Advanced Distributed Algorithms and Data Structures

Chapter 3: Link Primitives

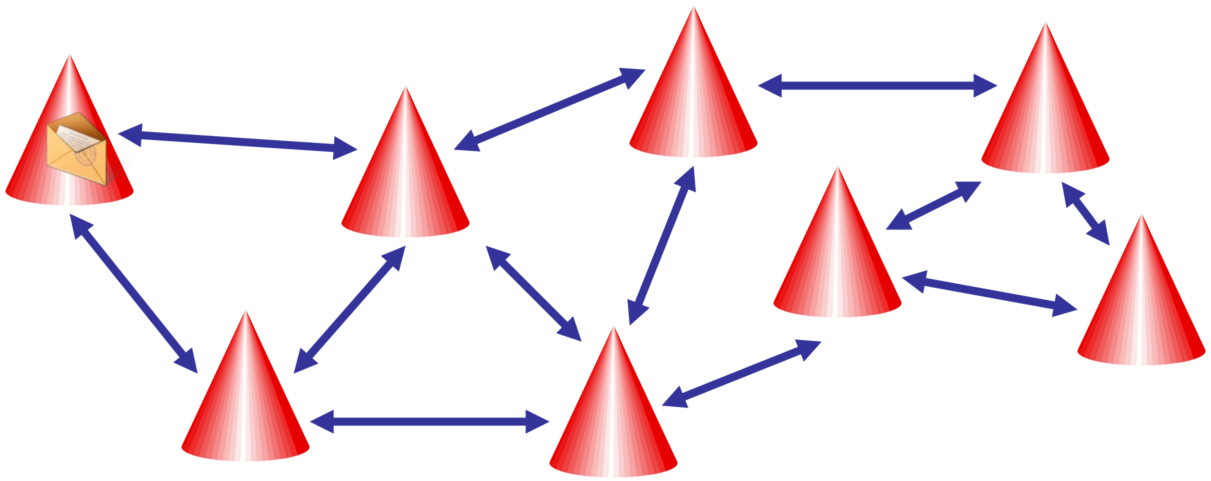
Christian Scheideler
Institut für Informatik
Universität Paderborn

Overview

- Model and basic primitives
- Universality
- Relays
- Joining and Leaving

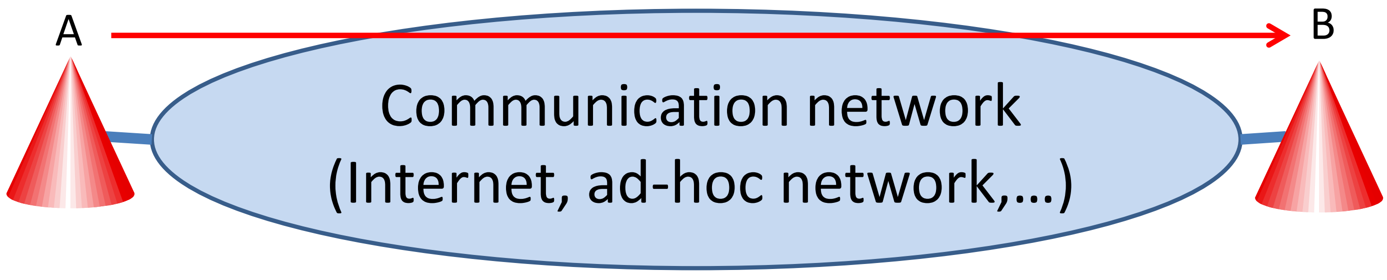
Process Model

- Processes can connect to each other



- Connections over some shared medium: overlay network

Model and Basic Primitives



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

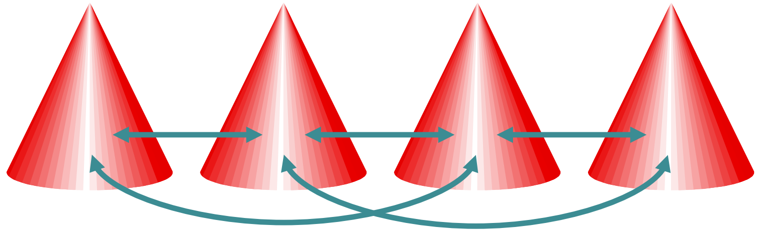
High-level view:

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

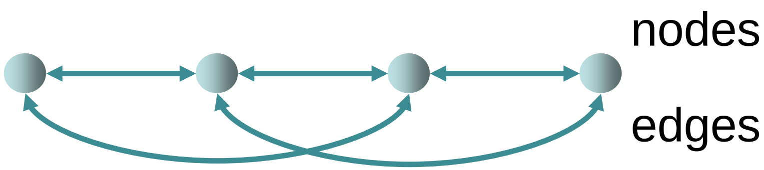
Set of all overlay edges: overlay network.

Model and Basic Primitives

- Overlay network established by processes:



- Graph representation:



- Edge A → B means: A knows / has access to B

Model and Basic Primitives

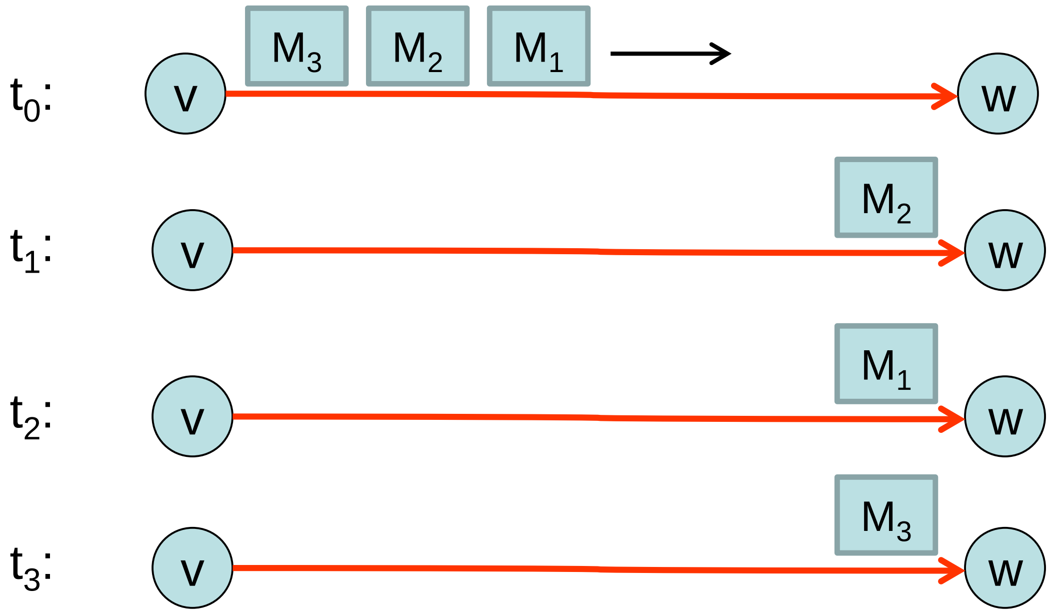
- Edge set E_L: set of pairs (v,w) where v knows w (explicit edges).

- 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.

Model and Basic Primitives

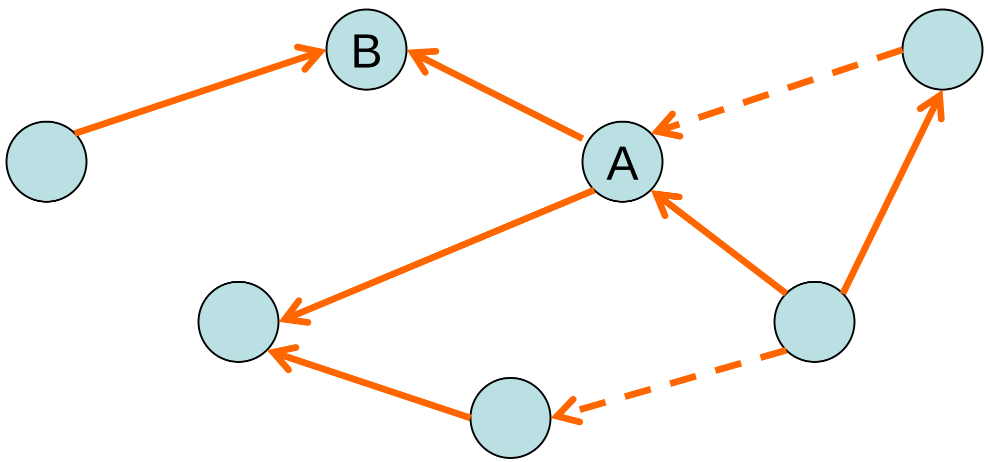
Asynchronous message passing



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

Model and Basic Primitives

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

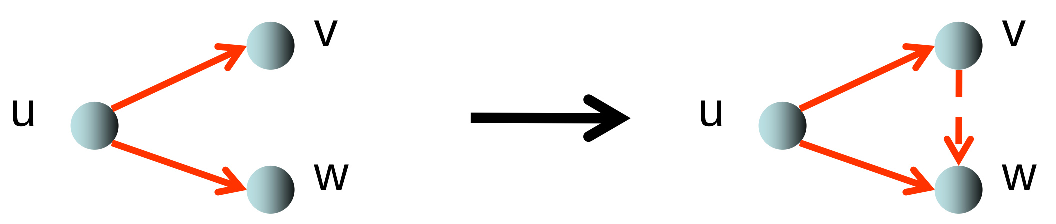


Fundamental rule: never just „throw away“ a reference!

Model and Basic Primitives

Admissible primitives for weak connectivity:

- Introduction:

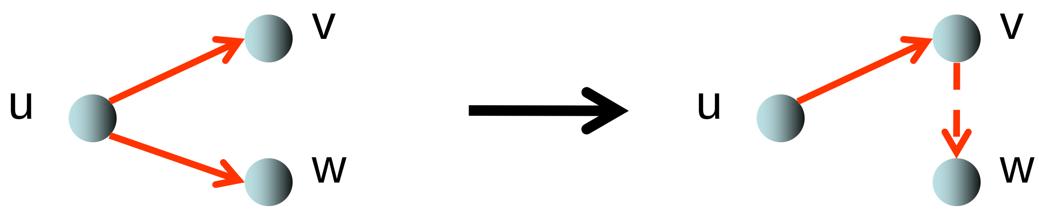


  u introduces w to v by sending a message to v containing a reference to w
- special case: u introduces itself to v

Model and Basic Primitives

Admissible primitives for weak connectivity:

- Delegation:



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

- Fusion:

Model and Basic Primitives

Admissible primitives for weak connectivity:

- Reversal:

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

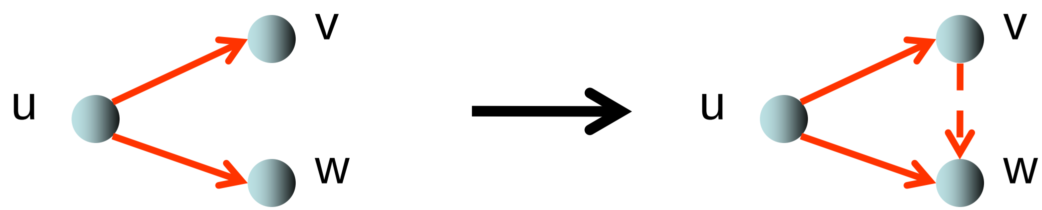
Remarks:

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

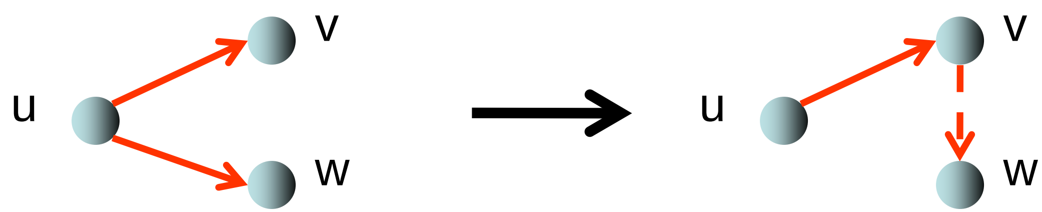
Universality

Theorem 3.1: The 3 primitives below are weakly universal, i.e., they can be used to transform any weakly connected graph G=(V,E) into any strongly connected graph G´=(V,E´).

introduction:



delegation:



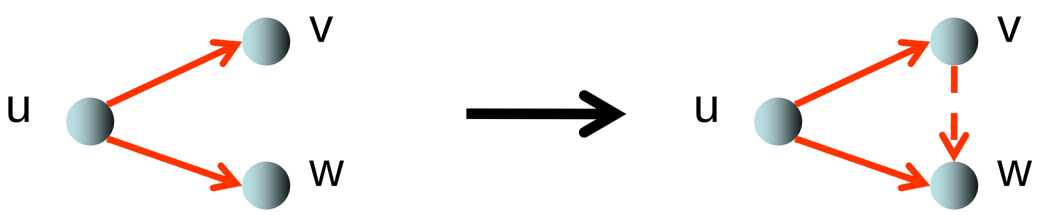
fusion:

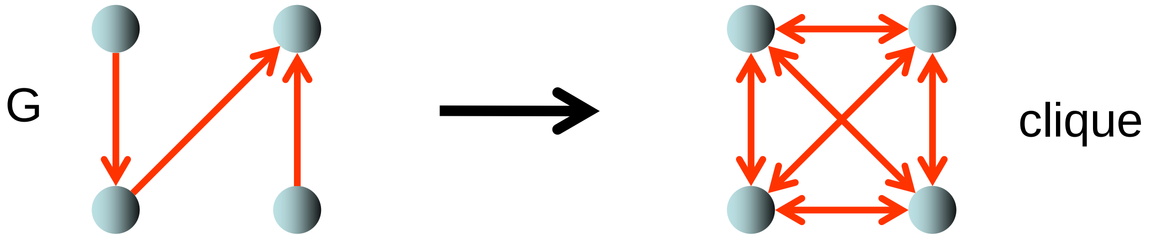
Universality

Proof: consists of two parts

1. Using the introduction primitive, one can get from any weakly connected graph G=(V,E) to the clique.

introduction:

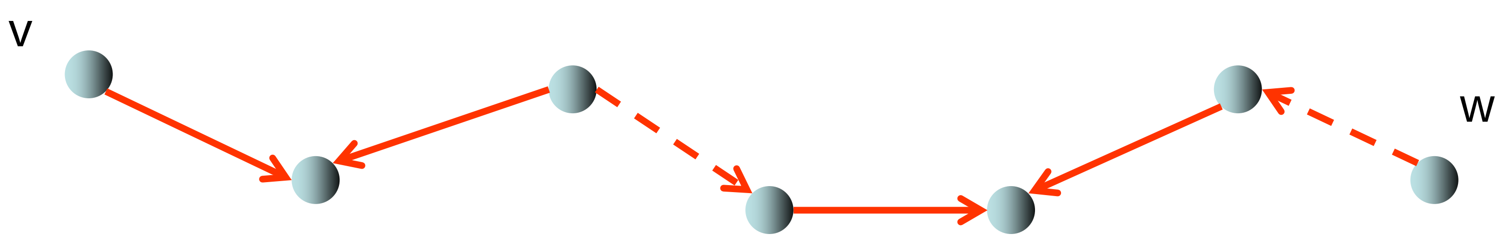




Universality

How does that work?

