ID2203 - Tutorial 1

Distributed Systems, Advanced Course

Cosmin Arad icarad@kth.se

Tallat Shafaat tallat@kth.se

Seif Haridi haridi@kth.se

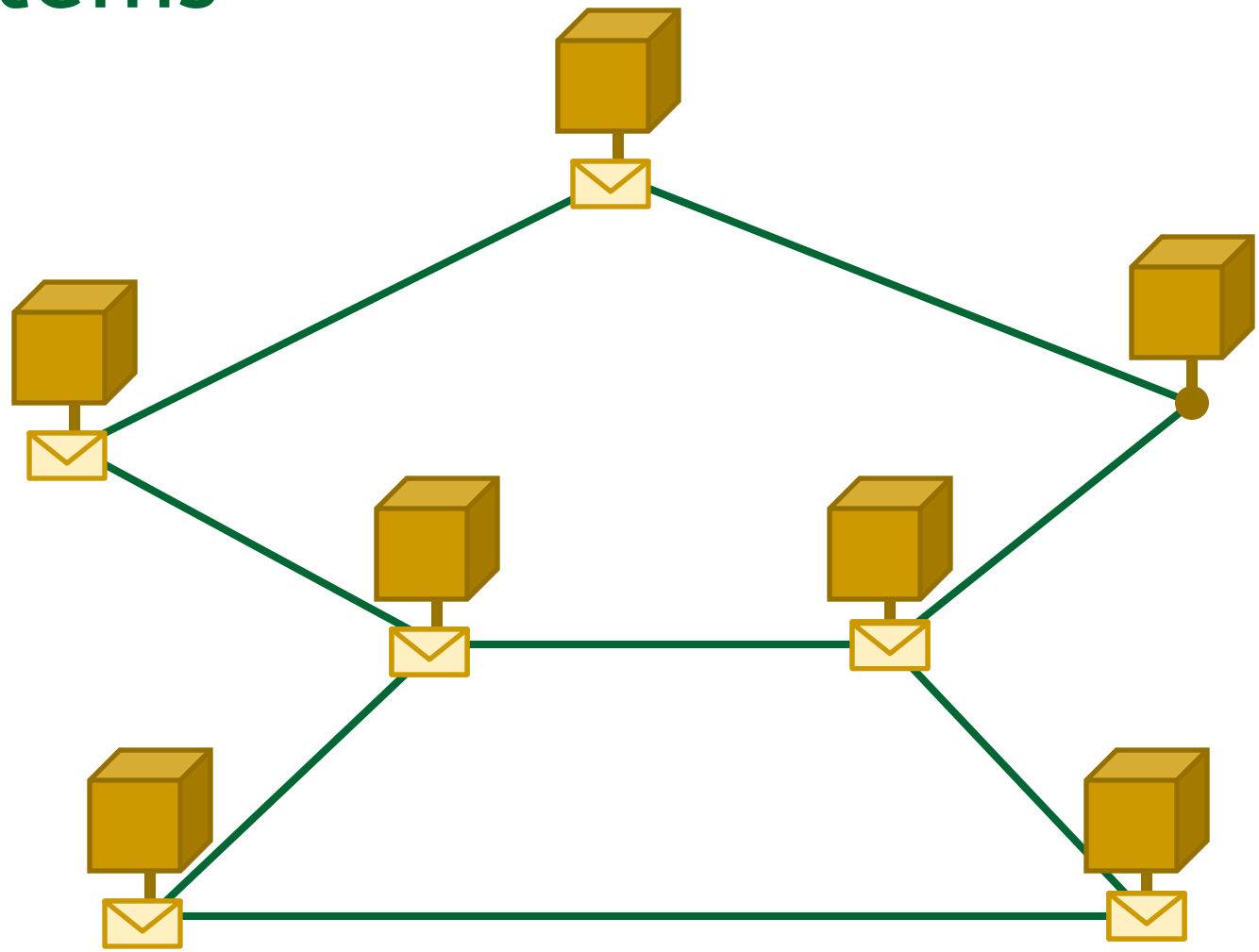
KTH - The Royal Institute of Technology

Overview

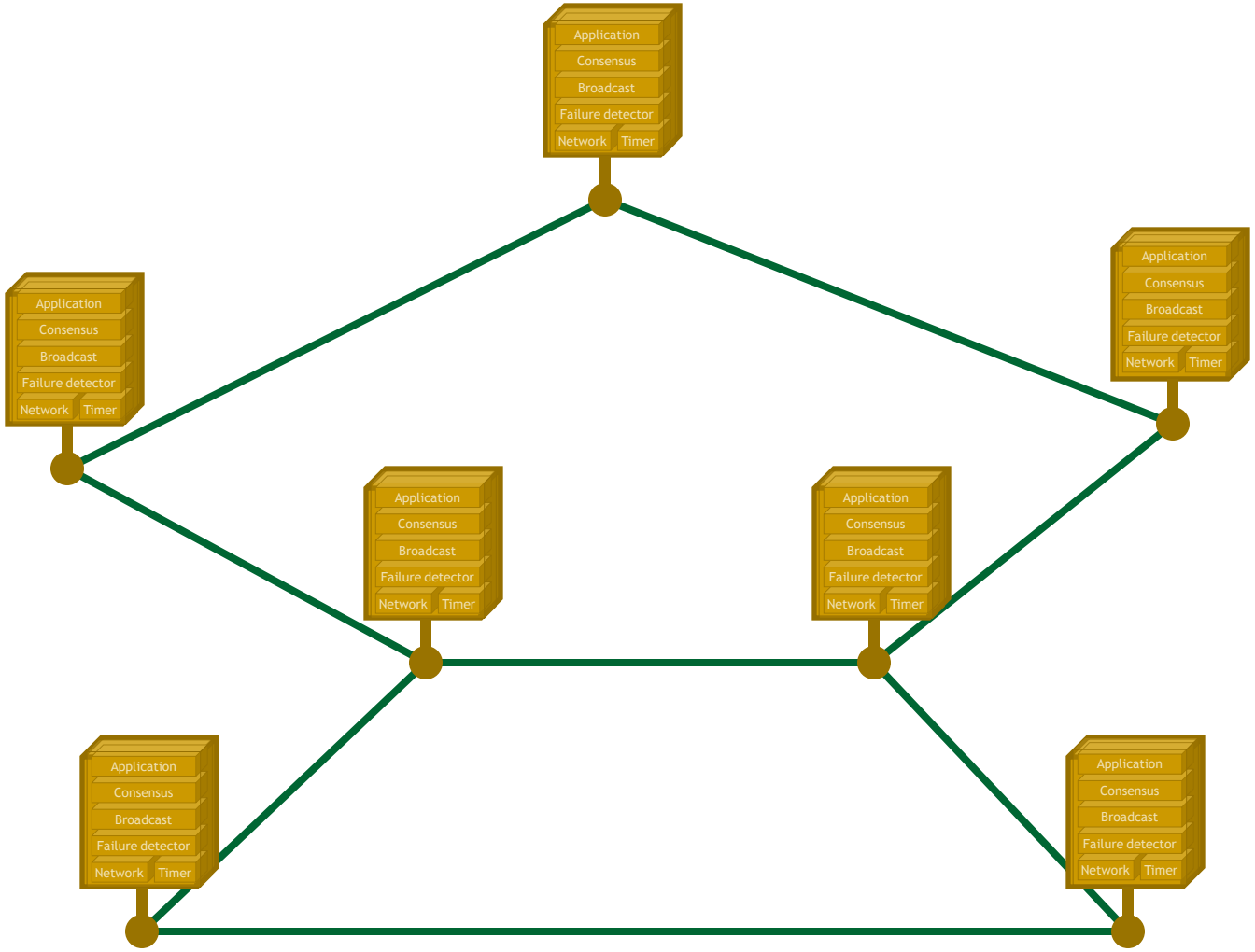
- Introduction to Kompics
- Relation to the textbook
- Assignments framework
- First assignment

