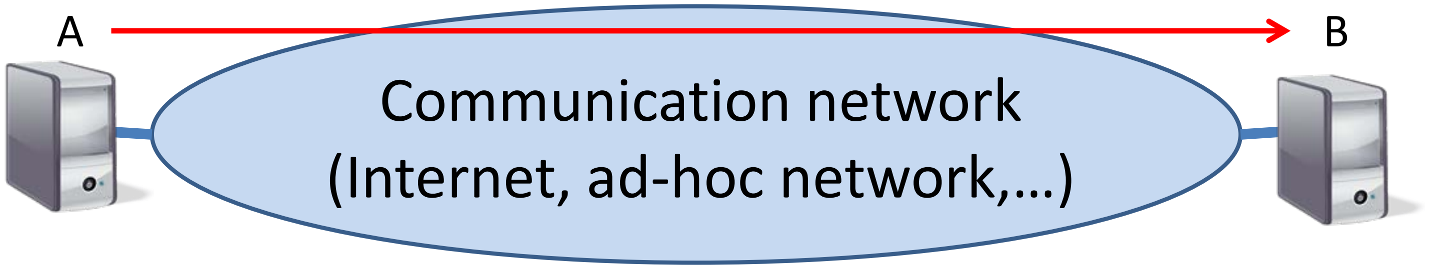
Advanced Distributed Algorithms and Data Structures

Chapter 9: Dynamic Overlay Networks

Christian Scheideler
Institut für Informatik
Universität Paderborn

Model and Basic Primitives



A knows (IP address, MAC address,… of) resp. has access autorization for B : network can send message from A to B

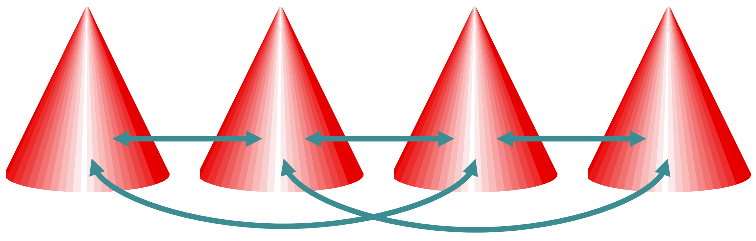
High-level view:

A knows B $\Rightarrow$ overlay edge (A,B) from A to B (A $\rightarrow$ B)

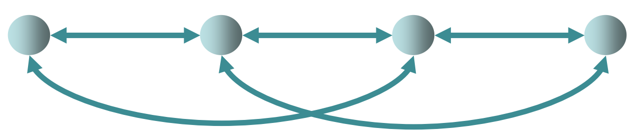
Set of all overlay edges forms directed graph known as overlay network.

Model and Basic Primitives

- Overlay network established by processes:



- Graph representation:

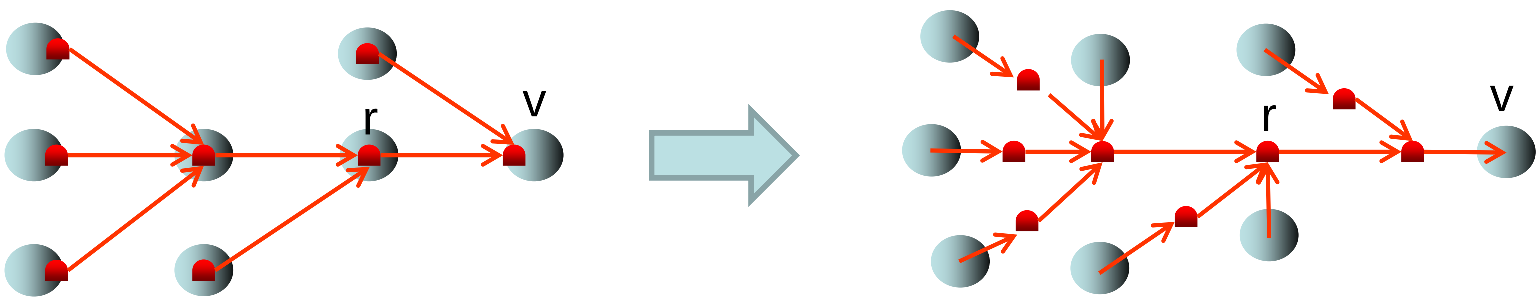


- Edge A $\rightarrow$ B means: A knows / has access to B

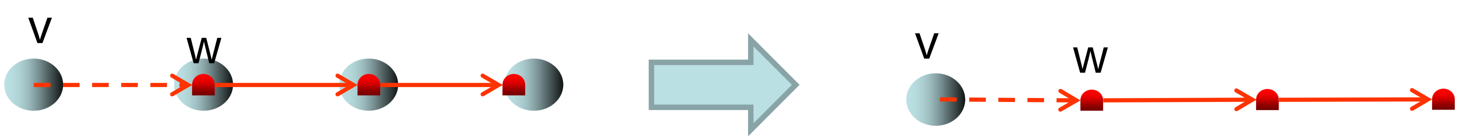
Model and Basic Primitives

Relay graph $G=(V,E_L \cup E_M)$:

- $V=R \cup P$, where R is the set of relays and P is the set of processes
- E_L (explicit edges): set of edges (v,w) where either ($v \in P$ and $w \in R$), or ($v \in R$ and $w \in R$), or ($v \in R$ and $w \in P$)

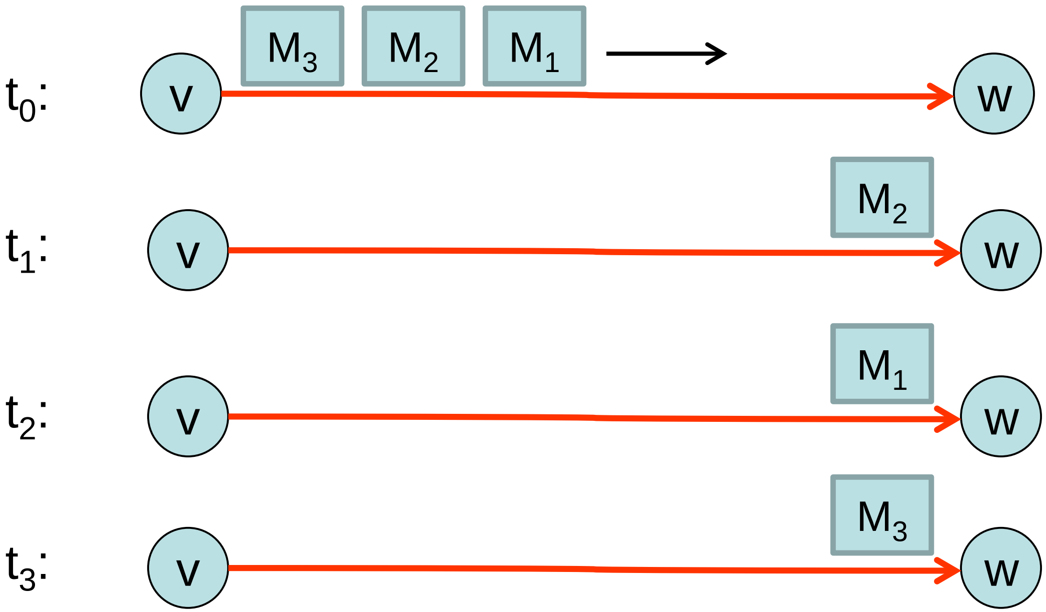


- E_M (implicit edges): set of edges (v,w) where $v \in P$ and $w \in R$, which represents a message in transit to v with a reference to relay w



Model and Basic Primitives

Asynchronous message passing



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

Problem

Problems:

- Processes continuously enter and leave the system.
- Processes might get faulty.

