THE ADVENTURES OF SELFMAN

KOMPICS™

Reactive Component Model for Distributed Computing

SELFMAN project

Cosmin Arad and Seif Haridi
KTH SICS

Collaborator: Roberto Roverso

ZIB Berlin, May 22nd 2008

Goals

- To make programming distributed systems an easy and painless job

- To enable the implementation of distributed systems in a way that reflects their nature
  - Concurrent activities
  - Reactive behavior
  - Complex interaction

Context

- Distributed protocols are inherently reactive
  - Broadcast
  - Gossip
  - Consensus
  - Distributed shared memory
  - Group membership
  - Overlay networks
- Distributed protocols should be reusable and compositional

Contribution

- A model for specifying and implementing distributed protocols as *components* that are…
  - Concurrent
  - Compositional
  - Event driven
  - Dynamically reconfigurable
  - Software fault-tolerant
  - Multi-core scalable

Contribution

- An implementation of Kompics in Java

- A set of reusable component libraries for distributed applications:
  - Overlay networks
  - Reliable group communication
  - Gossip based systems

Outline

- Model overview
- Concepts semantics
- Examples and Java implementation demo
- Outlook

Model overview

- The Kompics model is a concurrent extension of any sequential strongly typed programming language having the notion of types and subtypes
- We have a first implementation in Java
- We introduce the model entities using object oriented programming terminology

Concepts and artifacts

- Component
- Event
- Channel
- Event Handler
- Factory
- Component Type
- Component Membrane

Components…

- … are active objects
- … interact with each other by triggering asynchronous events
- … react to events by executing event specific procedures to handle the received events
- … are decoupled (through channels)
- … publish events into channels
- … subscribe to channels

Events…

- … are passive immutable objects
- … are typed and event types can form type hierarchies
- … are triggered by components and published into channels

Channels…

- … are interaction links between components
- … carry events from publishers to subscribers
- … are associated with a set of event types
- Components subscribe to a channel for a certain event type
- Components can only publish into a channel events of a type that is associated with the channel

An event handler…

- … is a component method that handles events
- … is executed as a reaction to an event
- … takes as argument an event of a certain type
- … can be guarded by a boolean guard
- … can trigger events

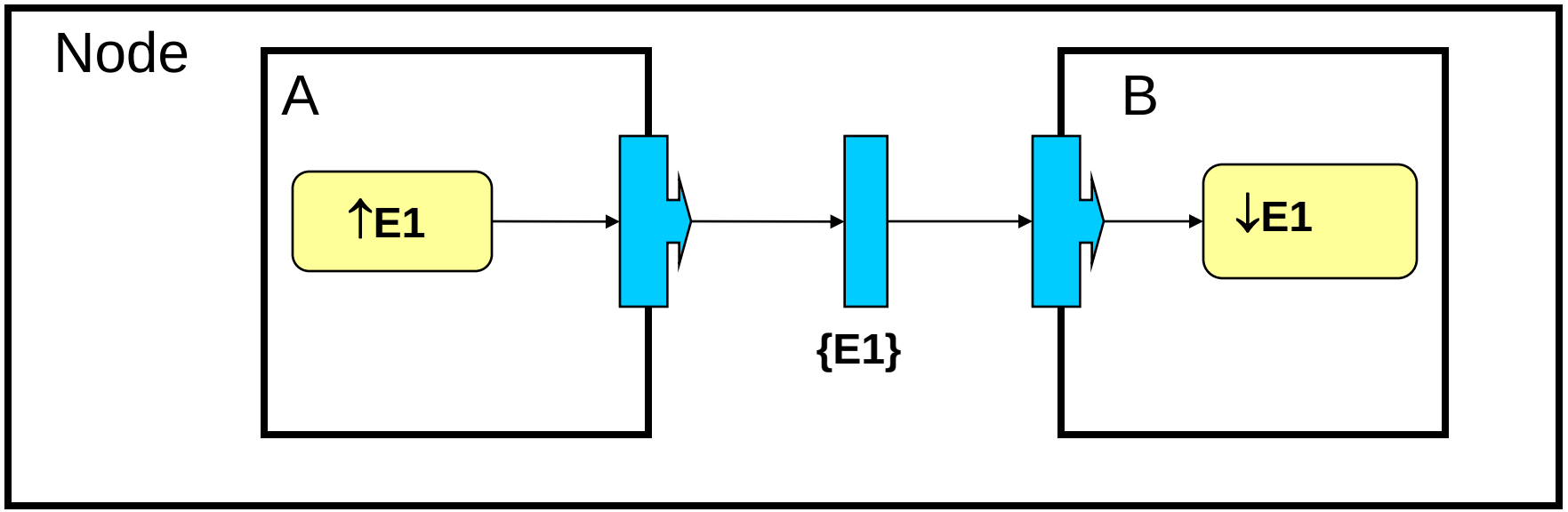
Component type CT

- Components interact with its environment through input and output channels
- CT represents the signature of a component
- CT specifies the set of *input* event types of any component that implements it
- CT specify the set of *output* event types that are *triggered* by any component that implements it

A Component Membrane…

- … is the collection of channels through which a component *instance a* interacts with its environment
- … maps every pair of (event type, in/out) in *a*'s signature (component type) to an actual channel *x*
- *a* is either subscribed to *x* (in), or *a* publishes events into *x* (out)

Example



↑E1 Event handler that triggers events of type E1

↓E1 Event handler that handles events of type E1

Component output channel parameter

Component input channel parameter

{E1} Channel carrying events of type E1

Factories…

- … are artifacts that take a component specification (class) and create components (instances)
- … create and initialize component instances
- … allow the creation of multiple component instances from the same component specification
- There is one factory for each component spec

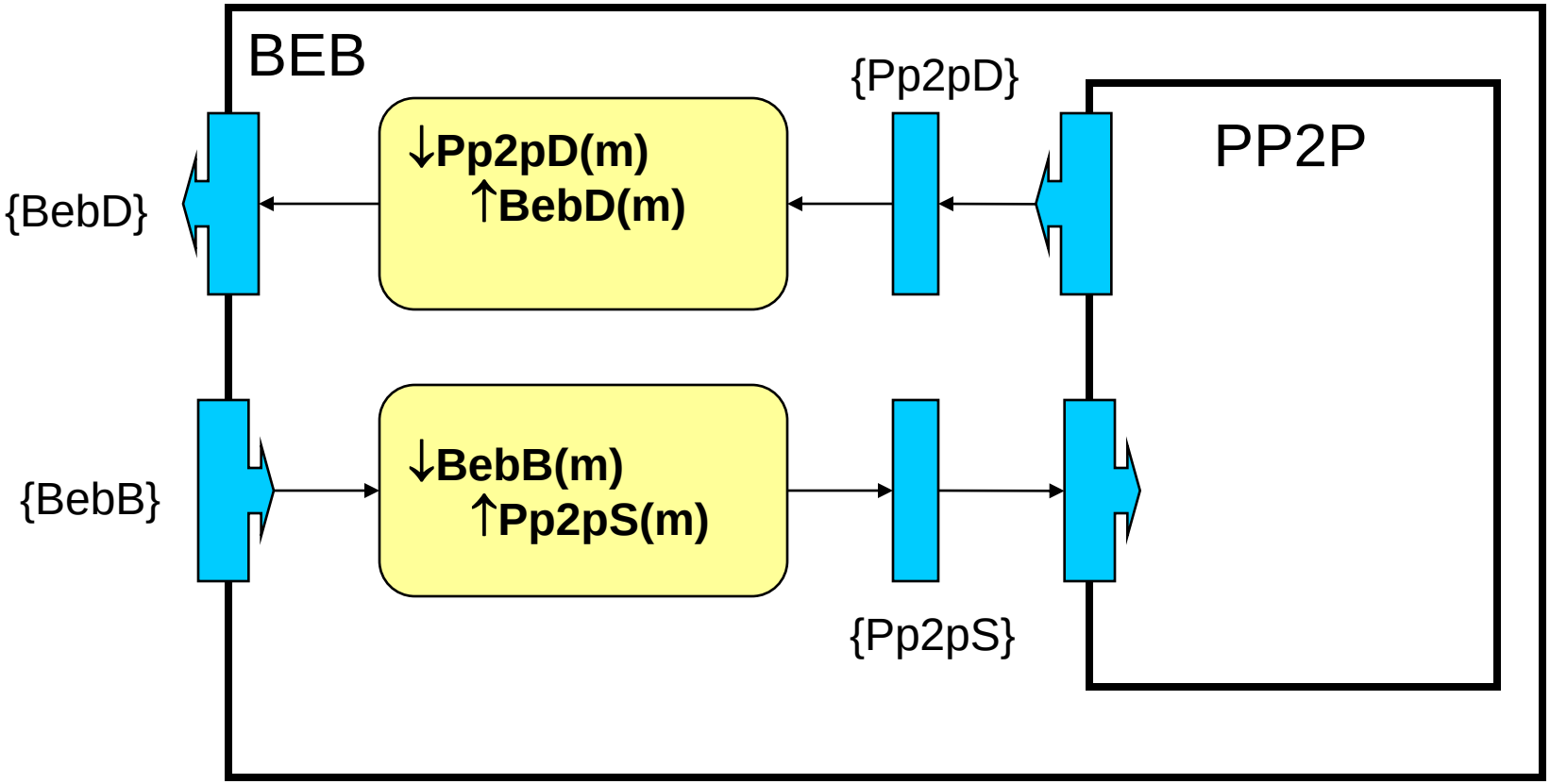
Composite components…

- … are components
- … encapsulate components and local channels in addition to state variables and event handlers
- … interact with their environment through input and output channels
- … interact with subcomponents through the local channels

Best Effort Broadcast (BEB) component

BebB(roadcast)
BebD(eliver)

Pp2pS(end)
Pp2pD(eliver)



Component abstract syntax example

component CompName {
    declare
        state variables
    create(**channel** $param_1$ … $param_n$) **do** {…}
    init(**any** $param_1$ … $param_m$) **do** { … }
    onEvent EventType1 e **do** {
        …
        trigger new EventType2(e)
        …
    }
    onEvent EventType3 e **do** { … }
    … /* private methods */
}

BEB Component example

```
component BEB {
    declare  ps /* the process group */
    …
    create(channel in, channel out) do {
        subscribe(in, BebB)
        Channel pp2pIn = new Channel({Pp2pS})
        Channel pp2pOut = new Channel({Pp2pD})
        Factory pp2pFactory = new Factory(PP2P)
        Component pp2p = pp2pFactory.create(pp2pIn, pp2pOut)
    }
    onEvent BebB b do {
        for all processes p in ps do
            trigger new Pp2pS(b.message)
    }
    ….
}
```