Component Model for Distributed Systems

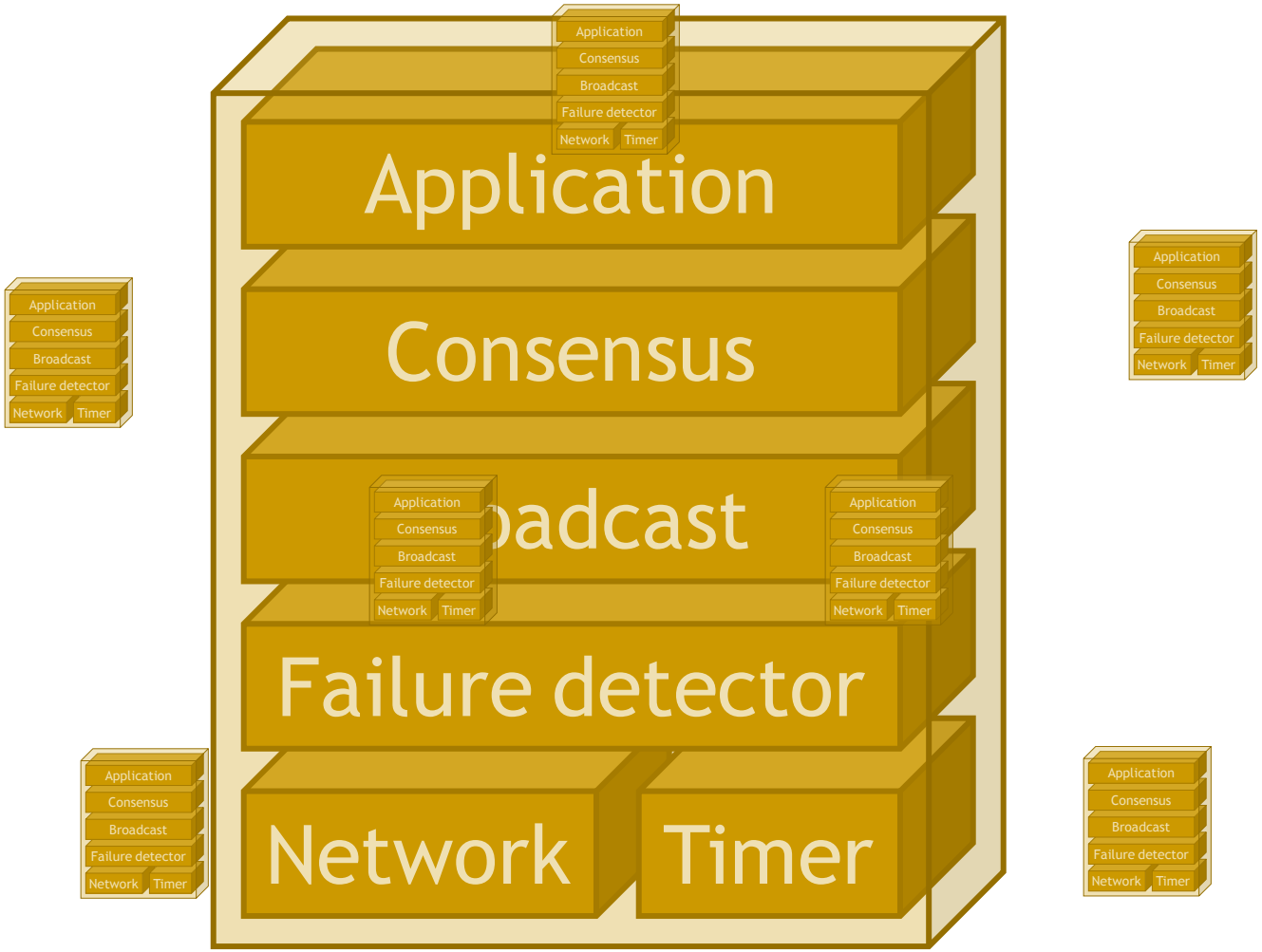
We want to build distributed systems



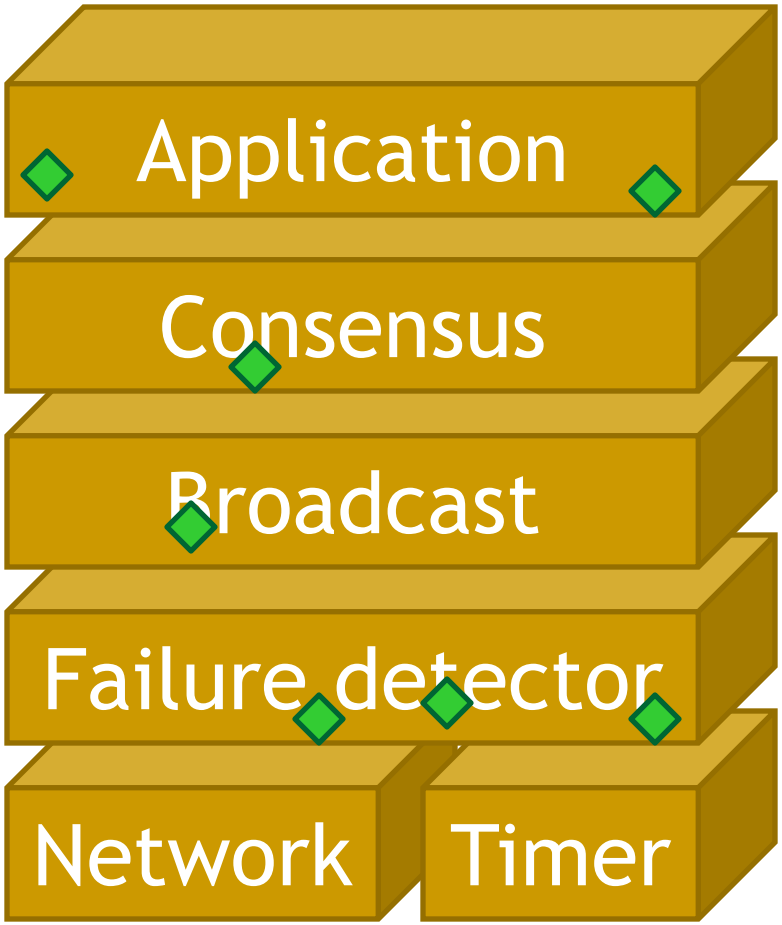
by composing distributed protocols



Implemented as reactive components

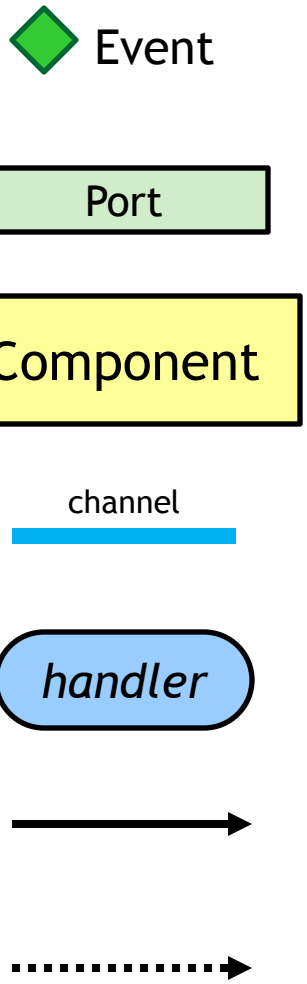


with message-passing concurrency



Concepts in Kompics

- Event
- Port
- Component
- Channel
- Handler
- Subscription
- Publication / Trigger



Events

Event

- Events are passive immutable objects
  - with typed attributes / fields
- Events are typed and can be sub-typed

```
class Message extends Event {
   Address source;
   Address destination;
}
class DataMessage extends Message {
   Data data;
   int sequenceNumber;
}
```