We need overlay networks that can handle that.

Basic approaches:

- Proactive: protect an overlay network from getting into an illegal state
- Reactive: make sure an overlay network can recover from any illegal state
  $\rightarrow$ self-stabilizing overlay networks

Overview

- Self-stabilization
- Self-stabilizing clique
- Self-stabilizing diameter 2 graphs

Self-Stabilization

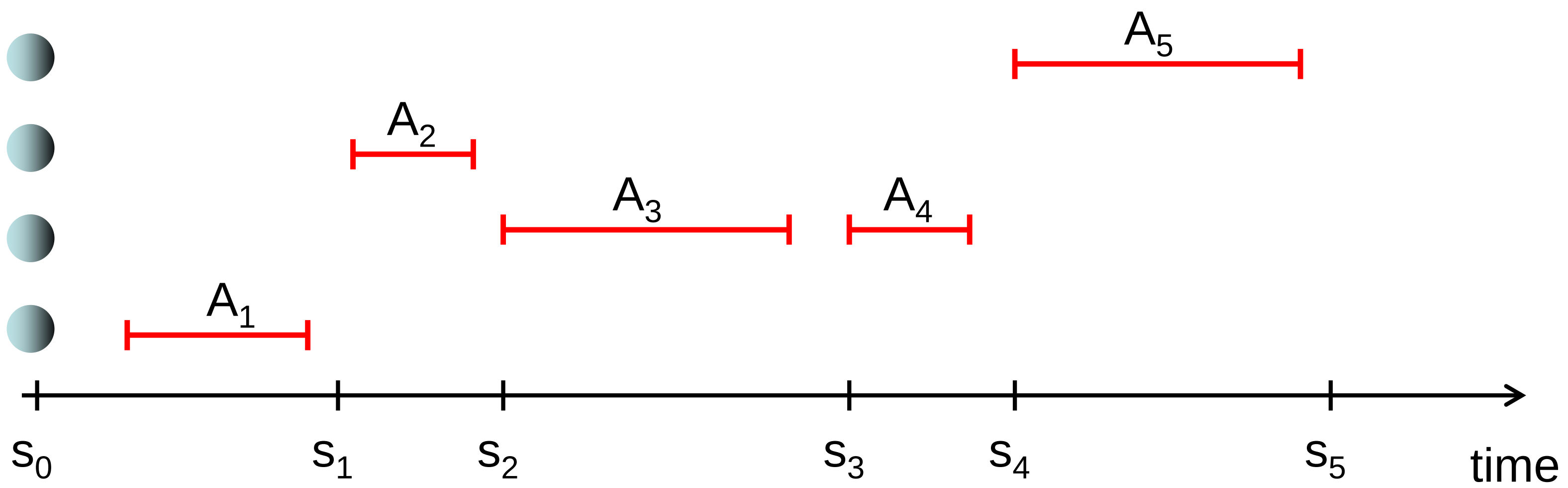
- State of a process: all data contained in it
- State of network: all messages currently in transit
- State of system: combination of the states of all processes and the state of the network

Computational problem P:
Given: initial system state S
Goal: eventually reach a system state $S' \in L_P(S)$
($L_P(S)$: set of all legal states of S w.r.t. P)

Example: Sorting problem
Given: any sequence of numbers
Goal: eventually reach a sorted sequence of numbers

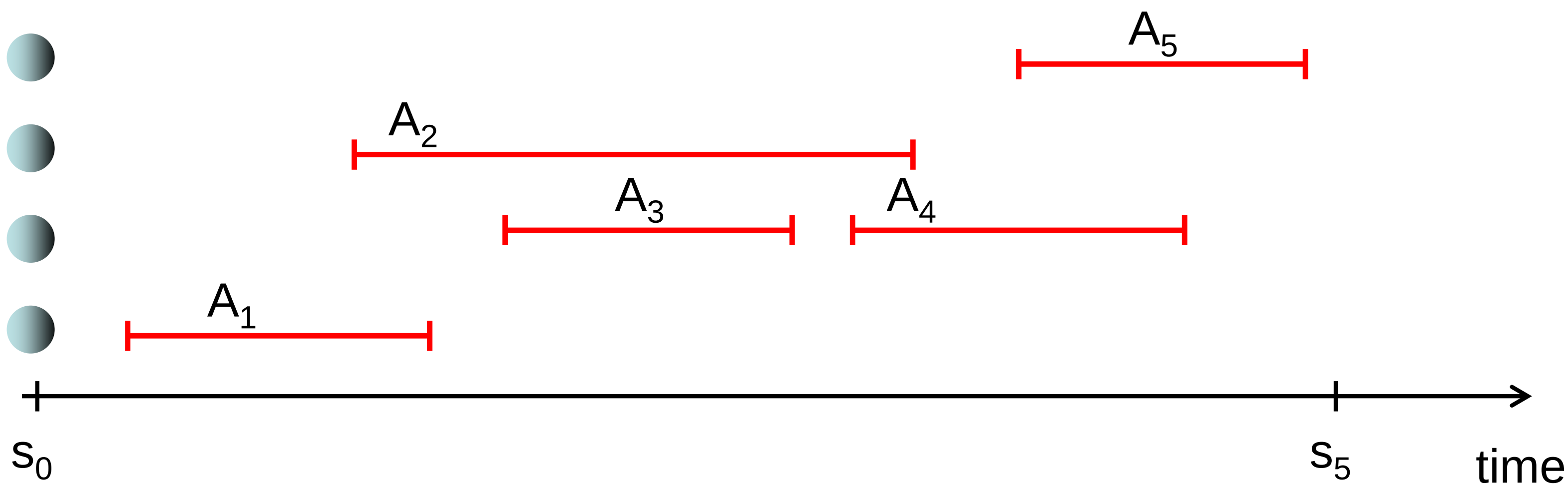
Self-Stabilization

- Simplifying assumption: in the entire system only one action can be executed at a time (globally atomic)
- Computation: potentially infinite sequence of system states $s_0, s_1, s_2, \ldots$, where state s_{i+1} is reached from s_i by executing some action
- Simple for a formal analysis, but not realistic