Consider any two nodes v and w. Since G is weakly connected, there is a path from v to w.



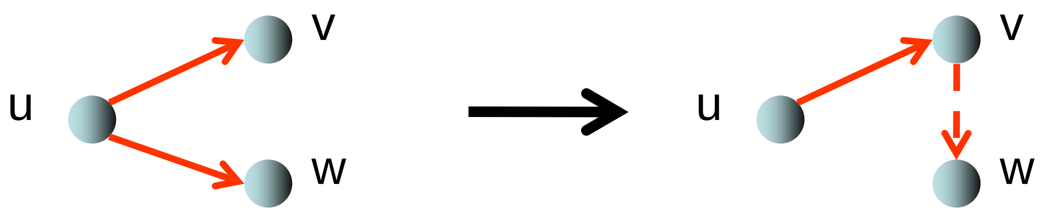
Exercise: If in each round every node introduces all of its neighbors and itself to all of its neighbors, then just O(log n) rounds are needed till the clique is reached.

Universality

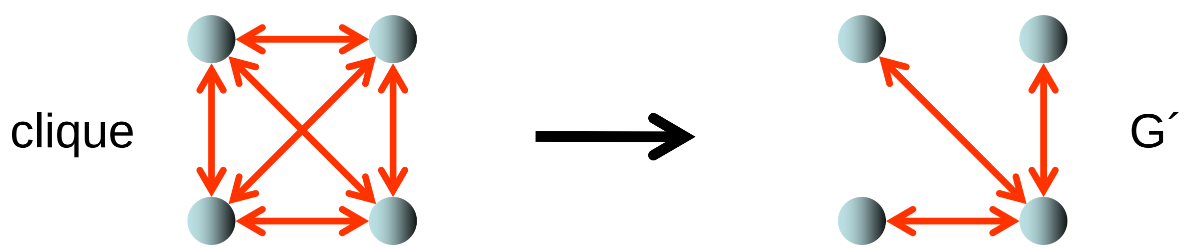
Proof:

2. Using the delegation and fusion primitives, one can get from the clique to G´=(V,E´).

delegation:



fusion:

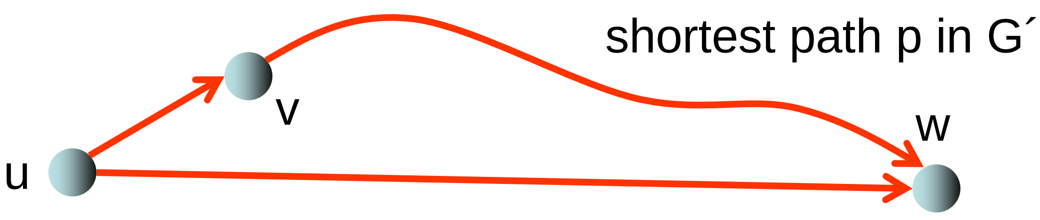


Universality

Proof: (details)

2. Suppose that G=(V,E) is a clique. Then G can be transformed into G´=(V,E´) in the following way without ever dropping edges of G´.

- Let (u,w) be an arbitrary edge that needs to be removed because it is not in E´. Since G´=(V,E´) is strongly connected, there is a directed path from u to w in G´. Let p be a shortest such path and let v be the next node along this path.

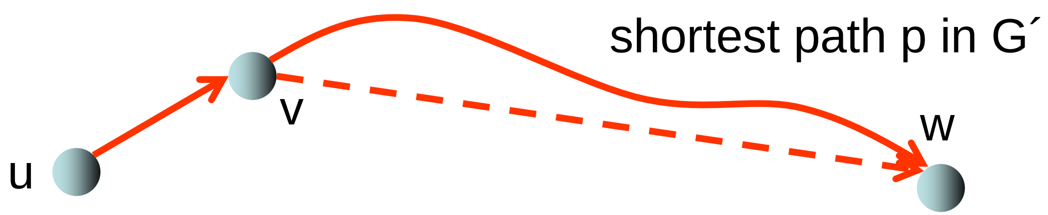


Universality

Proof: (details)

2. Suppose that G=(V,E) is a clique. Then G can be transformed into G´=(V,E´) in the following way without ever dropping edges of G´.

- Let (u,w) be an arbitrary edge that needs to be removed because it is not in E´. Since G´=(V,E´) is strongly connected, there is a directed path from u to w in G´. Let p be a shortest such path and let v be the next node along this path.
- Then node u delegates (u,w) to v, i.e., (u,w) is transformed into (v,w).

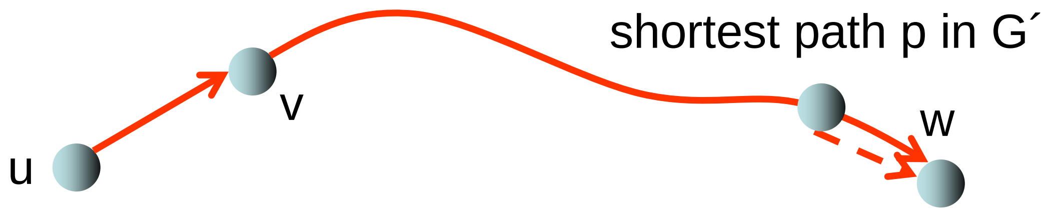


Universality

Proof: (details)

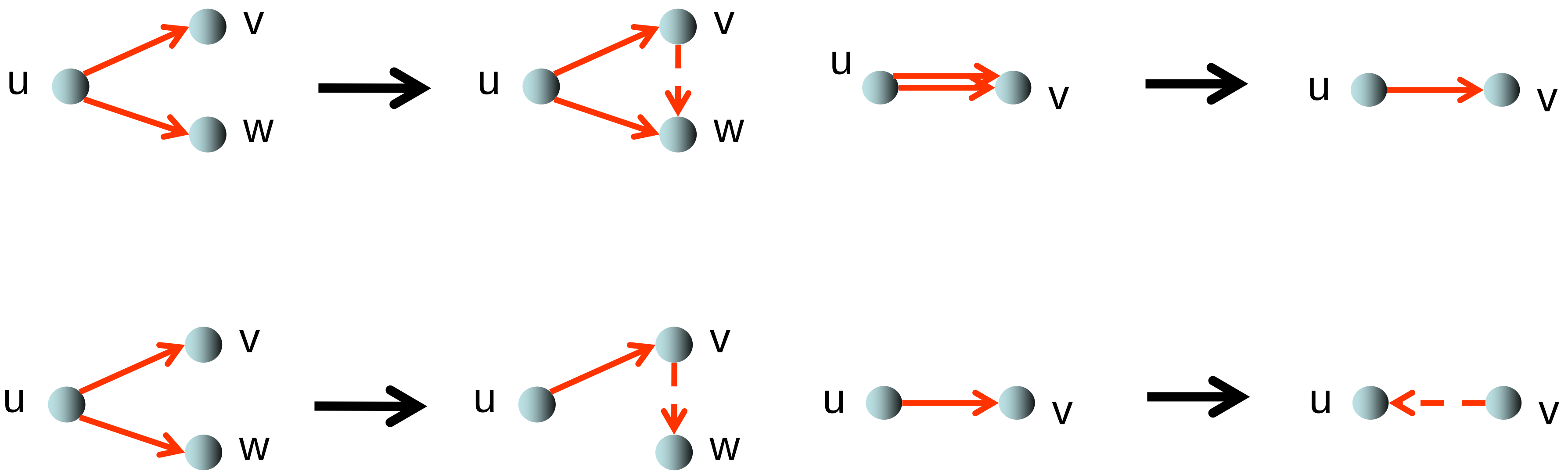
2. Suppose that G=(V,E) is a clique. Then G can be transformed into G´=(V,E´) in the following way without ever dropping edges of G´.

- Let (u,w) be an arbitrary edge that needs to be removed because it is not in E´. Since G´=(V,E´) is strongly connected, there is a directed path from u to w in G´. Let p be a shortest such path and let v be the next node along this path.
- Then node u delegates (u,w) to v, i.e., (u,w) is transformed into (v,w).
- After at most n-2 further delegations along p, the edge can be fused with an edge in G´. Doing that for all $(u,w) \notin E´$, we get G´.



Universality

Theorem 3.2: The 4 primitives below 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´).

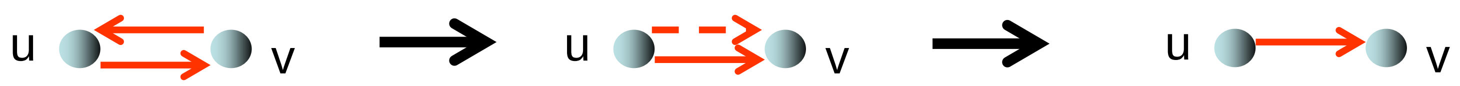


Universality

Theorem 3.2: The 4 primitives below are universal in a sense that one can get from any weakly connected graph $G=(V,E)$ to any other weakly connected graph $G'=(V,E')$.

Proof:

- Let $G''=(V,E'')$ be the bidirected version of G', i.e., for all $(u,v)\in E'$, $(u,v)\in E''$ and $(v,u)\in E''$.
- Certainly, G'' is strongly connected. (Why?)
- Theorem 3.1: we can get from G to G''.
- From G'' to G': use reversal and fusion primitive to remove wrong directions:



Universality

Remark:

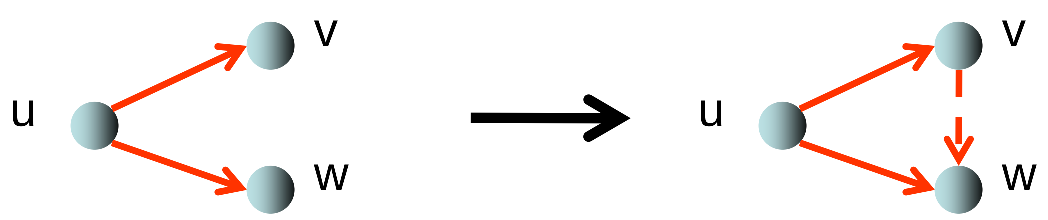
- Each of four primitives is necessary for universality.
  - Introduction: only one that generates new edge
  - Fusion: only one that removes edge
  - Delegation: only one that moves edge away
  - Reversal: only one that makes nodes unreachable
- Theorems 3.1 and 3.2 only show that in principle it is possible to get from any weakly connected graph to any other weakly resp. strongly connected graph. Designing distributed algorithms for specific topologies can be very challenging.

Overview

- Model and basic primitives
- Universality
- Relays
- Joining and Leaving

Relays

Recall the definition of the introduction primitive:



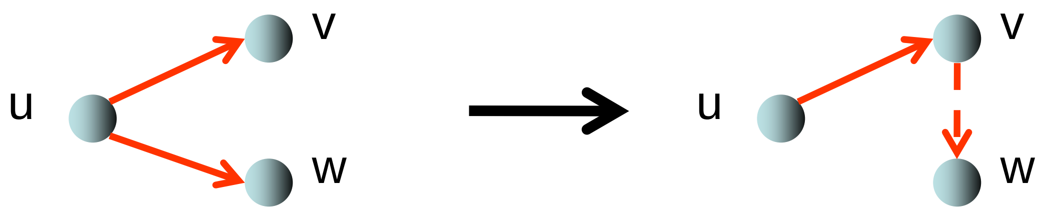
u introduces w to v by sending a message to v containing a reference to w

This violates w´s right to decide who shall connect to it.

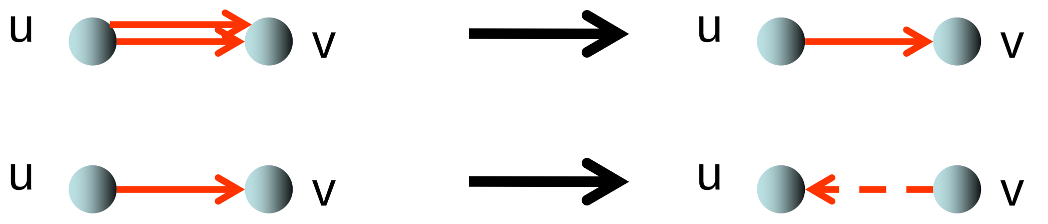
(But self-introduction is fine.)