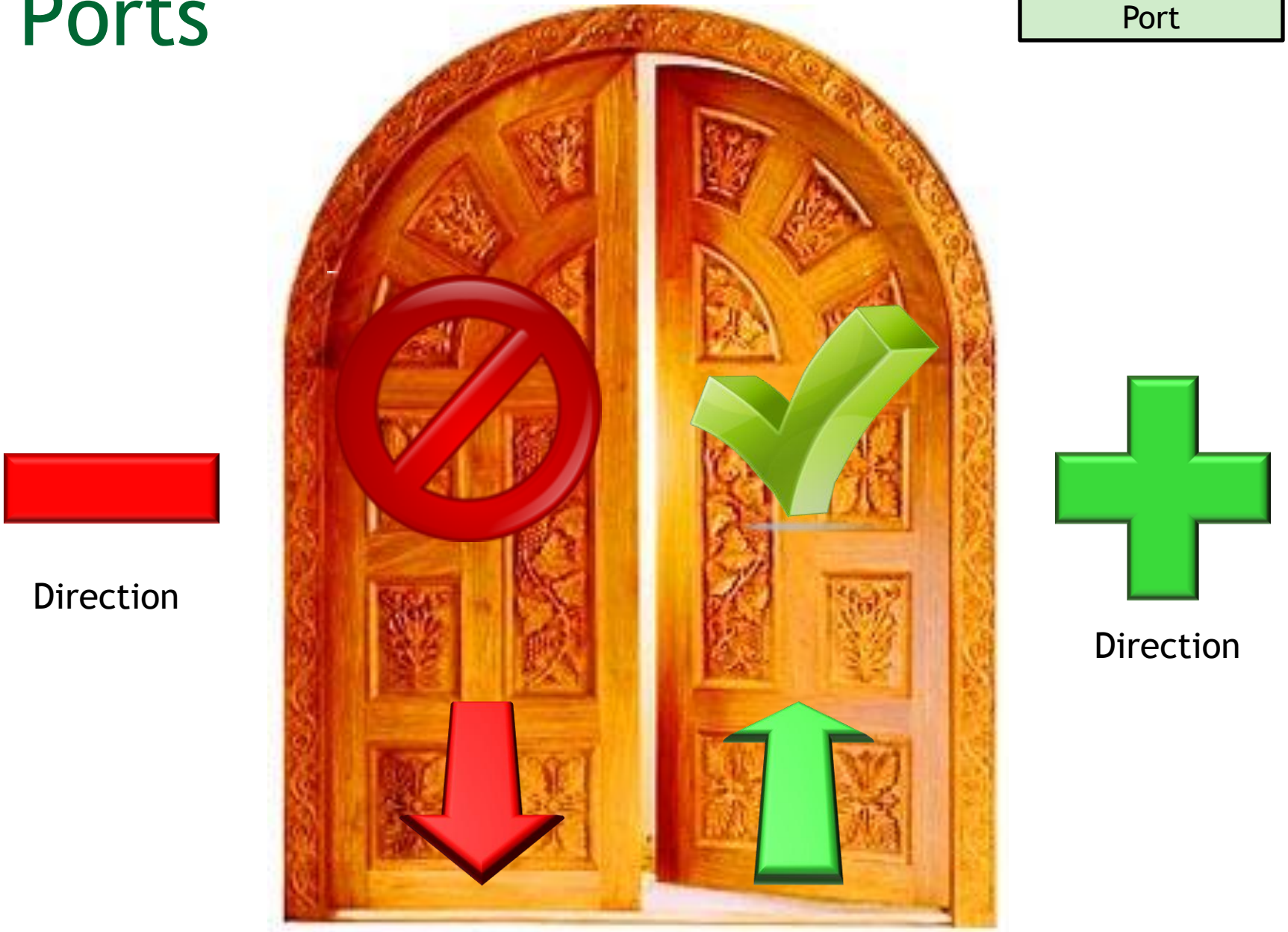
Message

DataMessage

DataMessage $\subseteq$ Message

Ports

Port

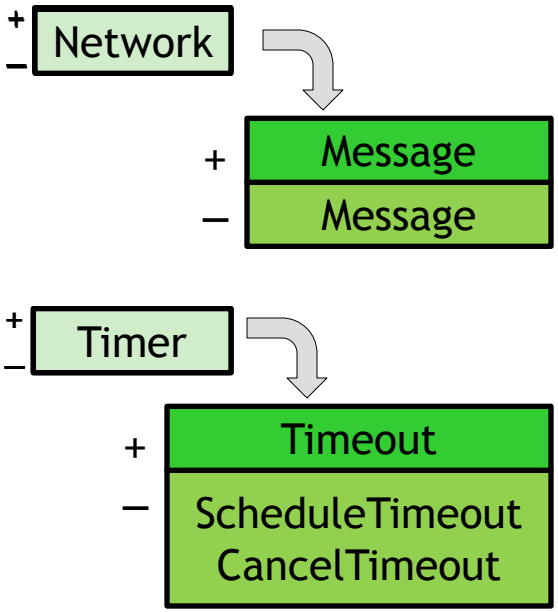


Ports

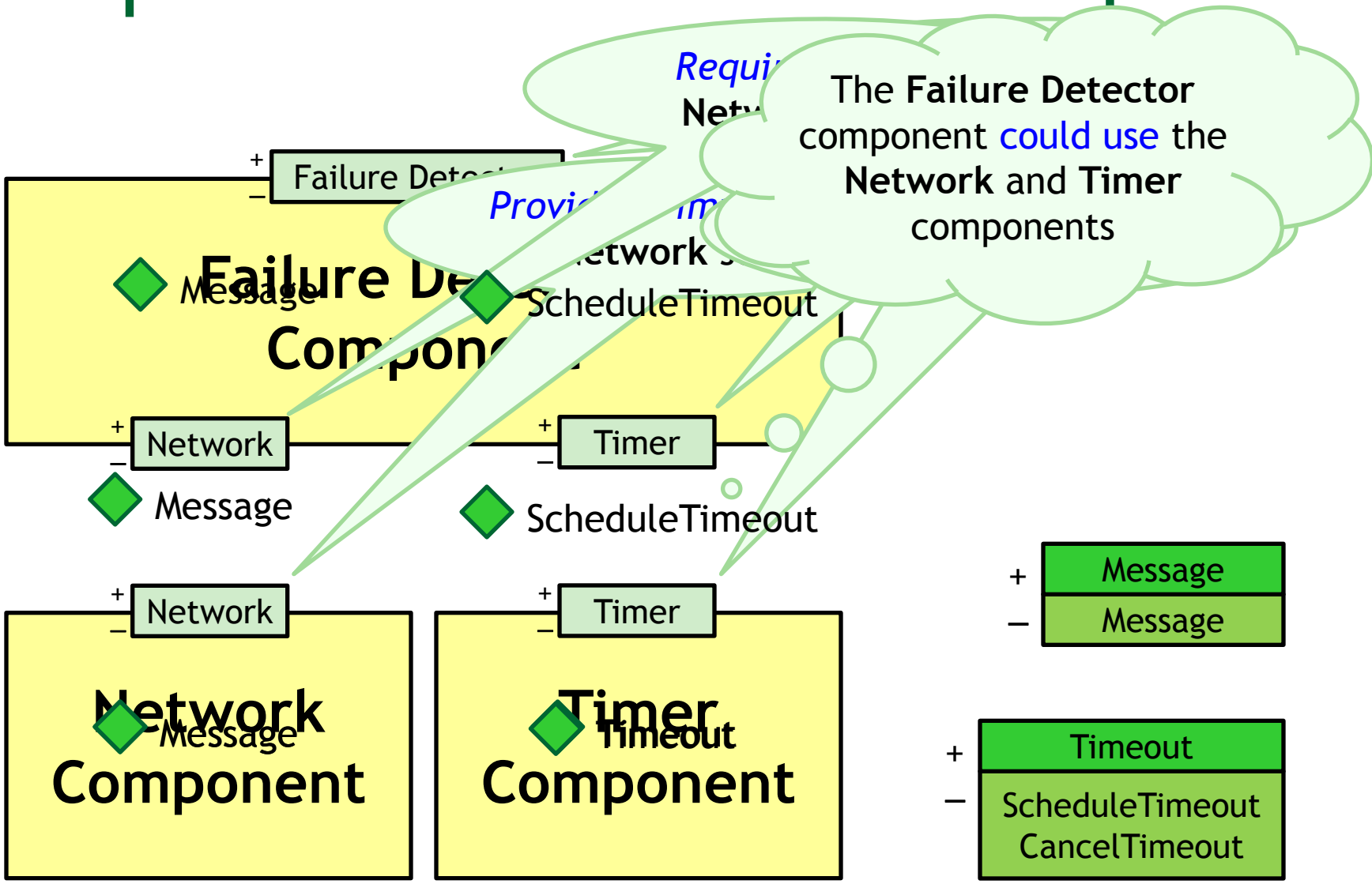
Port

- are bidirectional event-based comp interfaces
  - have a **positive (+)** and a **negative (–)** direction
- A *port type* consists of 2 sets of event types
  - one set of event types for each direction, **+** and **–**
  - represents a service/protocol abstraction

```
class Network extends PortType {{
        positive(Message.class);
        negative(Message.class);
}}
class Timer extends PortType {{
        positive(Timeout.class);
        negative(ScheduleTimeout.class);
        negative(CancelTimeout.class);
}}
```



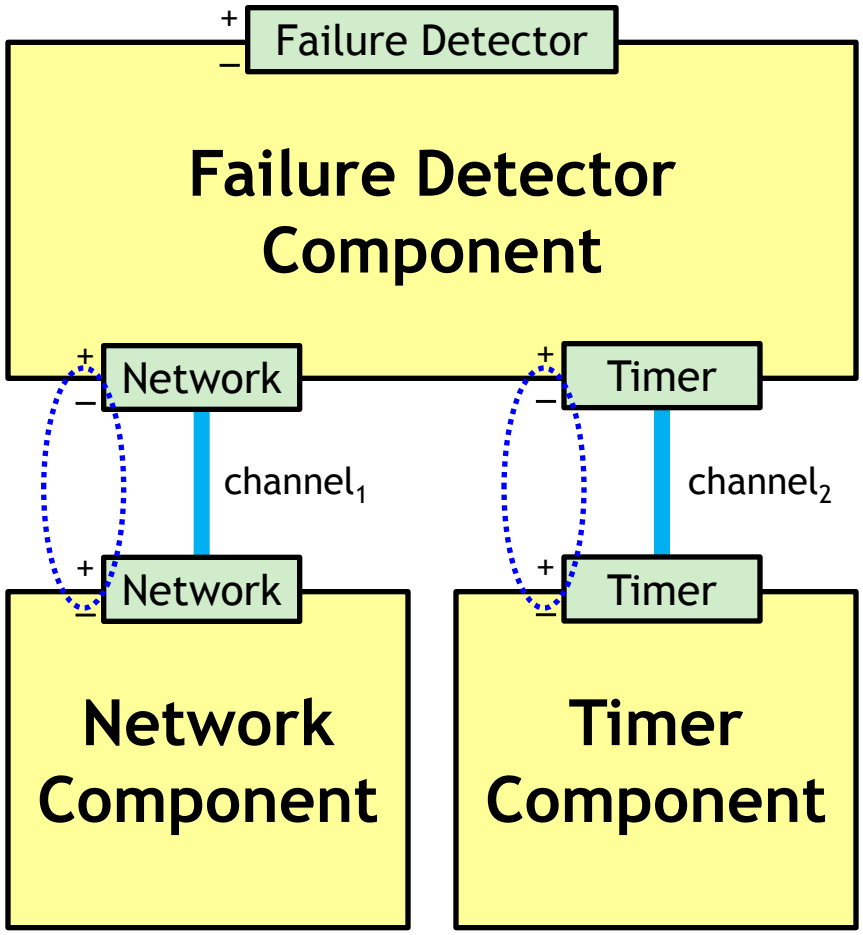
Components with Ports Example



Channels

channel

- Channels connect *complementary* ports of the *same* type
  - + to −
  - − to +
- Channels forward events in *FIFO* order in *both* directions