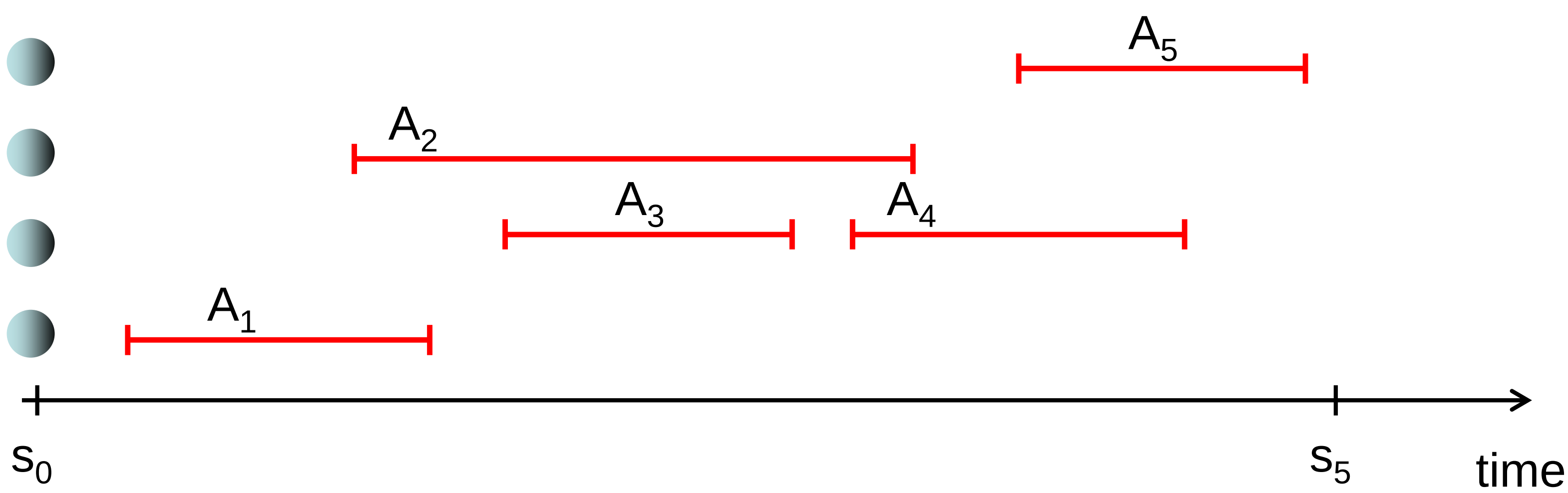
Self-Stabilization

- Simplifying assumption: in the entire system only one action can be executed at a time (globally atomic)
- Computation: potentially infinite sequence of system states $s_0, s_1, s_2, \ldots$, where state s_{i+1} is reached from s_i by executing some action
- In reality:



Self-Stabilization

More realistic assumption: in every process only one action can be executed at a time (locally atomic)



Self-Stabilization

More realistic assumption: in every process only one action can be executed at a time
(locally atomic)

Suppose that whenever a process is idle, its state does not change (i.e., there are no external changes affecting the state of a process like a physical clock). Then the following theorem holds.

Theorem 9.1: Within our process and network model, every finite locally atomic action execution can be transformed into a globally atomic action execution with the same final state.

→ All possible outcomes can be covered by globally atomic action executions.

→ „No bad globally atomic action execution“ implies „no bad locally atomic action execution“

Self-Stabilization

Theorem 9.1: Within our process and network model, every finite locally atomic action execution can be transformed into a globally atomic action execution with the same final state.

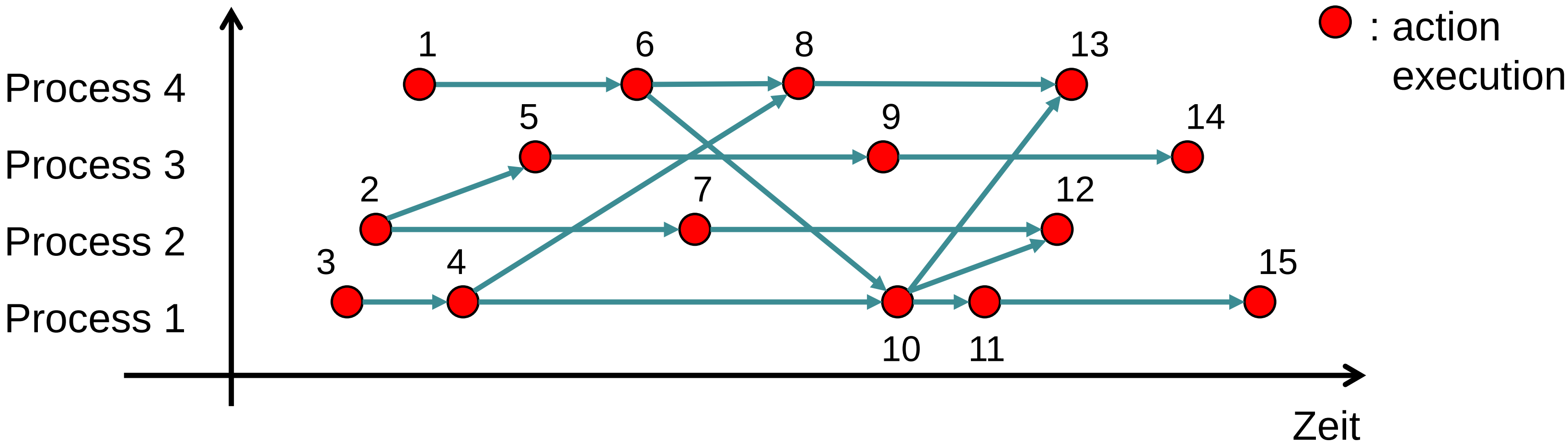
Proof:

- Recall that an action only depends on the local state and potentially the message that triggered it and can only access the local variables of the executing process.
- Consider the graph $G=(V,E)$, where V represents the set of all executed actions and (A,B) is an edge in E if and only if action A happened directly before action B in the same process or B was triggered by a message from A.
- For each edge $(A,B)\in E$ it holds that B can only start after A has started. Hence, G is acyclic (i.e., G has no directed cycle).
- Therefore, the nodes in G can be brought into a topological order (i.e., for all $(A,B)\in E$, $A<B$). It can be shown that when performing a globally atomic action execution in this order, it is a valid action execution, and the final state is the same as the one reached by the locally atomic action execution. (Proof: exercise)

Self-Stabilization

Illustration of Theorem 9.1:

- Locally atomic execution:



- numbers: topological order (= order in which actions are executed in globally atomic action execution)

Self-Stabilization

When does a process execute an action?

