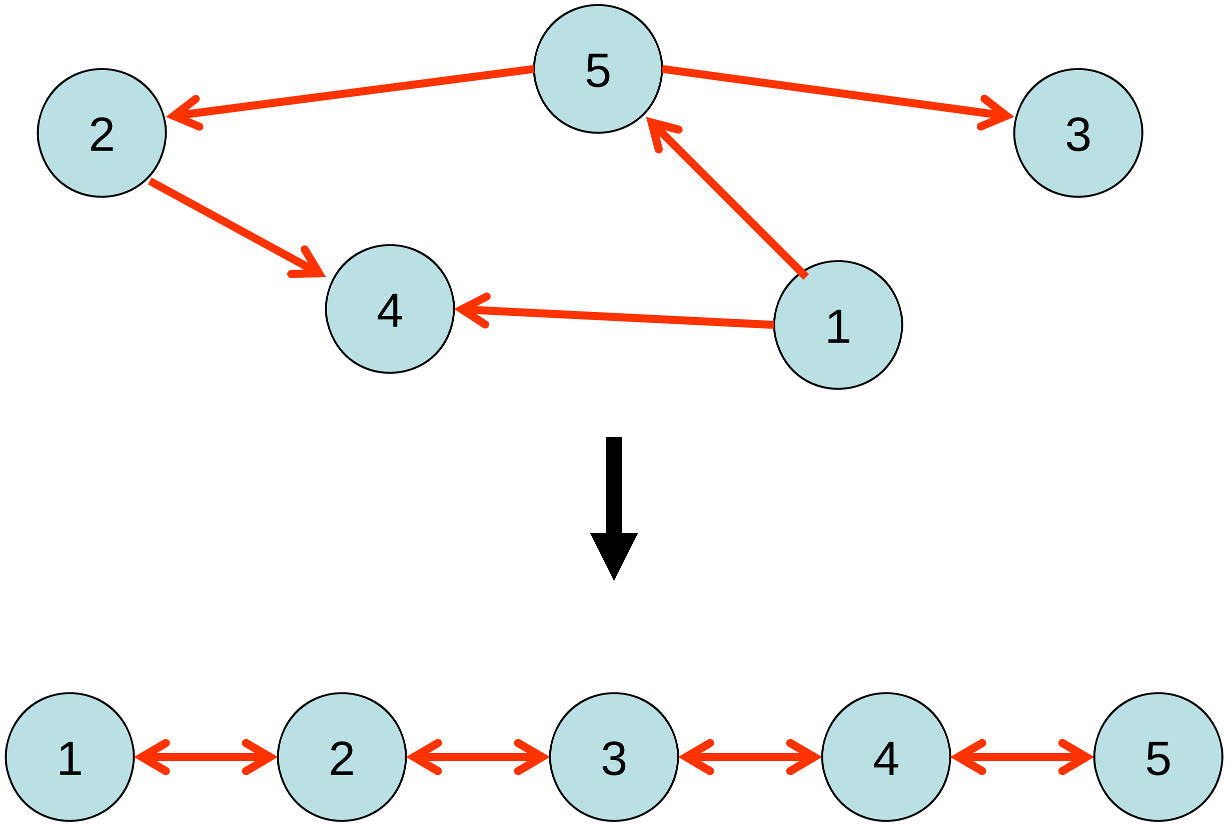
Time model:

- We allow an arbitrary asynchronous execution of the requests by the processes.
- A round is over once every process that has requests to execute executed at least one of these requests.
- We measure the runtime in the number of rounds.