Event handlers

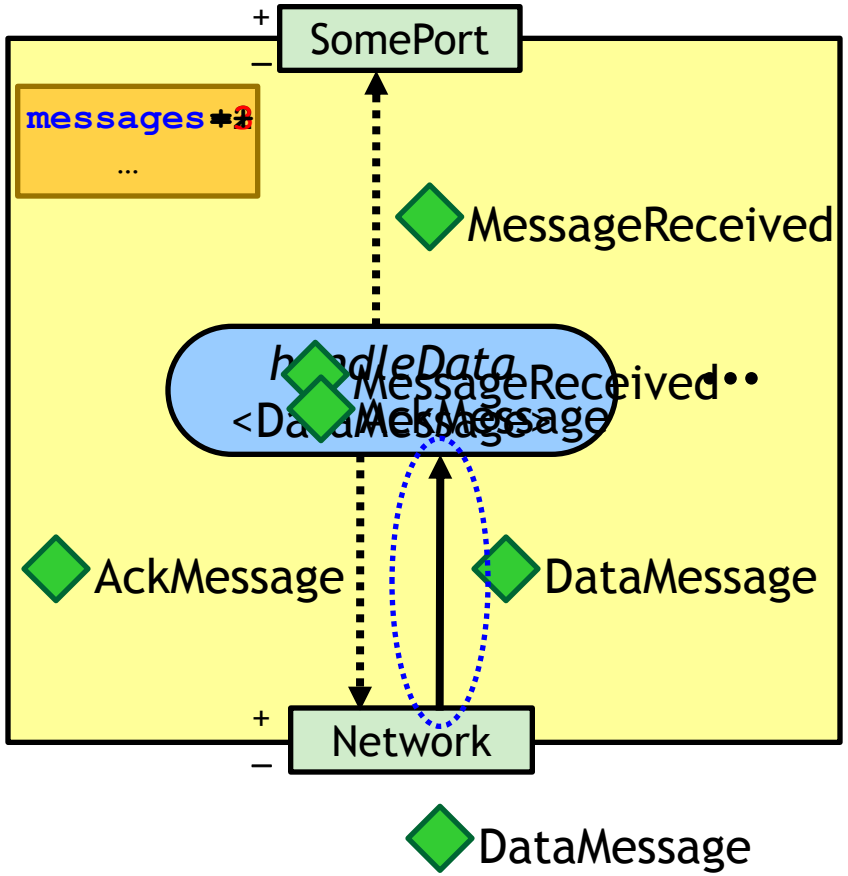
handler

- A handler is a first-class component procedure
- accepting a particular type of events
- executed reactively upon receiving an event
  - may mutate the local state and trigger new events
  - handlers of one component are mutually exclusive

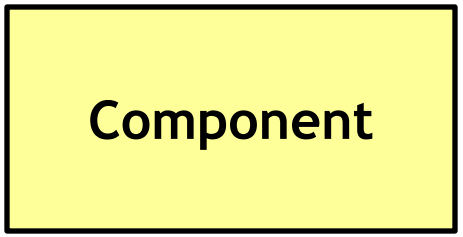
```
Handler<DataMessage> handleData = new Handler<DataMessage>() {
    public void handle(DataMessage dm) {
        messages++;                          // component state update
        trigger(new MessageReceived(dm.data), somePort);
        trigger(new AckMessage(myAddress, dm.source,
                dm.sequenceNumber), network); // event triggering
  }
};
```

Subscriptions & Publications

- A subscription binds an event handler, *h*, to a local component port *p*
- Let *e* be the type of *h*
  - let *f* be a supertype of *e*
  - *f* must come in on *p*
- After subscription
  - *h* will handle all events of any type *d*, subtype of *e*, coming in on *p*

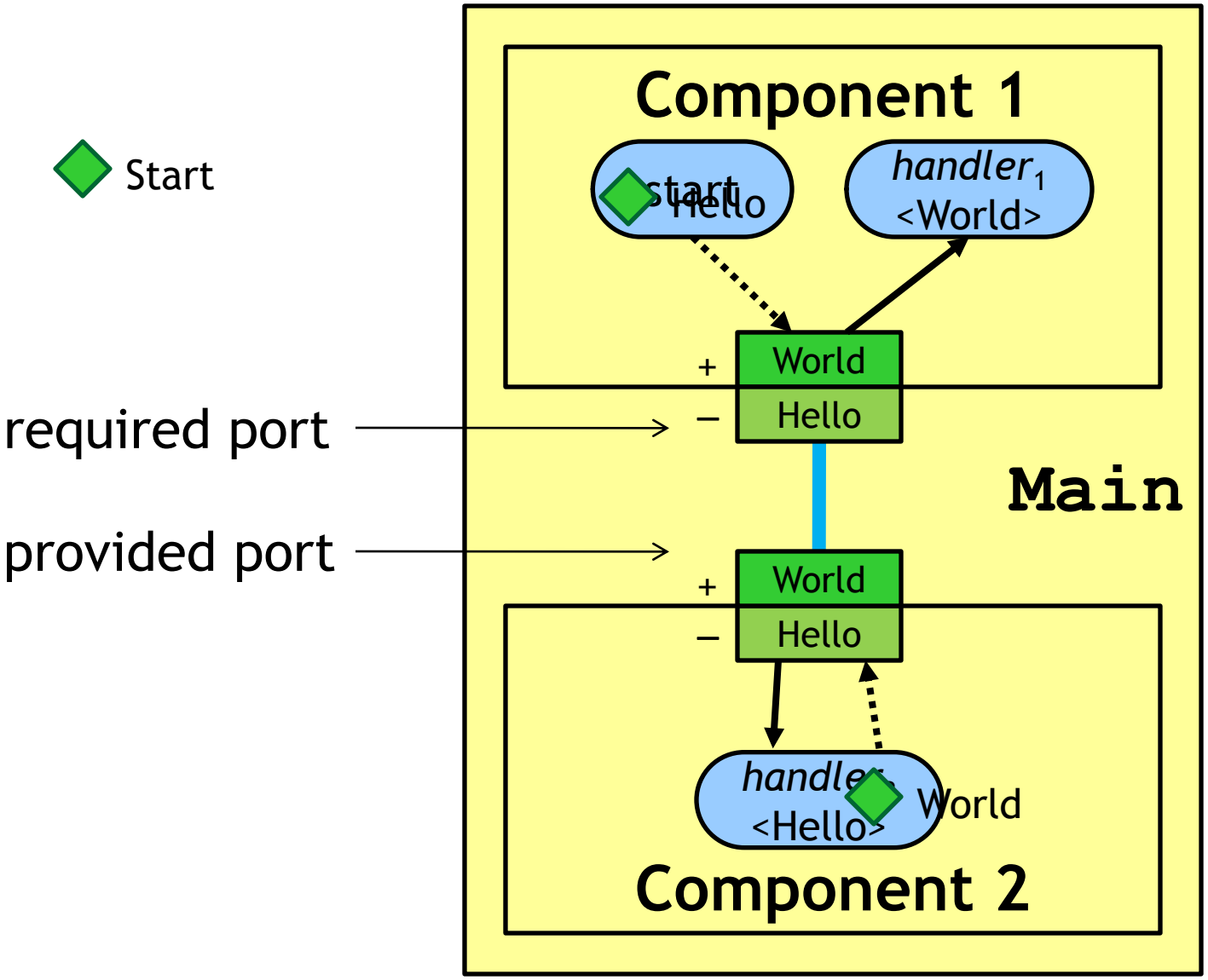


Components



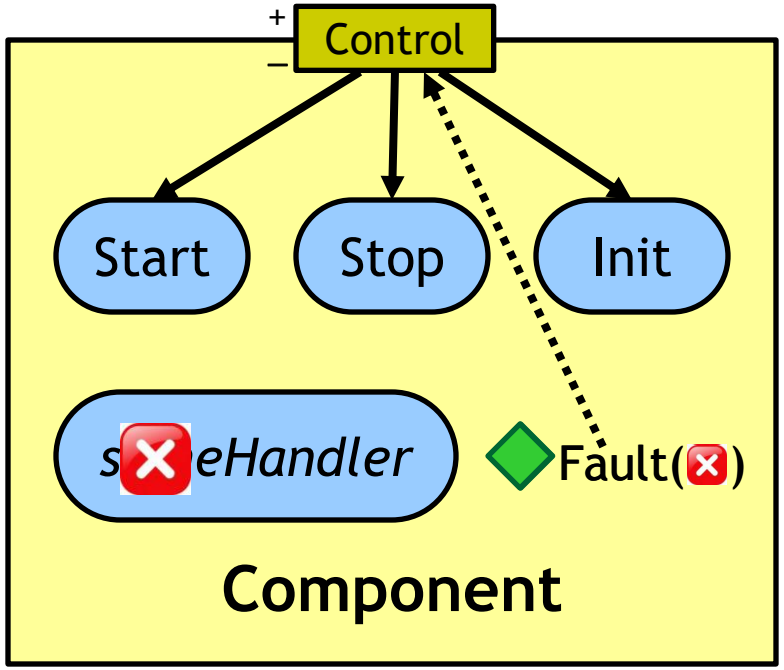
- are instantiated from component definitions
  - component definitions are Java classes
- A component instance is an object containing
  - local state variables
  - ports (provided or required interfaces)
  - event handlers
  - subscriptions
  - encapsulated subcomponents
  - channels
- form a containment hierarchy rooted at **`Main`**

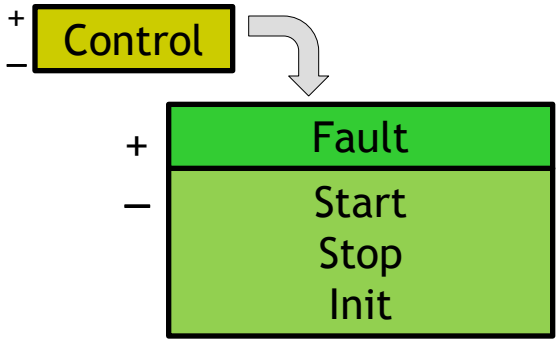
Hello World! Example



Control port

- Every component has a Control port
  - by default (provided)
  - not shown in diagrams
  - allows component to handle lifecycle events
- Exceptions / faults not caught inside a handler
  - wrapped into Fault event
  - triggered on control port