→ We assume fairness, i.e., no message and no action tiggered by a local predicate that is inifinitely often true has to wait infinitely long for its processing.

Action of type ⟨name⟩(⟨parameters⟩) → ⟨commands⟩:

- Triggered by local call by another action A: immediately executed (belongs to execution of A)
- Triggered by message: message is eventually processed, so corresponding action is eventually executed.

Action of type ⟨name⟩: ⟨predicate⟩ → ⟨commands⟩:

- Eventual execution only guaranteed if its predicate is true infinitely often (like the predicate true in timeout).

Self-Stabilization

Computational problem P:
Given: initial system state S
Goal: eventually reach legal system state $S' \in L_P(S)$
($L_P(S)$: set of all legal states of S w.r.t. P)

Assumptions:

- globally atomic execution
- fairness (but order of executions might be determined by an adversary)

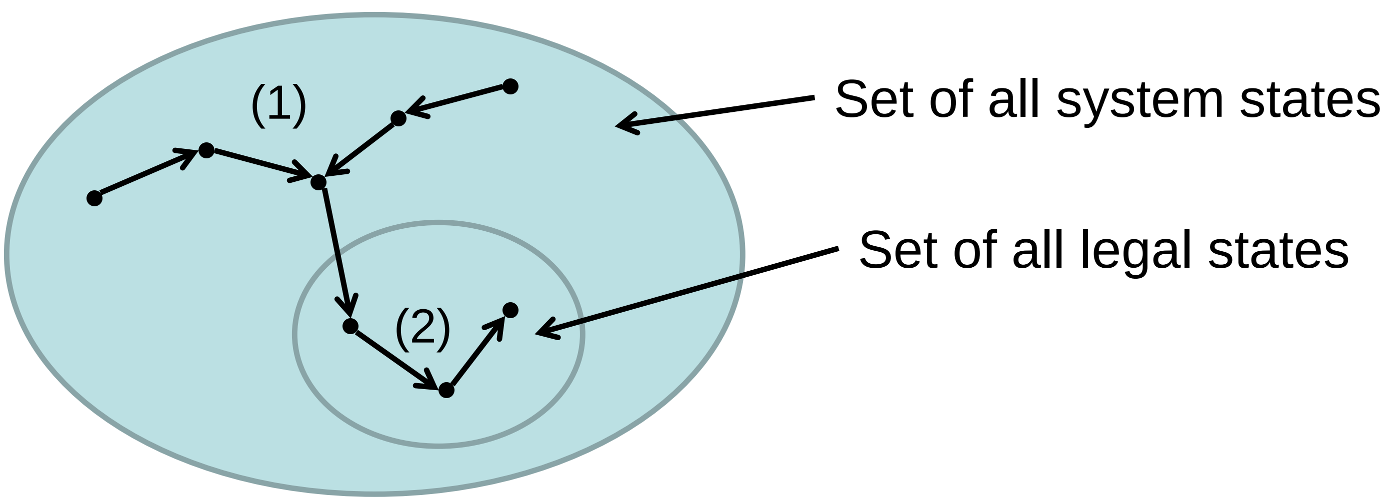
Definition 9.2: A system is self-stabilizing w.r.t. P if the following conditions hold under the assumption that the system does not undergo external changes or faults:

1. Convergence: For all initial system states S and any fair, globally atomic action execution, 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.

Self-Stabilization

Definition 9.2: 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 and any fair, globally atomic action execution, 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.



Self-Stabilization

Definition 9.2: 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 and any fair, globally atomic action execution, 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.

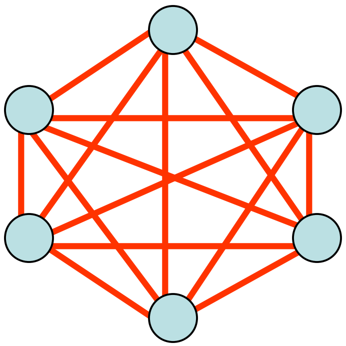
Remark: The convergence requirement has to be taken literally. ALL initial system states have to be considered, i.e., one cannot assume a well-initialized system state. Initially, the process states and the message might be corrupted in an arbitrary way. This complicates the design of self-stabilizing systems.

Overview

- Self-stabilization
- Self-stabilizing clique
- Self-stabilizing diameter 2 graphs

Self-stabilizing Clique

Legal state:



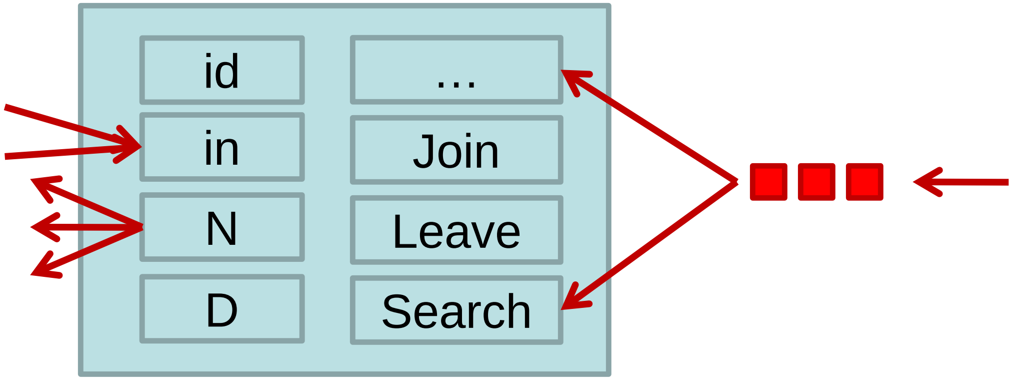
Operations:

- Join(v): add process v to clique
- Leave(): remove itself from clique
- Search(id): search for process with ID id

Clique

Variables within v:

- id: ID of v
- in: incoming relay of v
- $N \subseteq V$: current neighbor set of v (represented by a set of outgoing relays)
- D: set of to-be-delegated neighbors of v (due to indirect connections, which we do not want to have)



Clique

Variables within v:

- id: ID of v
- in: incoming relay of v
- $N \subseteq V$: current neighbor set of v (represented by a set of outgoing relays)
- D: set of to-be-delegated neighbors of v (due to indirect connections, which we do not want to have)

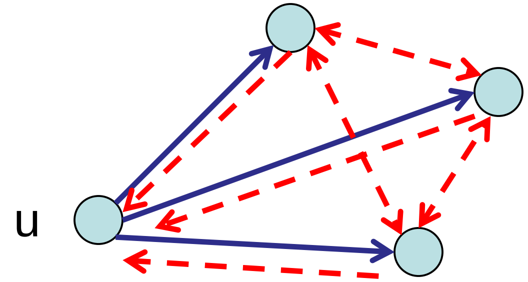
Legal state:

- For any process v let the (direct) neighborhood $\Gamma(v)$ of v be the set of all direct connections in v.N. (i.e., for any relay $r \in N$, there is a direct link from r to the r.sink).
- A state is legal if and only if $\bigcup_{v \in V} \Gamma(v)$ forms a clique.