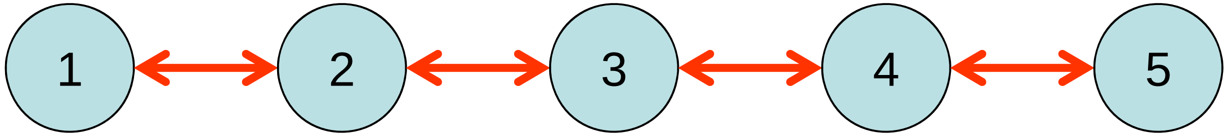
Sorted List

Build-List:



Sorted List

Ideal state:



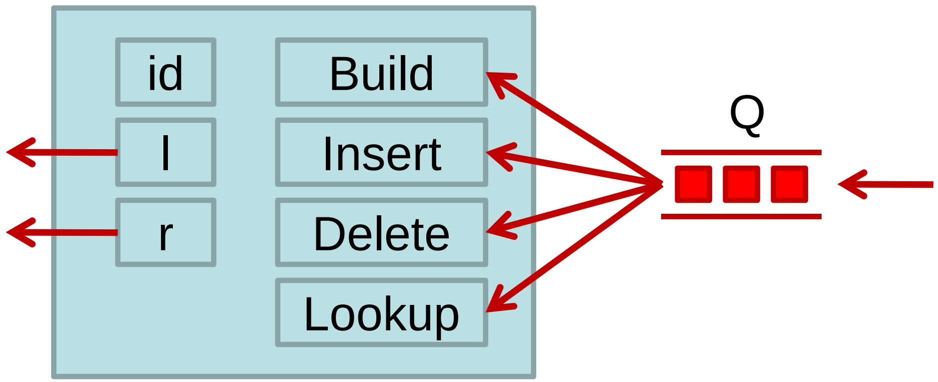
Operations:

- Build-List: forms a sorted list out of any weakly connected state
- Insert(v): insert node v into list
- Delete(v): remove node v from list
- Lookup(id): sends lookup request to that node w with id(w)=id

Sorted List

Variables in a node v:

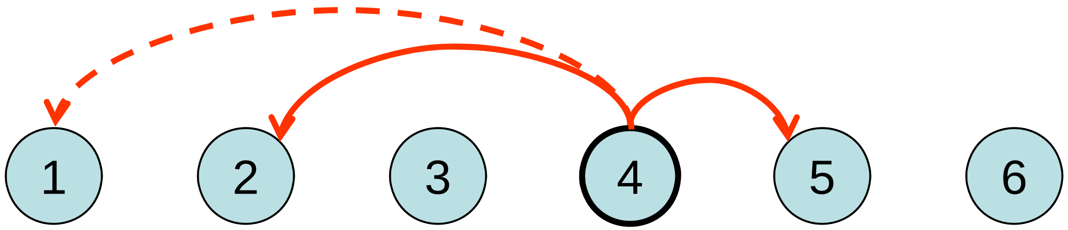
- v.id: ID of node v in some ordered space
- v.l $\in$ V$\cup${$\varnothing$}: closest left neighbor of v
- v.r $\in$ V$\cup${$\varnothing$}: closest right neighbor of v



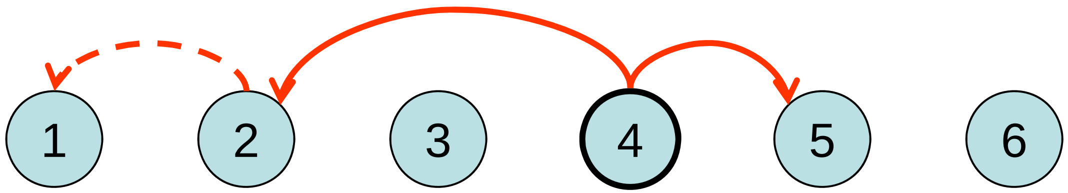
Sorted List

Build-List via linearization:

Idea: keep edges to closest neighbors and delegate rest.