Relays

Same problem with delegation:

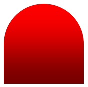


But fusion and reversal are fine:

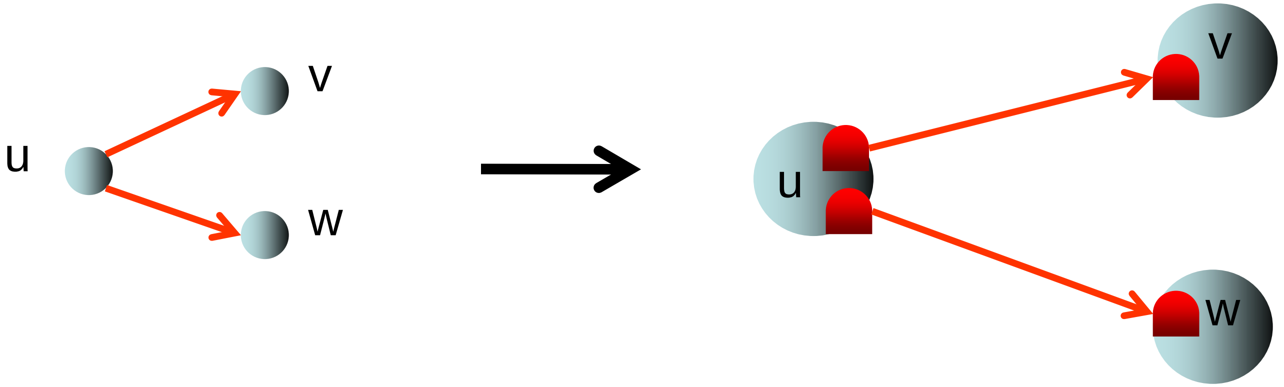


Relays

How to obtain safe forms of introduction and delegation?

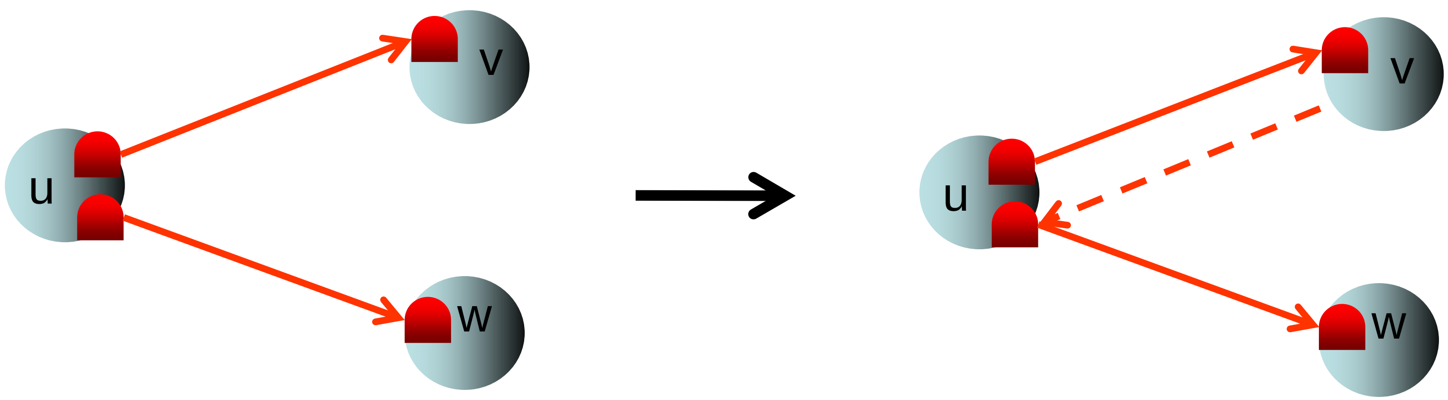
→Use the concept of relays ()

Extension of picture with relays:



Relays

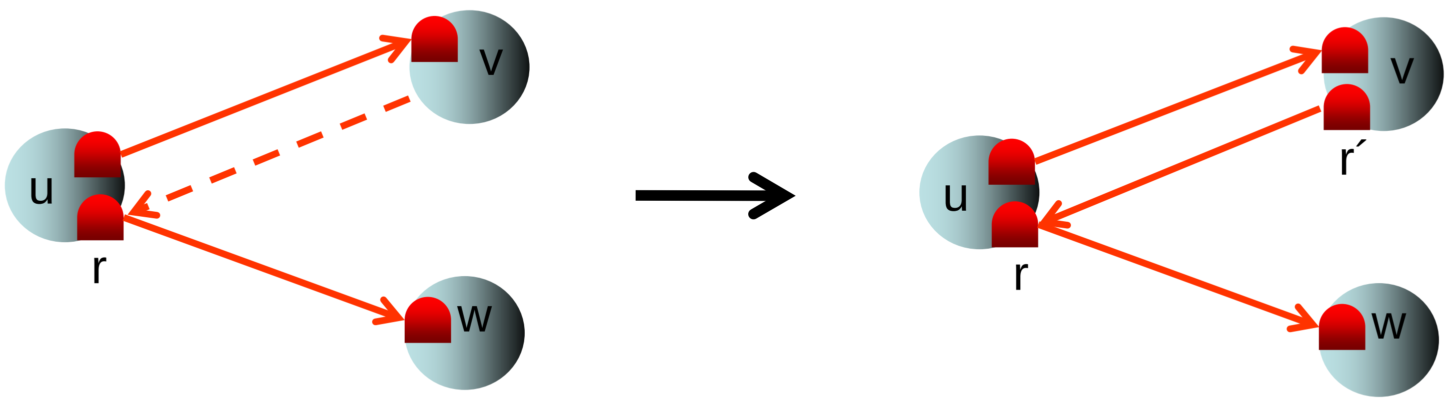
Safe introduction:



Instead of introducing w to v, u can only introduce its relay to w to v.

Relays

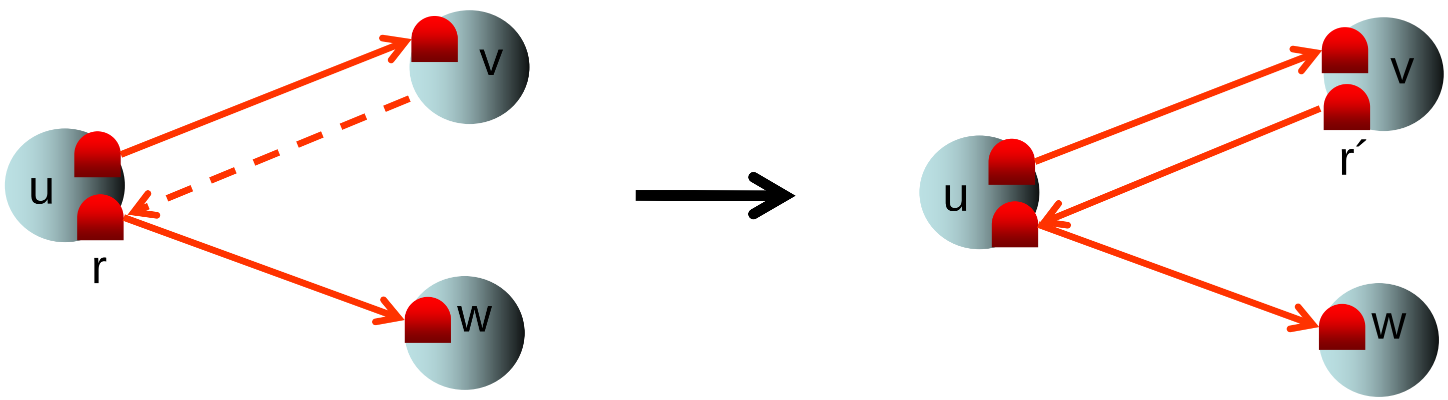
Safe introduction:



Once the reference of relay r to w is received by v, it is tied to a new relay r´ at v pointing to r.

Relays

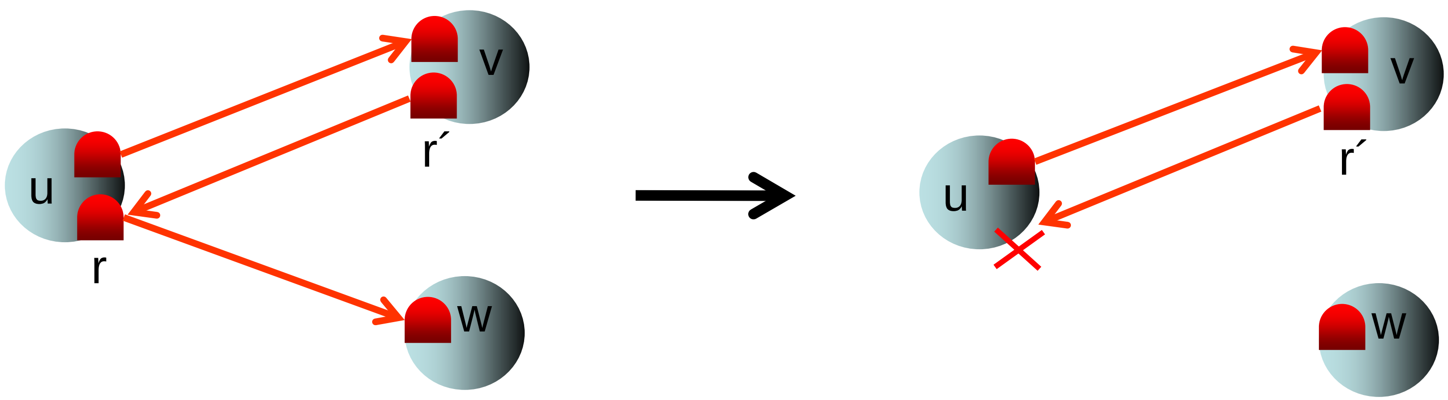
Safe introduction:



No access rights violated: u could have just forwarded anything from v to w by itself.

Relays

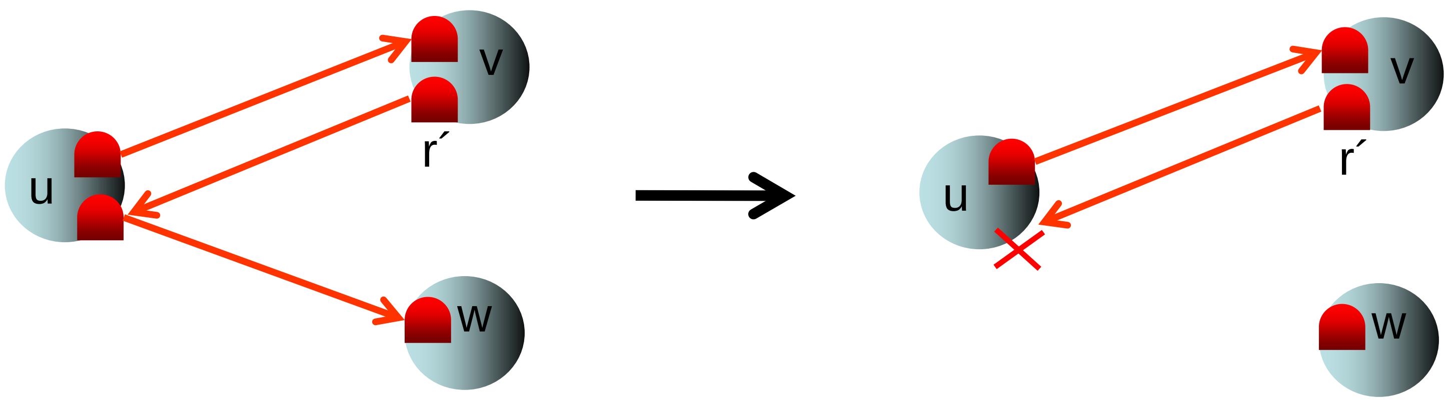
Safe introduction:



Most importantly, if u kills its relay to w, also v´s connection to w is gone.

Relays

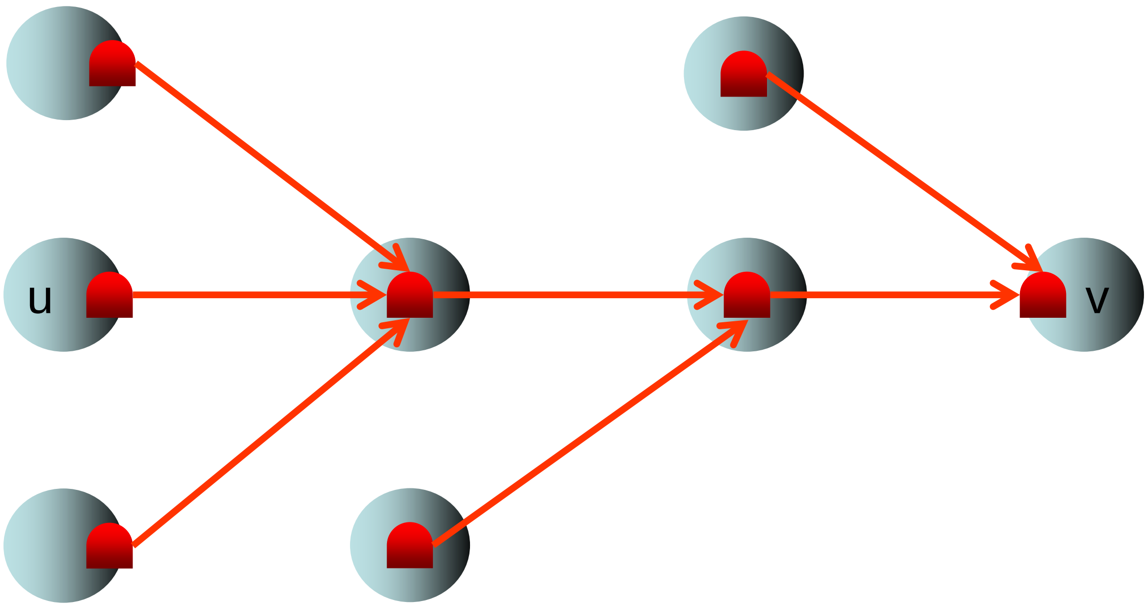
Safe introduction:



→ Principle of least exposure: when killing all relays with incoming links, no request can reach a node any more

Relays

Possible outcome of safe introductions:



Note that only process v will process message from u. The relays in between just forward the message.