Clique

Naive idea for building a clique:

Every process u continuously introduces itself and all of its neighbors to all of its neighbors.

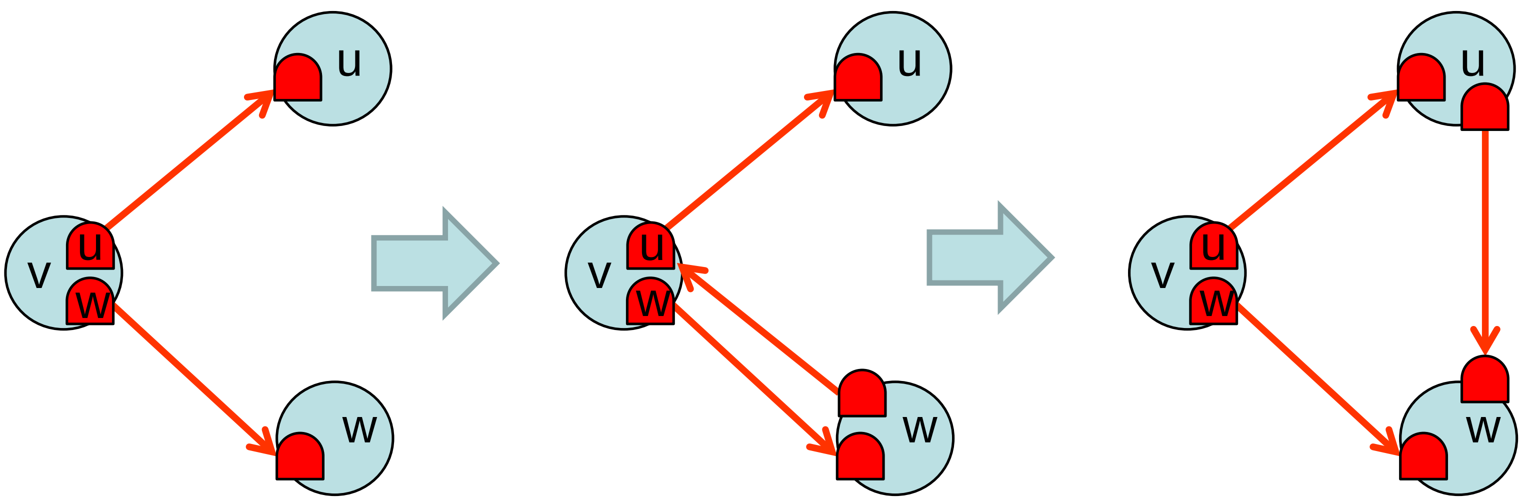


Problem: very high work in legal state!

Clique

Better idea:

Continuously, every process v selects a random pair of (relays to) processes u,w∈v.N or itself and safely introduces u to w. w will then safely introduce itself to u.

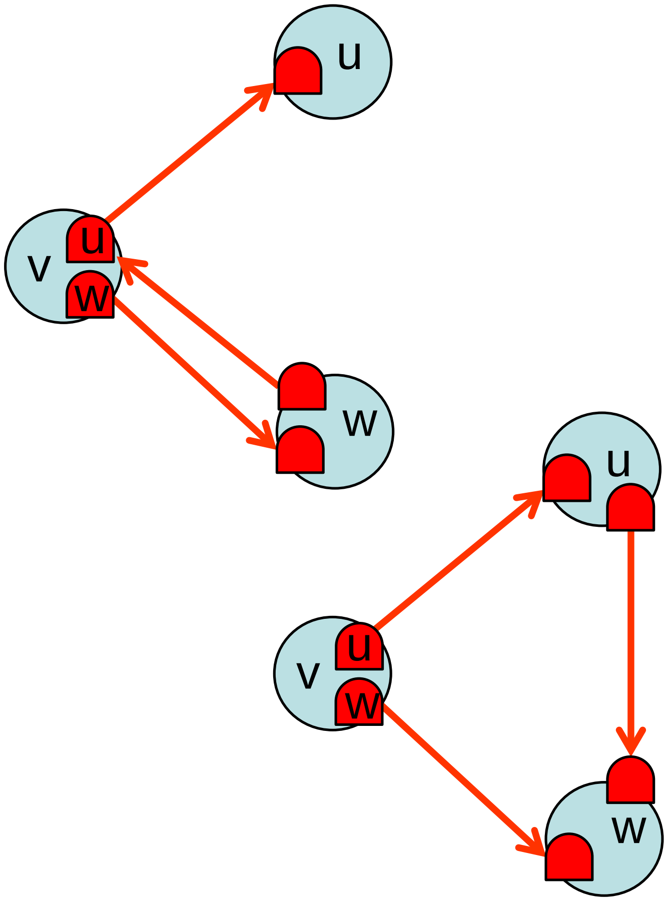


Build-Clique Protokoll

timeout: true →
  for all v∈N with v redundant or not v.direct do
    N:=N\{v}; D:=D∪{v}
  u:=random(N)
  w:=random(N∪{in})
  w←ask-for-intro(u)
  for all v∈D with not v.incoming do
    v←introduce(in)
    delete v

ask-for-intro(u) →
  { u is newly created, so no incoming links }
  if u.sink≠in then
    u←introduce(in)
  delete u

introduce(w) →
  { w is newly created, so no incoming links }
  if w.sink≠in and w is not redundant in N then
    if w.direct then N:=N∪{w}
            else D:=D∪{w}
  else
    delete w



Clique

Theorem 9.3 (Convergence): For any weakly connected relay graph, the Build-Clique protocol eventually reaches a legal state.

Proof:

- Certainly, the Build-Clique protocol preserves weak connectivity.
- Also, eventually we reach a state in which for every node v, $v.D=\emptyset$ and $v.N=\Gamma(v)$, and every introduce(w)-call still in transit will only establish a direct connection. Moreover, once this is reached, we will stay in such a state (Proof: exercise.)
- It remains to show that as long as $\bigcup_{v\in V} \Gamma(v)$ does not form a clique, the neighborhood of at least one node will eventually increase.
- Let u be a node whose neighborhood is not yet complete, and let w be a node that is not yet in its neighborhood.
- Since the graph is weakly connected, there is a (not necessarily directed) path from u to w.
- Let this path move along the nodes $u=v_0,v_1,\ldots,v_k=w$, and let this be a shortest possible path from u to w.
- If $k=1$, then w already knows u, so the probability is >0 that w will introduce itself to u (which happens if in timeout, $w=in$).
- If $k=2$, then we assume w.l.o.g. for $v:=v_1$ that v knows u and w (if not, this will eventually happen like in the case $k=1$). Then again the probability is >0 that v will introduce w to u.
- If $k>2$, then we reset w to v_2 so that we are back to the case $k=2$.

