

Advanced Distributed Algorithms and Data Structures

Chapter 5: TCM Model and Programming Environment

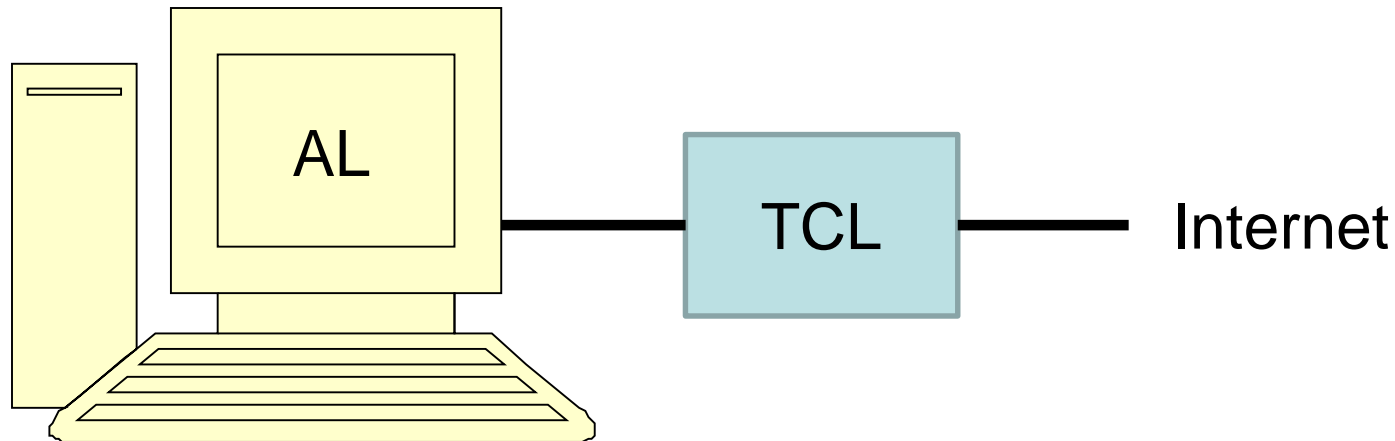
Christian Scheideler
Institut für Informatik
Universität Paderborn

Overview

- TCM model
- Pseudo-code and example programs
- Programming environment

TCM Model

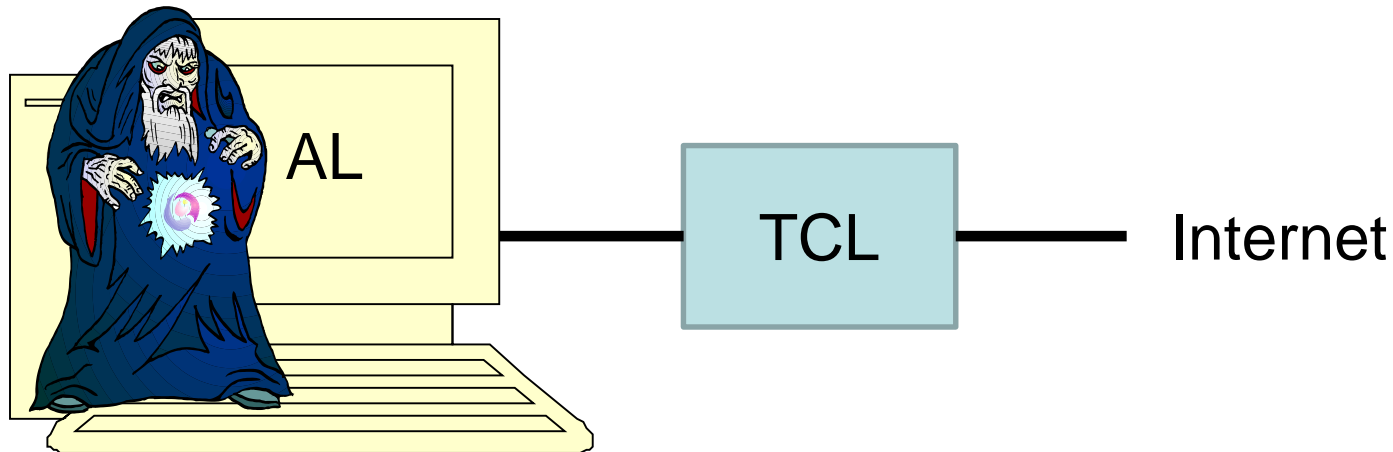
Trusted Communication Model (TCM):



- AL (**Application Layer**): large storage capacity and computational power, but potentially insecure
- TCL (**Trusted Communication Layer**): low storage capacity and computational power but can securely manage ports and keys and can securely execute basic primitives

TCM Model

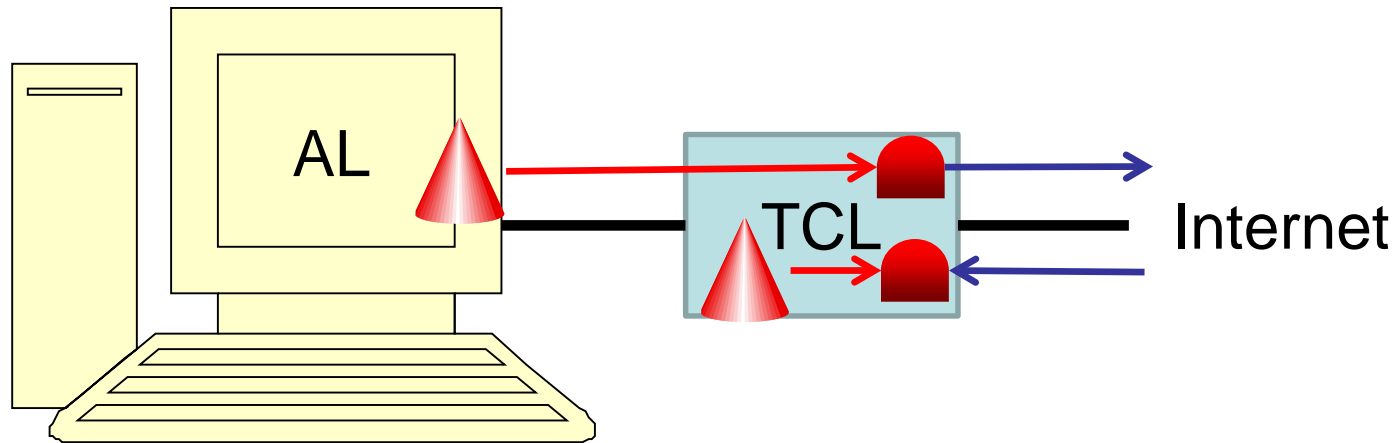
Trusted Communication Model (TCM):

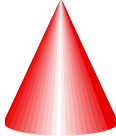




- AL: can be invaded
- TCL: cannot be invaded or inspected

Goal of TCL: support AL in ensuring availability, integrity, and confidentiality

TCM Model



- Processes  (at AL and TCL)
- Relays  (managed by the TCL)
- Processes have **references** () to local relays

TCM Model

Processes can interconnect (via relays) and execute **actions**.

General form of action: („→“ means „implies“)

⟨name⟩: ⟨event⟩ → ⟨commands⟩

In words, if ⟨event⟩ is true, then execute ⟨commands⟩.

Two types of actions:

- Triggered by a local/remote call:
⟨name⟩(⟨parameters⟩) → ⟨commands⟩
(Short form of ⟨name⟩: ⟨name⟩(⟨parameters⟩) called → ...
If ⟨name⟩(⟨parameters⟩)-call received, then execute ⟨commands⟩.)
- Triggered by a local state:
⟨name⟩: ⟨predicate⟩ → ⟨commands⟩
(If ⟨predicate⟩ is true, then execute ⟨commands⟩ .)

All messages are remote action calls.

TCM Model

Processes can interconnect (via relays) and execute **actions**.