Relay Semantics

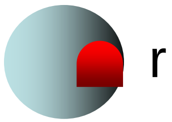
Processes have access to the following info about a relay r:

- r.incoming: number of incoming connections into r
- r.sink: identifier of sink relay of r (needed for safe form of fusion)
- r.direct∈{true,false}: is true if and only if r directly connects to its sink (or is a sink)

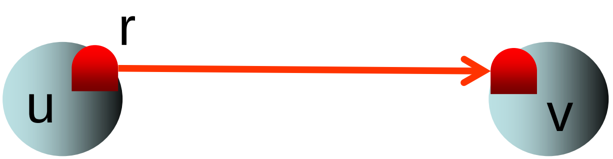
Relay Semantics

Commands:

- new Relay: creates new sink relay and returns reference r to calling process



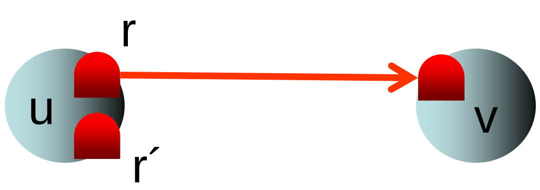
- u executes r←action(parameters): calls action(parameters) in the node hosting the sink relay of r (in example, node v)

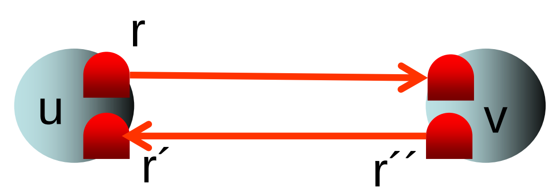


Relay Semantics

Commands:

- r←action(parameters): for any relay r´ in parameters, new outgoing relay r´´ is created at host of sink relay of r´ (i.e., v) and r´´ passed to v





A node only has access to local relays.

Relay Semantics

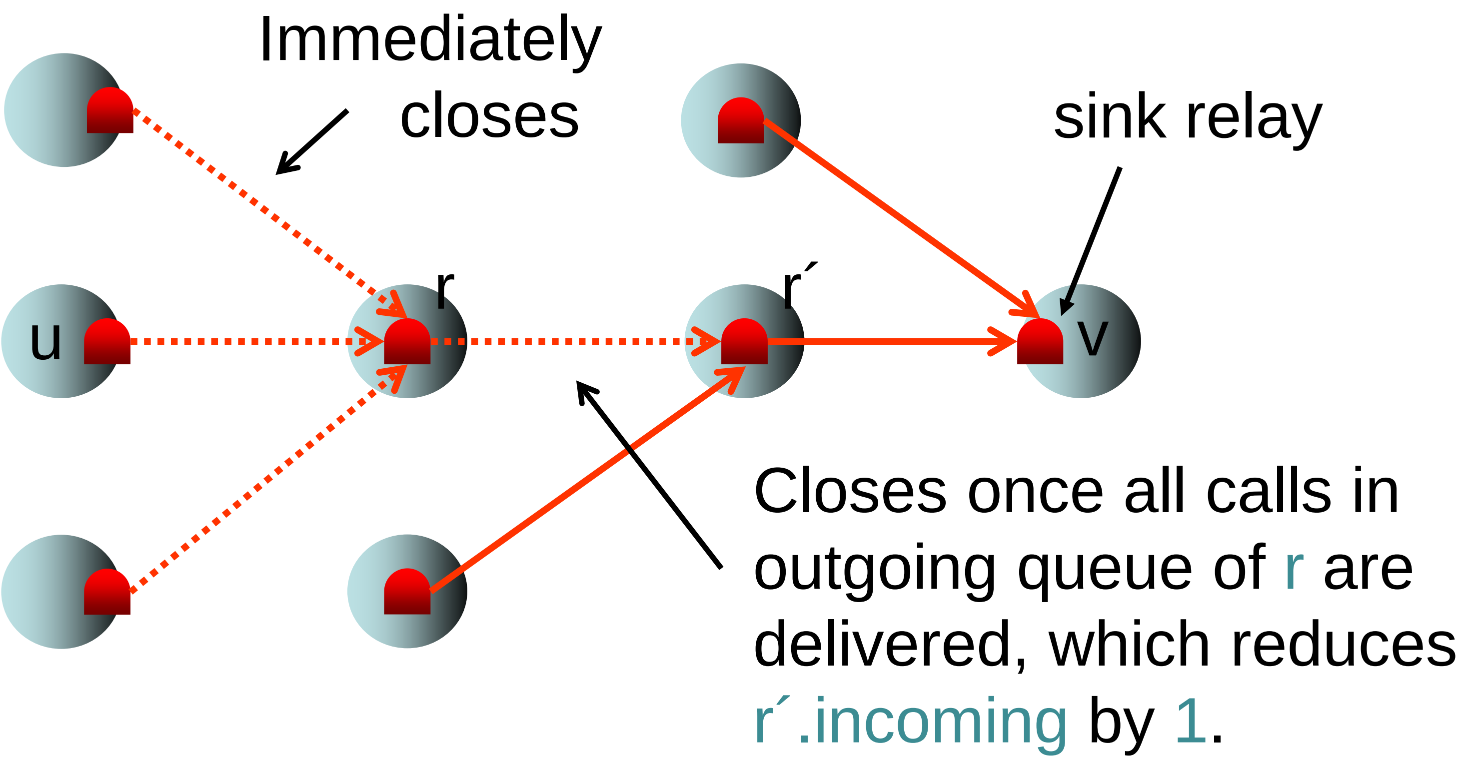
Commands:

- r←action(parameters): for any relay r´ in parameters, new outgoing relay r´´ is created at host of sink relay of r´ (i.e., v) and r´´ passed to v

- delete r: deletes relay r, which cuts off all relays behind it, but calls that have already been sent to/via it are still delivered. Outgoing connection of r closes once all of these calls have been delivered.

Relay Semantics

Outcome of deleting relay r:



Relay Semantics

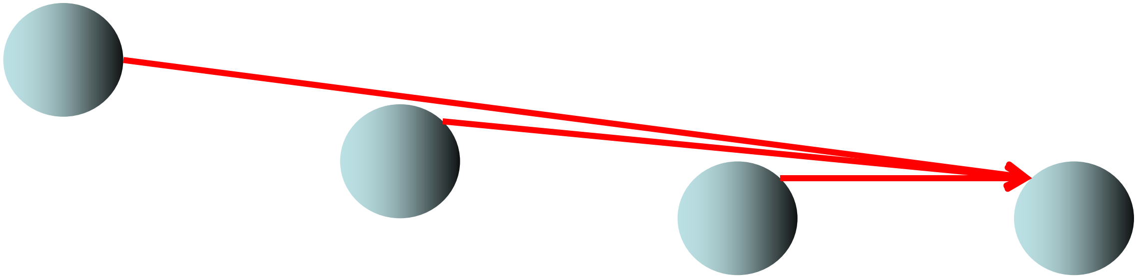
Remarks:

- Whenever a reference of some relay r is received, a local relay r´ is created in the receiving process pointing to r. This makes sure that processes only have references to local relays.
- Any relay newly created by a process is a sink relay (see v), i.e., all messages sent to it will be processed by v and only by v.

Relay Semantics

Possible outcome of safe introductions:

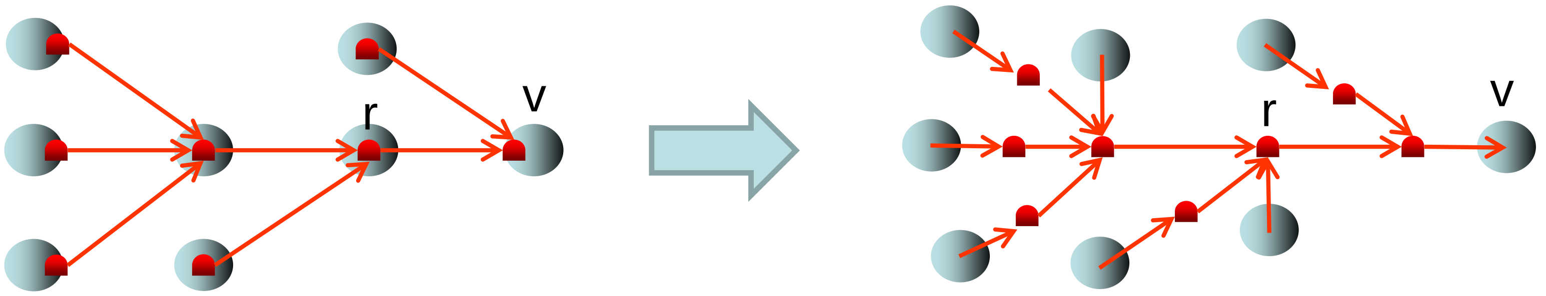
In our old graph terminology, this corresponds to the following connections (though there are now dependencies among them):



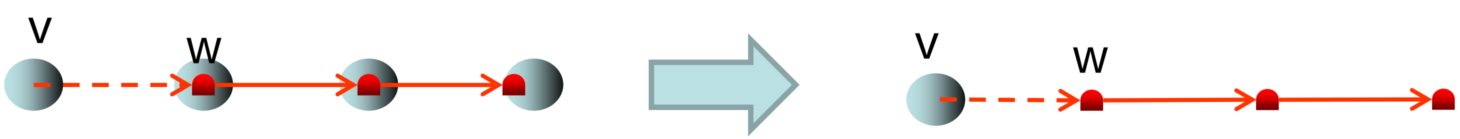
Relays

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



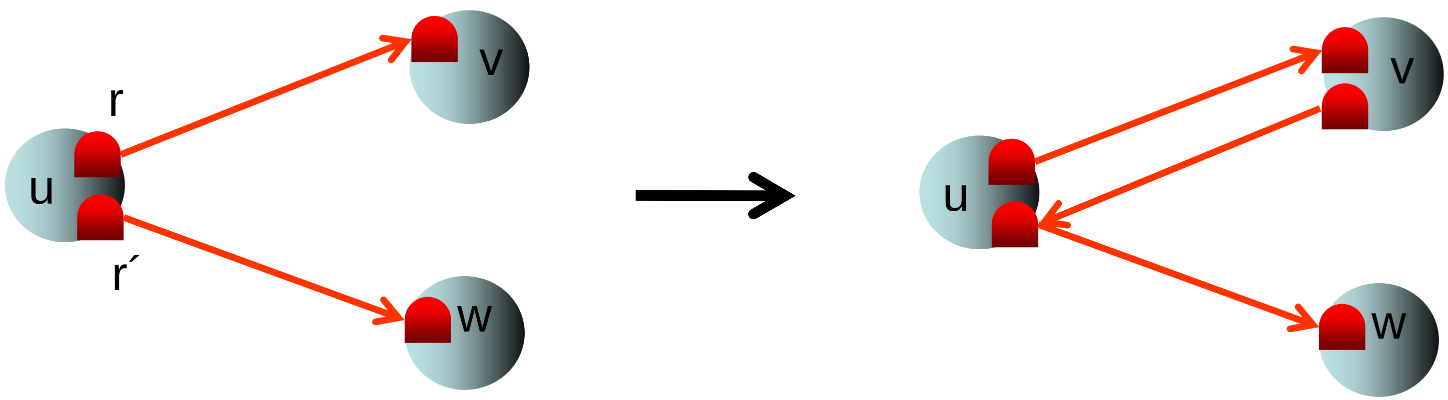
Relays

A relay graph $G=(R \cup P, E_L \cup E_M)$ is called

- weakly connected if for all pairs $v,w \in P$ there is a path from v to w in G when ignoring the directions of the edges
- strongly connected if for all pairs $v,w \in P$ there is a directed path from v to w in G

Relays

Safe introduction:

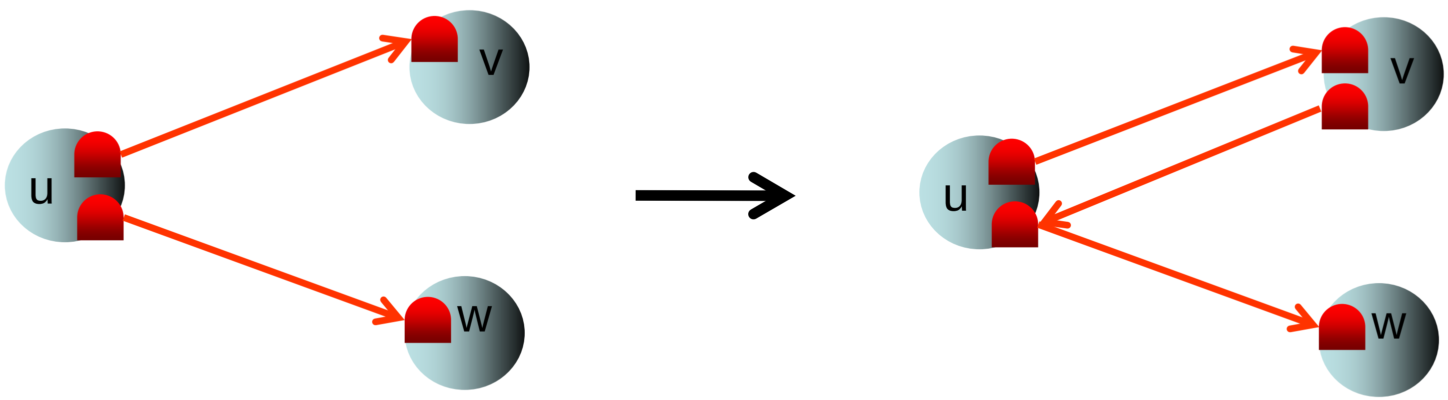


u executes: r←introduce(r´)

(introduce: just an example, could be any action)

Relays

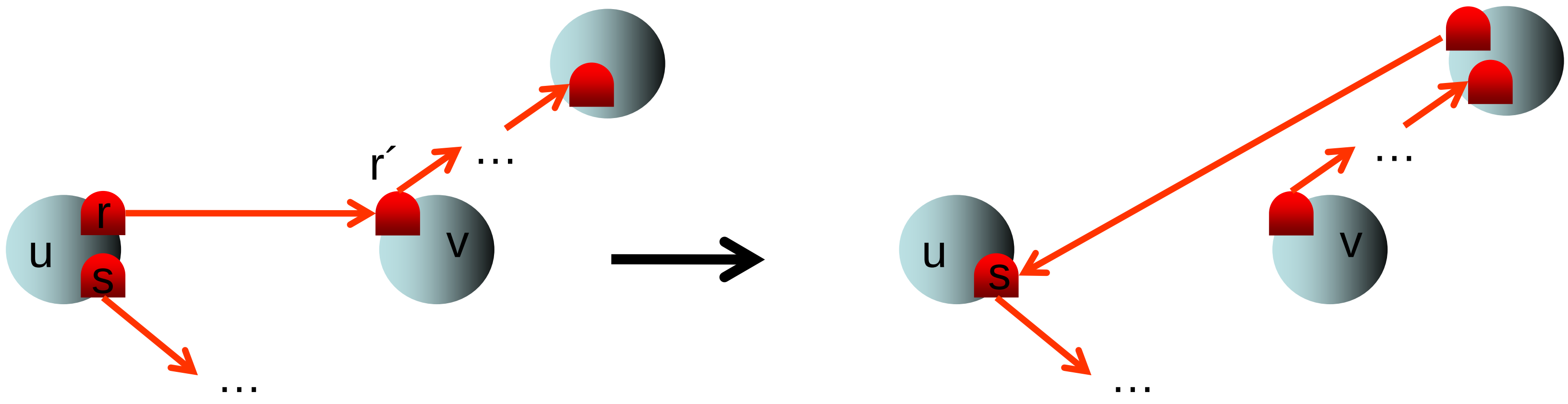
Safe introduction:



Certainly, safe introduction preserves weak (and strong) connectivity in relay graphs as this only adds an edge to G.