Clique

Theorem 9.4 (Closure): Once the processes have reached a legal state, they stay at a legal state.

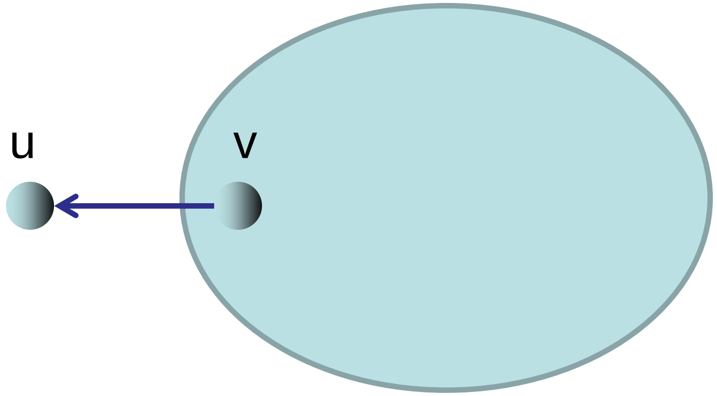
Proof:
Once a relay with a direct connection has been added to N, it is never removed.

Adversarial processes:
The Build-Clique protocol works for any number of adversarial processes (if we call a state to be legal once the set of honest processes forms a clique), as long as the graph of the honest processes is initially weakly connected.

Clique

Join(u):

- Suppose that some process v that is already in the system executes Join(u), where u is a relay to some process that wants to join the clique.
- Then v simply adds u to N.
- The Build-Clique protocol will then eventually integrate u into the clique.



Clique

Theorem 9.5: If all processes operate in synchronous rounds and in each round every process does a random introduction, then it takes at most $O(n \log n)$ rounds until a new process u is fully integrated into a clique of n processes.

Proof:

Number of rounds until everybody knows u:

- Suppose that at the beginning of the given round, u is already known by a set S of d out of n processes.
- For any $v \in S$,

$$\Pr[v \text{ introduces } u \text{ to some } w \notin S] = 1/(n+1) \cdot (n-d)/n$$

$$\Pr[v \text{ does not introduce } u \text{ to some } w \notin S] = 1-1/(n+1) \cdot (n-d)/n$$

$$\Pr[\text{no } v \in S \text{ introduces } u \text{ to some } w \notin S] = (1-1/(n+1) \cdot (n-d)/n)^d$$

$$\leq 1 - d/(n+1) \cdot (n-d)/n) + \binom{d}{2} \cdot (1/(n+1) \cdot (n-d)/n)^2$$

$$\leq 1 - d/(2(n+1)) \cdot (n-d)/n$$

Clique

Theorem 9.5: If all processes operate in synchronous rounds and in each round every process does a random introduction, then it takes at most $O(n \log n)$ rounds until a new process u is fully integrated into a clique of n processes.

Proof:

Number of rounds until everybody knows u (continued):

- Hence,

  $Pr[u$ is introduced to at least one $w \notin S] \geq d(n-d)/(2n(n+1))$
- Let $p:=Pr[u$ is introduced to at least one $w \notin S]$. Then it holds (exercise):

  $E[$#rounds until intro to some $w \notin S] = 1/p \leq 2n(n+1)/(d(n-d))$
- Therefore,

  $E[$#rounds until everybody knows $u]$

  $\leq \Sigma_{d=1}^{n-1} E[$#rounds until intro to some $w \notin S]$

  $= \Sigma_{d=1}^{n-1} 1/p = O(\Sigma_{i=1}^{n/2} n/i) = O(n \ln n)$

Clique

Theorem 9.5: If all processes operate in synchronous rounds and in each round every process does a random introduction, then it takes at most $O(n \log n)$ rounds until a new process u is fully integrated into a clique of n processes.

Proof:

Number of rounds until u knows everybody: exercise

Speeding up the protocol:

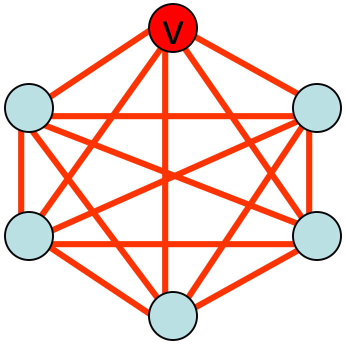
- Process u gives v feedback whether v introduced it to a new process or not.
- If so, this raises v´s probability to make another proposal to u, otherwise it decreases v´s probability (similar to contention resolution).

Clique

Leave(): we assume that a process v can only initiate Leave for itself

Simplest solution: process v just leaves the system. Since the clique has a very high expansion, there shouldn‘t be any danger for the connectivity of the rest.

Problem: a clique may not have been reached yet!



Solution idea:

- v does not let any new process connect to it.
- v tries to reverse all existing connections to it so that it does not have incoming connections any more.
- Once v does not have any incoming connections, it tries to get rid of all outgoing connections except one (the so-called anchor), and once it has succeeded with that, it leaves.

Clique

Variables needed for Leave operation:

- leaving: Boolean variable that indicates if the process wants to leave the system. Initially, it is set to false.
- a-out: relay to an anchor process, which is used by leaving processes. The variable can only be used once leaving is true, and initially it is set to $\perp$.
- a-in: incoming relay from current anchor. Like a-out, it can only be used once leaving is true, and initially it is set to $\perp$.
- D: set of relays that can be delegated away (once they have no incoming connections any more). Initially, it is set to $\emptyset$.

Leave operation:

Leave() $\rightarrow$
  leaving:=true

The rest is handled by an extension of Build-Clique.