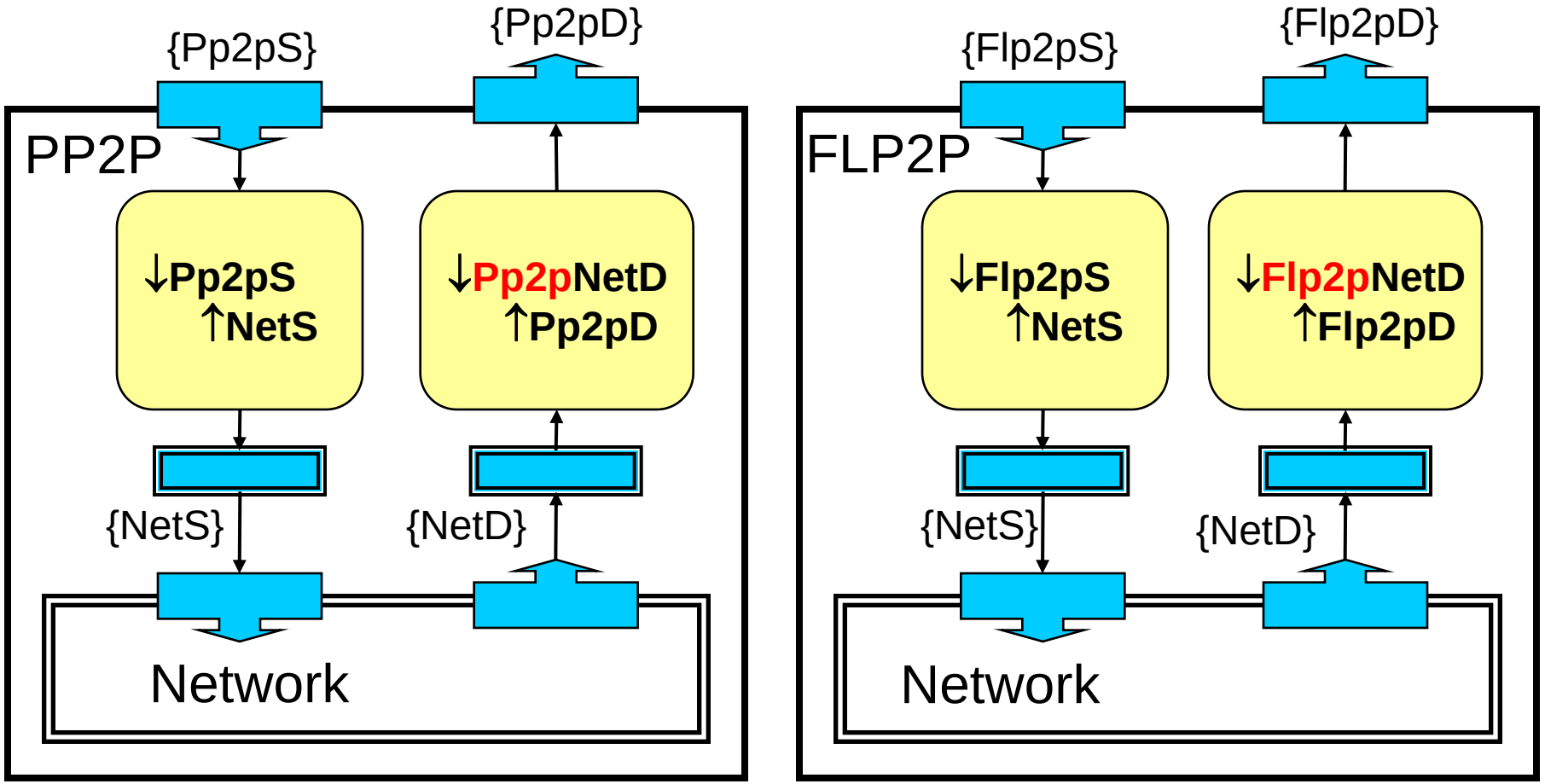
Event type hierarchy

```
event EventType1 {
   declare attributes;
   …
   /* public selector methods */
}

event EventType2 extends EventType1 {
   declare attributes;
   …
   /* public selector methods */
}
```

Sharing

- Subcomponents can be shared by multiple composite components
- Sharing a component is done by sharing the channels in the component's membrane
- To share a component *a*, a component membrane is created for *a* and registered under a name in a registry of shared components
- A composite component *b* that wants to use *a*, retrieves *a*'s membrane from the registry and subscribes to or publishes events into its channels

PP2P and FLP2P share the Network component



shared channel in shared component membrane

shared component

Flp2pNetD ⊂ NetD

Pp2pNetD ⊂ NetD

Sharing patterns

- Sharing a component *a* among different types of components (static sharing)
  - User components subscribe to subtypes of the event types published by *a*
  - Events filtering done by subtyping
- Sharing a component *a* among different instances of the same component type (instance sharing)
  - Events published by *a* have an attribute that identifies the target component
  - Events filtering done by demultiplexing on the identifier

Outline

- Model overview
- Concepts semantics
- Examples and Java implementation demo
- Outlook

Publish-subscribe semantics

- A component can publish into a channel only events that have the same type or a subtype of one of the channel event types

- A component can subscribe to a channel for events of the same type or of a subtype of one of the channel event types

Execution semantics

- The system contains *n* workers that execute event handlers on behalf of components
- Different component instances can execute event handlers in parallel
- The event handlers of one component instance are guaranteed to be executed sequentially by the execution model
- The execution of a handler h_1 of component c_1 is not atomic w.r.t to any other handler exec

Channel properties

- Channels are FIFO
  - each component subscribed to a channel *C*, executes events published in *C*, in the same order in which they are published
- A channel serializes the concurrent publication of events into the channel
- Events triggered sequentially by one component instance will be published in the channel in the order in which they were triggered

Subscription semantics

- Component *a* subscribes an event handler *h* for events of type *T* to channel *x*
  - A subscription (*a*, *T*, *h*) is registered at *x*
  - A FIFO work queue q_x is created at *a* for *x* (if it is not already created by a previous sub)
- Multiple handlers in the same component can subscribe to the same channel

Scheduling: component state

- A component *a* can be in one of three states:
  - **Busy**: a worker is currently executing an event handler of *a*
  - **Ready**: one or more of its queues q_x are not empty and the component is not busy
  - **Idle**: all its queues q_x are empty and it is not busy

Event triggering semantics

- Component *a* triggers event *e* of type *T* in channel *x*
- Let *S* be the subset of all subscriptions (*b*, *T'*, *h*) to *x*, where *T'* is *T* or a super-type of *T*
- For each (*b*, *T'*, *h*) in *S*
  - A work item (*e*, *h*) is enqueued at *b* in q_x
  - If *b* is idle then *b* becomes ready
- Free workers pick ready components (thus making them busy)

Worker picks a ready component *a*

- It makes *a* busy
- *a* has at least one work queue q_x that is not empty
- Dequeue one work item (*e*, *h*) from q_x
- Execute *a*.*h*(*e*)
- If all *a*'s work queues are empty make *a* idle
- Otherwise make *a* ready

Worker loop (Summary)

- Workers wait for components to become ready
- When a component *a* becomes ready, a worker *w* picks it and executes one work item (*e*, *h*) enqueued in some q_x at *a*
- The execution of handler *h*, may trigger new events e_i of type T_i, published in channels x_i
- Components subscribed to x_i for type T_i become ready if not busy
- Upon termination of *h*, worker *w* repeats the above steps

Multicore execution

- The number of workers can be proportional to the number of cores
- The semantics described guarantees serialization of handler in same component instance and FIFO execution of events in the same channel
- Locking is needed on channels and component queues q_x (no component locking)
- Initial results shows good multicore scalability

Fault isolation

- Every component gets an associated control channel when it is created
- Any error/exception that is not caught within an event handler triggers a fault event published into the control channel
- The control channel is available to the parent component
- This allow various fault supervision hierarchies

Dynamic reconfiguration

- Means that the software architecture can be changed at runtime
  - Components at any level
  - Channels and associated event types
  - Subscriptions to channels

Primitives of reconfiguration

- Components
  - Create components and factories
  - Destroy …
  - Subscribe/Unsubscribe an event handler from a channel
- Channels
  - Create
  - Modify the set of associated event types
  - Destroy

Security

- Mechanism for Principle Of Least Authority
- References to components and channels embed *revocable* capabilities (caretaker pattern)
  - Component capabilities
    - (un)subscribe, share, reconfigure, trigger events, create channel, create factory, etc.
  - Channel capabilities
    - publish, (un)subscribe, change associated event types

Resource control

- Computing resources can be reallocated to favor hot components
- Worker pools of various sizes can be created
- Each component is member of one worker pool at any one time and it shares the workers in the pool with the other member components
- A component can be moved from one worker pool to another

Threaded components

- Long running event handlers that execute RPCs need a blocking **receive** primitive
- A threaded component is a component that has an associated thread that executes its event handlers (does not use worker pool)
  - **Event** e **= receive**()
  - **Event** e **= receive**(EventType…, Channel…)
  - **Event** e **= receive**(Channel…)
  - **Event** e **= receive**(EventType, Channel…)
  - **Event** e **= receive**(EventType, Guard, Channel…)

Threaded components

- Threaded components look just like normal components to the environment
  - They publish/subscribe for events
- When a threaded component *a* executes a **receive**, the associated thread first checks the queues q_x of *a*, for an eligible event to return
- If no eligible event is found, it blocks
- Whenever a newly published eligible event is enqueued at *a*, the thread unblocks and returns it to the **receive** caller

Outline

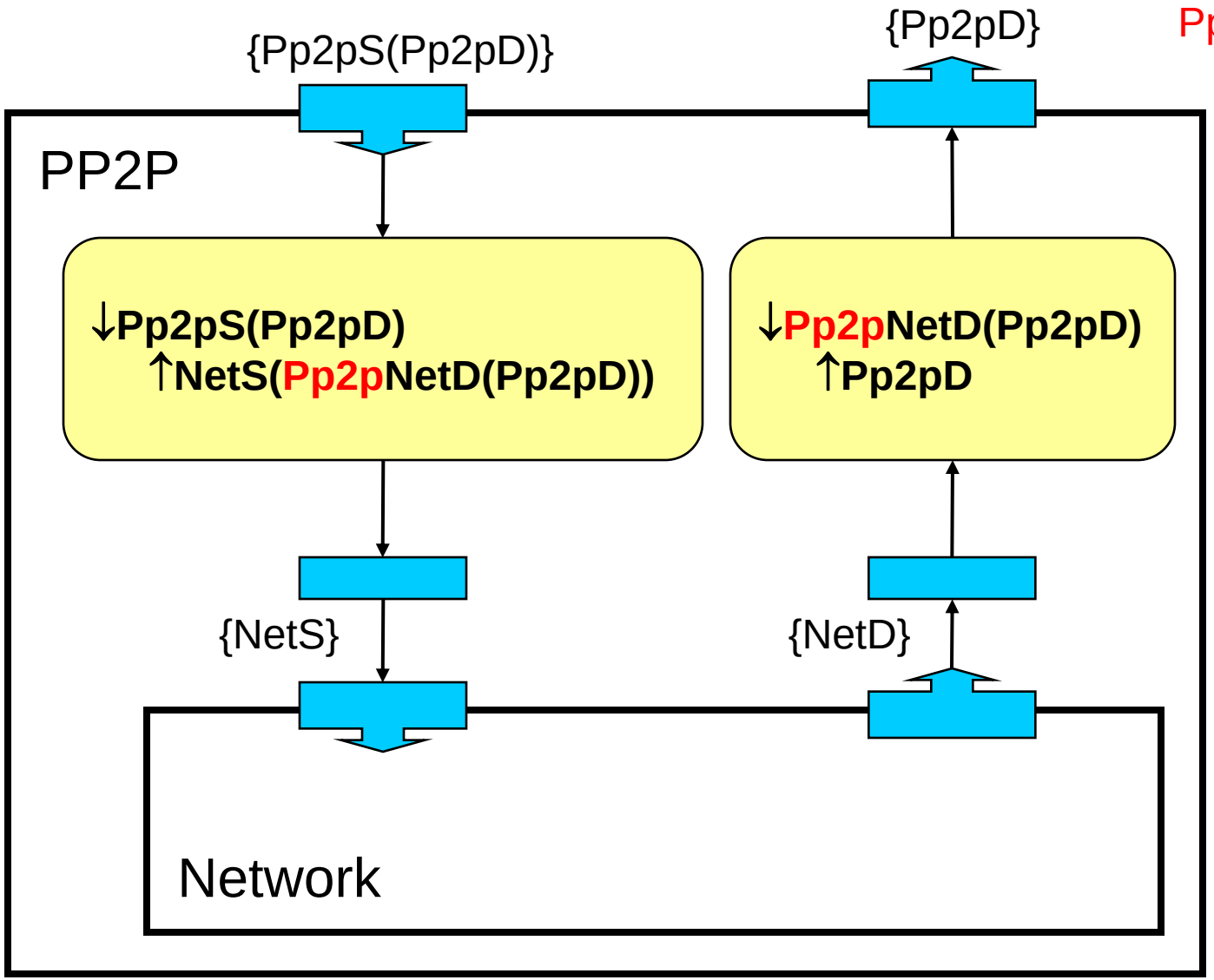
- Model overview
- Concepts semantics
- Examples and Java implementation demo
- Outlook