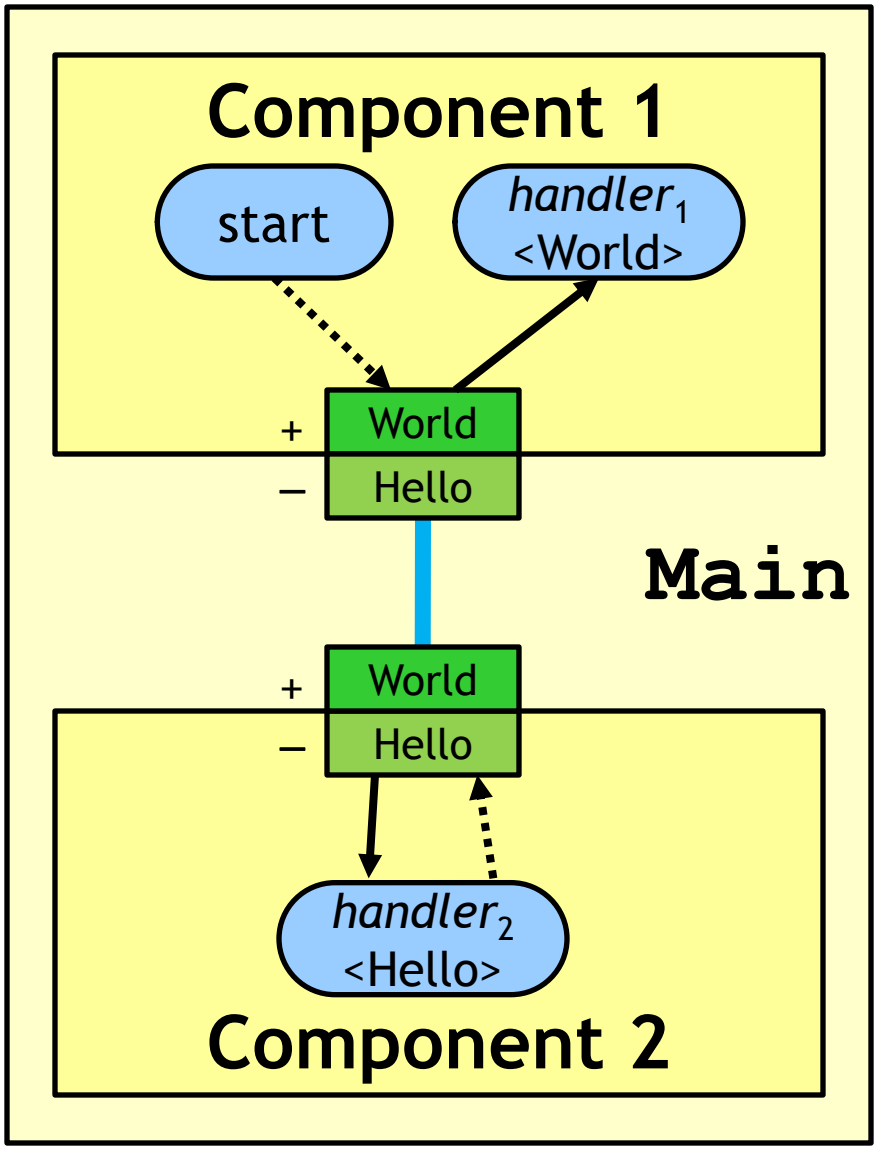


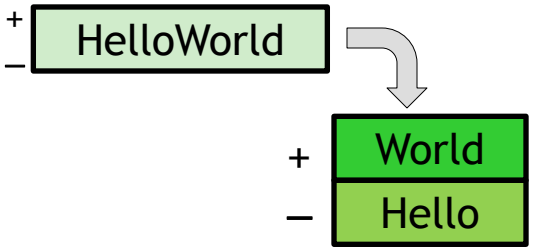


Init event

- Configures component with initial parameters
  - Specialized for each component definition
  - E.g. **`MyComponent`** handles **`MyInit`** ◆ MyInit ⊆ ◆ Init
- Init guaranteed to be the first event handled
  - "If a component subscribed an Init handler to its control port in its constructor, the component will not handle any other event before an Init event!"
  - The component will not do anything if the parent component does not trigger an Init event on its control port!

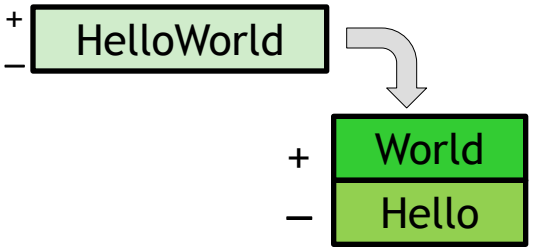
Hello World! Example





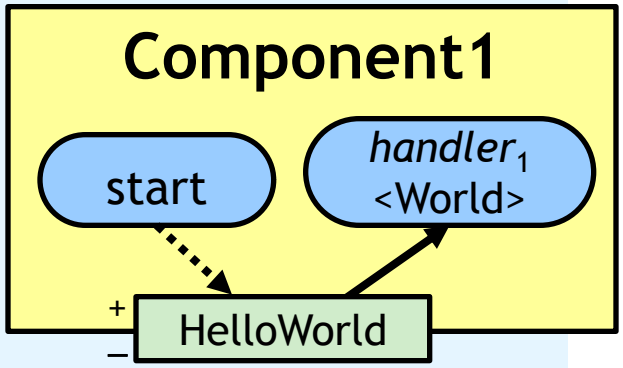
Source code: Events and Port

```
public final class Hello extends Event {
   private final String message;
   public Hello(String m) {
        message = m;
   }
   public String getMessage() {
        return message;
   }
}

public final class World extends Event {
}

public class HelloWorld extends PortType{{
   positive(World.class);
   negative(Hello.class);
}}
```



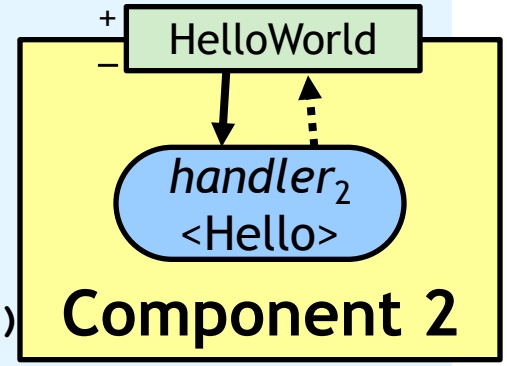
Source code: Component1

```
public class Component1 extends ComponentDefinition {

  private Positive<HelloWorld> hwPort = positive(HelloWorld.class);

  public Component1() {
    System.out.println("Component1 created.");
    subscribe(startHandler, control);
    subscribe(worldHandler, hwPort);
  }

  Handler<Start> startHandler = new Handler<Start>() {
    public void handle(Start event) {
      System.out.println("Component1 started. Triggering Hello...");
      trigger(new Hello("Hi there!"), hwPort);
  }};
  Handler<World> worldHandler = new Handler<World>() {
    public void handle(World event) {
      System.out.println("Component1 received World event.");
  }};
}
```

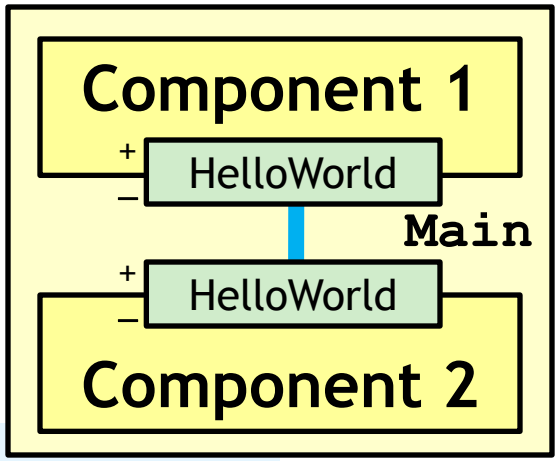


Source code: Component2

```
public class Component2 extends ComponentDefinition {
  private Negative<HelloWorld> hwPort = negative(HelloWorld.class);
  public Component2() {
    System.out.println("Component2 created.");
    subscribe(startHandler, control);
    subscribe(helloHandler, hwPort);
  }
  Handler<Start> startHandler = new Handler<Start>() {
    public void handle(Start event) {
      System.out.println("Component2 started.");
  }};
  Handler<Hello> helloHandler = new Handler<Hello>() {
    public void handle(Hello event) {
      System.out.println("Component2 received Hello event with "
                         + "message: " + event.getMessage());
      trigger(new World(), hwPort);
  }};
}
```



Source code: **Main**



- **Main** is a Java main class

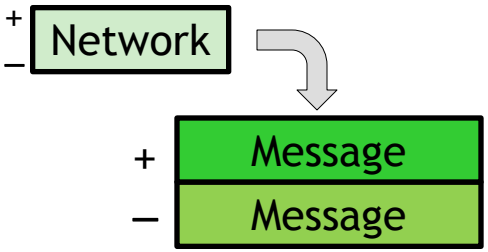
```
public class Main extends ComponentDefinition {
   private Component component1, component2;
   public Main() {
        System.out.println("Main created.");
        component1 = create(Component1.class);
        component2 = create(Component2.class);
        connect(component1.getNegative(HelloWorld.class),
                component2.getPositive(HelloWorld.class));
   }
   public static void main(String[] args) {
        Kompics.createAndStart(Main.class);
        Kompics.shutdown();
   }
}
```