Clique

Solution to „v does not let any new process connect to it“:

```
timeout: true →
    for all v∈N with v redundant or
                          not v.direct do
       N:=N\{v}; D:=D∪{v}
    if not leaving then
       u:=random(N)
       w:=random(N∪{in})
       w←ask-for-intro(u)
       for all v∈D with not v.incoming do
          v←introduce(in)
          delete v

introduce(w) →
    if w.sink≠in and w is not redundant in N then
       if w.direct then N:=N∪{w}
                   else D:=D∪{w}
    else
       delete w
```

```
ask-for-intro(u) →
if u.sink≠in then
   if not leaving then
      u←introduce(in)
      delete u
   else
      { leaving: no new incoming
        link, instead keep link for
        reversal so that incoming
        links removed }
      N:=N∪{u}
else delete u
```

Clique

Extension to „v tries to reverse all existing connections to it so that it does not have incoming connections any more“:

```
timeout: true →
    beginning as before
    else { leaving=true }
      for all v∈N do
        N:=N\{v}; D:=D∪{v}
      if not a-out.direct then
        D:=D∪{a-out}; a-out:=⊥
      for all v∈D with not v.incoming do
        { get rid of links to itself }
        v←ask-to-reverse(in)
        delete v
      if a-out≠⊥ and not a-in.incoming then
        { once no incoming anchor link,
          probe anchor again }
        a-out←ask-to-reverse(a-in)

ask-for-intro(u) and introduce(w)
    as before
```

```
ask-to-reverse(out) →
   for all v∈N with v.sink=out.sink do
      N:=N\{v}; D:=D∪{v}
   if leaving then
      if a-out=⊥ then
         out←ask-to-reverse(in)
      else
         if out.sink=a-out.sink then
            D:=D∪{a-out}; a-out:=⊥
         else
            out←reverse(a-out)
   else
      out←reverse(in)
   delete out
```

Clique

Solution to „once v does not have any incoming connections, it tries to get rid of all outgoing connections except one (the so-called anchor)“:

```
ask-to-reverse(out) →
    for all v∈N with v.sink=out.sink do
        N:=N\{v}; D:=D∪{v}
    if leaving then
        if a-out=⊥ then
            out←ask-to-reverse(in)
        else
            if out.sink=a-out.sink then
                D:=D∪{a-out}; a-out:=⊥
            else
                out←reverse(a-out)
    else
        out←reverse(in)
    delete out
```

```
reverse(out) →
    if not leaving then
        N:=N∪{out}
    else
        if a-out=⊥ then
            if out.direct then
                a-out:=out
            else
                out←ask-to-reverse(in)
                delete out
        else
            D:=D∪{out}
```

Clique

Solution to „once v does not have any incoming connections, it tries to get rid of all outgoing connections except one (the so-called anchor)":

```
timeout: true →
    beginning as before
    else { leaving=true }
        if N=∅ and D=∅ and not in.incoming and not a-in.incoming and
        not a-out.incoming then
            { only a-out non-empty, so only one link left, which means there
              is no danger of disconnecting graph by removing process }
            stop
        for all v∈N do
            N:=N\{v}; D:=D∪{v}
        if not a-out.direct then
            D:=D∪{a-out}; a-out:=⊥
        for all v∈D with not v.incoming do
            v←ask-to-reverse(in)
            delete v
        if a-out≠⊥ and not a-in.incoming then
            a-out←ask-to-reverse(a-in)
```

Clique

Search(sid):
if id=sid then „success"
if ∃w∈N: w.id=sid then w←Search(sid)
else „failure"

Problem: The convergence to a full clique is slow at the end because once a process knows almost everybody, the probability is small that it still learns about new processes, which may cause search failures.

Solution: As long as the destination has not been found, the message is forwarded to a random neighbor, but at most d times for a fixed, constant d.

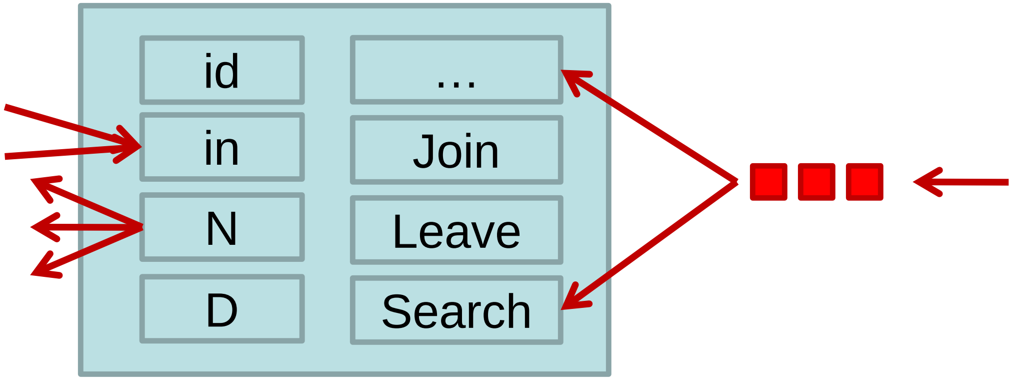
Overview

- Self-stabilization
- Self-stabilizing clique
- Self-stabilizing diameter 2 graphs

Diameter 2 Graph

Variables within v:

- id: ID of v
- in: incoming relay of v
- $N \subseteq V$: current neighbor set of v (represented by a set of outgoing relays)
- D: set of to-be-delegated neighbors of v (due to indirect connections, which we do not want to have)



Diameter 2 Graph

Theorem 9.6: Every graph of size n and diameter D must have a degree of at least $\lfloor n^{1/D} \rfloor$.

Proof: exercise

Hence, if we want to have a diameter 2 graph of size n, its degree must be at least $\sqrt{n}-1$.

Our goal: design a protocol for a self-stabilizing diameter 2 graph with degree $O(\sqrt{n})$. A useful lemma to achieve that is the following.

Lemma 9.7 (Birthday paradox): Suppose that we select k out of n balls uniformly and independently at random, where $k=o(n)$. Then the expected number of balls that is selected at least twice is

$$(1\pm o(1))\cdot k(k-1)/(2n).$$

Diameter 2 Graph

Lemma 9.7 (Birthday paradox): Suppose that we select k out of n balls uniformly and independently at random, where $k=o(n)$. Then the expected number of balls that is selected at least twice is

$$(1\pm o(1))\cdot k(k-1)/(2n).$$

Proof:

- Consider some fixed ball B.
- Pr[B not selected] = $(1-1/n)^k$
- Pr[B selected once] = $k\cdot(1/n)\cdot(1-1/n)^{k-1}$
- Hence,

  Pr[B selected at least twice]

  $= 1 - (1-1/n)^k - k\cdot(1/n)\cdot(1-1/n)^{k-1}$

  $= 1 - (1-k/n+\binom{k}{2}(1/n)^2 \pm O((k/n)^3)) - (k/n)(1-(k-1)/n \pm O((k/n)^2))$

  $= (1\pm o(1))\cdot k(k-1)/(2n^2)$
- Thus,

  E[#balls selected at least twice] = $(1\pm o(1))\cdot k(k-1)/(2n)$

Diameter 2 Graph

Lemma 9.7 (Birthday paradox): Suppose that we select k out of n balls uniformly and independently at random, where $k=o(n)$. Then the expected number of balls that is selected at least twice is

$$(1\pm o(1))\cdot k(k-1)/(2n).$$

Basic approach:

- Keep sampling neighbors at a 2-hop distance uniformly at random.
- Record the number of samplings between events where a node has been selected twice. If this happens too often (compared to what the birthday paradox predicts for a targeted degree of $\sim\sqrt{n}$), reduce the degree. Otherwise, slowly increase the degree over time.

Conjecture: the approach eventually arrives at a random diameter 2 graph of degree $O(\sqrt{n})$.