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)
```

Action „minimum“ is executed upon receipt of a request to call `minimum(x,y)`. **No return of values possible when called remotely!**

TCM Model

Processes can interconnect (via relays) and execute **actions**.

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 the action is **periodically executed** by the given peer.

TCM Model

Execution of actions: Processes can act concurrently but within a process the actions must be executed in a strictly sequential, and therefore **atomic** way.

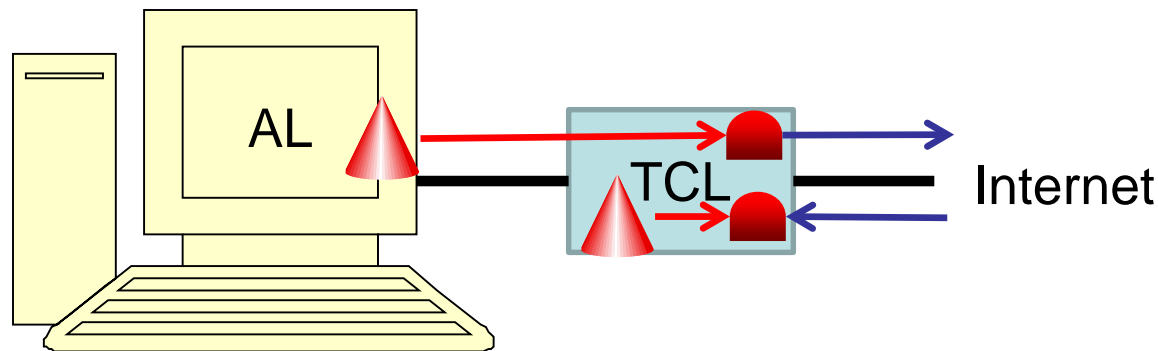
→ every action must eventually terminate!

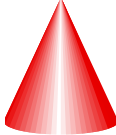


This simplifies correctness proofs.

Timing of actions: In order to avoid that actions are called too frequently, a lower bound for the repeated execution of an action can be set by calling **enable(<name>,<min-time>)**, ensuring that only after **min-time** steps it is enabled again (i.e., ready to be executed). This is, for example, useful to control the executions of timeout actions (see prev. slide).

TCM Model

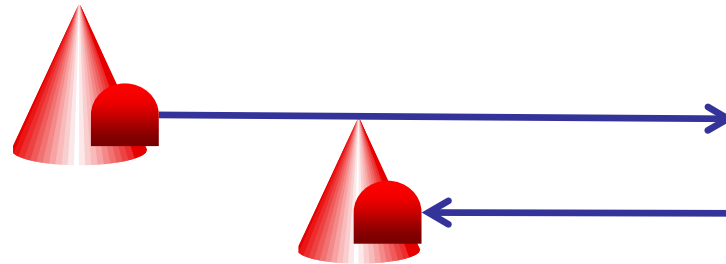
Next we discuss how to handle relays.

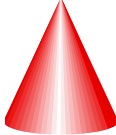




- Processes  (at AL and TCL)
- Relays  (managed by the TCL)
- Processes have **references** () to local relays

TCM Model

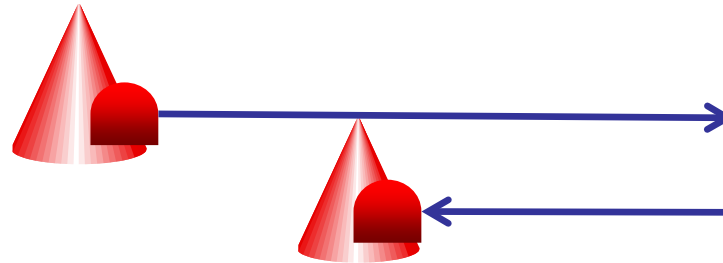
Abstraction:



- Processes  (at AL and TCL)
- Relays  (managed by the TCL)
- Processes have **references** () to local relays

TCM Model

Abstraction:

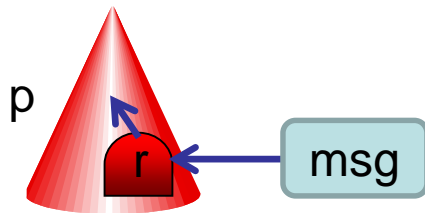


- Only primitives allowed for a relay:
new, delete, safe introduction

As we know, this is sufficient for universality.

TCM Model

- **new Relay**: creates **new sink relay** (i.e., a relay whose outgoing link points to the process that created it) and returns a reference to it (which is only locally valid)



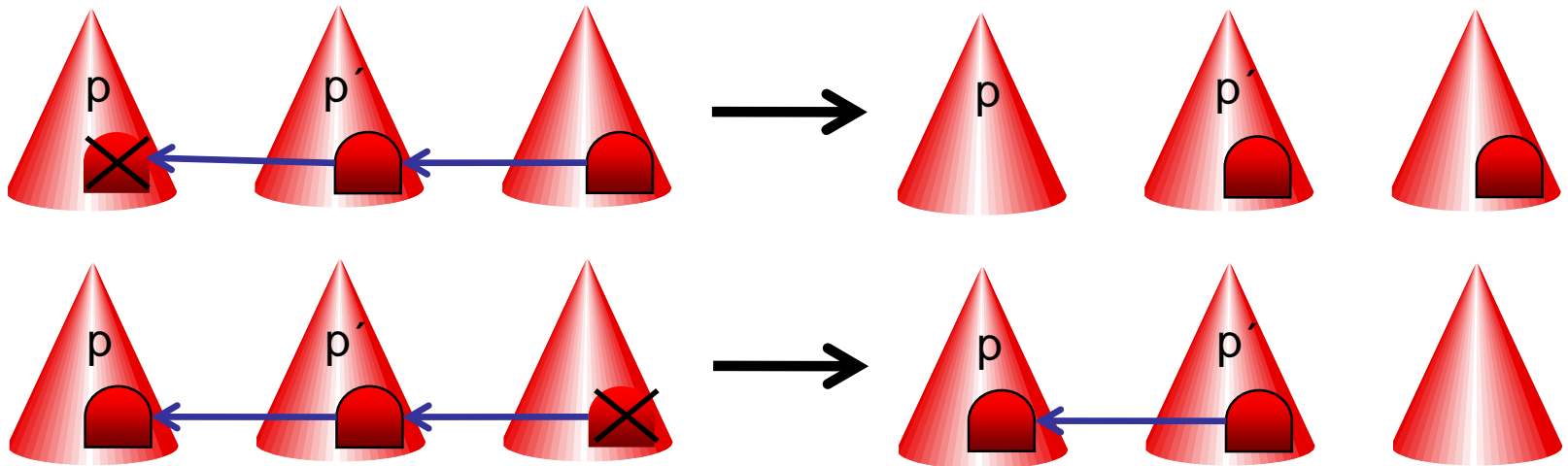
if **p** created **r**, then message sent to **r** is processed by **p**