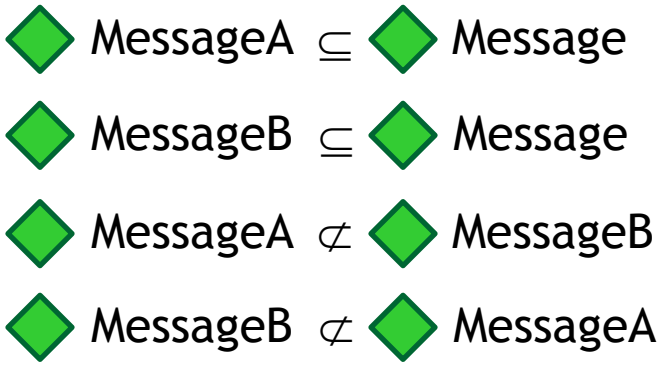
Output

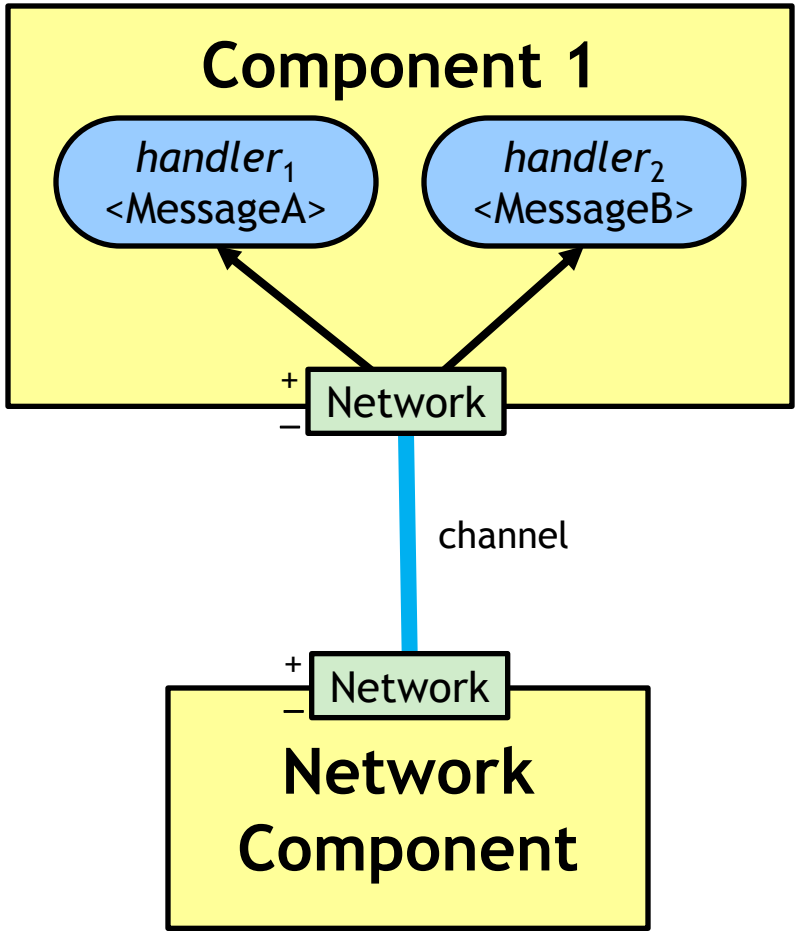
```
prompt:$ java Main
Main created.
Component1 created.
Component2 created.
Component1 started. Triggering Hello...
Component2 started.
Component2 received Hello event with message: Hi there!
Component1 received World event.
prompt:$ _
```

- Kompics runtime creates and starts **`Main`**
  - **`Main`** recursively creates and starts **`c1`**, **`c2`**
  - **`c1`**'s start handler triggers Hello event
    - handled by **`c2`**, which triggers World event
      - handled by **`c1`**