Diameter 2 Graph

How should the degree be balanced?

- Let $m=|N|$.
- v_i and v_j are a twin: $v_i.sink=v_j.sink$
- Pr[there is a twin in N] $\le \binom{m}{2} 1/n = m(m-1)/(2n)$
- N is small: $m \le \sqrt{n}/2$
- Pr[there is a twin in a small N] $\le 1/8$

- Pr[there is no twin in N] $\le n(n-1)\ldots(n-m+1)/n^m$

$$= (n/n) \cdot (n-1)/n \cdot (n-2)/n \cdot \ldots \cdot (n-m+1)/n$$
$$= 1 \cdot (1-1/n) \cdot (1-2/n) \cdot \ldots \cdot (1-(m-1)/n)$$
$$\le e^0 \cdot e^{-1/n} \cdot e^{-2/n} \cdot \ldots \cdot e^{-(m-1)/n} = e^{-m(m-1)/(2n)}$$

- N is large: $m \ge 3\sqrt{n}$
- Pr[there is no twin in a large N] $\le 1/8$

Concrete approach:

- Organize N as FIFO queue
- For each dequeued node v of N:
  - if v belongs to twin, delete v (reduces |N|)
  - else if N has a twin then replace v by a new random node (preserves |N|)
  - else if N has no twin then add a new random node to N (increases |N|)

Build-D2G Protokoll

```
timeout: true →
    for all v∈N with not v.direct do
        N:=N\{v}; D:=D∪{v}
    v:=dequeue(N)
    if v is a twin then delete v
    else
        if N has a twin then
            D:=D∪{v} { replace v by random node }
        else
            v←ask-for-intro(in)
            enqueue(N,v)
    for all v∈D with not v.incoming do
        v←ask-for-intro(in)
        delete v

ask-for-intro(u) →
    if u.sink≠in then
        w:=random(N)
        u←introduce(w)
    delete u

introduce(w) →
    if w.sink≠in then
        w←ask-for-connect(in)
    delete w

ask-for-connect(u) →
    if u.sink≠in then
        u←connect(in)
    delete u

connect(w) →
    if w.sink≠in then
        N:=N∪{w}
    else
        delete w
```

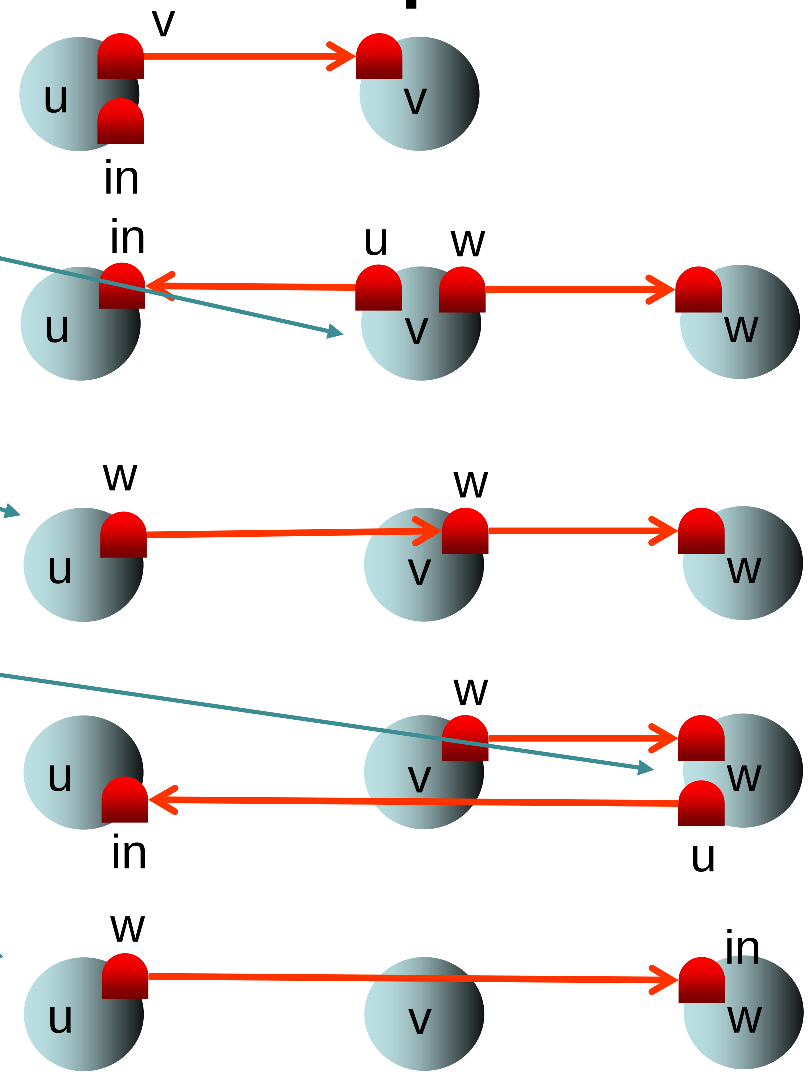
Diameter 2 Graph

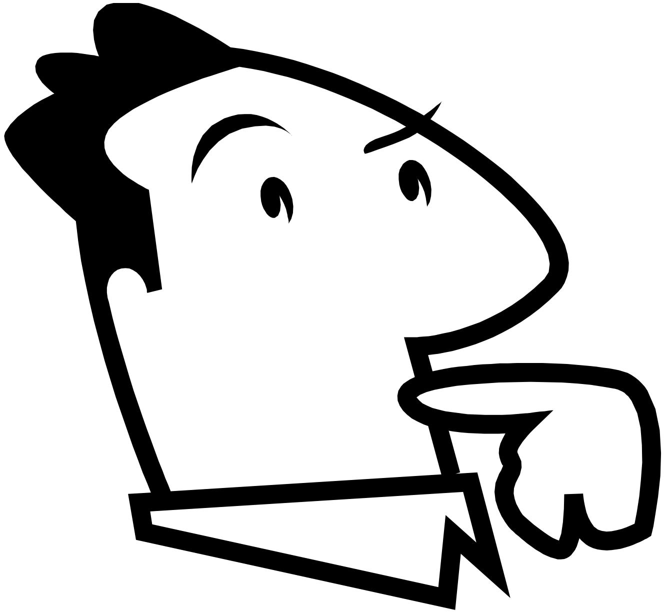
```
ask-for-intro(u) →
    if u.sink≠in then
        w:=random(N)
        u←introduce(w)
    delete u

introduce(w) →
    if w.sink≠in then
        w←ask-for-connect(in)
    delete w

ask-for-connect(u) →
    if u.sink≠in then
        u←connect(in)
    delete u

connect(w) →
    if w.sink≠in then
        N:=N∪{w}
    else
        delete w
```





Questions?