Relays

Safe reversal:

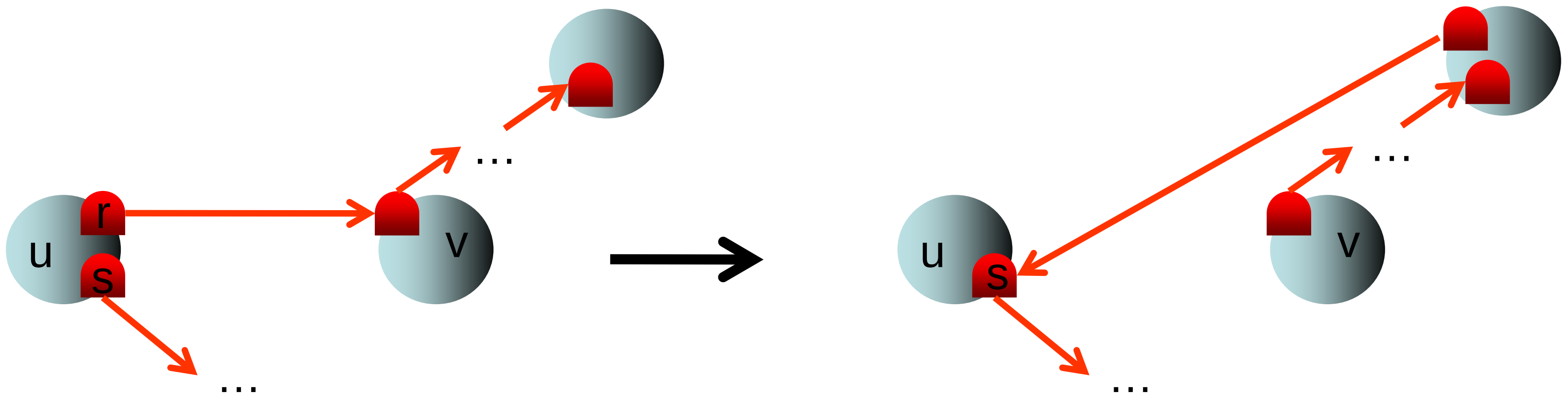


Given r has no incoming connections, u executes:
r←introduce(s); delete r

Note: r is only closed once s-ref. has reached r´.

Relays

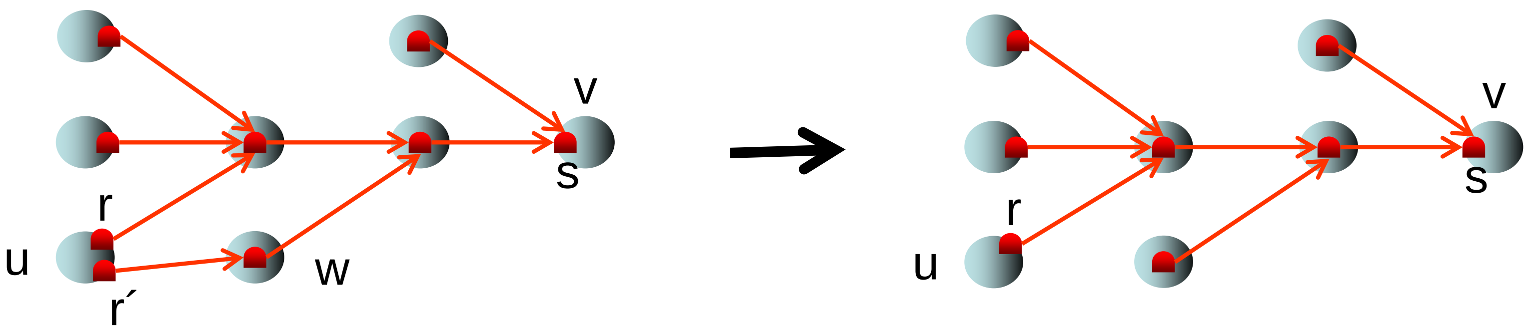
Safe reversal:



Certainly, safe reversal preserves weak connectivity since the connected components of u and v stay weakly connected.

Relays

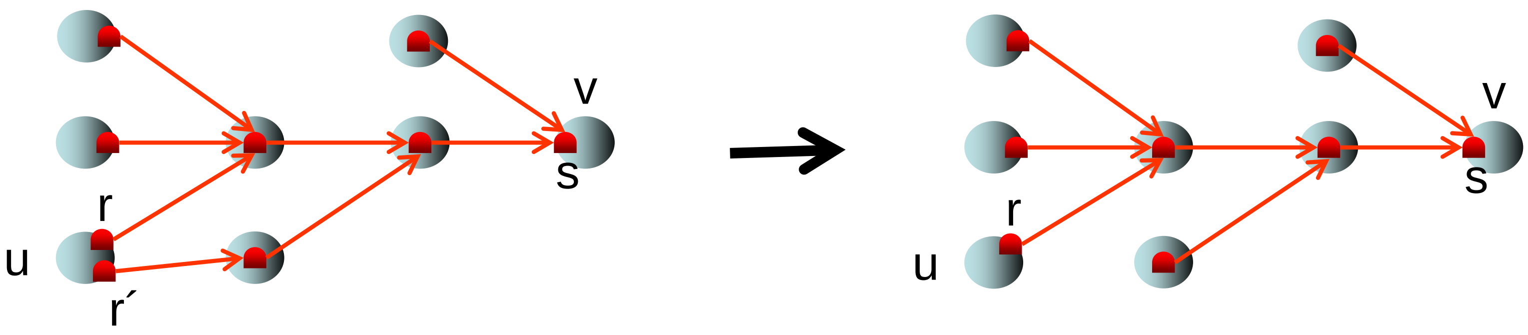
Safe fusion:



Given r´ has no incoming connections, u executes:
if r.sink=r´.sink then delete r´
Exercise: safe fusion preserves weak and strong connectivity.

Relays

Safe fusion:



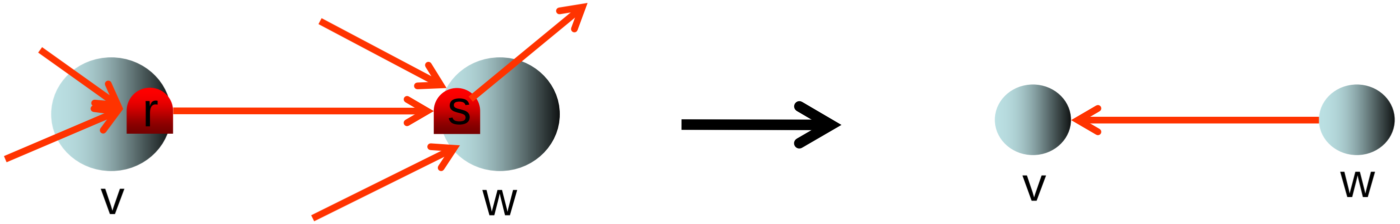
Remark: Processes only know whether two references point to the same relay or not (not to the same process). This allows processes to maximize anonymity since different relays can be used for different tasks.

Relays

Theorem 3.3: Safe introduction, fusion, and reversal are universal in a sense that one can get from any weakly connected relay graph $G=(R\cup P,E)$ to any other weakly connected relay graph $G'=(R\cup P,E')$ (where w.l.o.g. E and E' consist solely of explicit edges).

Proof:

- For any process $v\in P$ let $R(v)$ be the set of all relays local to v.
- Let $G_1=(P,E_1)$ be the graph where $(w,v)\in E_1$ if and only if there is an edge $(r,s)\in E$ with $r\in R(v)$ and $s\in R(w)$. Define $G_2=(P,E_2)$ in the same way for E'.



Relays

Theorem 3.3: Safe introduction, fusion, and reversal are universal in a sense that one can get from any weakly connected relay graph $G=(R \cup P,E)$ to any other weakly connected relay graph $G'=(R \cup P,E')$ (where w.l.o.g. E and E' consist solely of explicit edges).

Proof:

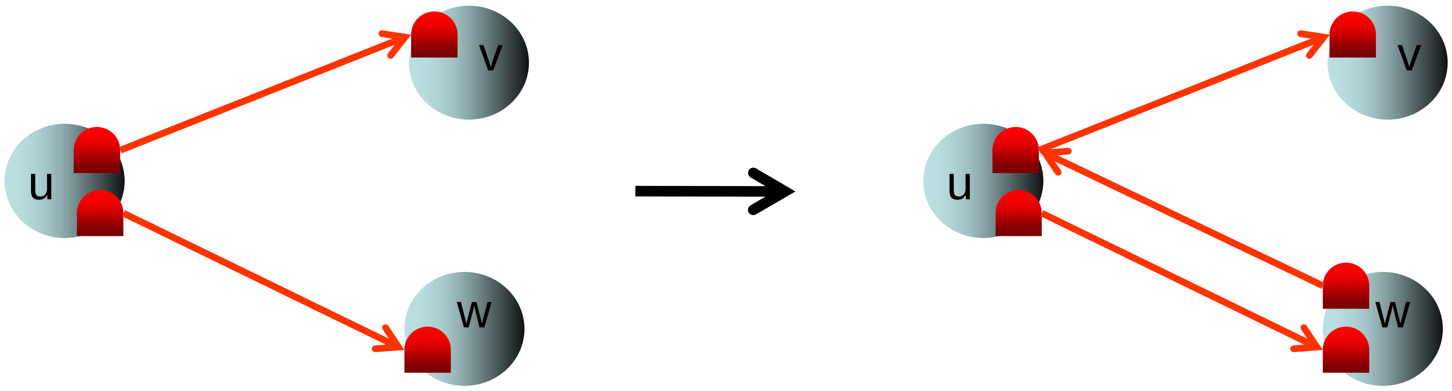
- For any process $v \in P$ let $R(v)$ be the set of all relays local to v.
- Let $G_1=(P,E_1)$ be the graph where $(w,v) \in E_1$ if and only if there is an edge $(r,s) \in E$ with $r \in R(v)$ and $s \in R(w)$. Define $G_2=(P,E_2)$ in the same way for E'.

First, we show how to emulate the standard introduction and delegation rules by our safe rules. The remaining proof then proceeds in three parts:

1. Transform G into G_1.
2. Transform G_1 into G_2.
3. Transform G_2 into G'.

Proof of Theorem 3.3

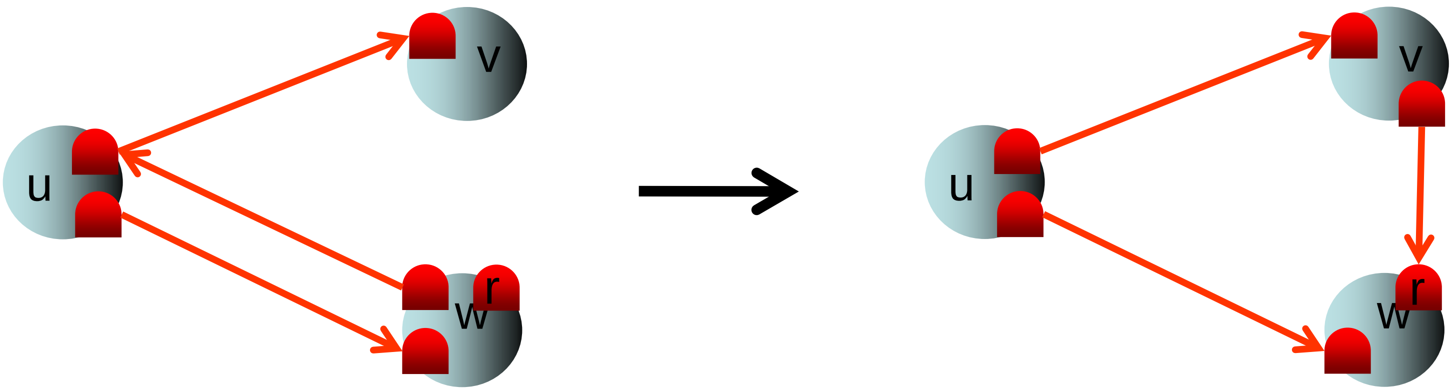
Emulation of introduction rule (u introduces w to v):



First, u introduces w to its relay to v (using the safe introduction rule).