having a relay in a relay tree with sink **r** allows one to send a message to **p**

TCM Model

- **delete r**: kills relay **r**, which has cascading effect on **incoming** links, enforced by the TCL (how to realize that is shown in the next chapter)

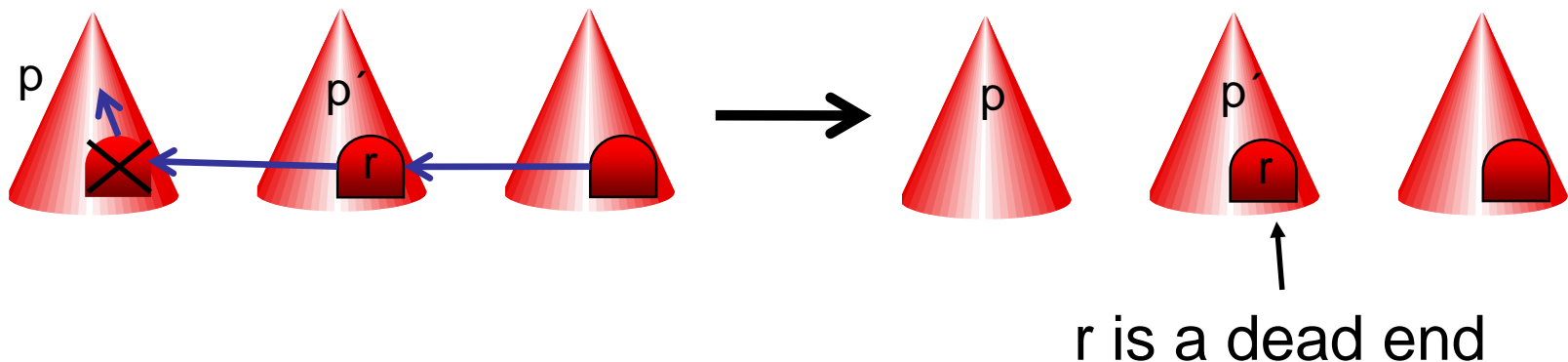
Examples:



TCM Model

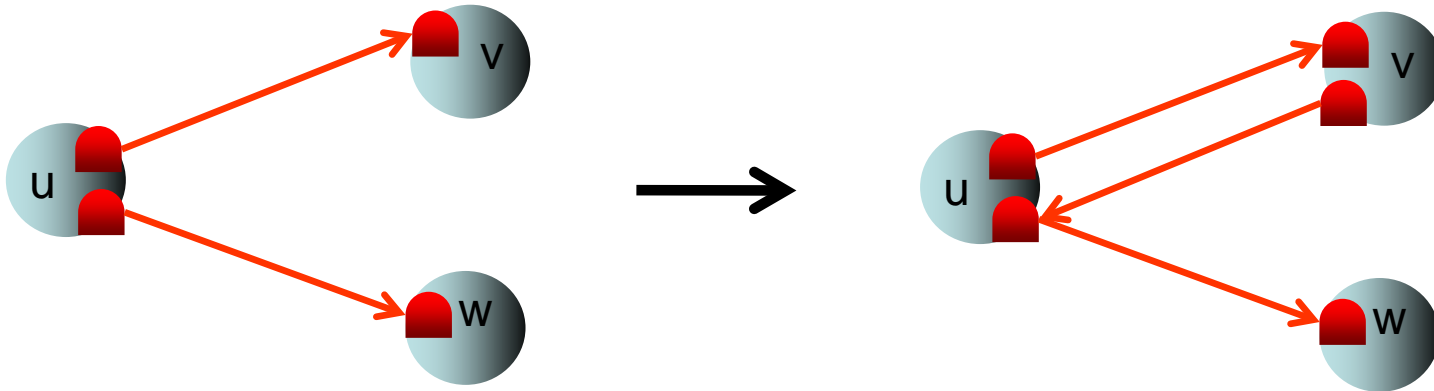
- `dead(r)`: returns `true` if `r` does not exist (because it has been deleted) or represents a dead end (i.e., it has no outgoing link any more, which can be due to a deleted relay or process)

Example:



TCM Model

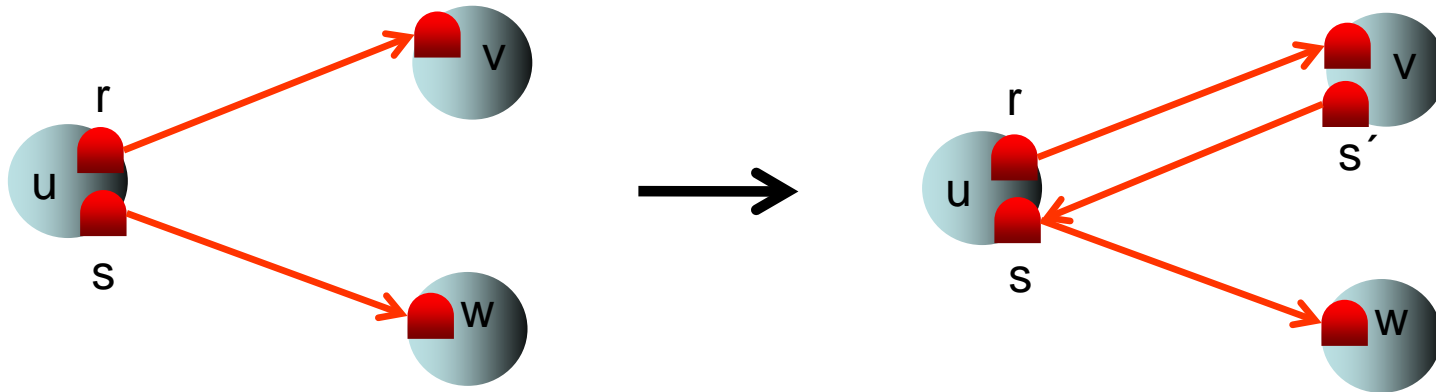
Recall the safe introduction rule:



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

TCM Model

Recall the safe introduction rule:

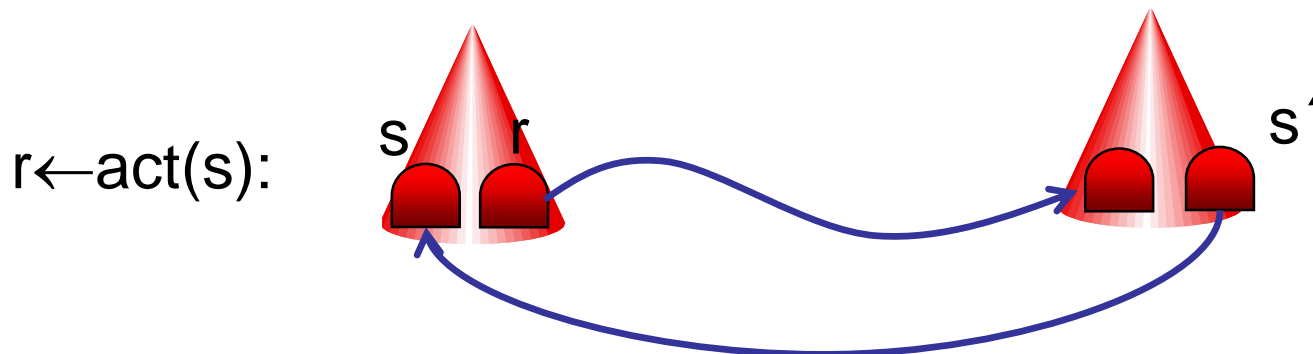


To safely introduce w , u calls $r \leftarrow \text{act}(s)$ for some action „ act “.

TCM Model

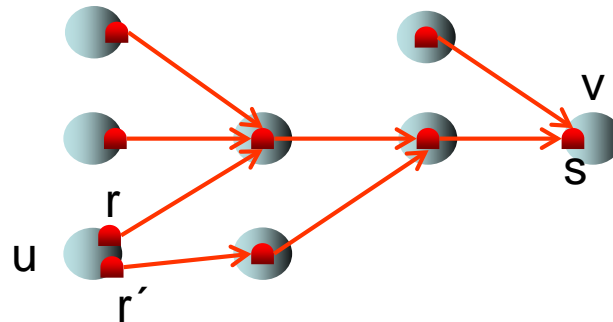
- Local call of an action:
 $\langle \text{name} \rangle (\langle \text{parameters} \rangle)$
Relays in $\langle \text{parameters} \rangle$ stay as they are.
- Remote call of an action:
 $\langle \text{relay} \rangle \leftarrow \langle \text{name} \rangle (\langle \text{parameters} \rangle)$
Transforms any relay r in $\langle \text{parameters} \rangle$ into local r' with connection to r when executing $\langle \text{name} \rangle$ in destination.

handled by TCL



TCM Model

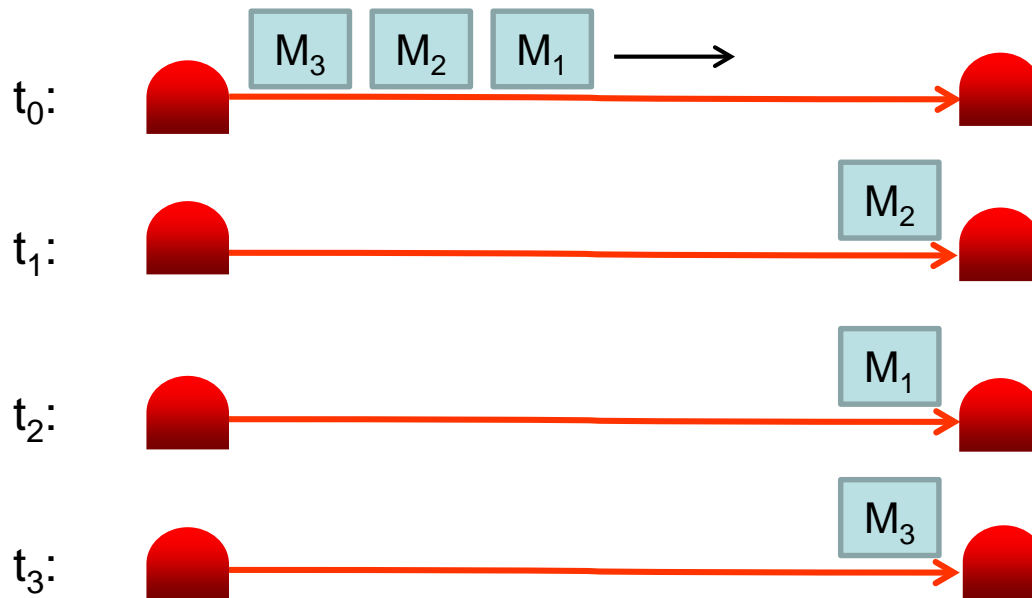
- r : (locally valid) ID or relay r
- $r.sink$: (locally valid) sink ID of relay r



- Test of equal sinks: $r.sink=r'.sink$
- Relays can be ordered by a process according to their IDs or their sink IDs so that a quick searching for equal sinks is possible.
- $r.incoming$: Boolean variable that is true if r has incoming connections
- $r.direct$: Boolean variable that is true if r has a direct connection (i.e., without intermediate relays) to a sink

TCM Model

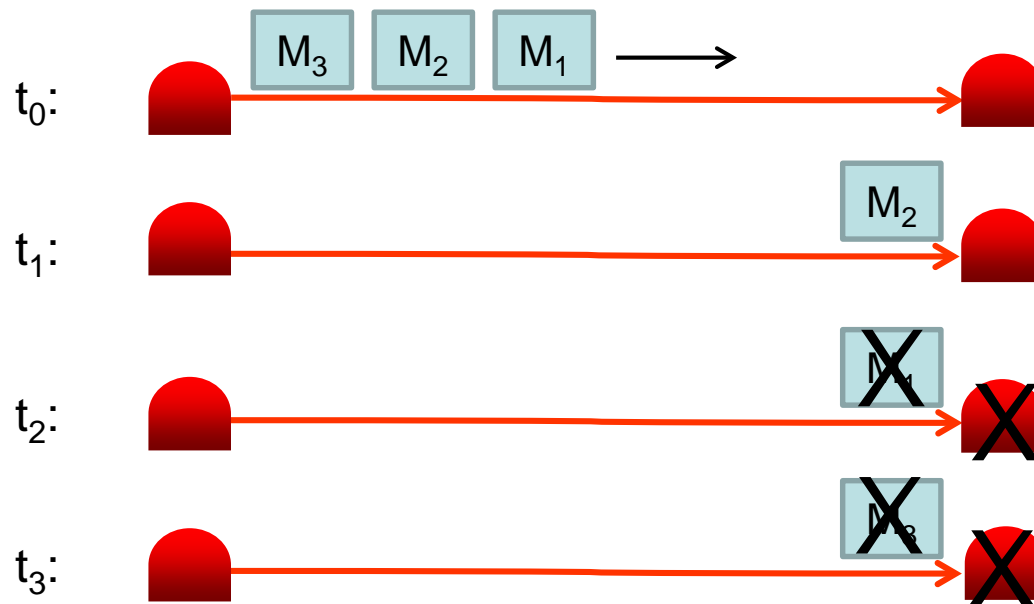
Asynchronous message passing



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