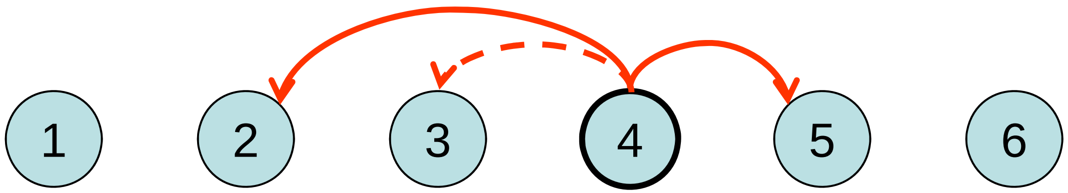
Upon Build-List(1): 4 generates request 2←Bild-List(1)



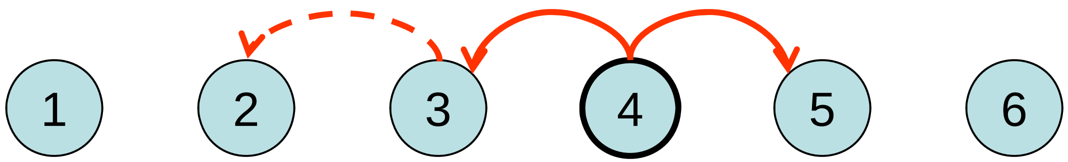
Sorted List

Build-List via linearization:

Idea: keep edges to closest neighbors and delegate rest.



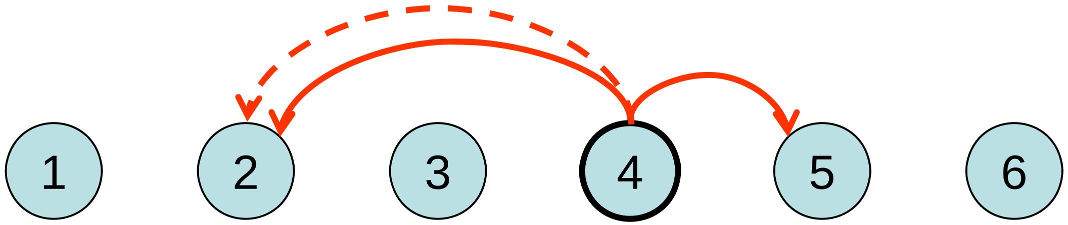
Upon Build-List(3): 4 sets 4.l:=3 and generates request
3←Build-List(2)



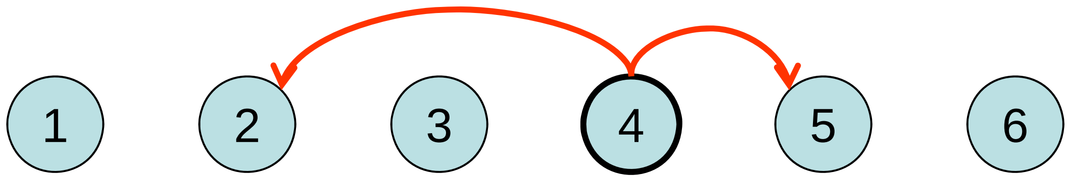
Sorted List

Build-List via linearization:

Idea: keep edges to closest neighbors and delegate rest.



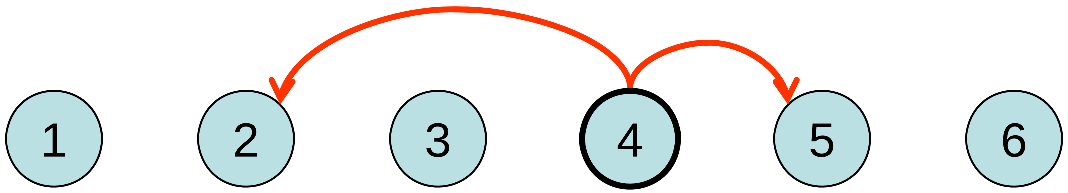
Upon Build-List(2) oder Build-List(5): 4 fuses that with existing edge.