Proof of Theorem 3.3

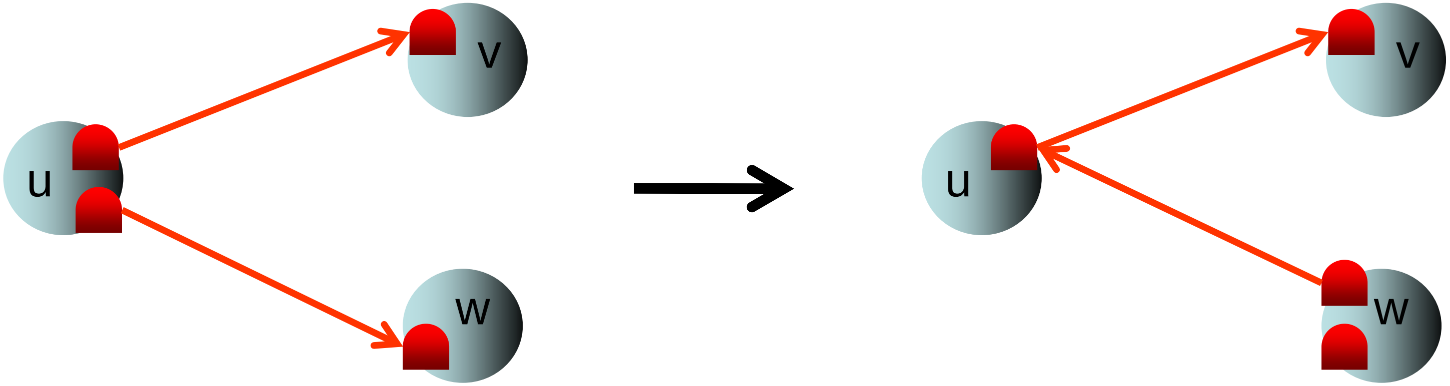
Emulation of introduction rule (u introduces w to v):



Then w establishes a new relay r, sends its reference via u to v and drops its relay to u (which resembles the safe reserval rule).

Proof of Theorem 3.3

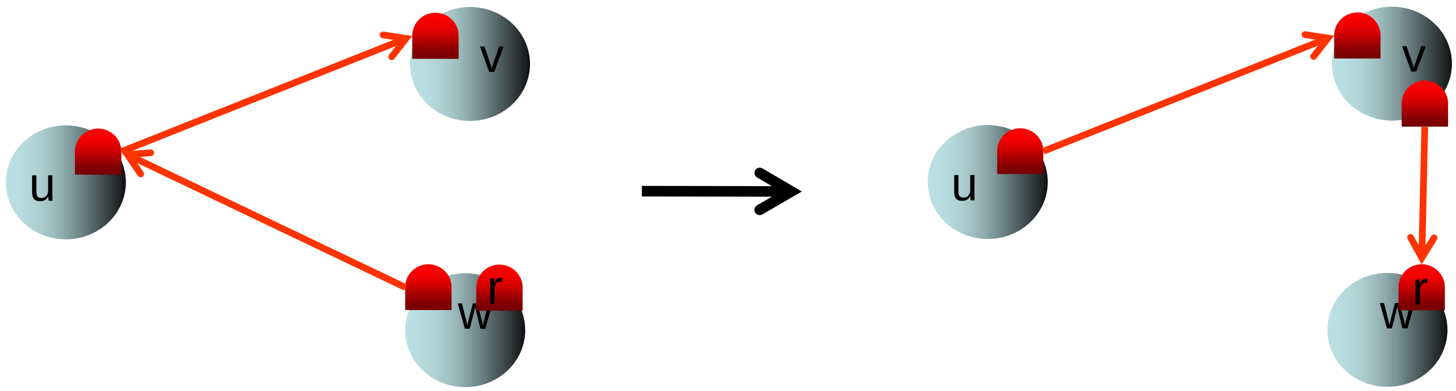
Emulation of delegation rule (u delegates w to v):



First, u introduces w to its relay to v and drops its relay to w (which resembles the safe reversal rule).

Proof of Theorem 3.3

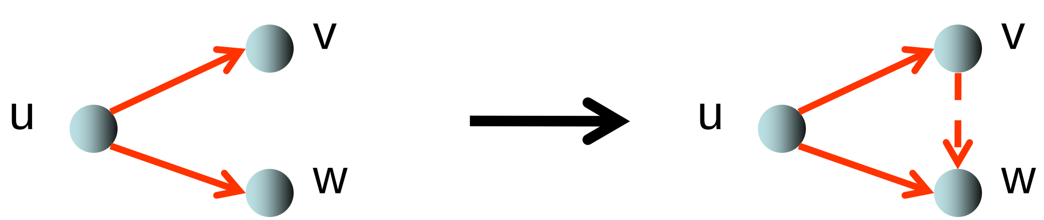
Emulation of delegation rule (u delegates w to v):



Then w establishes a new relay r, sends its reference to u (which will be forwarded to v) and drops its relay to u (which resembles the safe reserval rule).

Proof of Theorem 3.3

Remark: Since now w is always directly involved whenever it is introduced or delegated to a node v, w can also ensure that no corrupted information about it is sent to v. This is not guaranteed by the old way introduction and delegation is handled:

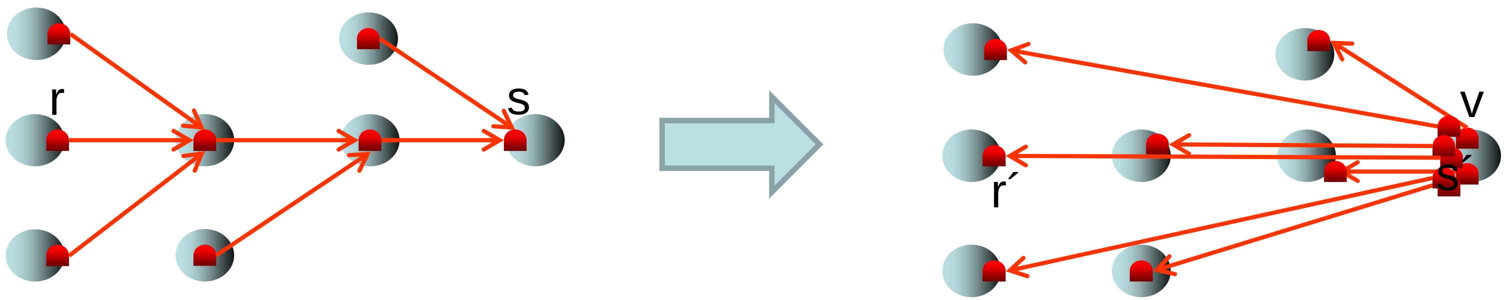


u sends a message to v containing w´s reference.

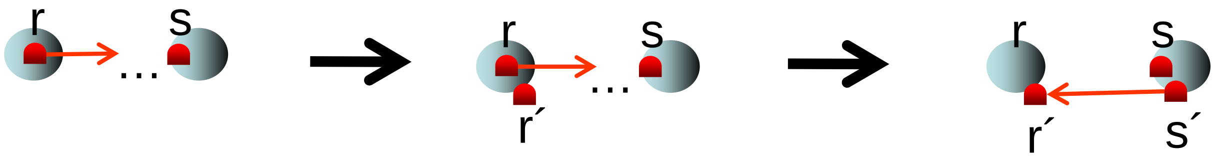
Proof of Theorem 3.3

Transforming G into G_1:

First, transform any relay tree in the following way starting with the most distant relays r from s



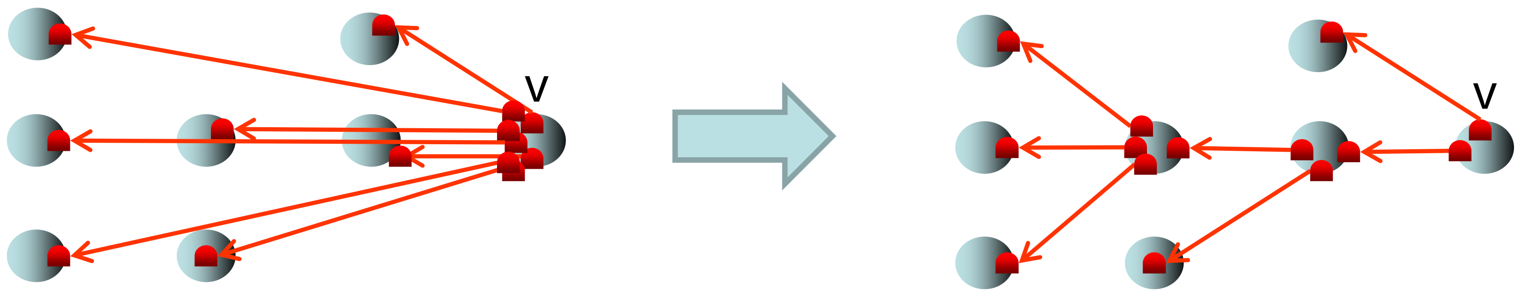
using safe reserval for any pair (r,s):



Proof of Theorem 3.3

Transforming G into G_1:

Then, transform the star back into the original tree, but with reversed, isolated edges



using the safe rules emulating the standard delegation rule. Since at the end just isolated edges are left, we can simplify that to our standard graph on processes, G_1.

Proof of Theorem 3.3

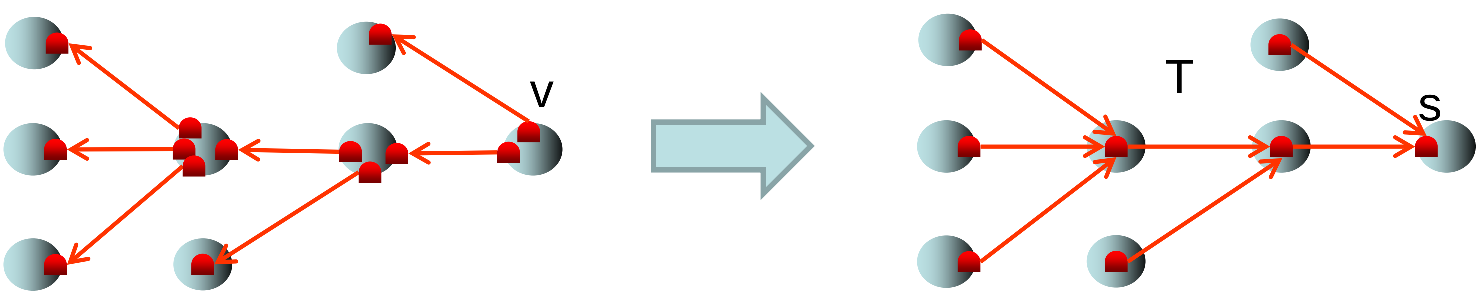
Transforming G_1 into G_2:

This follows from Theorem 3.2 since introduction, delegation, fusion, and reversal can be emulated by our safe primitives.

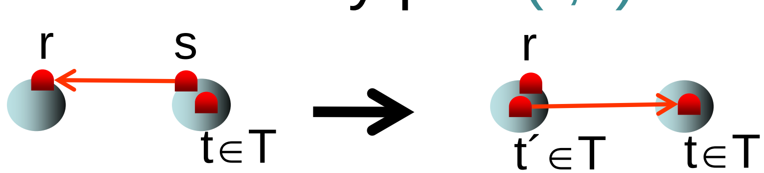
Proof of Theorem 3.3

Transforming G_2 into G´:

For any relay tree T in G´, transform the individual edges belonging to it in G_2 into that tree starting with the closest relays to v

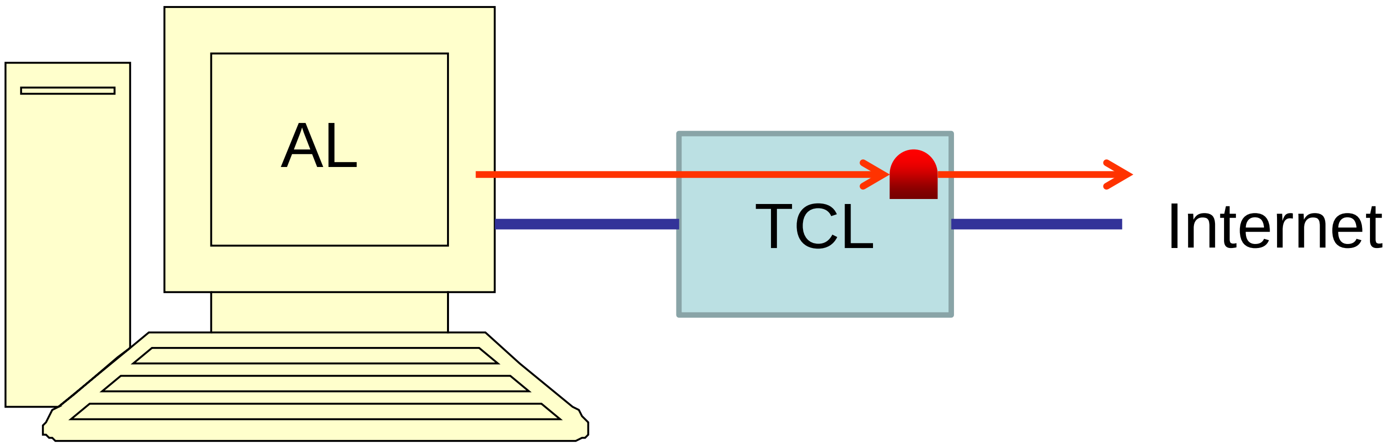


by using safe reserval for any pair (r,s):



Realization of Relays

Embedding into Trusted Communication Environment:



- AL: application layer, manages processes
- TCL: trusted communication layer, manages relays

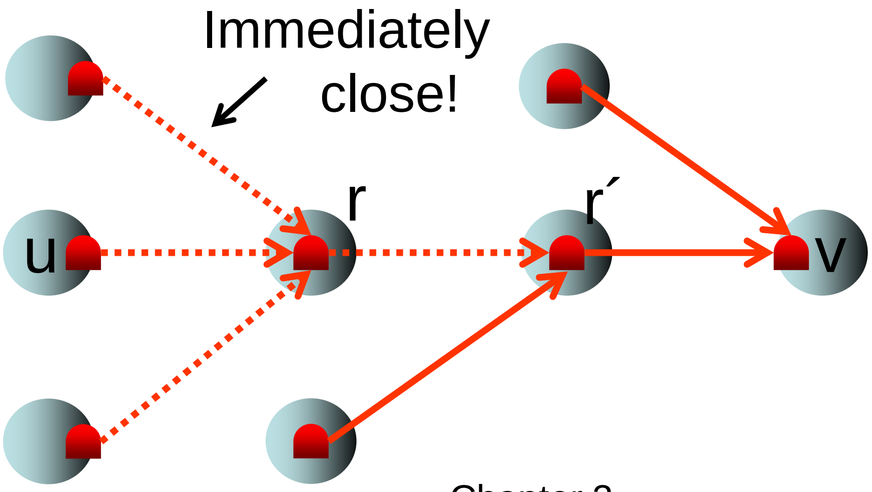
Why Relays?