TCM Model

Asynchronous message passing



- Eventual delivery: guaranteed by TCP/IP
- **But links might fail due to process or TCL failures!**

TCM Model

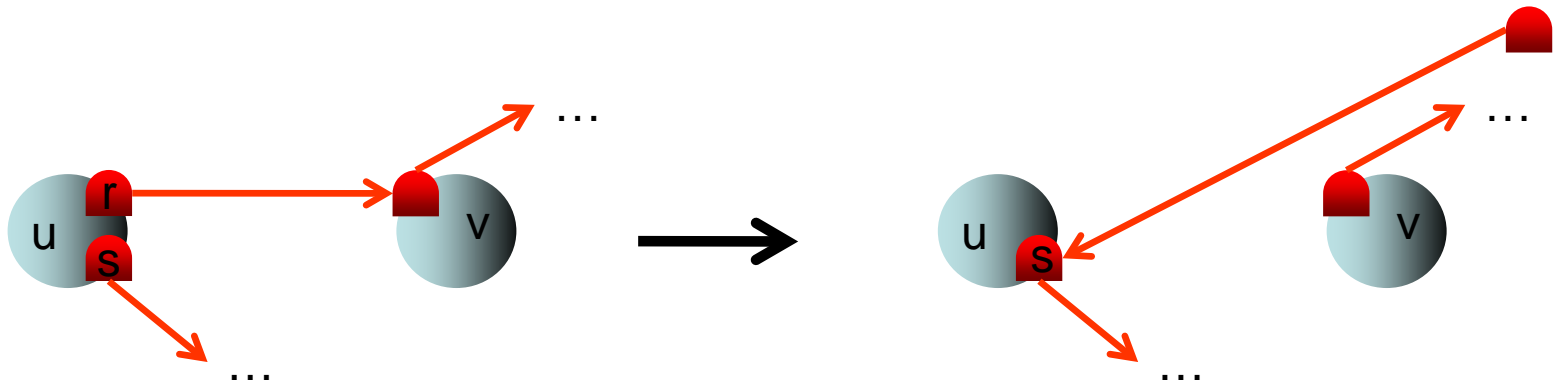
Asynchronous, faulty message passing:

- Messages are eventually delivered or fail
- If an acknowledgement is received for a message m , then m was correctly delivered. (However, due to our model it may take arbitrarily long for an acknowledgement of a successful transmission to be received.)

Are the primitives new, delete, and safe intro sufficient to support this model?

TCM Model

Recall the safe reversal rule:

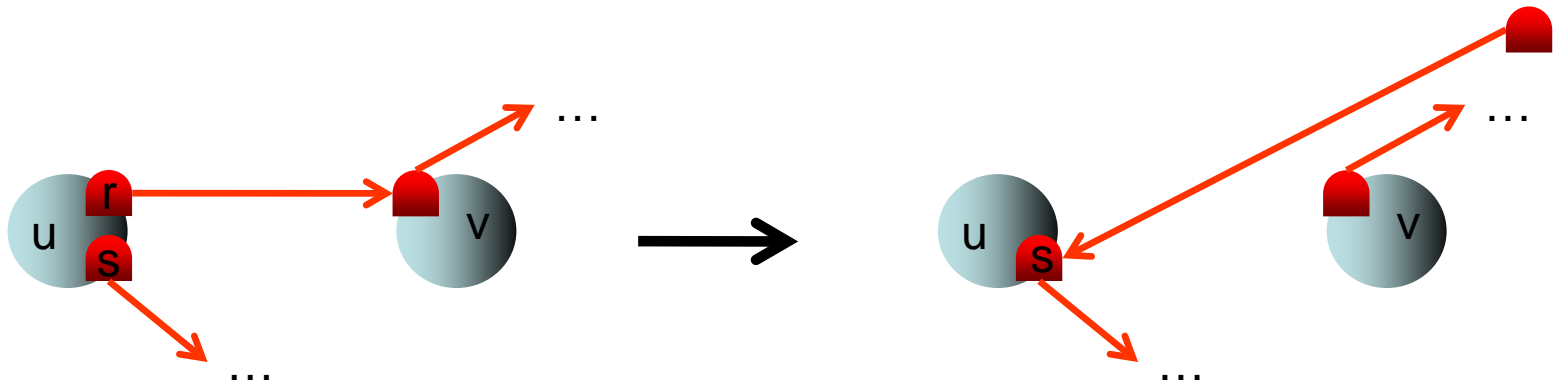


u safely introduces relay **s** to the process owning the sink relay of **r** and **deletes r**.

Does not preserve weak connectivity if message with reference to s is lost!

TCM Model

Recall the safe reversal rule:

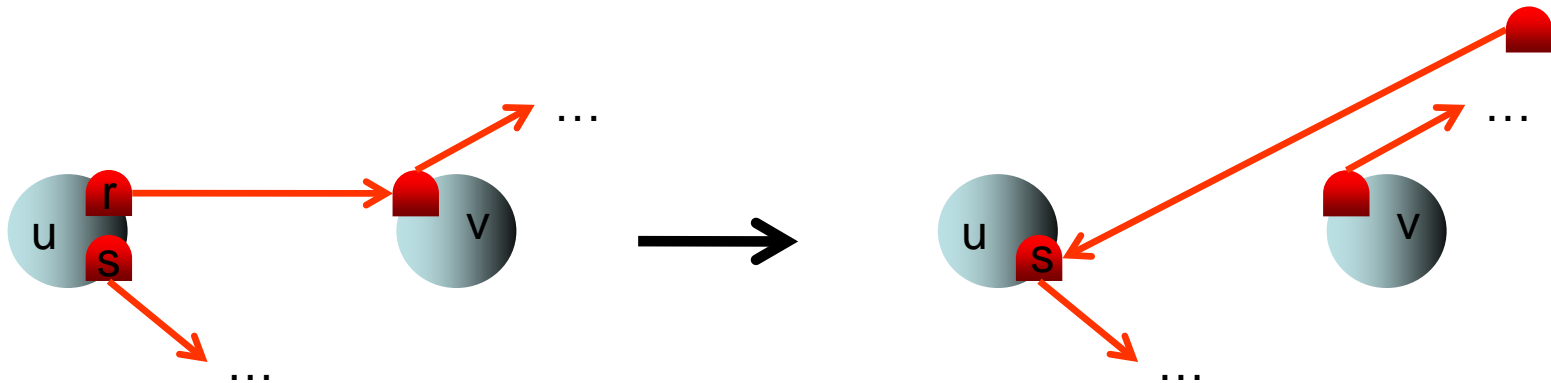


u safely introduces relay s to the process owning the sink relay of r and **deletes** r .

However, if message with s gets lost, then one of the processes from r to its sink must be broken (or must have a broken TCL).

TCM Model

Recall the safe reversal rule:

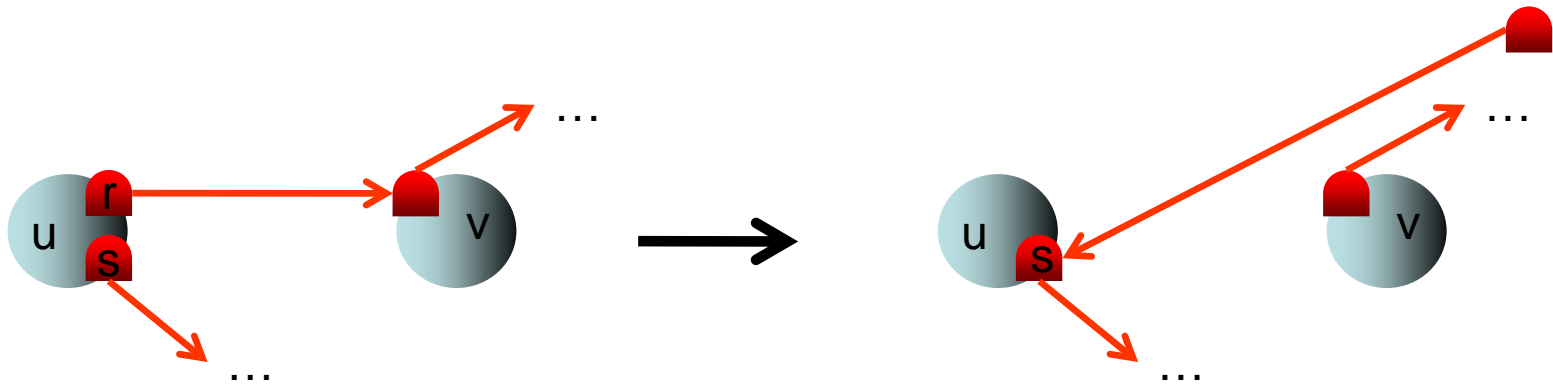


u safely introduces relay s to the process owning the sink relay of r and **deletes** r .

If we only need to deal with permanent failures, then r would not be of any use anymore, so its immediate deletion would be fine.

TCM Model

Recall the safe reversal rule:

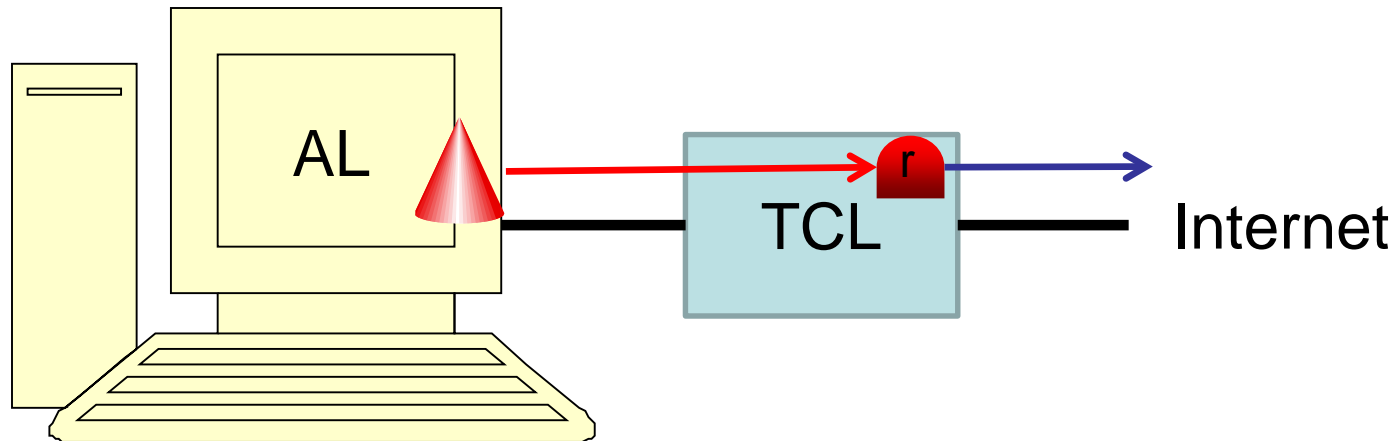


But what if failures are temporary?

1. TCL can recover: message can go on
 2. TCL needs to be reset: connection lost
- In both cases, the deletion of **r** is still fine.

TCM Model

Handling message failures for other kinds of info:



Use standard redundancy methods (in processes) and acknowledgements so that lost messages do not cause lost information.

Overview

- TCM model
- Pseudo-code and example programs
- Programming environment

TCM Model

Pseudo code like in object-oriented programming:

```
Subject <Name>: // declares process type
  local variables
  actions
```

Types of actions:

```
<ActionName>(<Parameters>) →
  commands in pseudo code
```

```
<ActionName>: <Predicate> →
  commands in pseudo code
```

Special action:

```
init(<Parameters>) → // constructor
  commands in pseudo code
```

Pseudocode

- Assignment via `:=`
- Loops (for, while, repeat)
- Conditional branching (if – then – else)
- Comment via `{ }`
- Block structure via indentation
- Call of action via relay: `r←act(...)`
- Relay: stores reference to a relay (empty reference: \perp)
- Creation of new processes or relays: `new`
(`new` calls `init` in process)

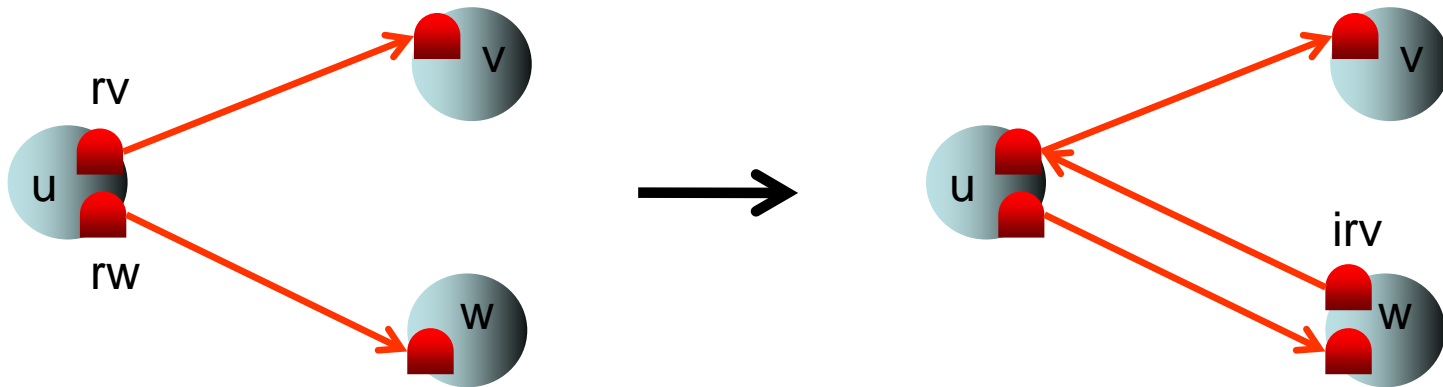
Pseudocode

Examples:

- introduction
- delegation
- broadcast service
- self-stabilizing sorted list

Introduction

Step 1:

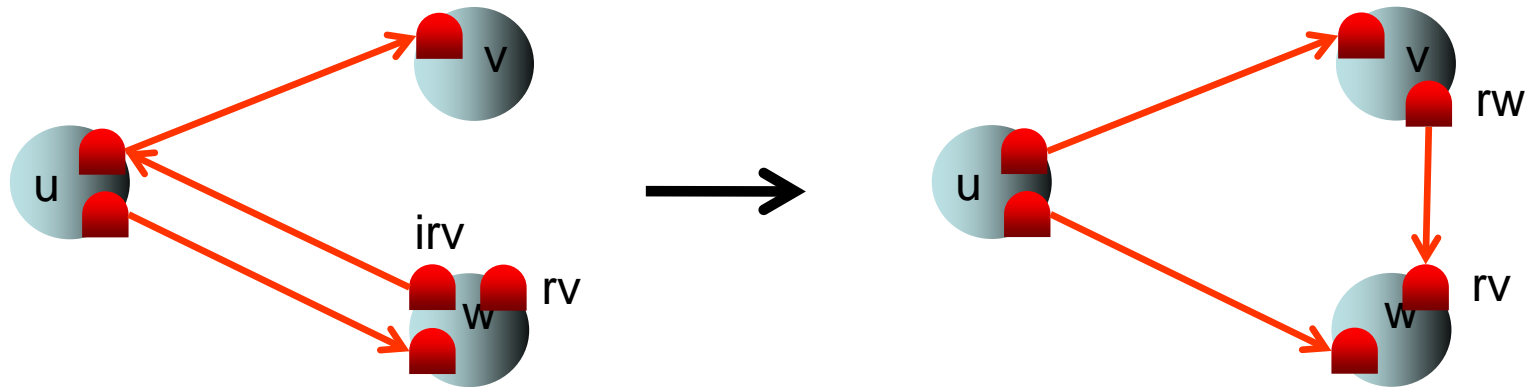


In node u :

$rw \leftarrow \text{ask-for-intro}(rv)$

Introduction

Step 2:

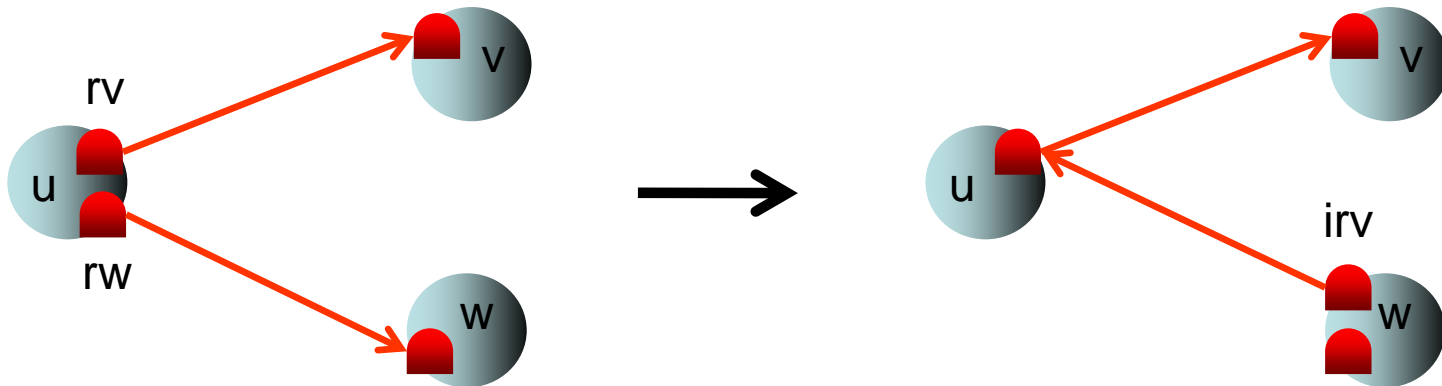


In node w :