Composite component example

- Reliable Broadcast
  - Best Effort Broadcast
    - Perfect Point-to-point Links
      - Network
  - Perfect Failure Detector
    - Perfect Point-to-point Links
      - Network
    - Timer

- Static sharing of the PP2P component

Perfect Point-to-point Links component (PP2P)

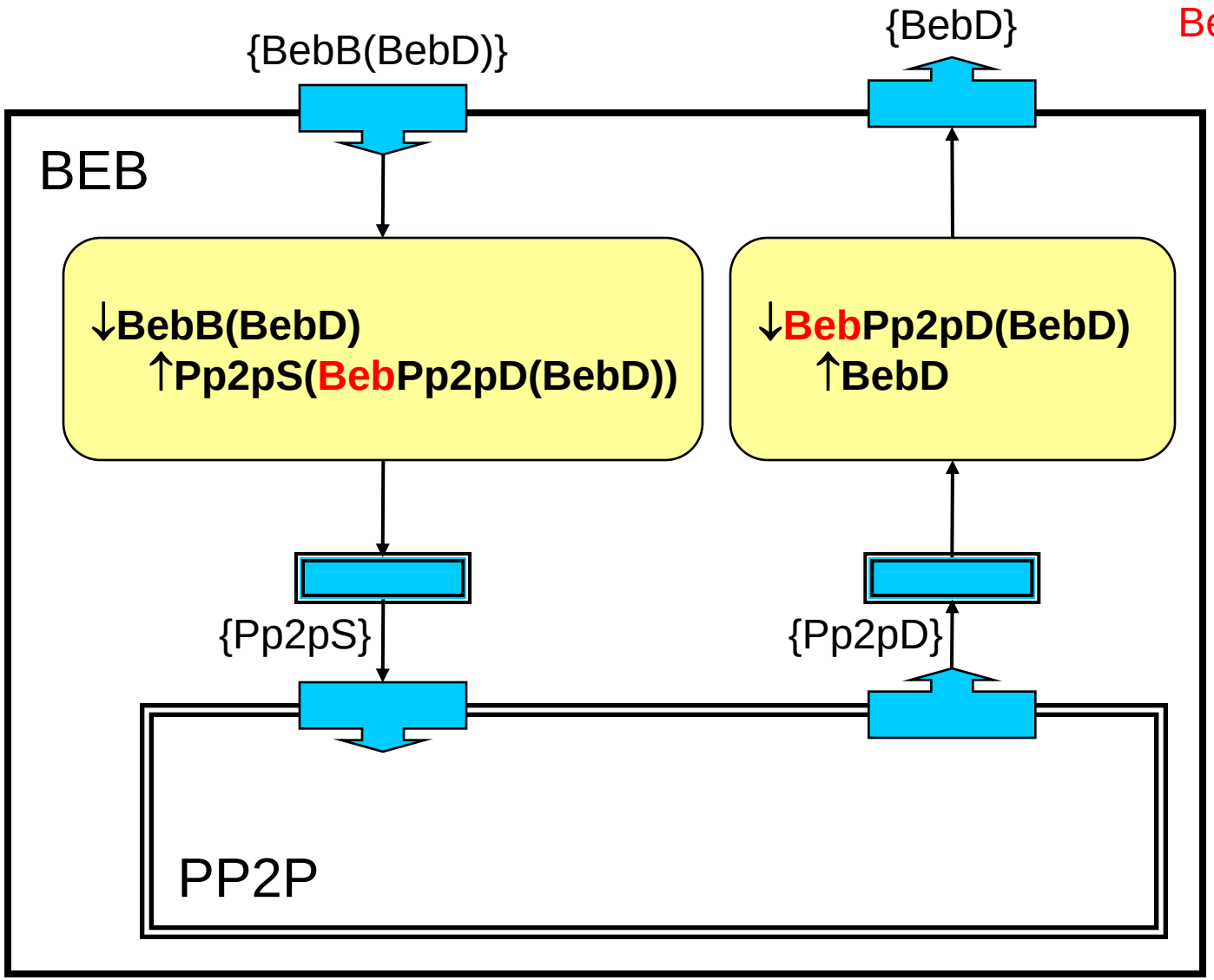


Pp2pNetD ⊂ NetD

Pp2pS(end)
Pp2pD(eliver)

NetS(end)
NetD(eliver)

Best Effort Broadcast component (BEB)

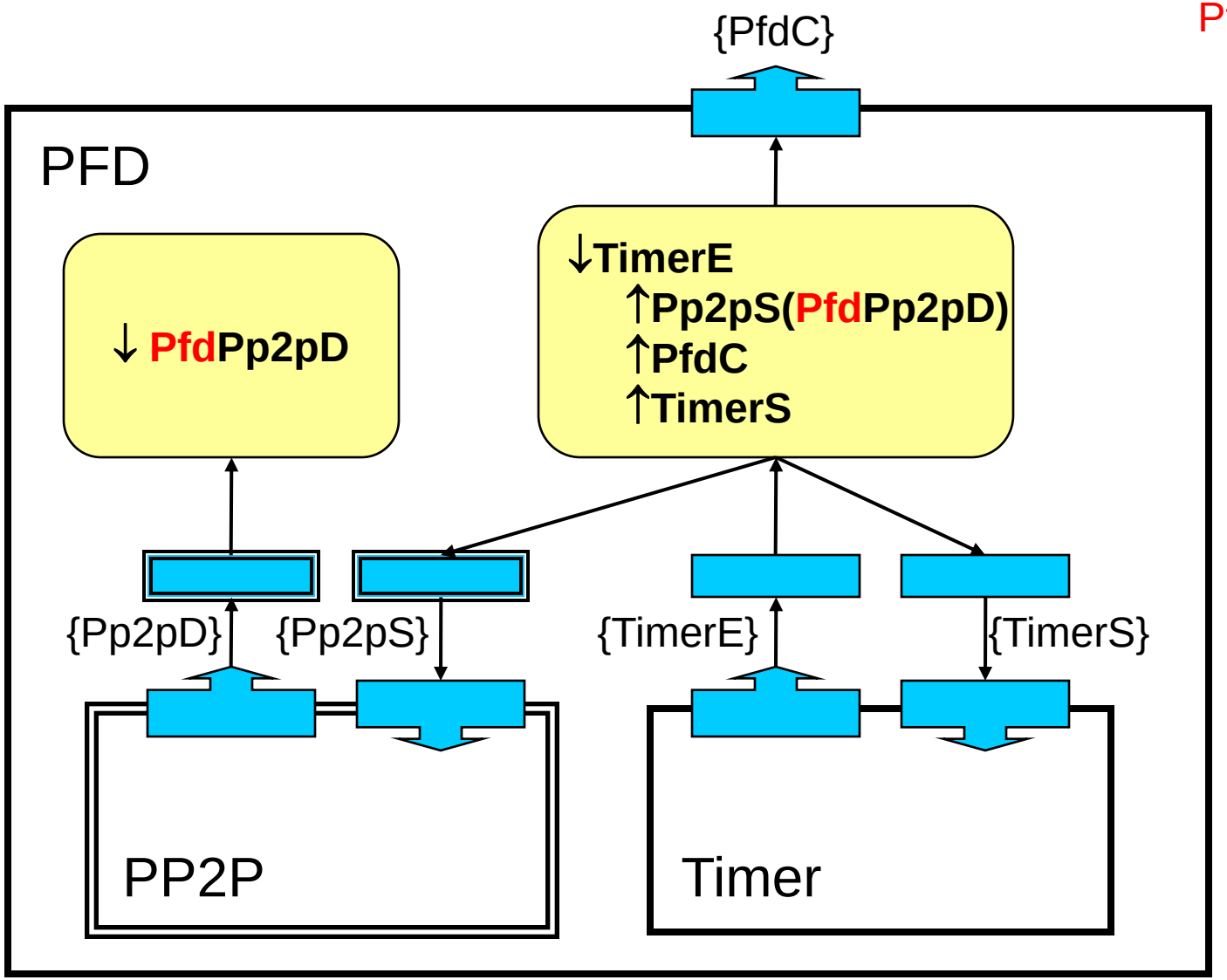


BebPp2pD ⊂ Pp2pD

BebB(broadcast)
BebD(eliver)

Pp2pS(end)
Pp2pD(eliver)

Perfect Failure Detector component (PFD)



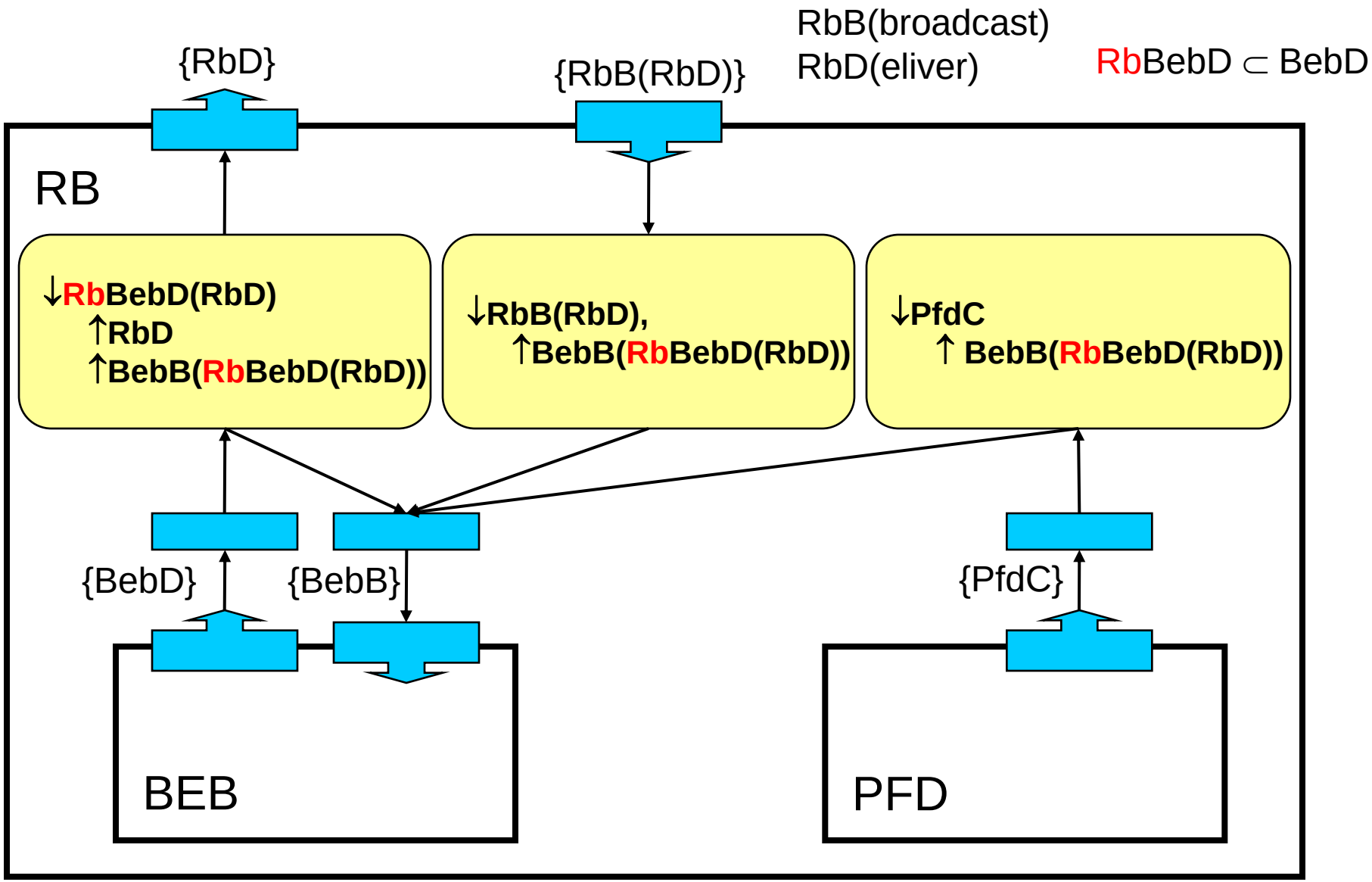
PfdPp2pD ⊂ Pp2pD

PfdC(rash)