Standard assumption for adversarial behavior in theory of distributed systems:

- Adversarial nodes do not overwhelm other nodes with messages.

With relays, this assumption is not needed any more since adversarial nodes can be isolated.

Relay r deleted:



Why Relays?

Important access control requirements:

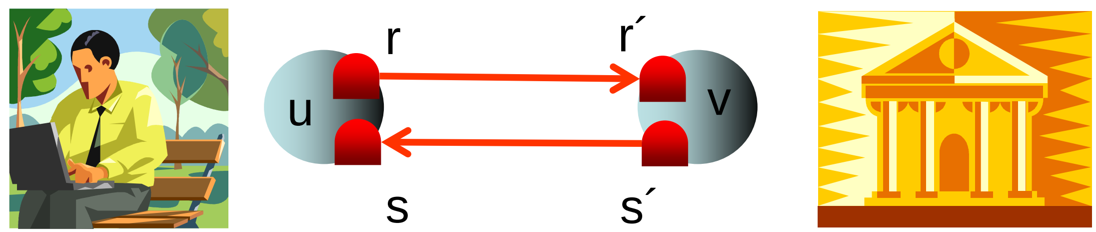
- Integrity: It should not be possible to construct, tamper with or steal an access right.
- Propagation: There should be mechanisms for controlling the transfer of access rights.
- Revocation: It should be possible to revoke an access right.

If relays are managed by reliable and protected TCL, these requirements can be satisfied.

$\rightarrow$ Researchers in distributed computing can now consider denial-of-service and access control problems

Why Relays?

When using sink relays as pseudonyms, authentication is possible:



If u executes r←buy(x,s), a new relay s´´ in v will connect to s, allowing v to check via s´.sink=s´´.sink and s´´.direct=true that request came from u.

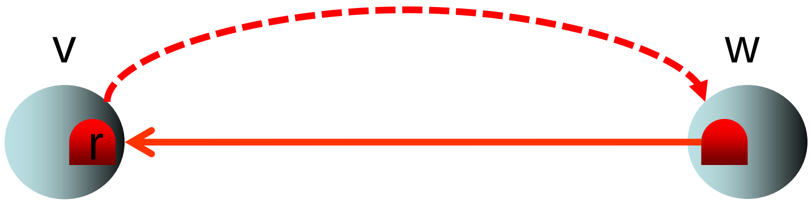
Overview

- Model and basic primitives
- Universality
- Relays
- Joining and Leaving

Joining an Overlay

Decentralized approach:

- Node v sends (encrypted) access info about r via some external (potentially insecure) channel (like emails) to w.
- Node w feeds this info into its TCL to connect to r.

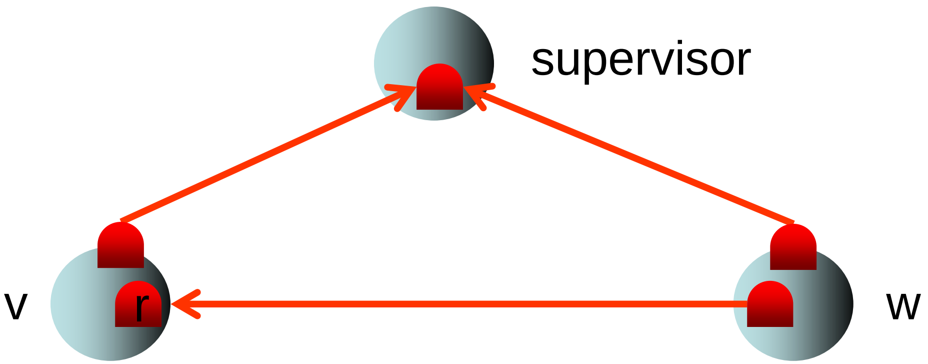


Problem: Access info can get stolen (to hijack connection) or replaced by other info (for identity theft)

Safely Joining an Overlay

Supervised approach:

- When TCL is initialized, every node is connected to a preset, trusted supervisor.
- Supervisor safely introduces v to w.

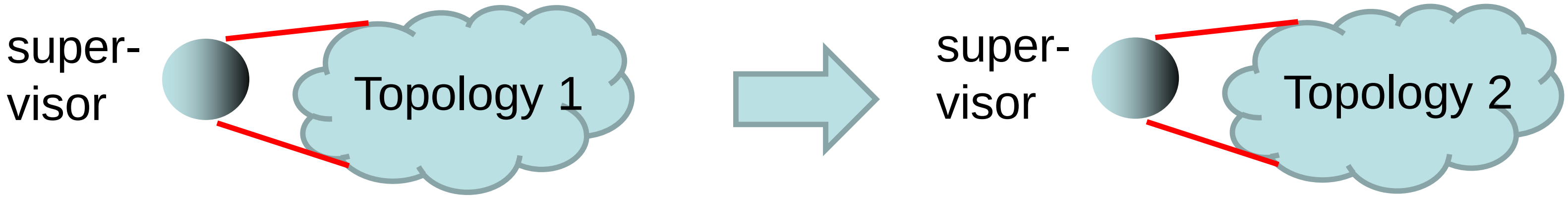


Useful for virtual private networks (VPNs)!

Supervised Overlays

Remarks:

- Advantage: safe joining of overlay
- Supervisor may also be used to transform overlay, i.e., nodes wait for supervisor commands to change connections.

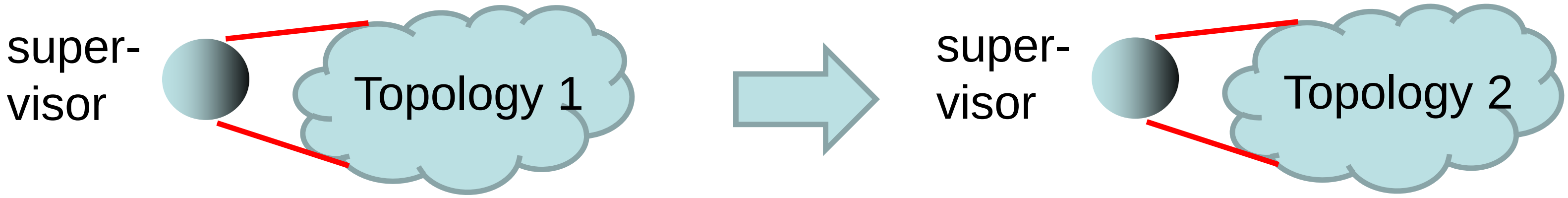


- Advantage of supervised transformations: supervisor can compute minimum number of transitions needed to get from Topology 1 to Topology 2, thereby minimizing the work of the nodes and the disruptions, topology transformation may cause to the functionality of the overlay (similar approach in Software Defined Networks!).
- We just recently developed an algorithm for computing near-minimum number of transformations (submitted to ICALP).

Supervised Overlays

Remarks:

- Advantage: safe joining of overlay
- Supervisor may also be used to transform overlay, i.e., nodes wait for supervisor commands to change connections.



- What if supervisor is malicious? It could start Sybil attacks (flooding an overlay with fake identities) or Eclipse attacks (disconnecting parts of the overlay)
- Not a problem if supervisor only suggests changes that the nodes could have done themselves (safe intro, delegation and fusion), since these cannot introduce new nodes to the system and these cannot disconnect nodes from the overlay.
- Exercise: Why?

Safely Leaving an Overlay

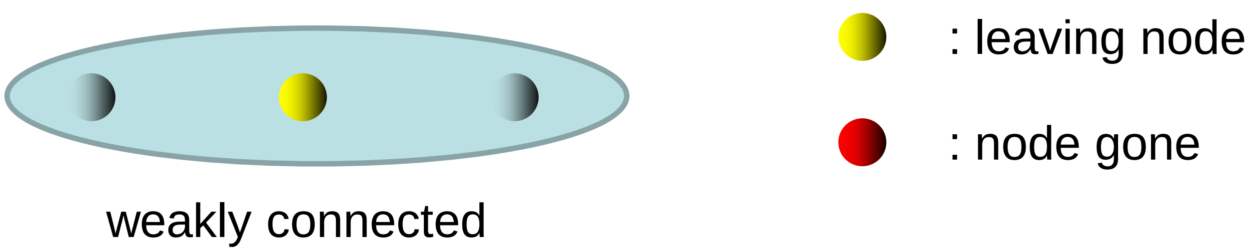
Safe Departure Problem (SDP): leave overlay without disconnecting it.

Decentralized approach:

Theorem 3.4: The SDP cannot be solved in the standard link model (without relays).

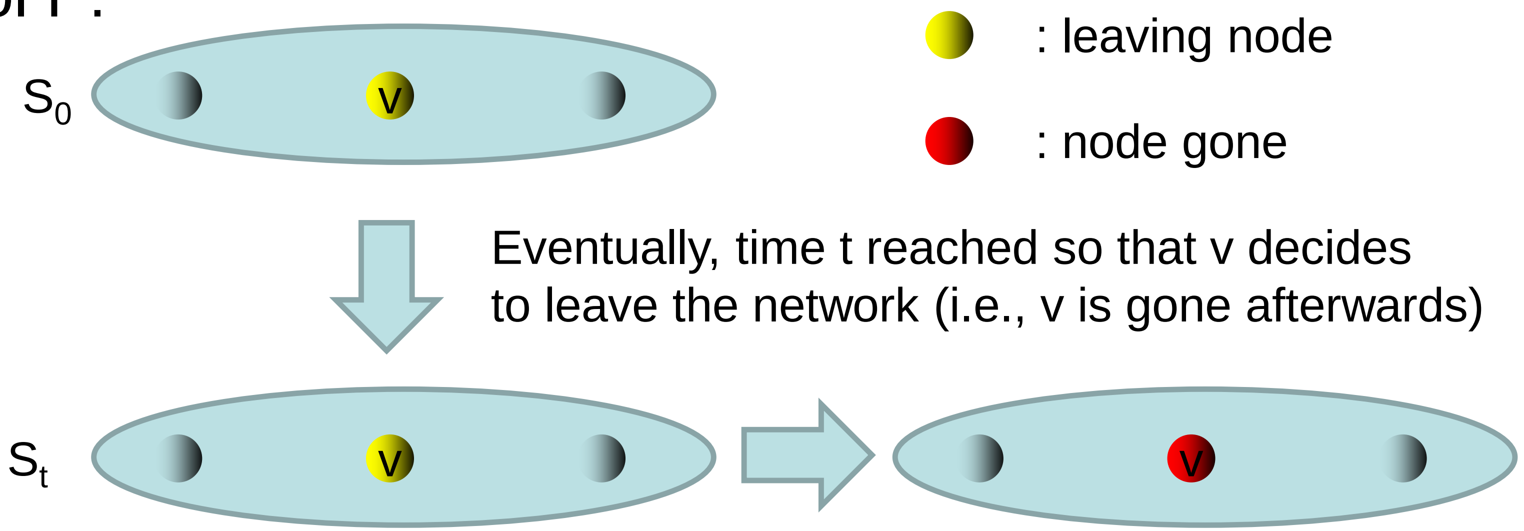
Proof:

- Suppose there is a distributed protocol P that can solve the SDP problem.
- Consider the following initial state S_0:

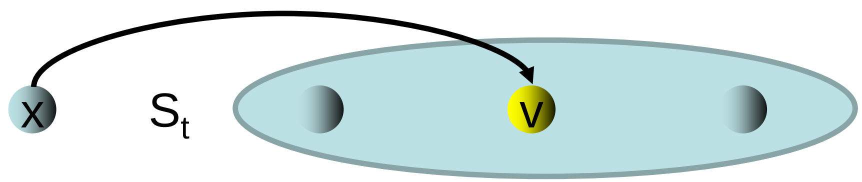


Safely Leaving an Overlay

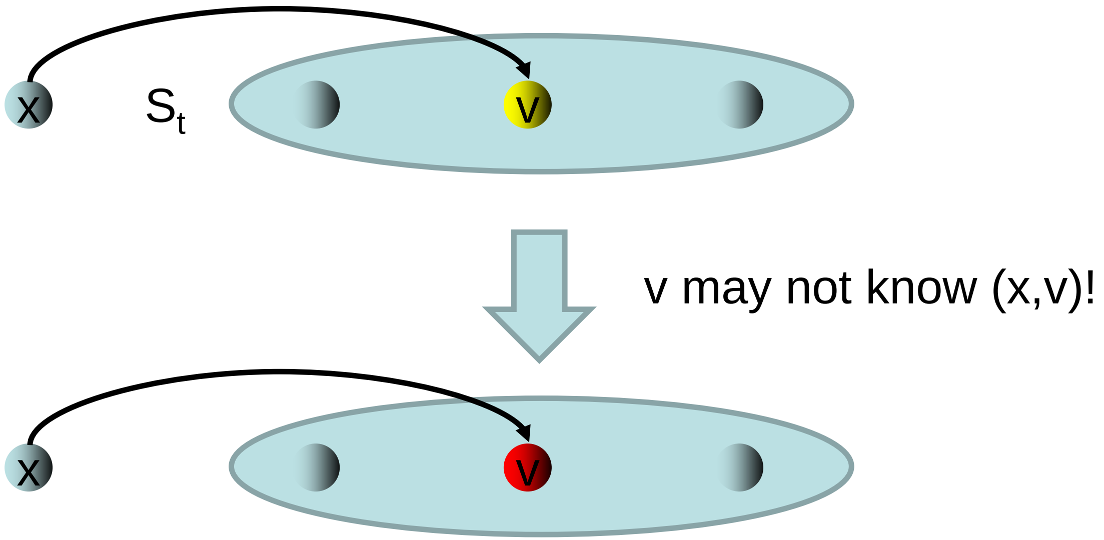
Protocol P:



Consider now the following initial state:

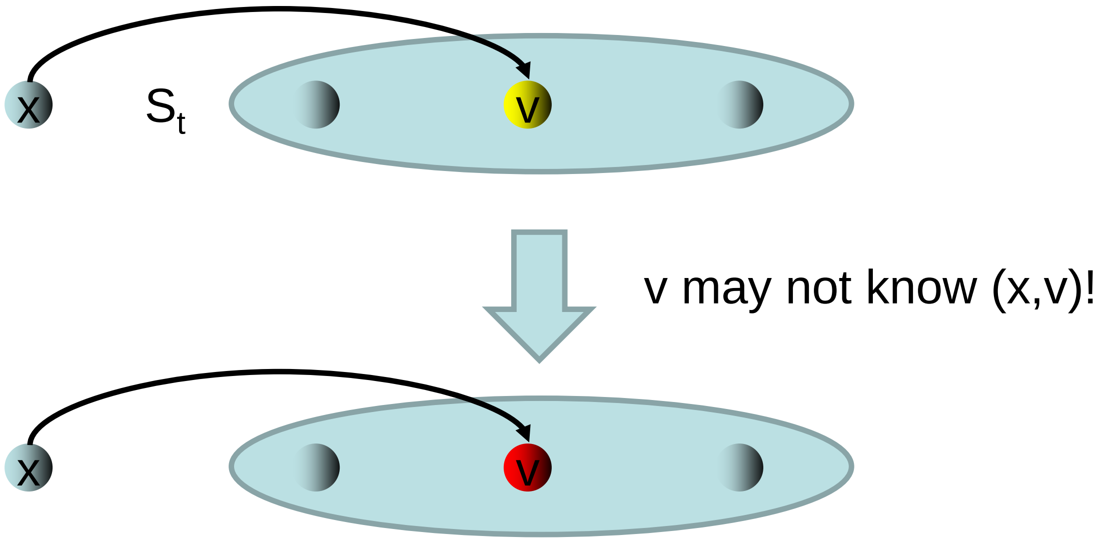


Safely Leaving an Overlay



Problem: v may still decide to leave the system since it may not be aware of the fact that x has a link to v! But if v leaves, then x is isolated, i.e., Protocol P does not solve the SDP problem. Contradiction!