```
ask-for-intro(irv) →  
rv := new Relay  
irv ← introduce(rv)  
delete irv
```

Delegation

Step 1:

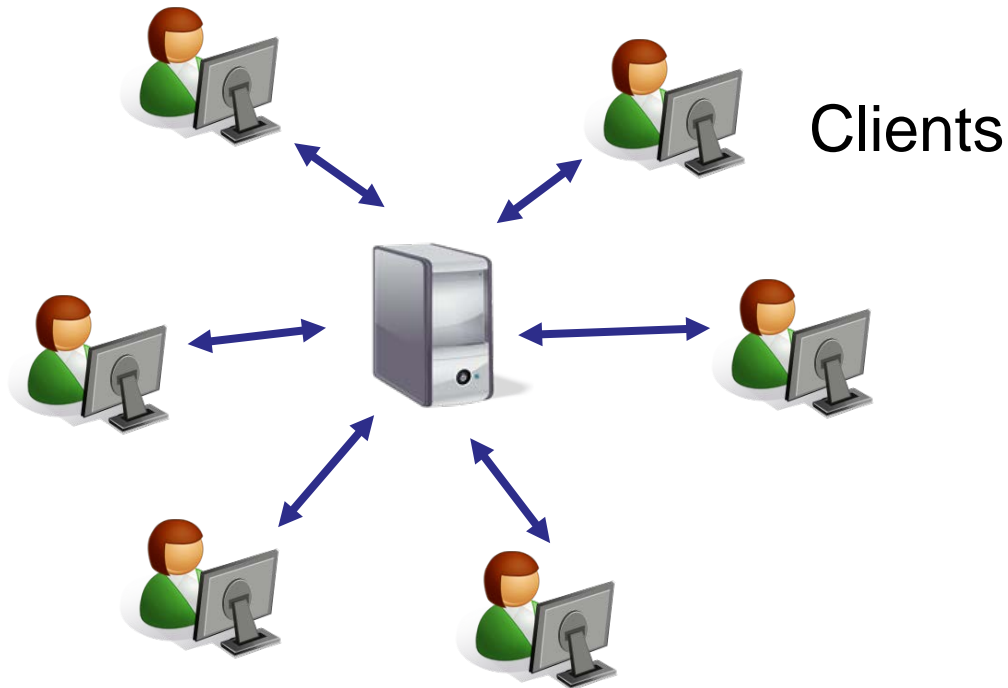


In node u :

```
rw ← ask-for-intro(rv)  
delete rw
```

Broadcast Service

Simple broadcast service via server



Broadcast Service

Subject Server:

n: Integer { stores number of clients }
toServer: Relay { relay for clients to connect to }
toClient: Array[1..MAX] of Relay { stores refs to clients }

init() → { constructor }

n:=0

toServer:=new Relay { sets up relay for clients }

register(C) → { register new client with reference C }

n:=n+1

toClient[n]:=C

broadcast(M) → { send M to all clients }

for i:=1 to n do

M' := new Object(M) { new object containing M }

toClient[i] ← output(M')

Broadcast Service

Subject Client:

toServer, toClient: Relay

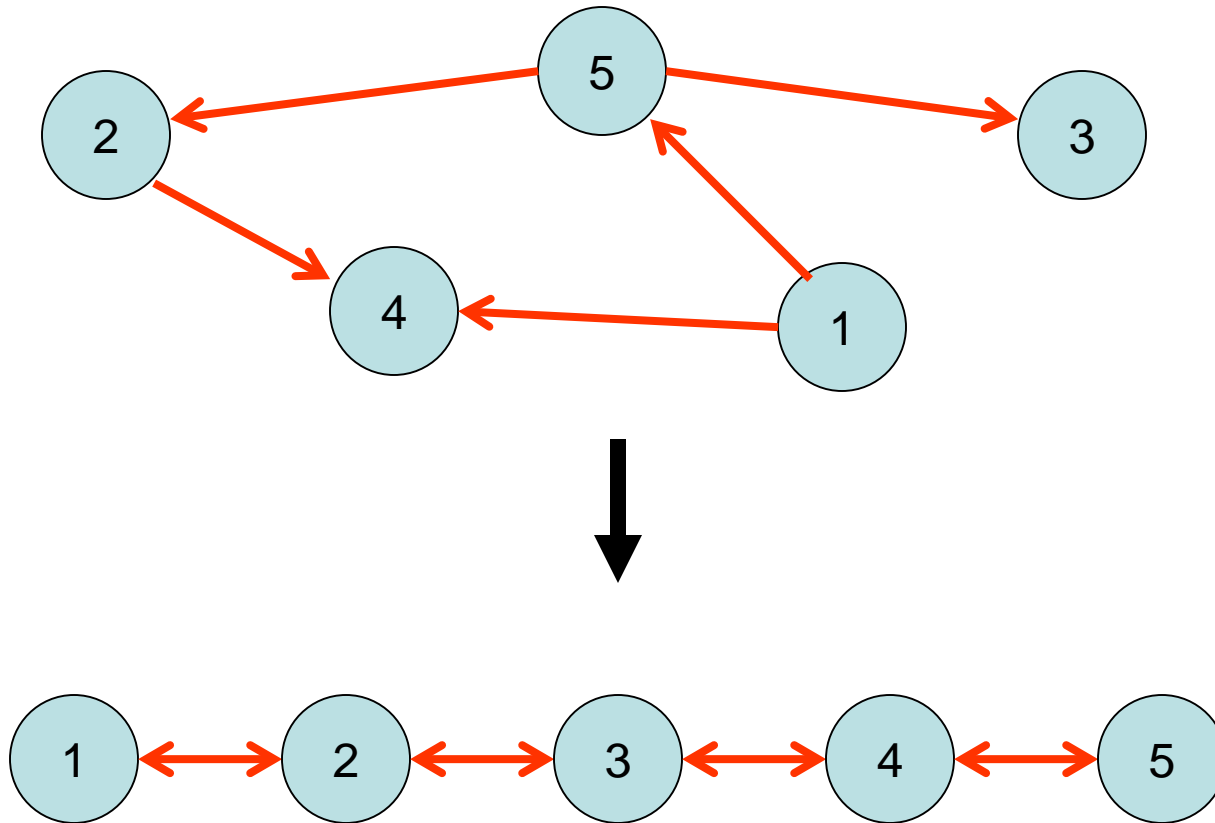
```
init(S) → { constructor }  
toServer:=S { S: reference to server }  
toClient:=new Relay  
toServer←register(toClient)  
{ send ref. to client to server }
```