Pp2pS(end)
Pp2pD(eliver)

TimerS(et)
TimerE(xpired)

Reliable Broadcast component (RB)

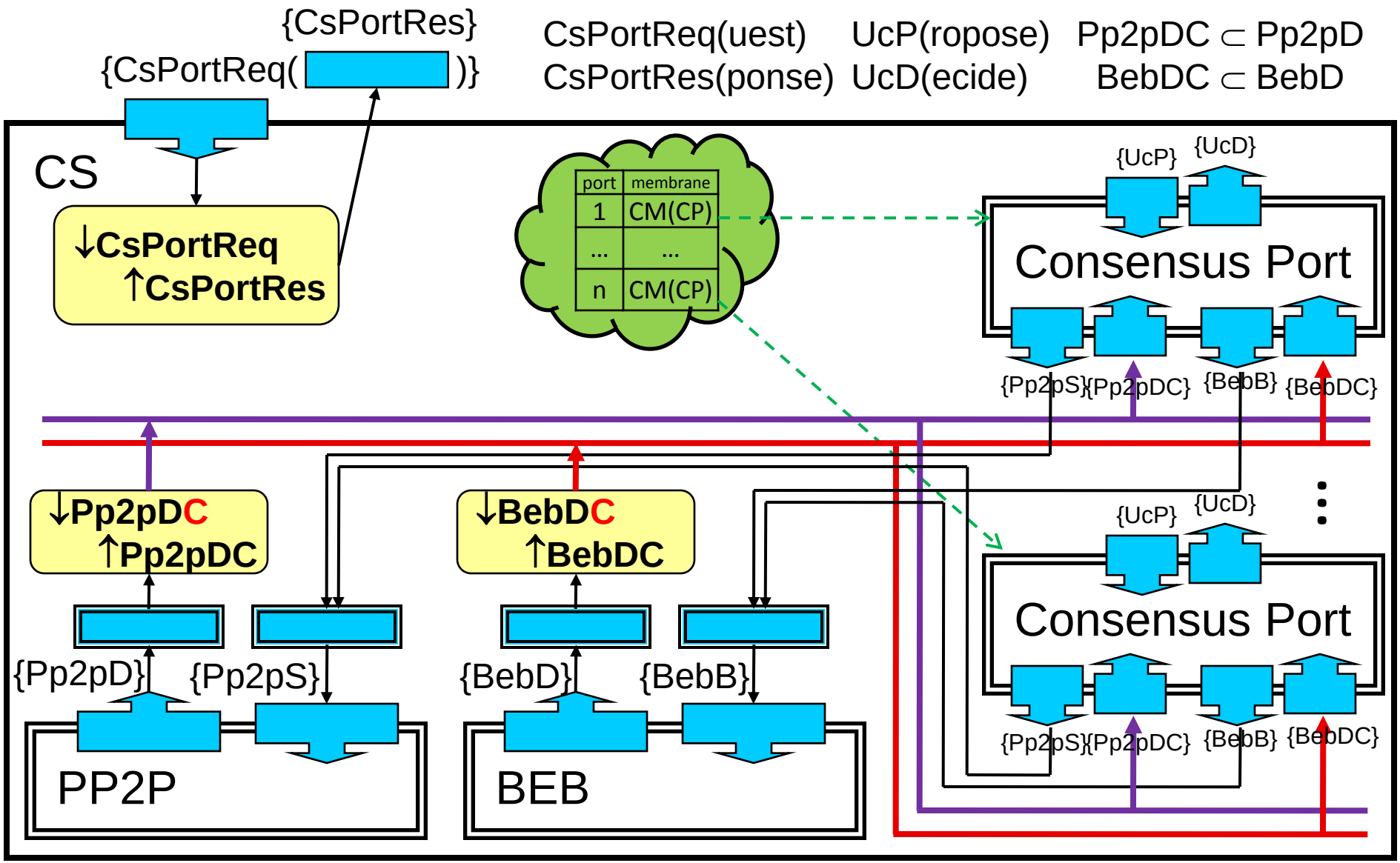


Compositionality with many instances

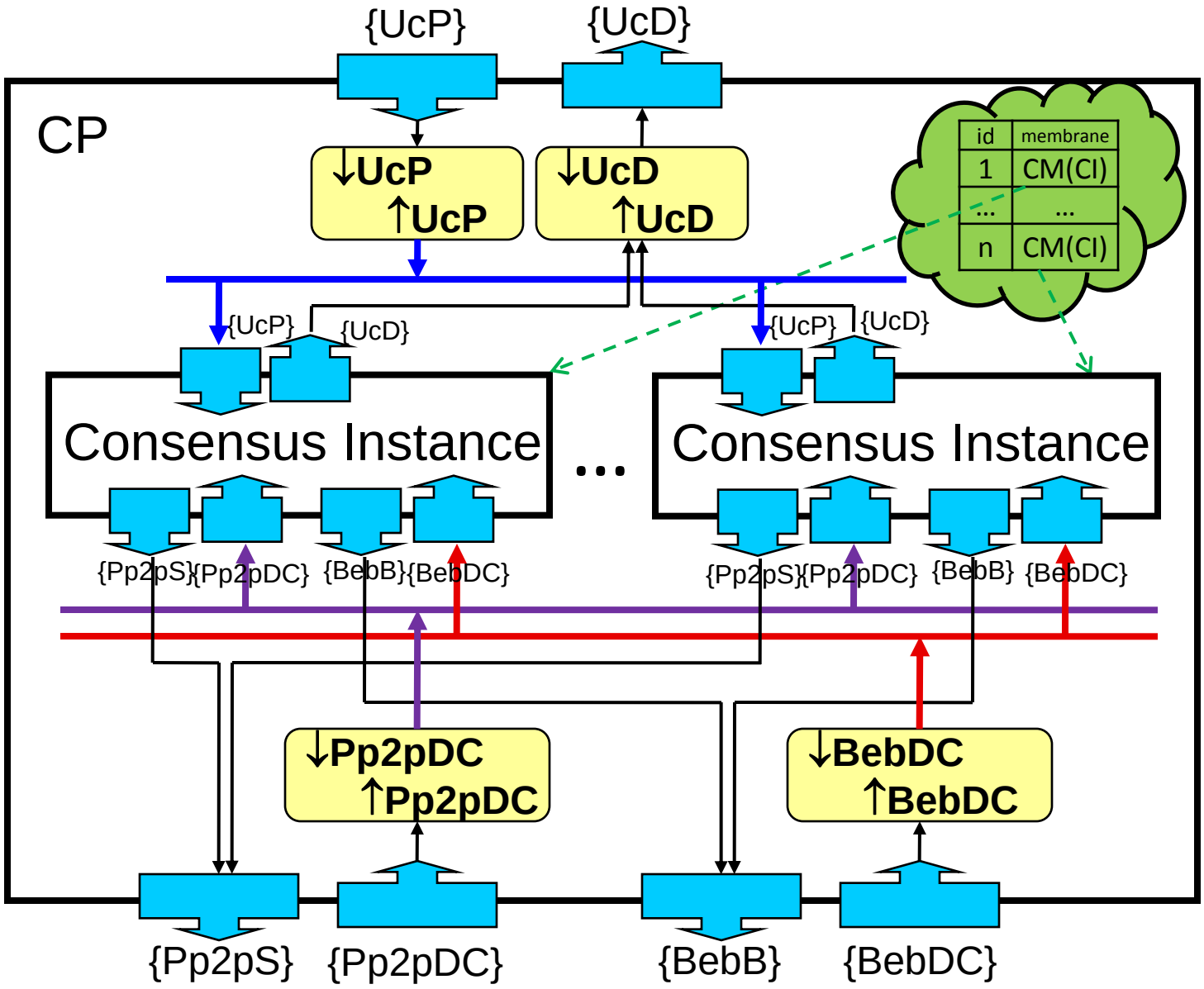
- Consensus Service
  - Consensus Port (#)
    - Consensus Instance (#)
      - Abortable Consensus
        - » Best Effort Broadcast
          - Perfect Point-to-point Links
            - Network
      - Eventual Leader Detector
        - » Timer
        - » Best Effort Broadcast
          - Perfect Point-to-point Links
            - Network
- Instance sharing of PP2P and BEB

Consensus Service component (CS)

new



Consensus Port component (CP)



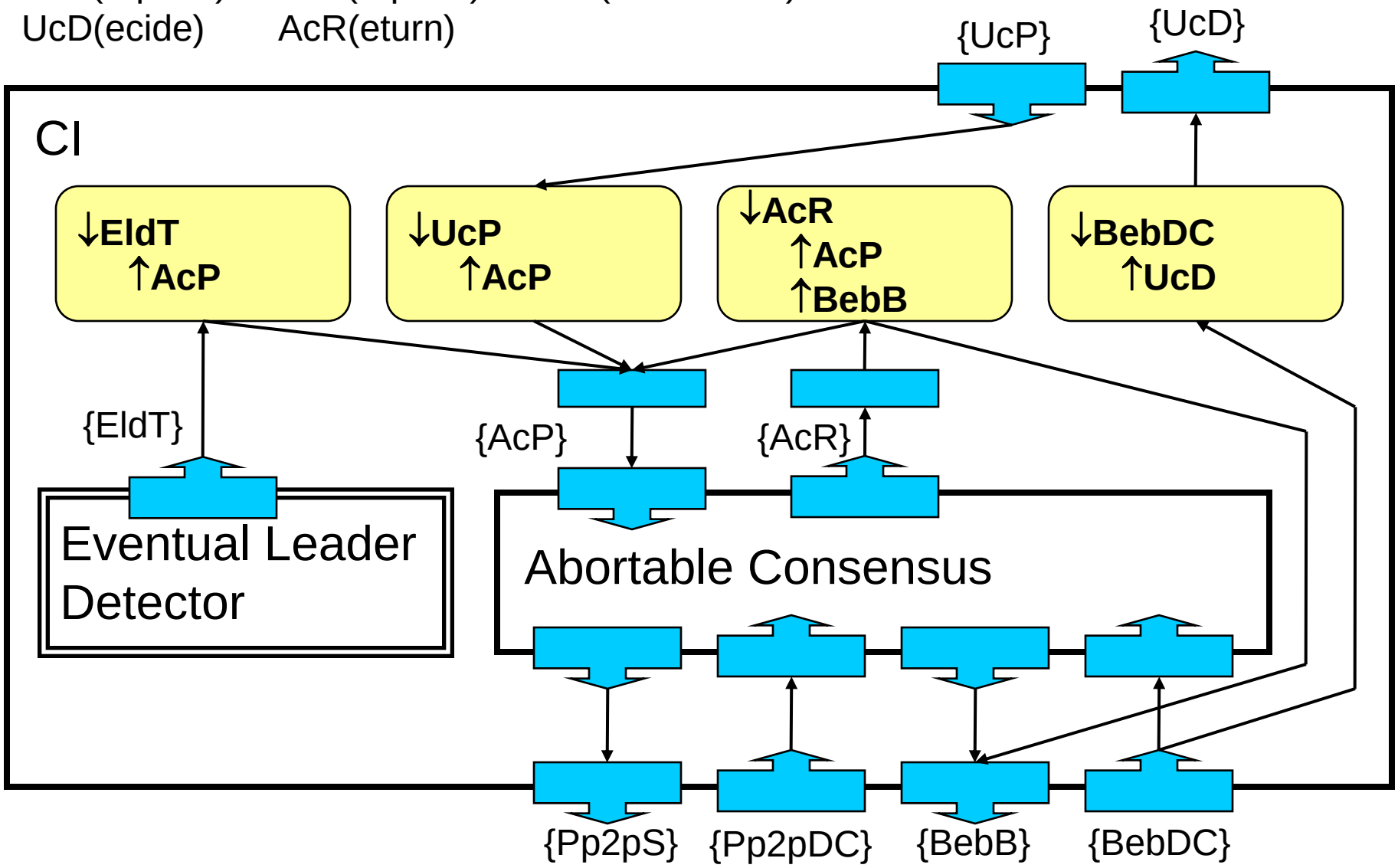
UcP(ropose)
UcD(ecide)