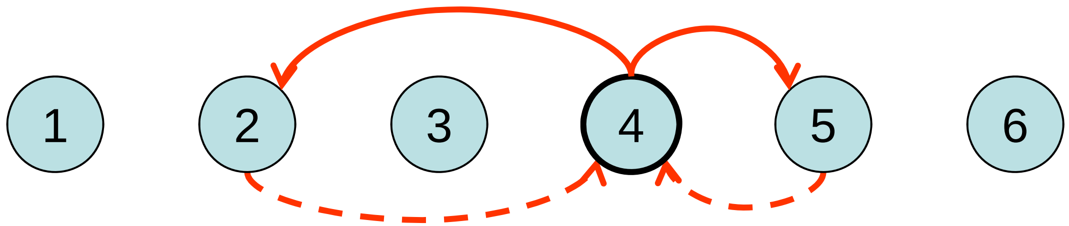
Sorted List

Build-List via linearization:

Idea: keep edges to closest neighbors and delegate rest.



Periodically, we also execute Build-List(): 4 generates requests 2←Build-List(4) und 5←Build-List(4) .

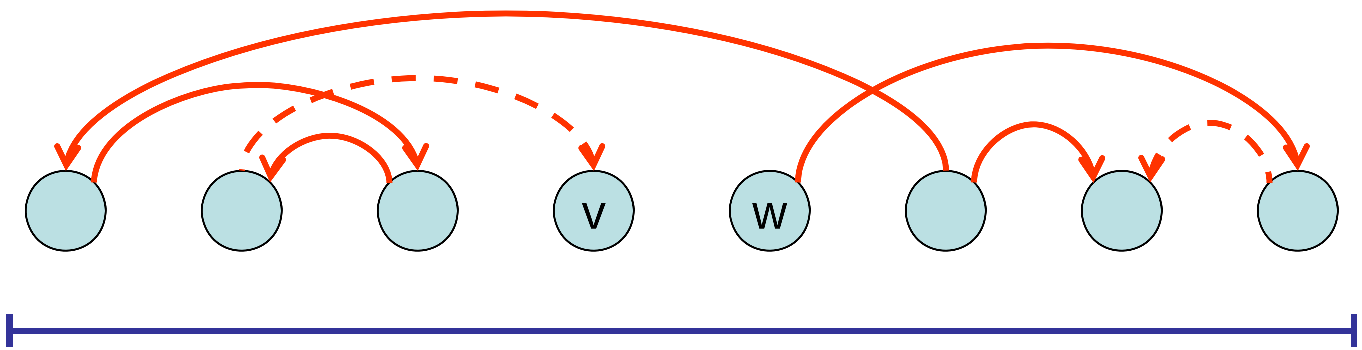


Sorted List

Theorem 4 (Convergence): For any weakly connected graph $G=(V,E_L \cup E_M)$, Build-List generates a sorted list.

Proof sketch:

- Consider an arbitrary neighboring pair v,w w.r.t. sorted list.
- Since G is weakly connected, there is a (not necessarily directed) path in G from v to w.

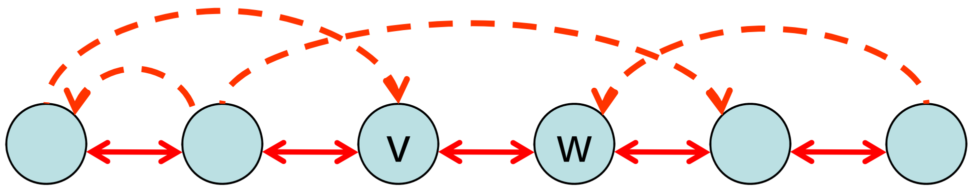


range of path from v to w shrinks monotonically

Sorted List

Theorem 5 (Closure): If the explicit edges already form a sorted list, then these edges will be preserved under any Build-List call.

Proof:



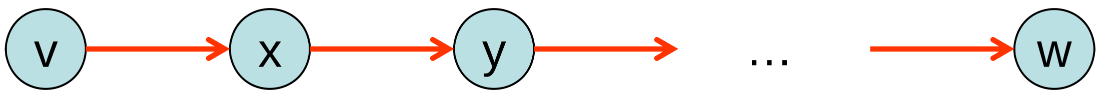
- An explicit edge is only given up if the node learns about a closer node.
- Once the explicit edges form a sorted list, this does not happen any more. Indeed, in this case the implicit edges will only be delegated further until they merge with an explicit edge.
- Hence, at the end we are only left with the sorted list.