```
broadcast(M) → { broadcast M via server }  
toServer←broadcast(M)
```

```
output(M) → { output M }  
print M
```

Self-stabilizing Sorted List

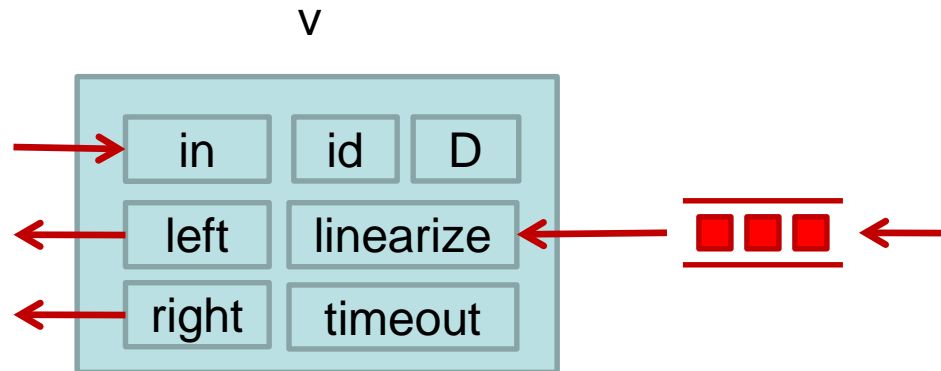
Goal:



Self-stabilizing Sorted List

Variables in a node v :

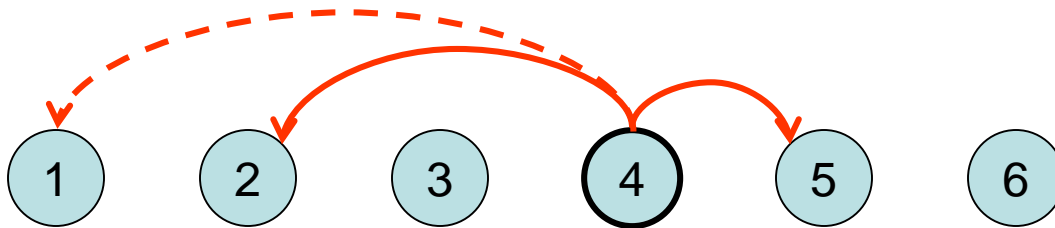
- $v.id$: ID of v
- $v.in$: incoming relay of v
- $v.left \in V \cup \{\perp\}$: relay to closest left neighbor of v
($l.sink$: sink of $left$, $left.id$: ID of $left$, $left.id < v.id$)
- $v.right \in V \cup \{\perp\}$: relay to closest right neighbor of v
- $v.D$: set of relays to to-be-delegated neighbors of v



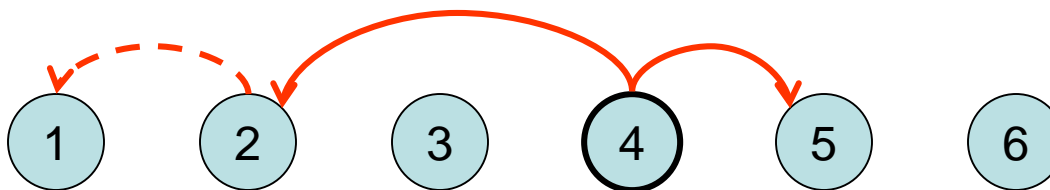
Self-stabilizing Sorted List

Linearize action:

Idea: keep edges to **closest** neighbors and delegate rest.



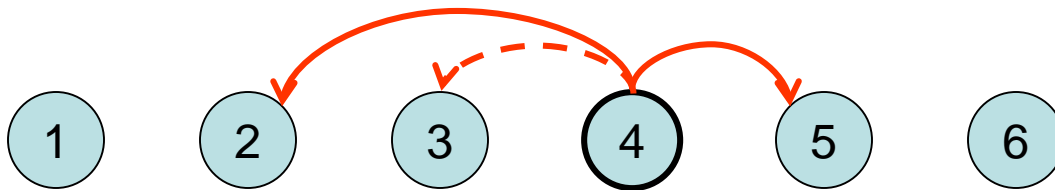
Upon `linearize(1)`: 4 delegates 1 to 2



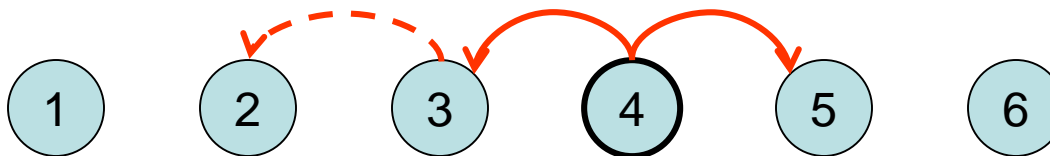
Self-stabilizing Sorted List

Linearize action:

Idea: keep edges to **closest** neighbors and delegate rest.



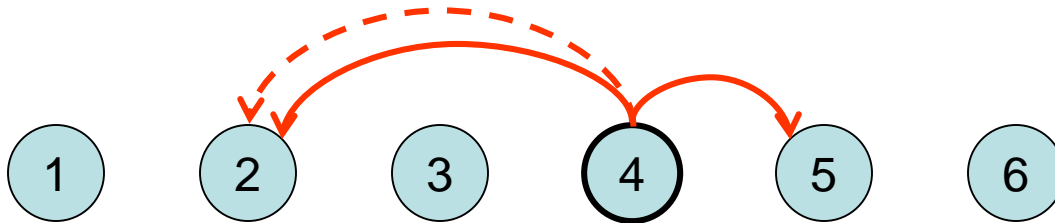
Upon `linearize(3)`: 4 sets `4.left:=3` and **delegates** 2 to 3



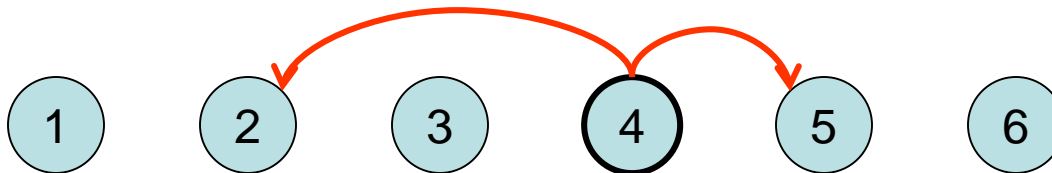
Self-stabilizing Sorted List

Linearize action:

Idea: keep edges to **closest** neighbors and delegate rest.



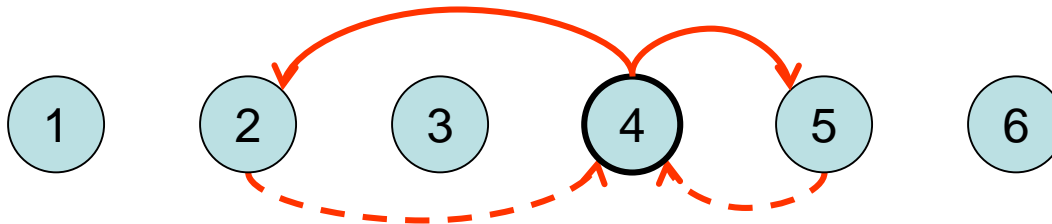
Upon `linearize(2)` oder `linearize(5)`: drop reference.



Self-stabilizing Sorted List

Periodically, every node also executes a timeout action.

Upon timeout, 4 introduces itself to 2 and 5.



Theorem 5.1: Linearize and timeout guarantee that we have a self-stabilizing sorted list.

Proof: see BA lecture.

Self-stabilizing Sorted List

Simplifying assumptions:

- If $\text{left}=\perp$, then we assume for comparisons that $\text{left.id}=-\infty$.
- If $\text{right}=\perp$, then we assume for comparisons that $\text{right.id}=\infty$.
- A call $r\leftarrow\text{act}(s)$ is only executed if r and s are not equal to \perp .

Remark: We need D because in certain cases a node v is not interested any more in some relay r , either because it is not a direct connection to some sink relay, or it leads to a node that is further away than its current left or right neighbor. We cannot directly delete these relays because they may still have incoming connections due to some previous safe introduction or delegation, so we have to wait until all of its incoming connections are closed before deleting it.

Self-stabilizing Sorted List

Subject Sorted_List:

id: Integer

left, right, in: Relay

D: Set of Relay

Correctly set left and right
($u.\text{left.id} < u.\text{id}$ and $u.\text{right.id} > u.\text{id}$):

timeout: true \rightarrow

{ executed by node u }

if $\text{left.id} \geq \text{id}$ or $\text{left.sink} = \text{in}$ or not left.direct then

$D := D \cup \{\text{left}\}$; $\text{left} := \perp$

else

$\text{left} \leftarrow \text{linearize}(\text{in})$

if $\text{right.id} \leq \text{id}$ or $\text{right.sink} = \text{in}$ or not right.direct then

$D := D \cup \{\text{right}\}$; $\text{right} := \perp$

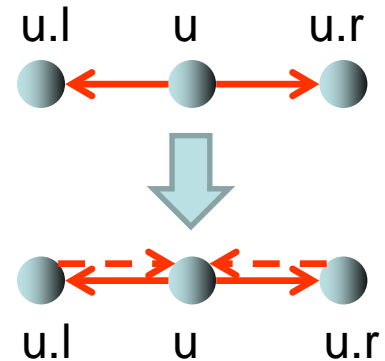
else

$\text{right} \leftarrow \text{linearize}(\text{in})$

for all $v \in D$ with $v.\text{incoming} = \text{false}$ do

if $v.\text{sink} \neq \text{in}$ then $\text{linearize}(v)$

else delete v

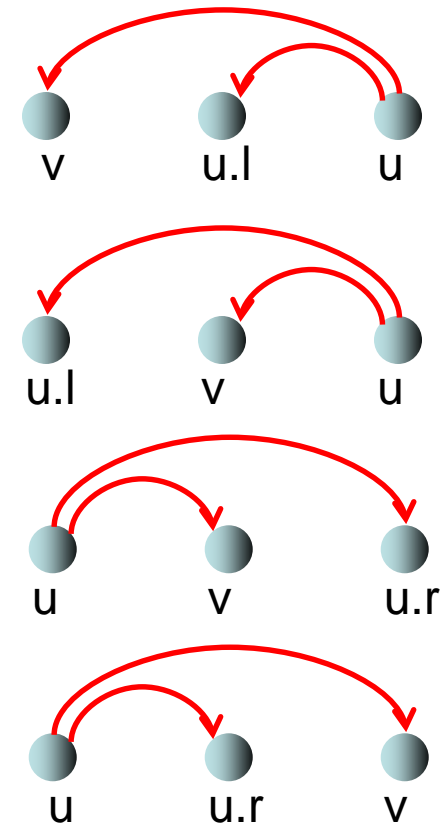


u.l: short form of u.left
u.r: short form of u.right

Self-stabilizing Sorted List