new

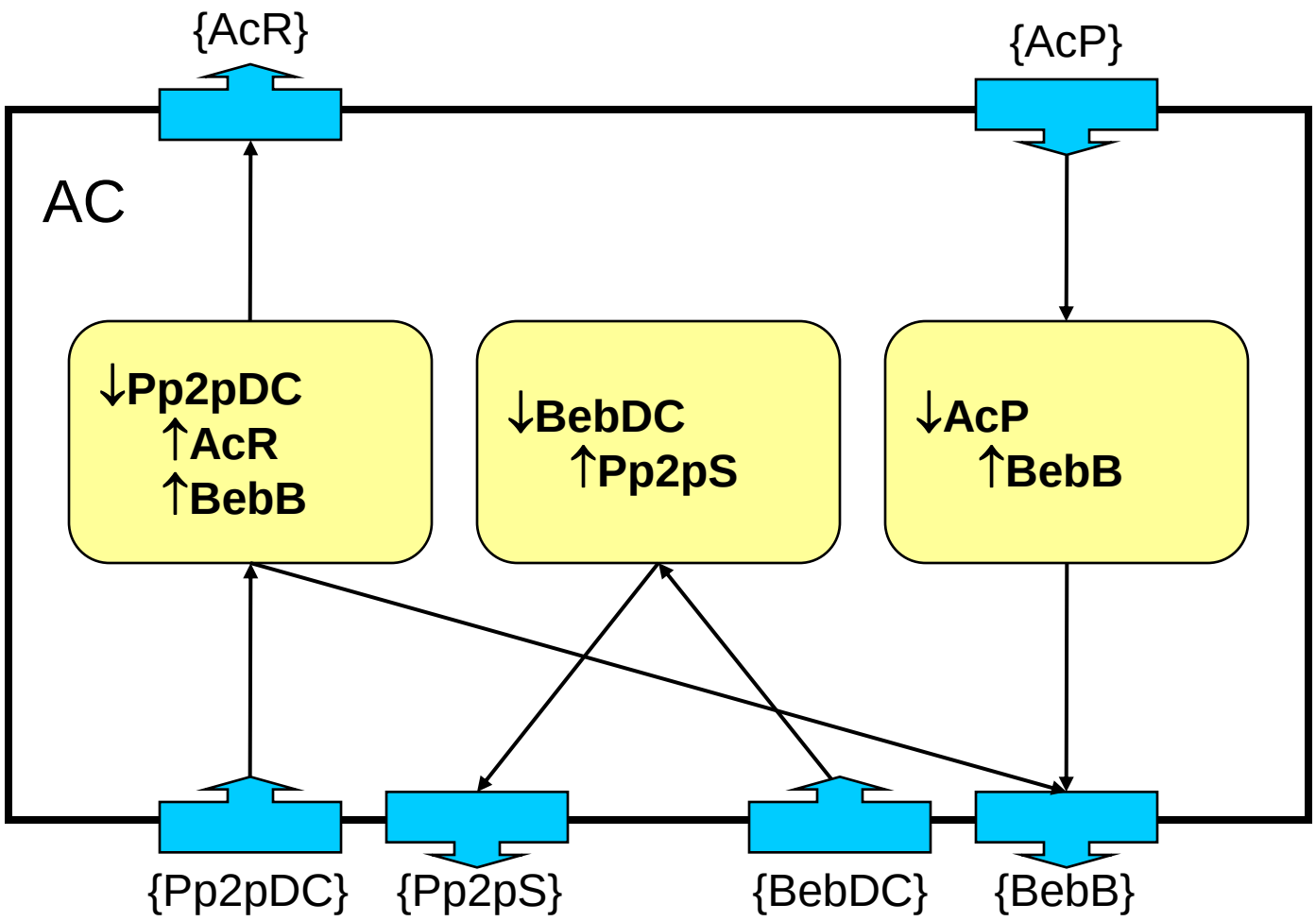
Consensus Instance component (CI)

new

UcP(ropose) AcP(ropose) EldT(rustLeader)
UcD(ecide) AcR(eturn)



Abortable Consensus component (AC)



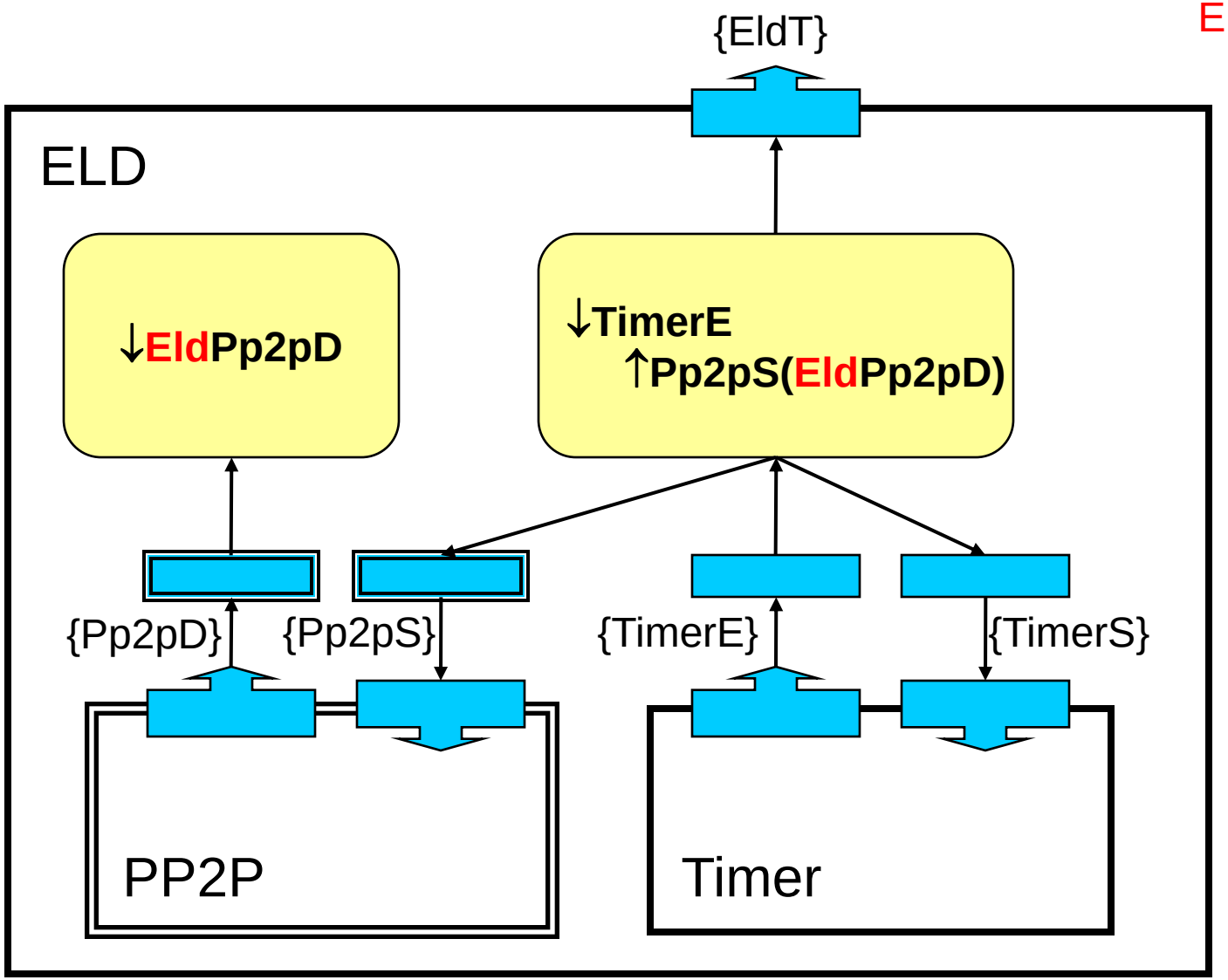
AcP(ropose)
AcR(eturn)

Pp2pS(end)
Pp2pD(eliver)
C(onsensus)

BebB(roadcast)
BebD(eliver)
C(onsensus)

new

Eventual Leader Detector component (ELD)



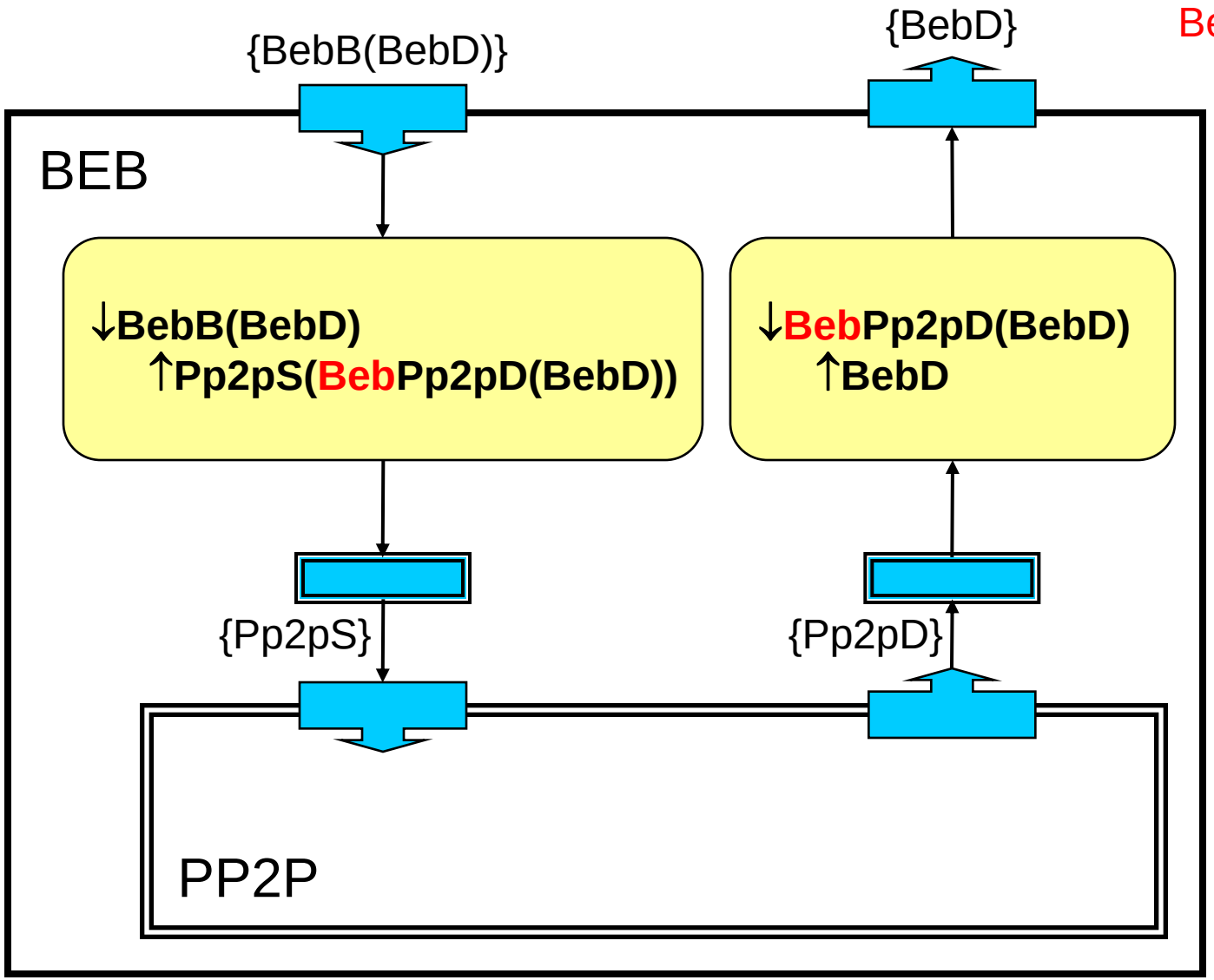
EldPp2pD ⊂ Pp2pD

EldT(rust)

Pp2pS(end)
Pp2pD(eliver)

TimerS(et)
TimerE(xpired)

Best Effort Broadcast component (BEB)