Sorted List

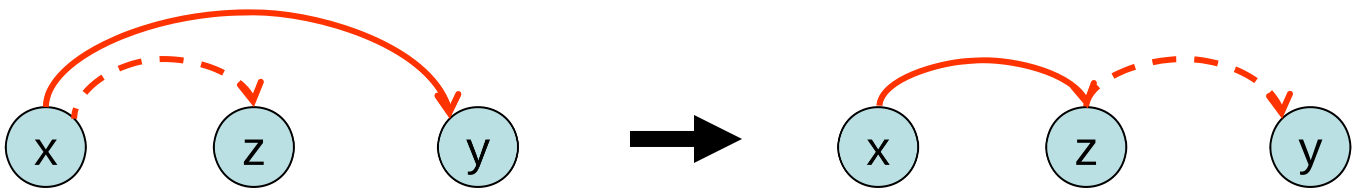
Theorem 6: Build-List guarantees monotonic List-reachability.

Proof sketch:

- Node w is list-reachable from node v if there is a sequence of (u,u.r)-edges (resp. (u,u.l)-edges) that leads from v to w.



- This property can be violated if an edge is delegated.

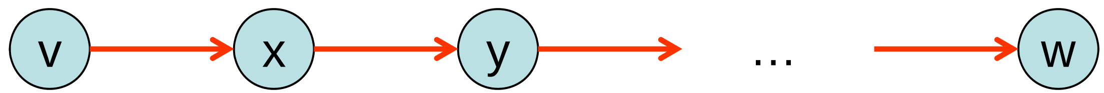


Sorted List

Theorem 6: Build-List guarantees monotonic List-reachability.

Proof sketch:

- Node w is list-reachable from node v if there is a sequence of (u,u.r)-edges (resp. (u,u.l)-edges) that leads from v to w.



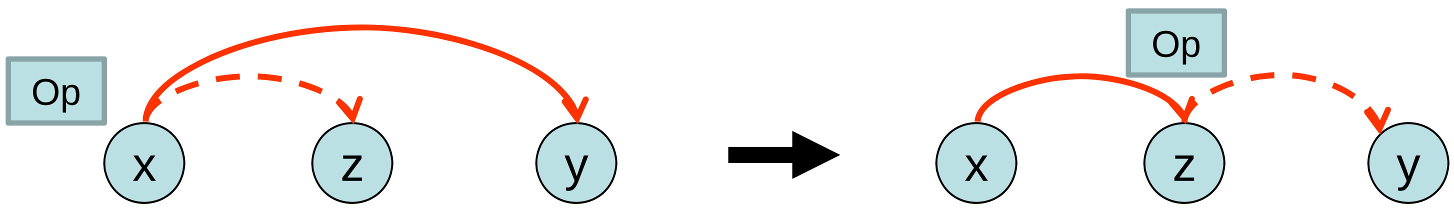
- However, we only need a weaker condition for the operations: If an operation is able to get from v to w, then there must be a sequence of (u,u.r)-edges (resp. (u,u.l)-edges) over time that leads from v to w.

Line Metric

Theorem 6: Build-List guarantees monotonic List-reachability.

Proof sketch:

- Suppose that we are in a situation in which Op is to be sent to y and Op is executed after Build-List(y) in x.