Safely Leaving an Overlay



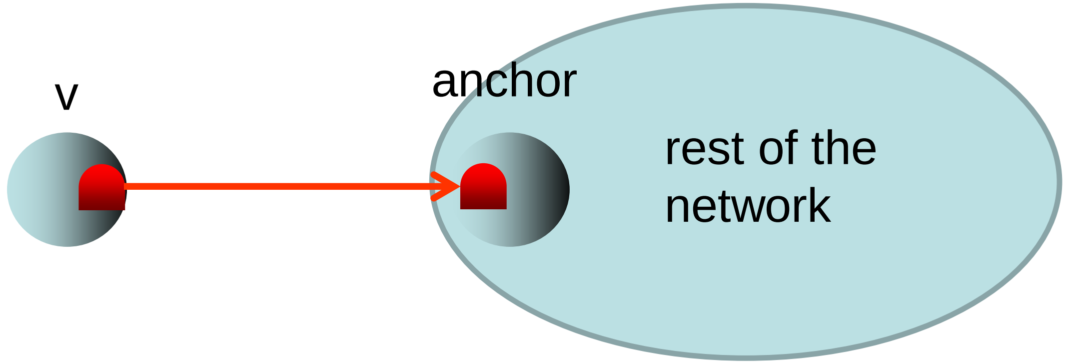
- This problem will not happen with the relay approach because v must have given the permission to x to connect to v and therefore is aware of the link (x,v)!
- In fact, for relays, a distributed protocol is known (presented by us at SSS 2018) that can solve the SDP problem.

Safely Leaving an Overlay

Safe Departure Problem (SDP): leave overlay without disconnecting it.

Basic idea to solve the SDP with relays (that works for isolated departures):

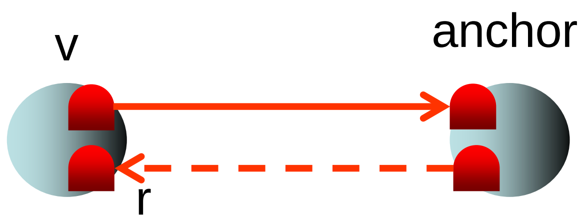
- Phase 1: v looks for any relay connection to a non-leaving node and declares it its anchor.
- Phase 2: v safely delegates its connections to its anchor until it is only connected to its anchor. Once this is completed, v leaves.



Safely Leaving an Overlay

Phase 1: v looks for any relay connection to a non-leaving node and declares it its anchor.

- Node v may pick any outgoing neighbor as its anchor, but to be on the safe side, it may periodically check (via timeout) whether its anchor is still a non-leaving node. It does so by sending a reference to r with its request so that the anchor can reply with its state.

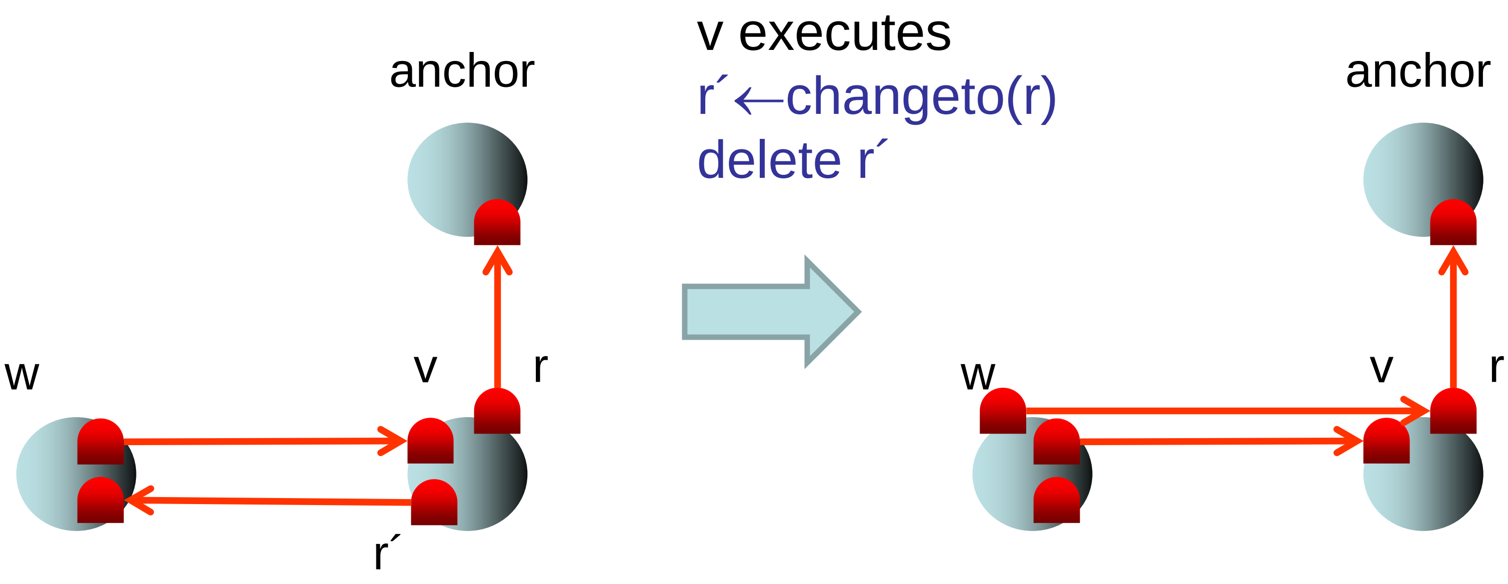


- The anchor immediately closes the link to r after replying (but remember that its answer will still be delivered!) so that v just has an outgoing but no incoming connection to its anchor.

Safely Leaving an Overlay

Phase 2: v safely delegates its connections to its anchor until it is only connected to its anchor. Once this is completed, v leaves.

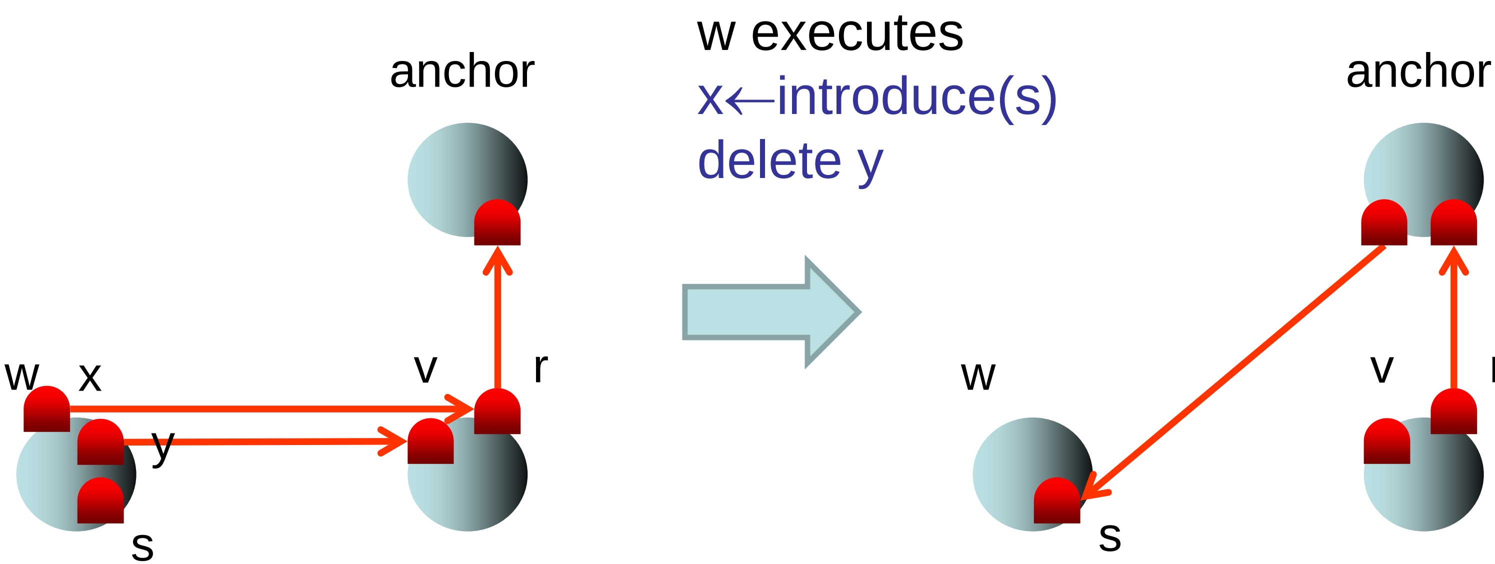
Example:



Safely Leaving an Overlay

Phase 2: v safely delegates its connections to its anchor until it is only connected to its anchor. Once this is completed, v leaves.

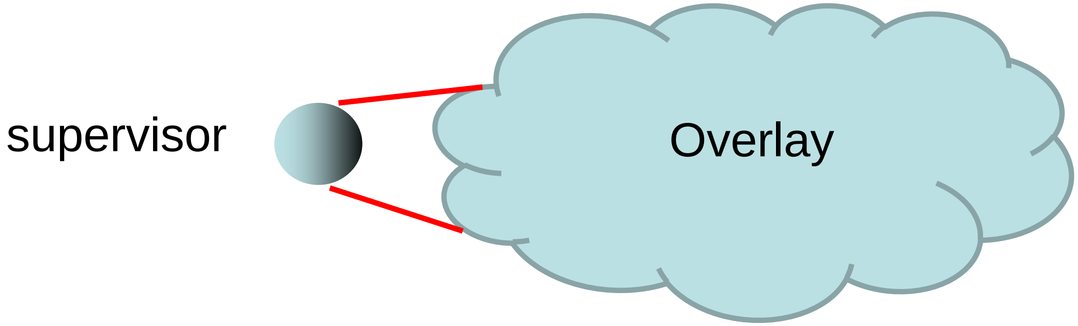
Example:



Safely Leaving an Overlay

Safe Departure Problem (SDP): leave overlay without disconnecting it.

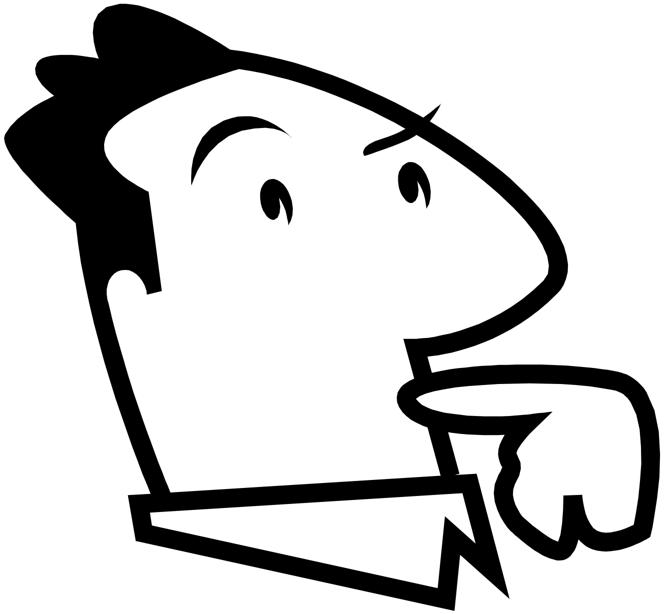
Supervised approach:



This is easy since supervisor knows the connections between the processes, so it can make the appropriate introductions in order to avoid disconnectivity. (Exercise)

References

- Kishore Kothapalli and Christian Scheideler. Supervised Peer-to-Peer Systems. ISPAN 2005: 188-193.
- Dianne Foreback, Andreas Koutsopoulos, Mikhail Nesterenko, Christian Scheideler, and Thim Strothmann. On Stabilizing Departures in Overlay Networks. In SSS 2014: 48-62.
- Andreas Koutsopoulos. Dynamics and Efficiency in Topological Self-Stabilization. PhD Thesis, Paderborn University, December 2015.
- Andreas Koutsopoulos, Christian Scheideler, and Thim Strothmann. Towards a universal approach for the finite departure problem in overlay networks. Inf. Comput. 255: 408-424 (2017).
- Christian Scheideler and Alexander Setzer. Relays: A New Approach for the Finite Departure Problem in Overlay Networks. In SSS 2018: 239-253.
- Christian Scheideler and Alexander Setzer. On the Complexity of Graph Transformations. Unpublished manuscript.



Questions?