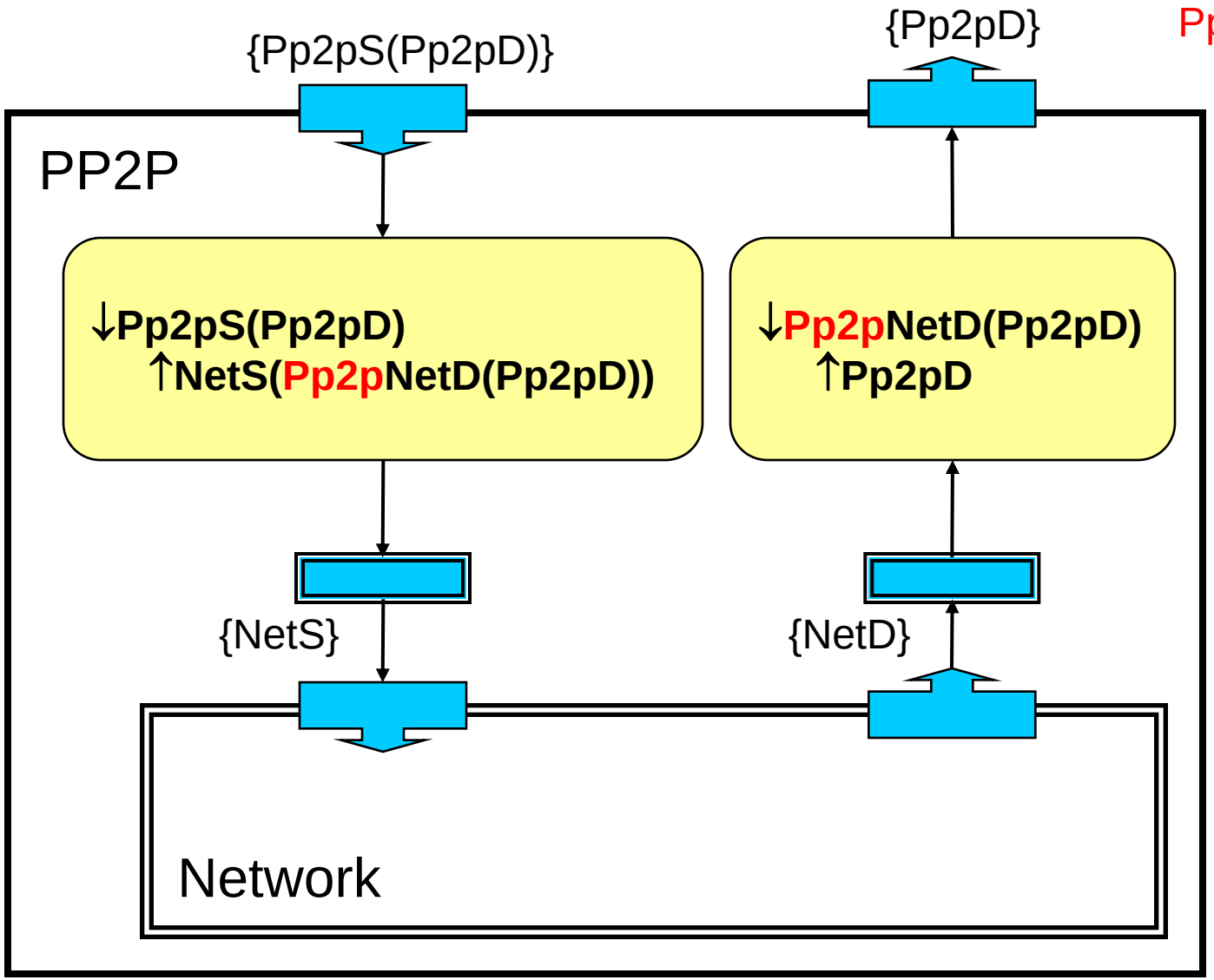


BebPp2pD ⊂ Pp2pD

BebB(broadcast)
BebD(eliver)

Pp2pS(end)
Pp2pD(eliver)

Perfect Point-to-point Links component (PP2P)



Pp2pNetD ⊂ NetD

Pp2pS(end)
Pp2pD(eliver)

NetS(end)
NetD(eliver)

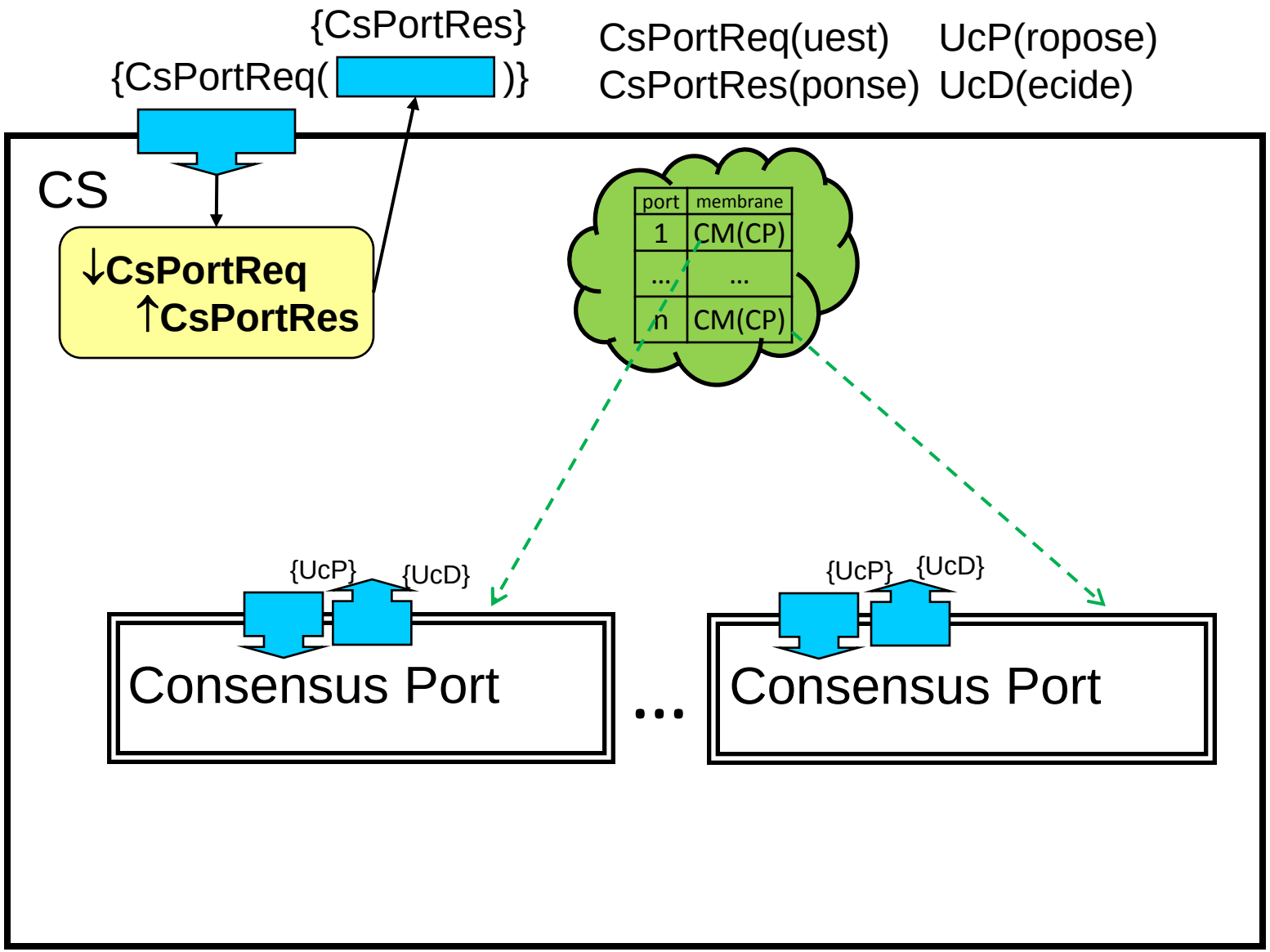
Observation 1

- We achieved decoupling of functional and non-functional aspects of consensus
- The Consensus Instance component implements a single-instance Paxos consensus algorithm (functional)
- The Consensus Port component deals with creating and garbage collecting consensus instances (non-functional)
- The Consensus Service component creates different Consensus Ports for client components (non-functional)

Observation 2

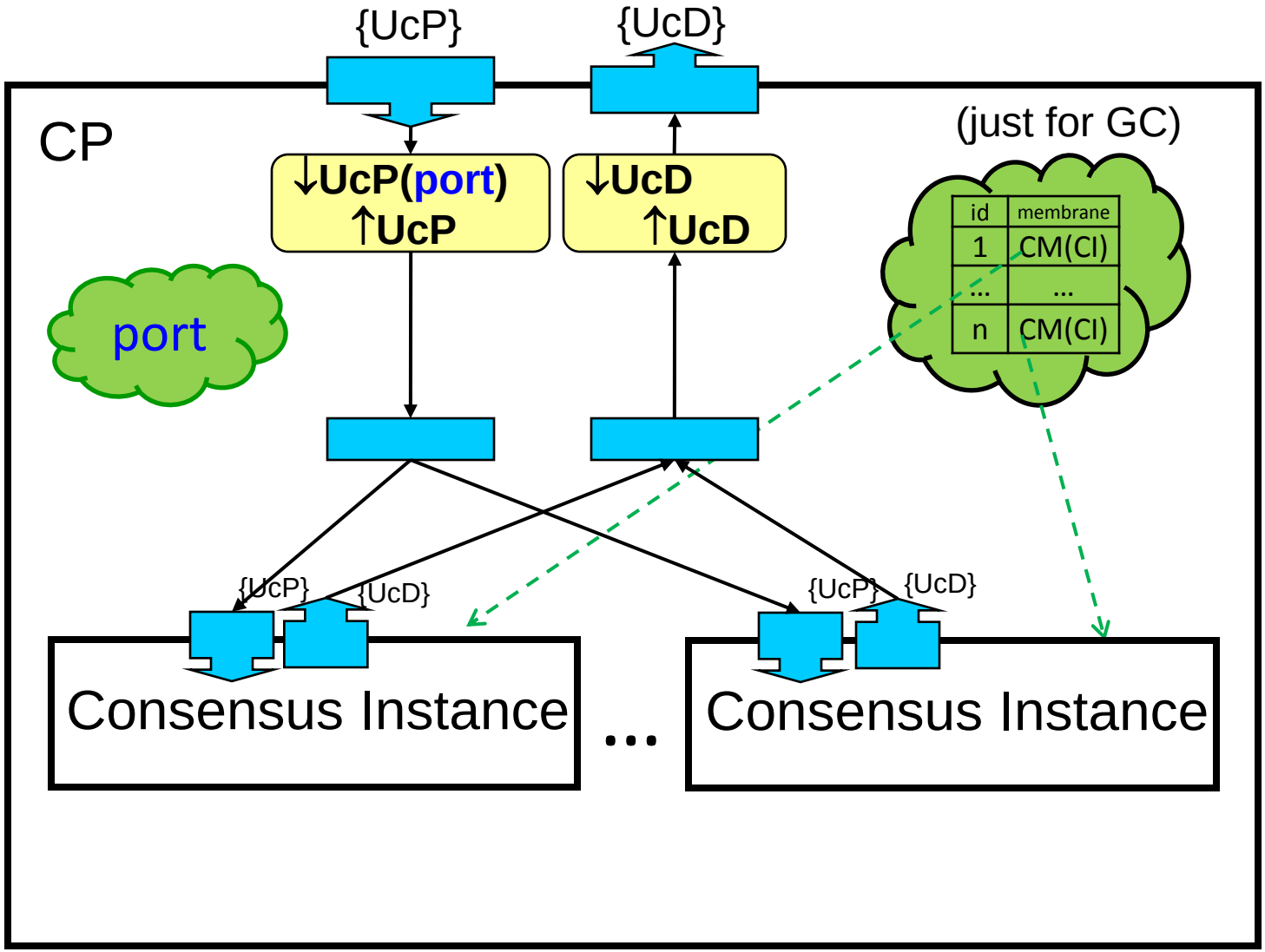
- We used an instance sharing pattern where demultiplexing is done in the high level components (CS and CP)
- This forces us to pass the channels of the shared component in the interface of the lower level components (CP, CI, and AC)
- The alternative is to provide channel subscription by event attribute and do the event demultiplexing in the channel