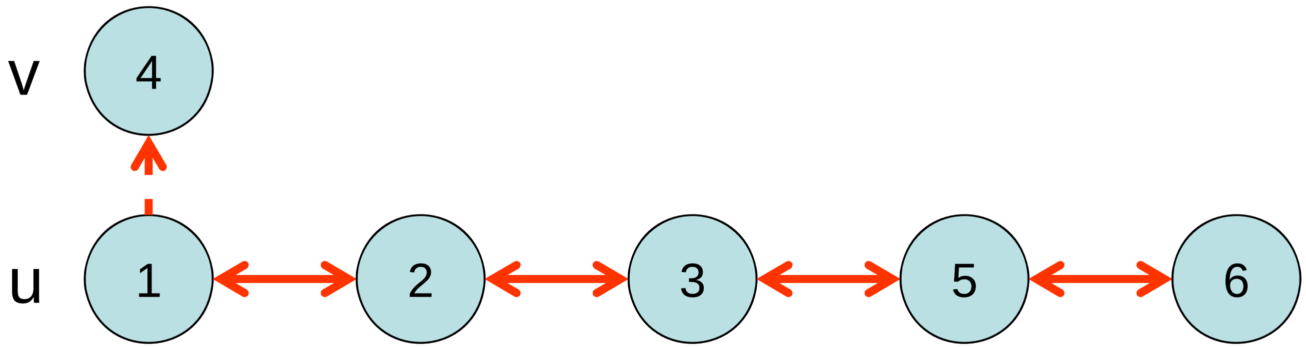


- Then y is delegated to z and afterwards, Op is delegated to z as well (or a closer node), so that Op is still executed after Build-List(y) due to the FIFO rule on links.
- Inductive proof: Op eventually reaches y.

Sorted List

Insert(v):

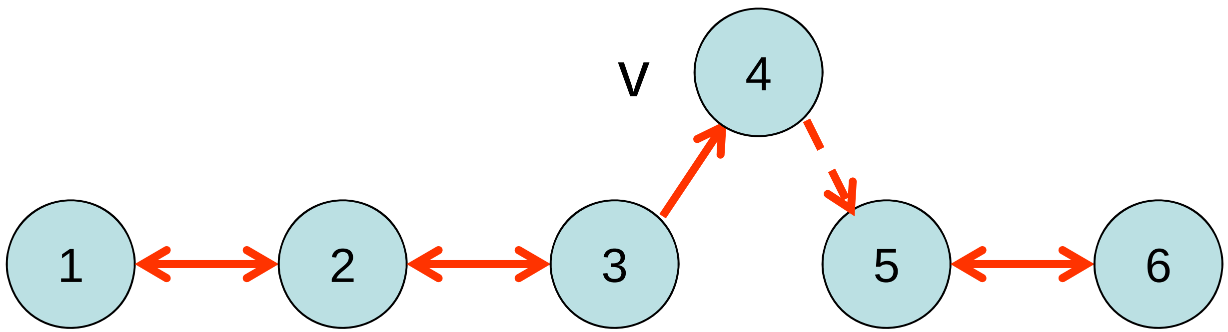
- Suppose that node u executes the request Insert(v).
- Then u simply calls u←Build-List(v).
- The Build-List protocol will then incorporate v in the right position in the ordered list, i.e., Build-List stabilizes the Insert operation.



Sorted List

Insert(v):

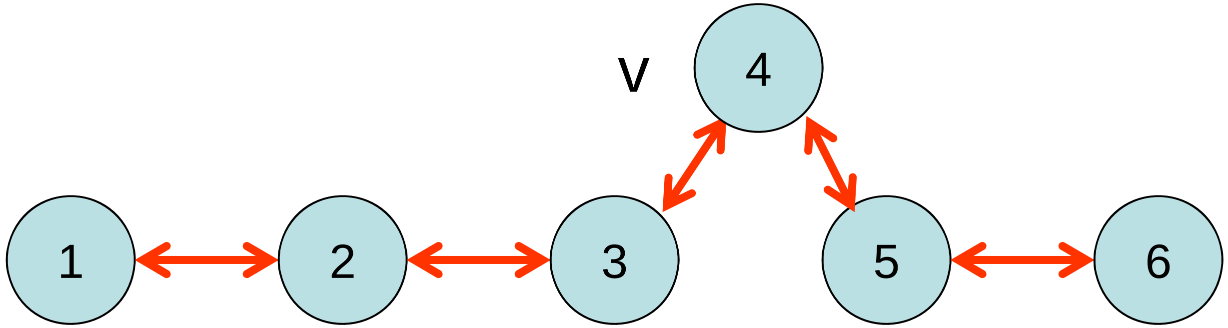
- Suppose that node u executes the request Insert(v).
- Then u simply calls u←Build-List(v).
- The Build-List protocol will then incorporate v in the right position in the ordered list, i.e., Build-List stabilizes the Insert operation.