```

linearize(v) →
  { executed by node u }
  if v.sink ∈ {in, left.sink, right.sink} then delete v
  else
    if v.id < left.id then
      v ← ask-for-lin(left) { safe reversal }
      delete v
    if left.id < v.id < id then
      left ← ask-for-lin(v) { safe reversal }
      D := D ∪ {left} { might have inc. conn.! }
      left := v
    if id < v.id < right.id then
      right ← ask-for-lin(v)
      D := D ∪ {right} { might have inc. conn.! }
      right := v
    if right.id < v.id then
      v ← ask-for-lin(right)
      delete v
  
```



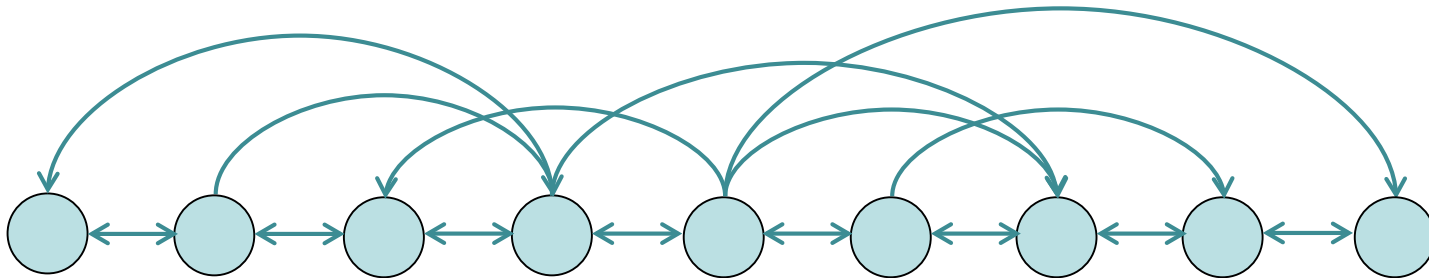
Self-stabilizing Sorted List

ask-for-lin(v) \rightarrow
 { executed by node u }
 $v \leftarrow \text{linearize}(\text{in})$ { safe reversal }
 delete v

Problem: A sorted list is not a very robust structure.

Solutions:

- keep old connections (\rightarrow multilist, see BA lecture)



- construct a clique (Ch. 9)

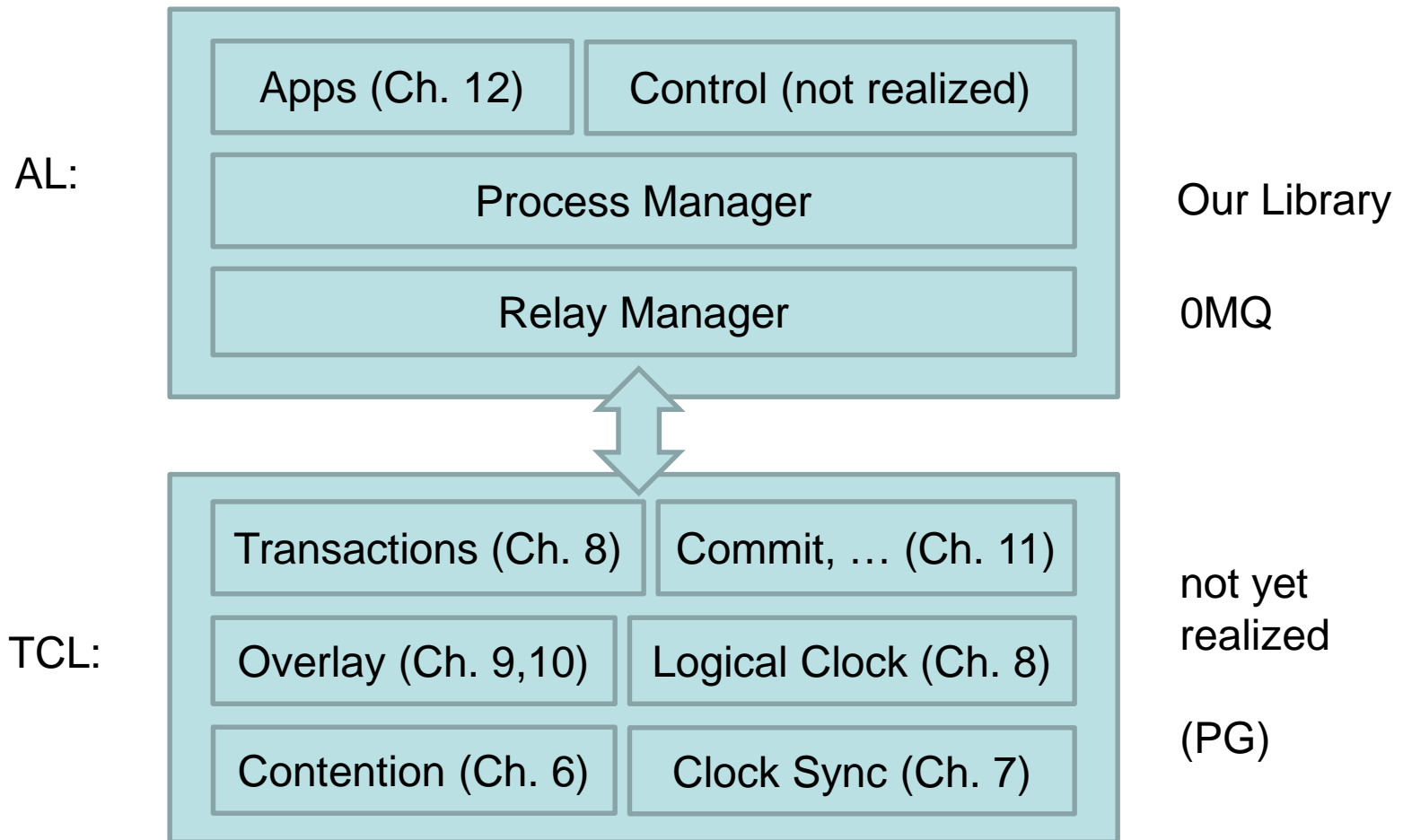
Overview

- TCM model
- Pseudo-code and example programs
- **Programming environment**

Programming Environment

- **Origin:** Hewitt's Actor Model (1973) for neural networks
- Since then, a lot of work in the area of **programming languages** (E, Scala,...)
- We will use an extension based on C++ and OMQ to implement the pseudo code.

Programming Environment



Programming Environment

Next, we present an example program realizing ping-pong communication between two processes. More information on the environment will be given in the tutorial next week.

Programming Environment

```
#define BOOST_TEST_DYN_LINK
#define BOOST_TEST_MODULE ApplicationTests
#include <boost/test/unit_test.hpp>
#include <relaymq.h>

#include "../tests/messages/addressbook.pb.h"

#include "../include/Relay.h"
#include "../include/Socket.h"
#include "../include/ApplicationContext.h"
#include "../include/Subject.h"

class PingPongSubject: public Subject{

public:
    PingPongSubject(int limit):
        limit(limit),
        curr(0){
    }

private:
    int limit;
    int curr;
    RelayRef theOtherGuy;
```

Programming Environment

```
bool onTimeout(){
    return (curr < limit);
}

virtual bool onMessage(RelayRef receiver, std::string msg, std::vector<RelayRef> refs){
    if(msg == "INIT"){//Aus der MAIN
        theOtherGuy = refs[0];
        bool ok = theOtherGuy->Send("INTRODUCE", {receiver});
        assert(ok);
    }else
    if(msg == "INTRODUCE"){//Vom anderen Subject
        theOtherGuy = refs[0];
        theOtherGuy->Send("PING",{});
    } else
    if(msg == "PING"){
        puts(msg.c_str());
        theOtherGuy->Send("PONG", {});
        curr++;
    }
    else
    if(msg == "PONG"){
        puts(msg.c_str());
        theOtherGuy->Send("PING", {});
        curr++;
    }
    return true;
}

};
```

Programming Environment

```
BOOST_AUTO_TEST_CASE(ping_pong_test){
    puts("===== ping_pong_test() =====");

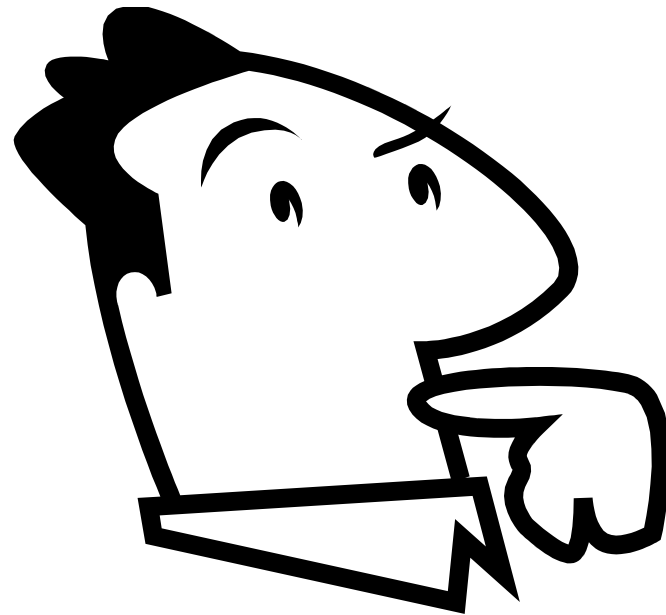
    /*Boilerplate-Code that sets up everything in the background */
    ApplicationContext::Init();

    /* Every subject is supposed to send 1000 messages */
    int limit = 1000;

    /* creates a subject and sets limit */
    auto ping = ApplicationContext::Create<PingPongSubject>(limit);
    auto pong = ApplicationContext::Create<PingPongSubject>(limit);

    /*
     * sends message and reference to the subject
     */
    pong->Send("INIT", {ping});

    /*
     * Starts application:
     * 1) The main method blocks till all subjects are deleted
     * 2) The delivery of messages starts
     */
    ApplicationContext::Start();
}
```



Questions?