Consensus Service component (CS)



old

Consensus Port component (CP)

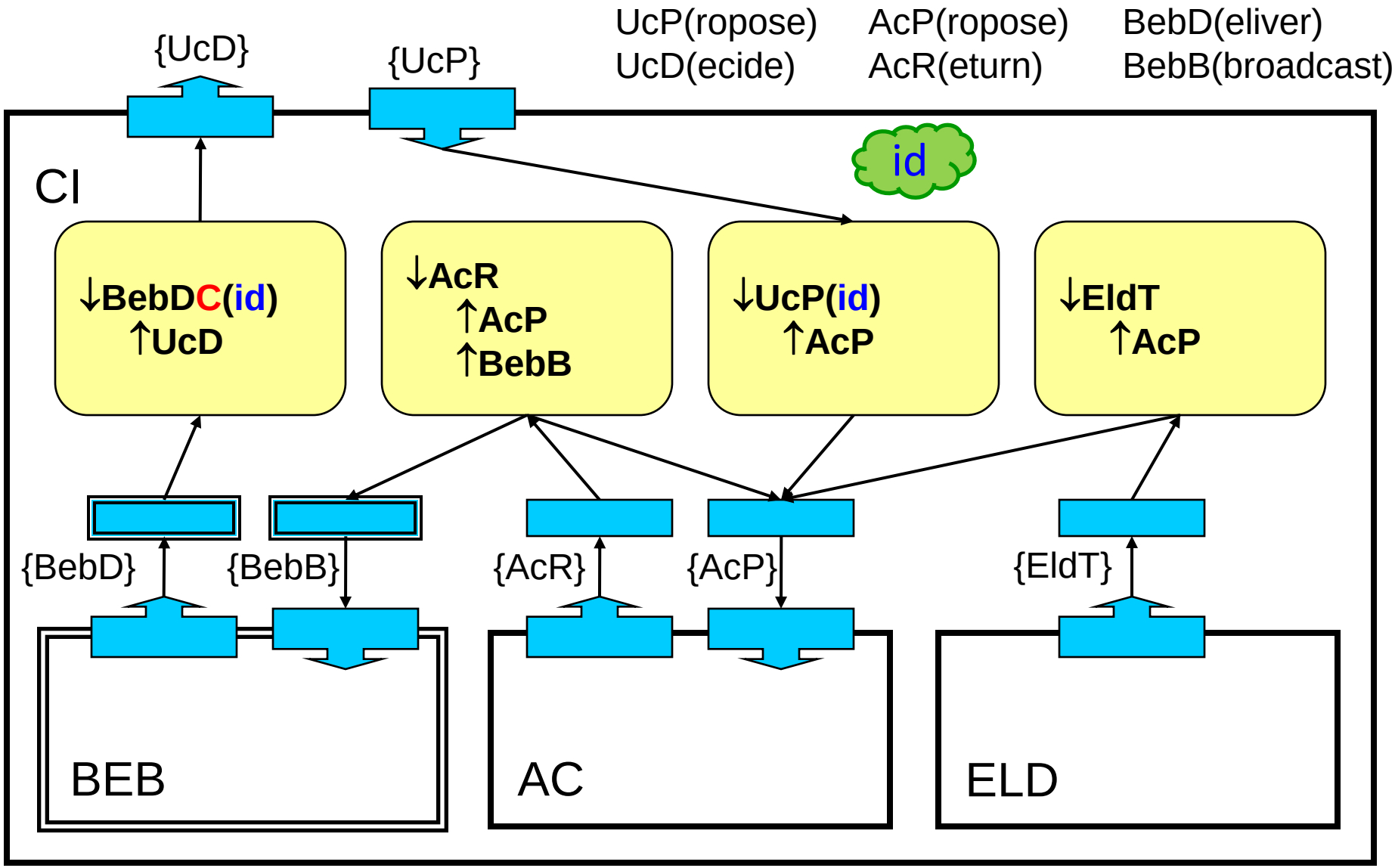


UcP(ropose)
UcD(ecide)

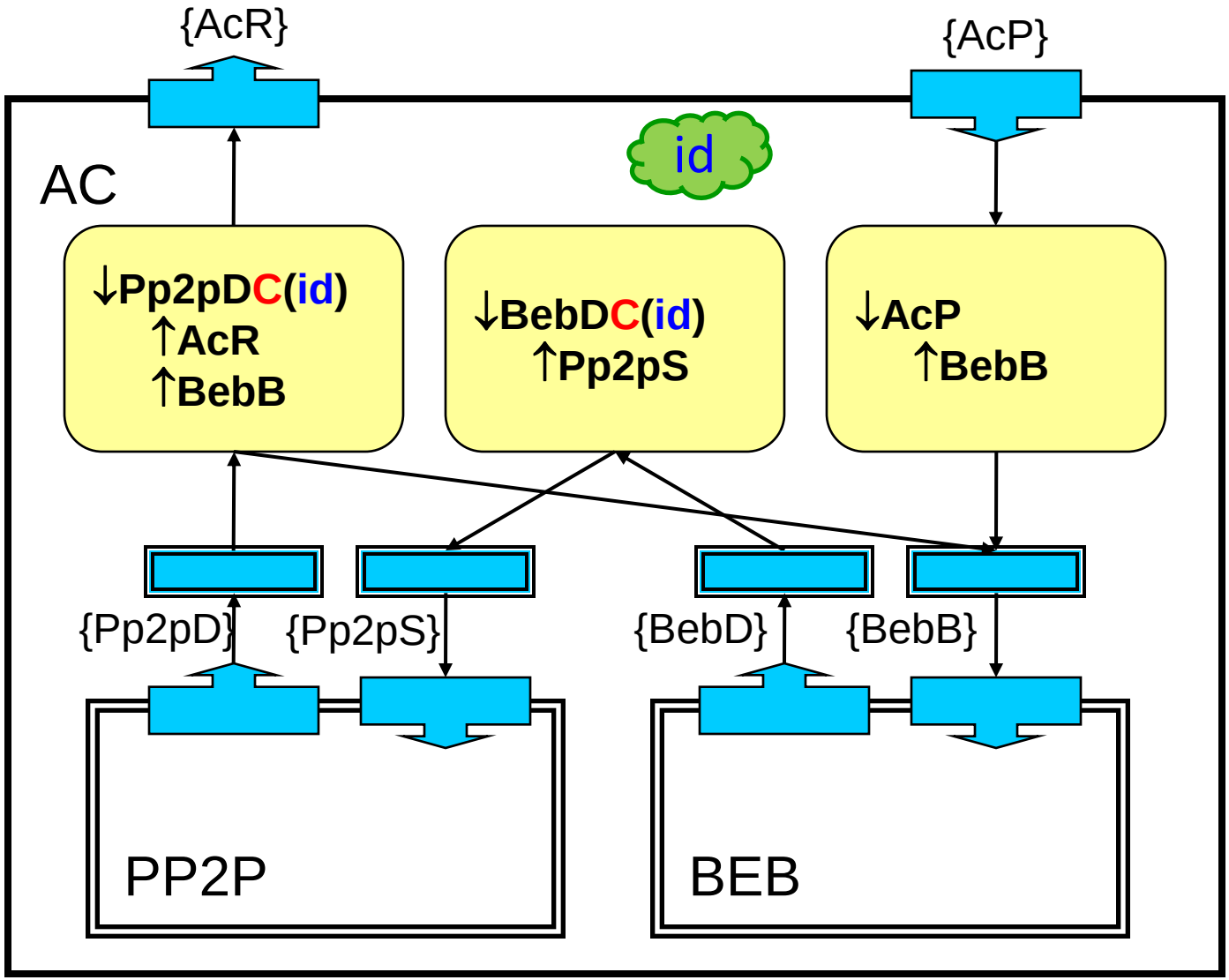
old

Consensus Instance component (CI)

old



Abortable Consensus component (AC)



AcP(ropose)
AcR(eturn)

Pp2pS(end)
Pp2pD(eliver)

BebB(roadcast)
BebD(eliver)

old

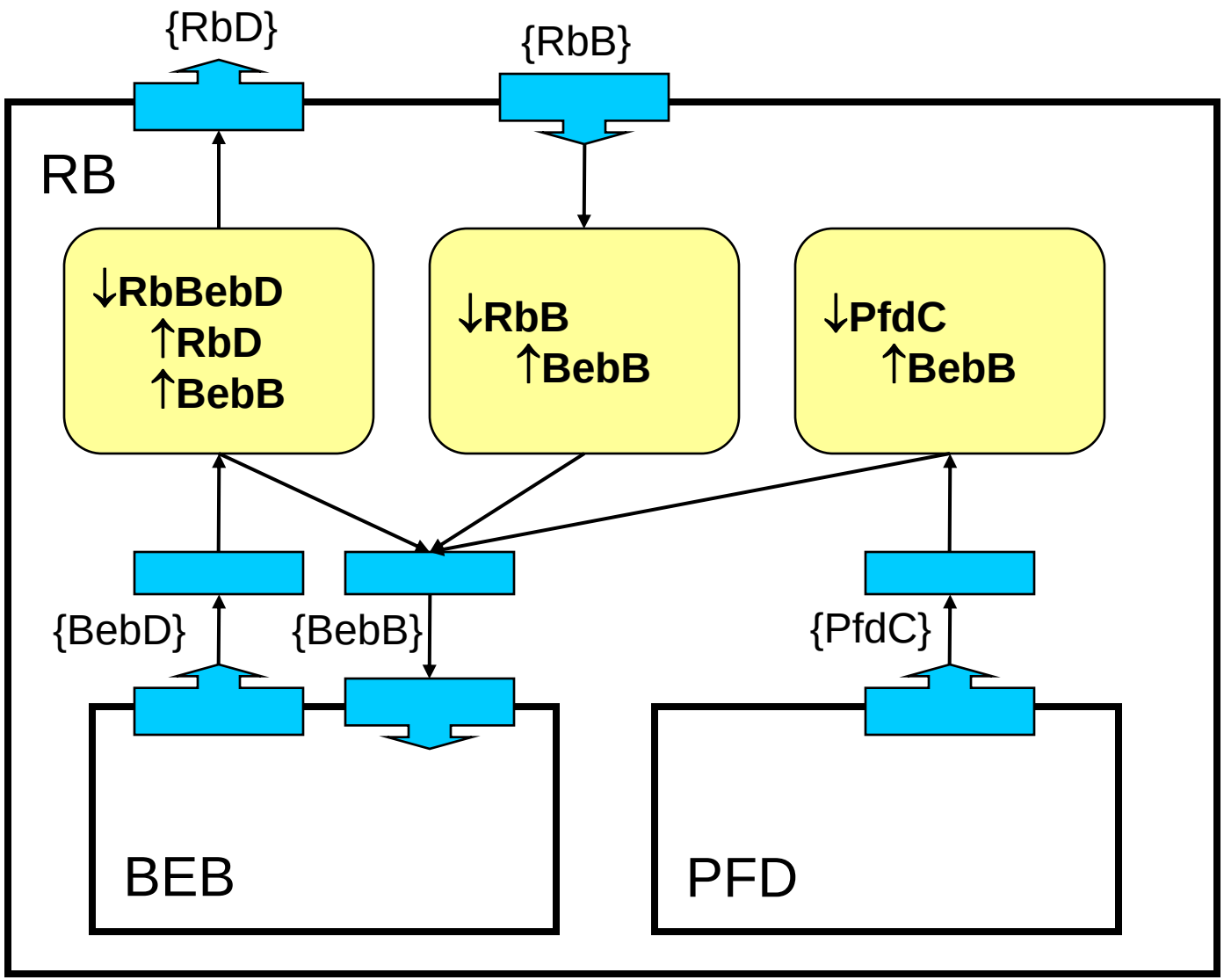
Observation 3

- We showed demultiplexing of events from BEB and PP2P in the CS and CP
- We showed demultiplexing of events from BEB and PP2P in the channel by subscription by attribute (topic)
- A third alternative is demultiplexing events of BEB and PP2P directly in BEB and PP2P
  - This is done in conjunction with the registration of user components to the shared components
  - Each component registers its own channel(s) (example CS)