Sorted List

Insert(v):

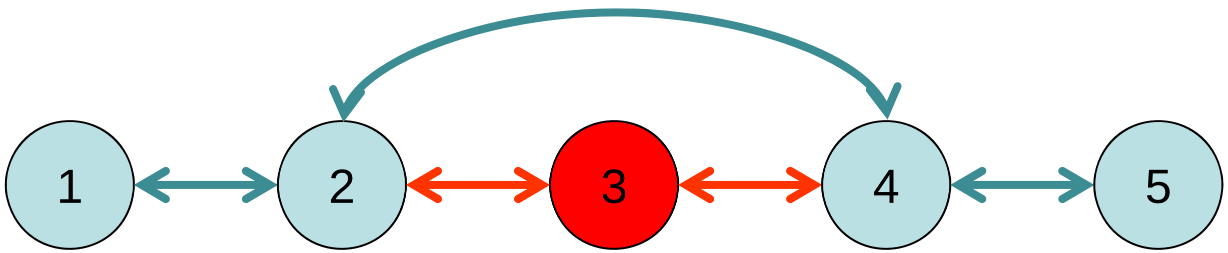
- Suppose that node u executes the request Insert(v).
- Then u simply calls u←Build-List(v).
- The Build-List protocol will then incorporate v in the right position in the ordered list, i.e., Build-List stabilizes the Insert operation while preserving monotonic list-reachability.



Sorted List

Delete(v): we assume that a node v can only delete itself.

How to stabilize Delete(v) so that monotonic list-reachability is preserved?

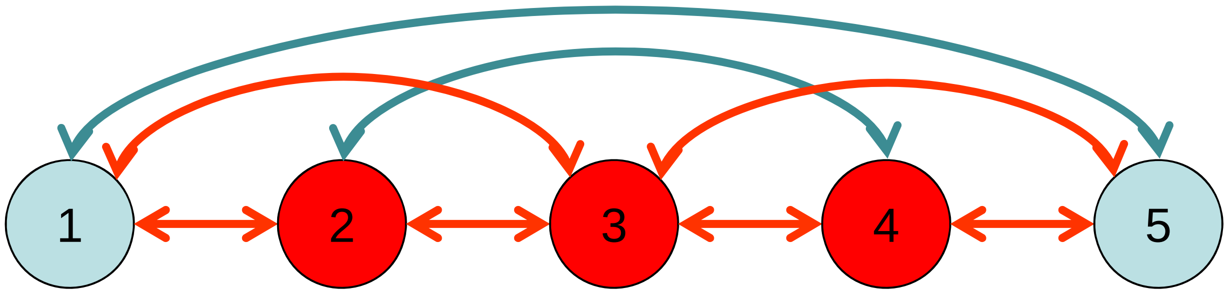


- a leaving node v starts converting its explicit edges into special leave edges and adds a new edge that connects its current predecessor and successor

Sorted List

Delete(v): we assume that a node v can only delete itself.

How to stabilize Delete(v) so that monotonic list-reachability is preserved?

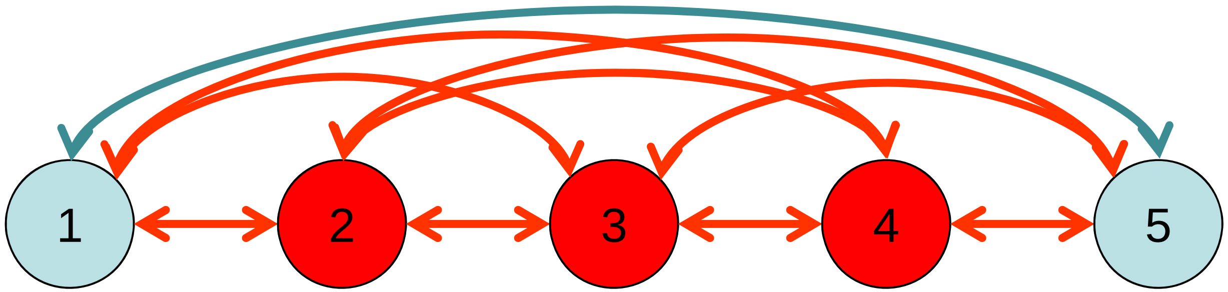


- the leaving nodes continue the conversions till no leaving node is connected to a non-leaving edge