Java implementation example

- We now show a Java implementation of the Reliable Broadcast (RB) component
- RB is a composite component containing a BEB and a PFD component
- In this example, the BEB and PFD components are not shared and completely encapsulated in the RB component
- BEB and PFD are created by RB's creation routine

Reliable Broadcast component (RB)



RbB(broadcast)
RbD(eliver)

BebB(broadcast)
BebD(eliver)

PfdC(rash)

RbBebD(eliver)

RbBebD $\subset$ BebD

RbBroadcastComponent.java 1/6

```
public class RbBroadcastComponent {
   // reference to the "core" of the component
   private Component component;
   // parameters and internal channels
   private Channel rbBroadcastChannel, rbDeliverChannel, pfdCrashChannel,
                   bebBroadcastChannel, bebDeliverChannel;
   // state variables
   private HashSet<Address> correct;
   private HashMap<Address, Set<RbDeliverEvent>> from;
   private HashSet<RbDeliverEvent> delivered;
   private NeighbourLinks neighbourLinks;
   // Java constructor
   public RbBroadcastComponent(Component component) {
        this.component = component;
        this.correct = new HashSet<Address>();
        this.from = new HashMap<Address, Set<RbDeliverEvent>>();
        this.delivered = new HashSet<RbDeliverEvent>();
   }
        ...
```

RbBroadcastComponent.java 2/6

```
    ...
@ComponentCreateMethod
public void create(Channel rbBroadcastChannel, Channel rbDeliverChannel)
        throws ClassNotFoundException {

   this.rbBroadcastChannel = rbBroadcastChannel;
   this.rbDeliverChannel = rbDeliverChannel;

   pfdCrashChannel = component.createChannel();
   bebBroadcastChannel = component.createChannel();
   bebDeliverChannel = component.createChannel();

   pfdCrashChannel.addEventType(PfdCrashEvent.class);
   bebBroadcastChannel.addEventType(BebBroadcastEvent.class);
   bebDeliverChannel.addEventType(BebDeliverEvent.class);

    ...
```

RbBroadcastComponent.java 3/6

```
    ...
  Factory pfdFactory = component.createFactory(
       "se.sics.kompics.protocols.pfd.PerfectFailureDetectorComponent");
  Factory bebFactory = component.createFactory(
       "se.sics.kompics.protocols.beb.BestEffortBroadcastComponent");
  Channel faultChannel = component.getFaultChannel();

  Component pfdComponent = pfdFactory.createComponent(faultChannel,
       pfdCrashChannel);
  Component bebComponent = bebFactory.createComponent(faultChannel,
       bebBroadcastChannel, bebDeliverChannel);

  component.subscribe(this.rbBroadcastChannel,"handleRbBroadcastEvent");
  component.subscribe(pfdCrashChannel, "handlePfdCrashEvent");
  component.subscribe(bebDeliverChannel, "handleRbBebDeliverEvent");
}
    ...
```

RbBroadcastComponent.java 4/6

```
    ...
@ComponentInitializeMethod
public void init(NeighbourLinks neighbourLinks) {
   this.neighbourLinks = neighbourLinks;
   correct.addAll(this.neighbourLinks.getAllNodes());
}

@EventHandlerMethod
@MayTriggerEventTypes(BebBroadcastEvent.class)
public void handleRbBroadcastEvent(RbBroadcastEvent event) {
   RbDeliverEvent rbDeliverEvent = event.getRbDeliverEvent();
   RbBebDeliverEvent bebDeliverEvent = new RbBebDeliverEvent(
                rbDeliverEvent);
   BebBroadcastEvent bebBroadcastEvent = new BebBroadcastEvent(
                bebDeliverEvent);
   component.triggerEvent(bebBroadcastEvent, bebBroadcastChannel);
}
    ...
```

RbBroadcastComponent.java 5/6

```
    ...
@EventHandlerMethod
@MayTriggerEventTypes( { RbDeliverEvent.class, BebBroadcastEvent.class })
public void handleRbBebDeliverEvent(RbBebDeliverEvent event) {
   RbDeliverEvent rbDeliverEvent = (RbDeliverEvent) event
        .getRbDeliverEvent();
   Address sourceNode = event.getSource();

   if (!delivered.contains(rbDeliverEvent)) {
       delivered.add(rbDeliverEvent);
       component.triggerEvent(rbDeliverEvent, rbDeliverChannel);
       saveMessage(sourceNode, rbDeliverEvent);
       if (!correct.contains(sourceNode)) {
           BebBroadcastEvent bebBroadcastEvent = new
                       BebBroadcastEvent(event);
           component.triggerEvent(bebBroadcastEvent,bebBroadcastChannel);
        }
   }
}    ...
```

RbBroadcastComponent.java 6/6

```
    ...
@EventHandlerMethod
@MayTriggerEventTypes(BebBroadcastEvent.class)
public void handlePfdCrashEvent(PfdCrashEvent event) {
   Address crashedNode = event.getCrashedNodeAddress();
   correct.remove(crashedNode);

   if (from.containsKey(crashedNode)) {
        for (RbDeliverEvent rbDeliverEvent : from.get(crashedNode)) {
                RbBebDeliverEvent bebDeliverEvent = new
                        RbBebDeliverEvent(rbDeliverEvent);
                BebBroadcastEvent bebBroadcastEvent = new
                        BebBroadcastEvent(bebDeliverEvent);
                component.triggerEvent(bebBroadcastEvent,
                        bebBroadcastChannel);
        }
        from.remove(crashedNode);
   }
}
    ...
```

BebBroadcastEvent.java

```
@EventType
public final class BebBroadcastEvent implements Event {
   private Address source;
   private BebDeliverEvent bebDeliverEvent;
   public BebBroadcastEvent(BebDeliverEvent bebDeliverEvent,
                Address source){
        this.source = source;
        this.bebDeliverEvent = bebDeliverEvent;
   }
   public BebBroadcastEvent(BebDeliverEvent bebDeliverEvent) {
        this(bebDeliverEvent, Address.getLocalAddress());
   }
   public final Address getSource() {
        return source;
   }
   public final BebDeliverEvent getBebDeliverEvent() {
        return bebDeliverEvent;
   }
}
```

BebDeliverEvent.java

```
@EventType
public abstract class BebDeliverEvent implements Event, Serializable {

   private transient Address source;

   public final Address getSource() {
        return source;
   }

   public final void setSource(Address source) {
        this.source = source;
   }
}
```

RbBebDeliverEvent.java

```
@EventType
public final class RbBebDeliverEvent extends BebDeliverEvent {

   private static final long serialVersionUID = -79477826379778166O9L;

   RbDeliverEvent rbDeliverEvent;

   public RbBebDeliverEvent(RbDeliverEvent rbDeliverEvent) {
        super();
        this.rbDeliverEvent = rbDeliverEvent;
   }

   public final RbDeliverEvent getRbDeliverEvent() {
        return rbDeliverEvent;
   }
}
```

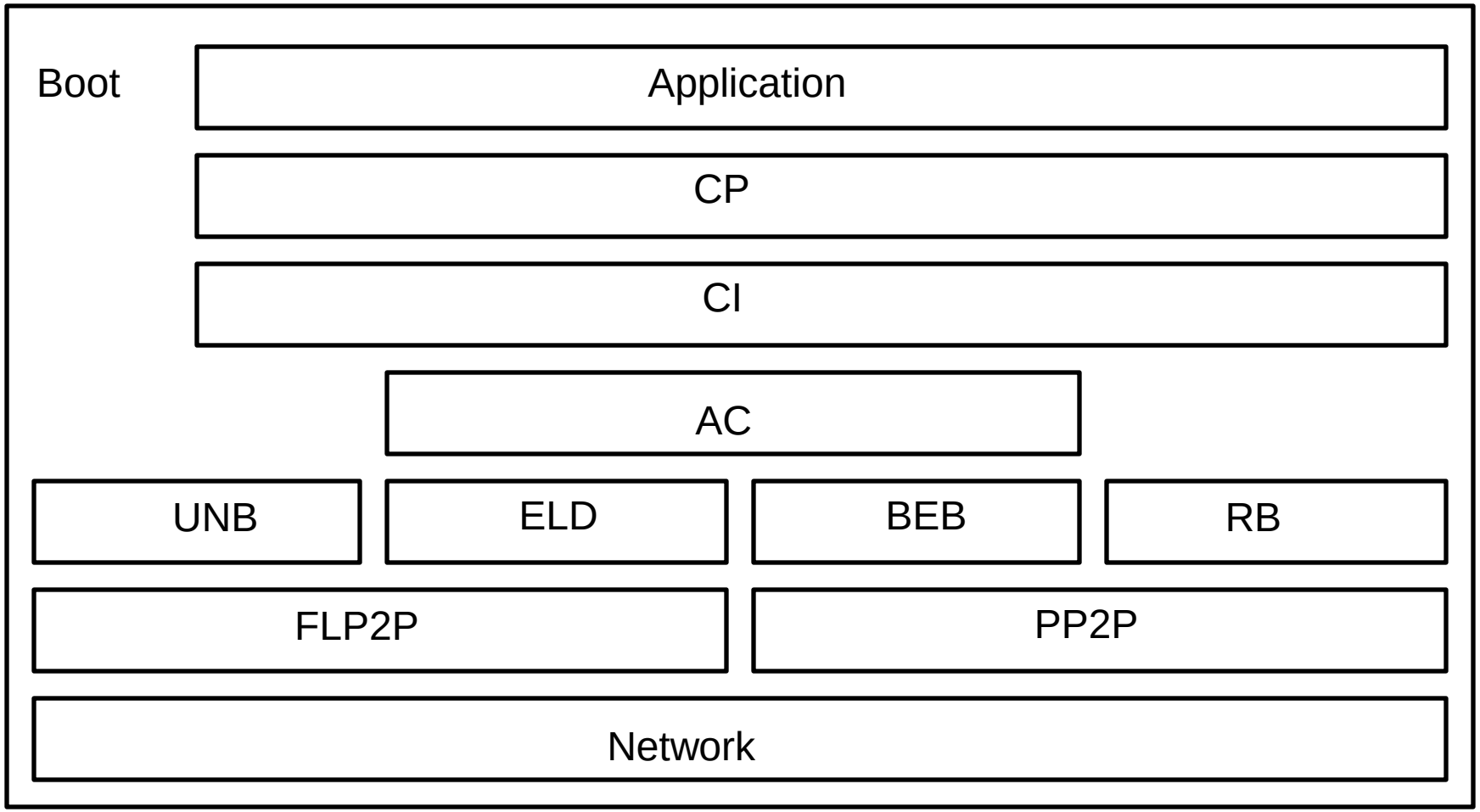
PfdCrashEvent.java

```
@EventType
public class PfdCrashEvent implements Event {

   private Address address;

   public PfdCrashEvent(Address address) {
        this.address = address;
   }

   public Address getCrashedNodeAddress() {
        return address;
   }
}
```

Demonstration scenario

- We run some protocols from "*Introduction to Reliable Distributed Programming*" by Rachid Guerraoui and Luís Rodrigues, 2006
- We start *n* processes and some of them can die (we use fail-stop protocols)
- We have a *topology* describing the characteristics of the links between the processes

Demonstration scenario



Demonstration scenario

- Each process executed a sequence of operations

Outline

- Model overview
- Concepts semantics
- Examples and Java implementation demo
- Outlook

Outlook

- Wrap Kompics into a Fractal interface
- Dynamic reconfiguration: scenarios for real systems
- Deployment: binary dependencies management
- Extensive comparison with existing protocol frameworks
  - Appia
  - SEDA
- Translator from abstract syntax or visual representation to specific programming languages
  - Java
  - C++

THANK YOU!

Questions?