Sorted List

Delete(v): we assume that a node v can only delete itself.

How to stabilize Delete(v) so that monotonic list-reachability is preserved?

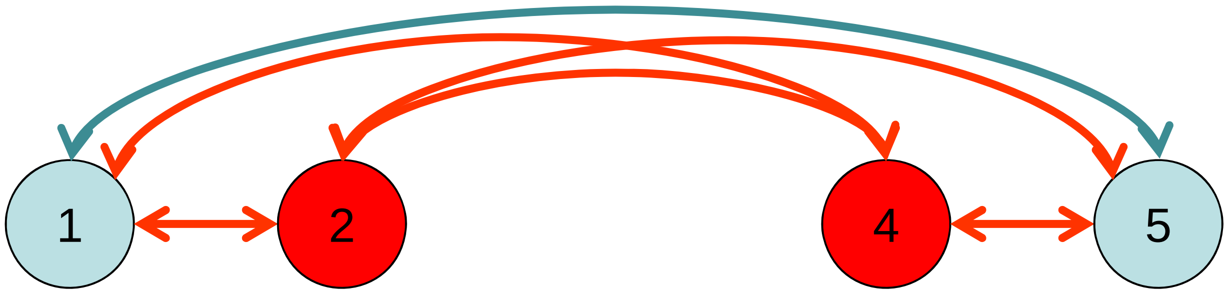


- non-leaving nodes only use standard (non-leaving) edges to forward requests
- leaving nodes do not generate any further requests

Sorted List

Delete(v): we assume that a node v can only delete itself.

How to stabilize Delete(v) so that monotonic list-reachability is preserved?



- once a leaving node has no requests any more, it leaves the system (and takes all of its edges with it)

Sorted List

Delete(v): we assume that a node v can only delete itself.

How to stabilize Delete(v) so that monotonic list-reachability is preserved?

- once a leaving node has no requests any more, it leaves the system (and takes all of its edges with it)

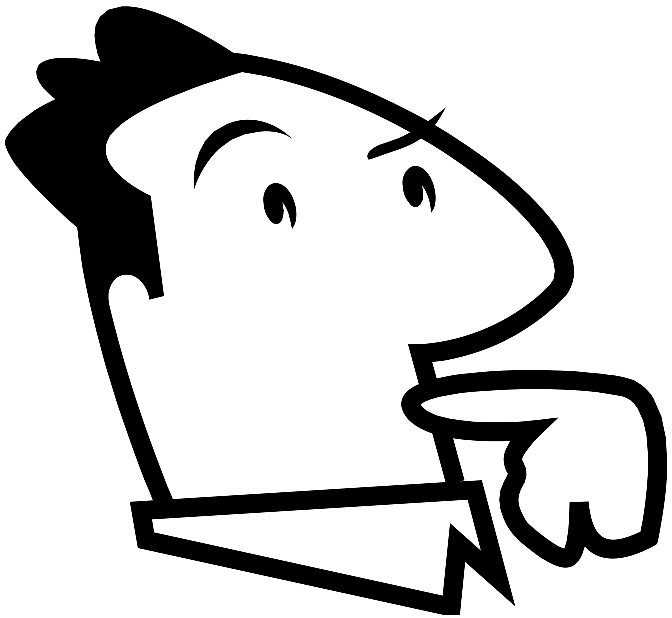
Conclusion

Our hope: starting point to design efficient and robust distributed data structures for large distributed systems.

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]

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



Questions?