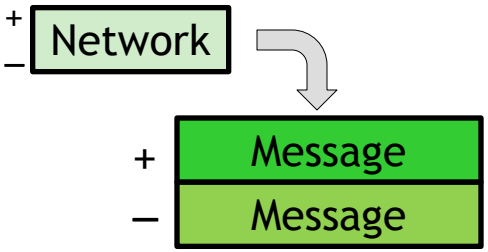
Publish / Subscribe

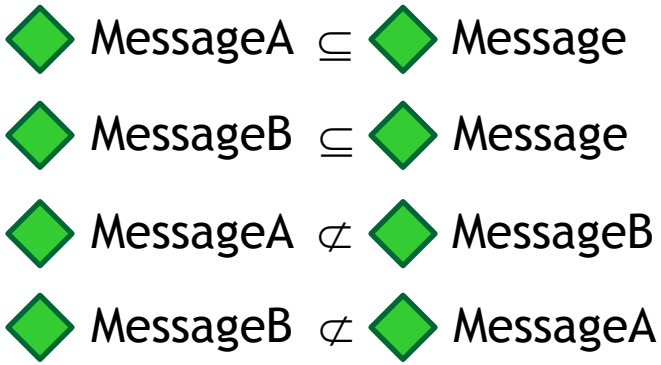


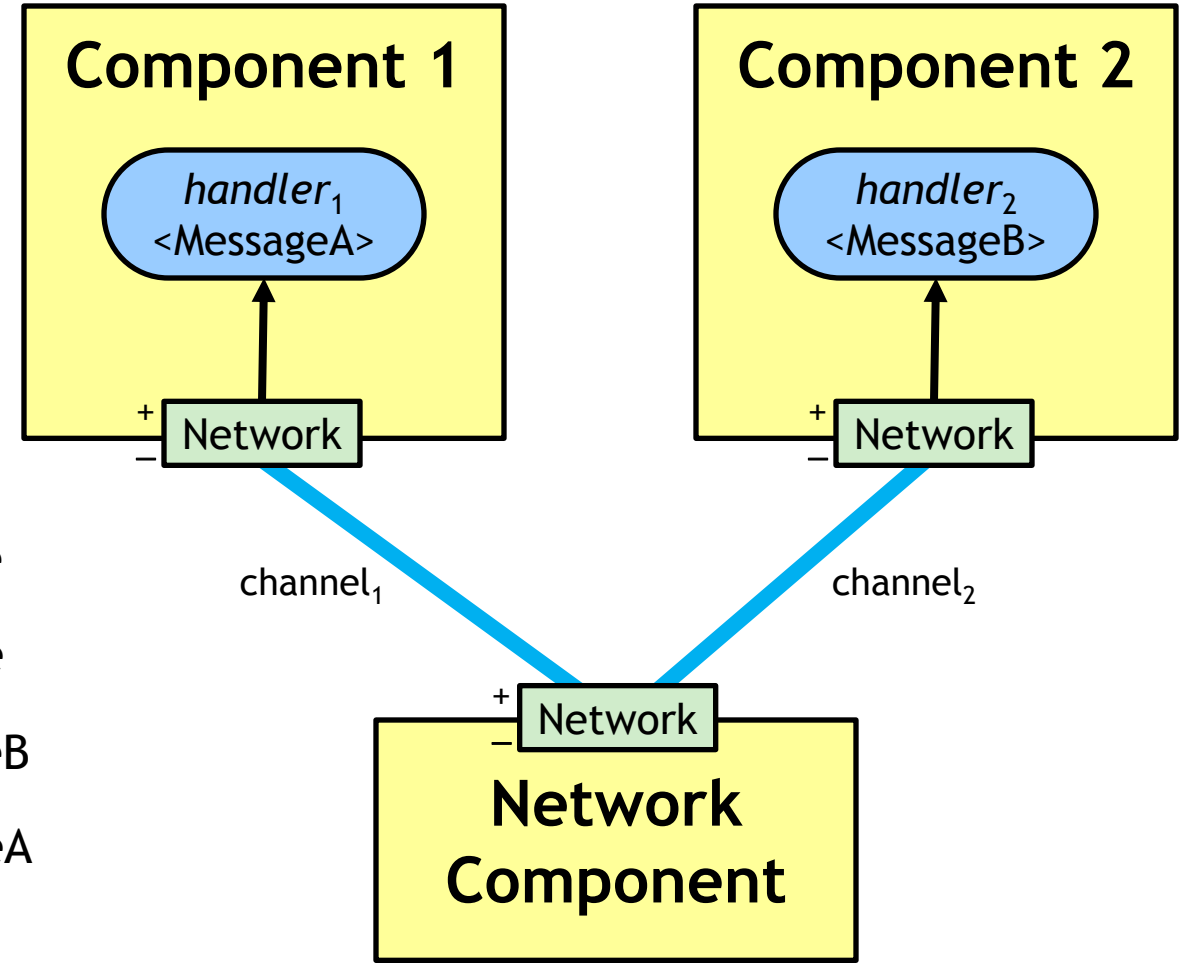




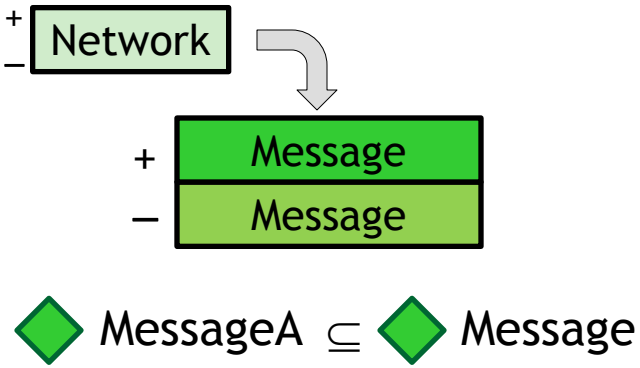
Publish / Subscribe

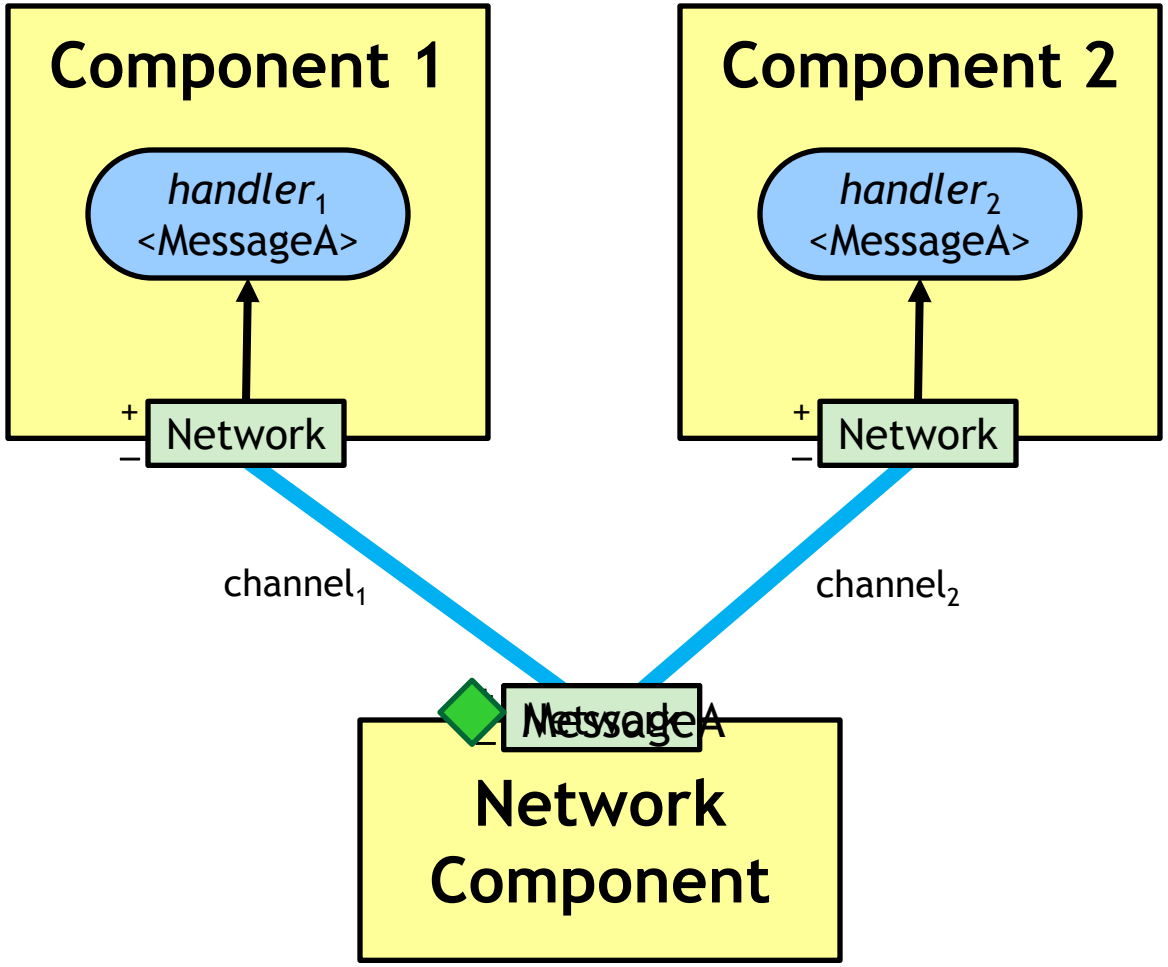




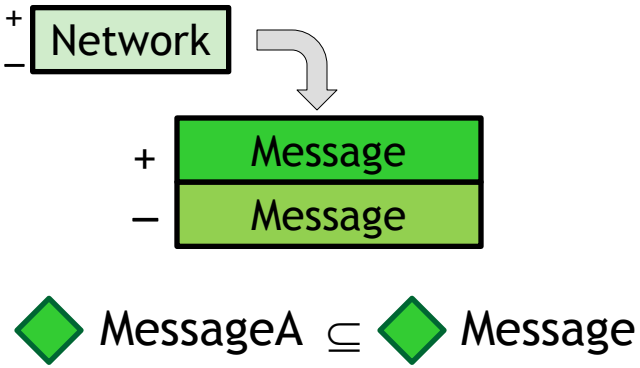


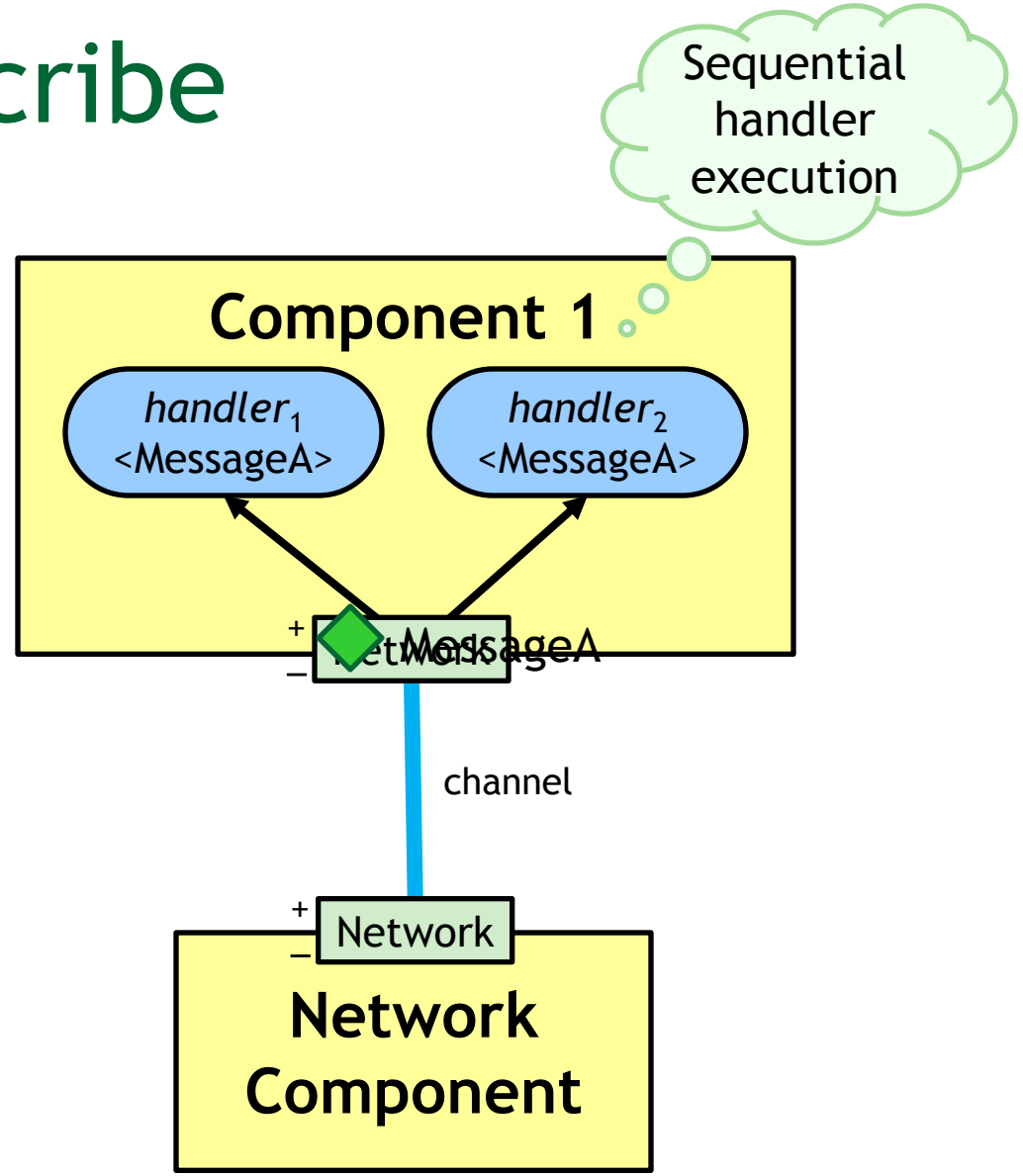
Publish / Subscribe



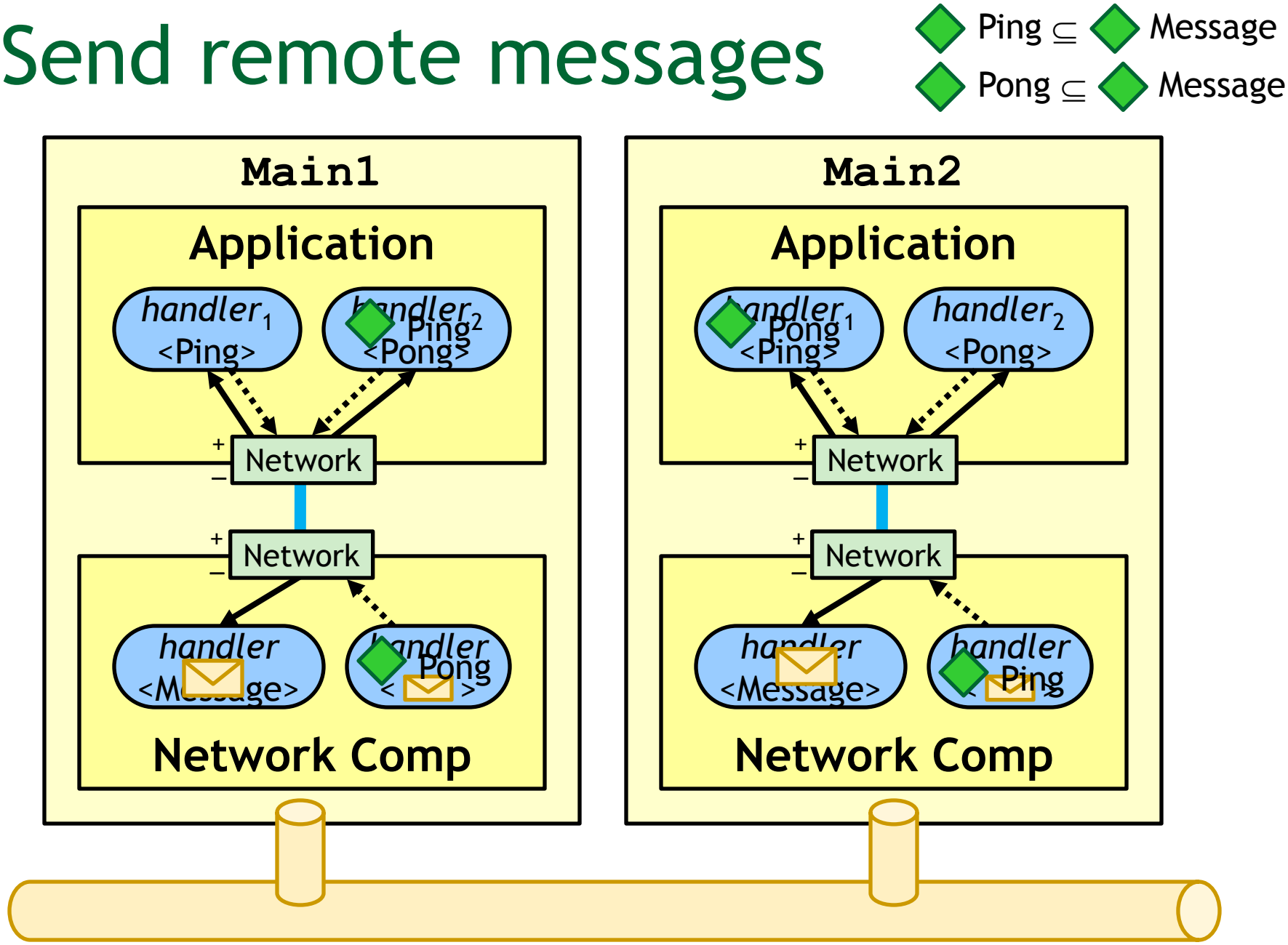


Publish / Subscribe

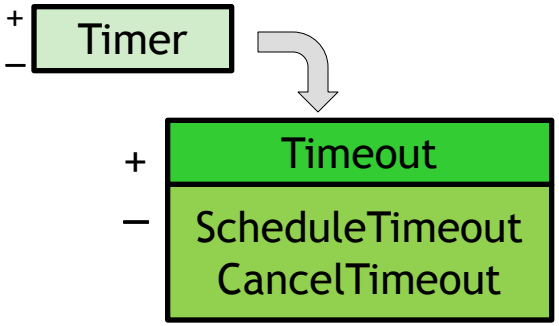


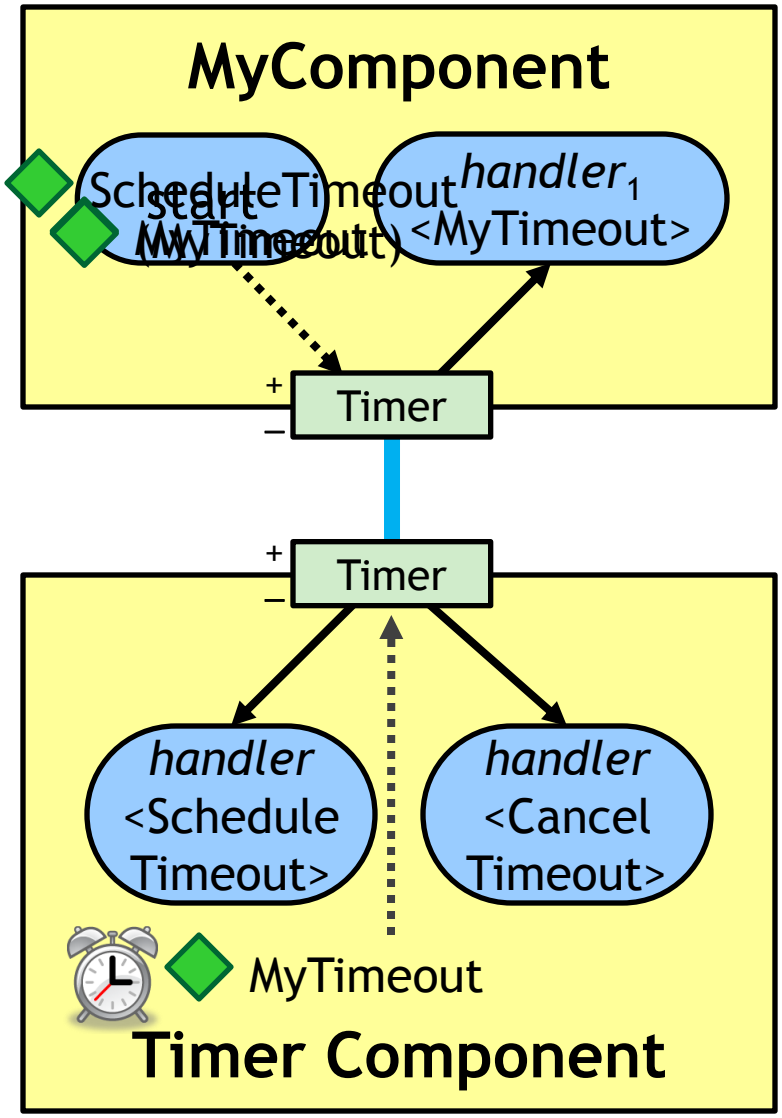


Send remote messages

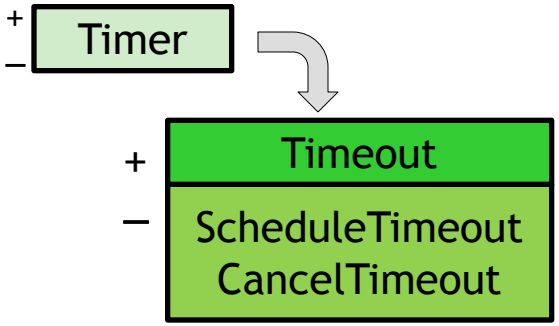


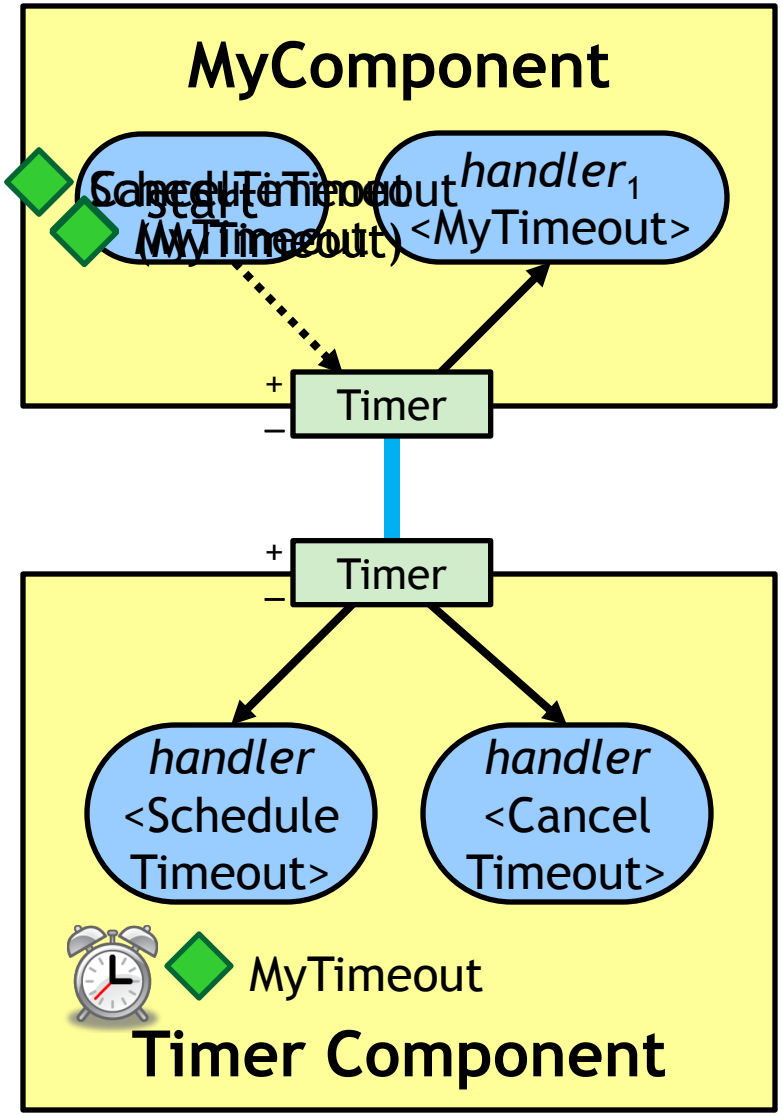
Scheduling a Timeout





Canceling a Timeout





Scheduling & Canceling a Timeout

MyTimeout ⊆ Timeout

```
class MyComponent extends ComponentDefinition {
  Positive<Timer> timer = positive(Timer.class); //required

  Handler<Start> startHandler = new Handler<Start>() {
    public void handle(Start event) {
      // scheduling a timeout
      long delay = 5000; // milliseconds
      ScheduleTimeout st = new ScheduleTimeout(delay);
      st.setTimeoutEvent(new MyTimeout(st));
      UUID timeoutId = st.getTimeoutEvent().getTimeoutId();
      trigger(st, timer);
      // canceling a timeout
      CancelTimeout ct = new CancelTimeout(timeoutId);
      trigger(ct, timer);
  }};
}
```

Software engineering view

- Events and ports are interfaces
  - service abstractions, modules
  - packaged together as libraries
- Components are implementations
  - provide or require modules / interfaces
  - dependencies on provided / required modules
    - expressed as library dependencies
    - multiple implementations for some module
      - separate libraries
    - deploy-time composition

Kompics software

- Open source project
- Latest release: 0.4.2.6
- http://kompics.sics.se
  - Java source code (SVN)
  - Javadocs
  - Test reports
- Ports and components (e.g. Network, Timer)
  - are packaged as JARs
    - You can use Maven to download them automatically
    - see http://kompics.sics.se/trac/browser/trunk/pom.xml

Relation to the textbook

How we use Kompics to model the abstractions in the textbook / course

Correspondences

Textbook / Lectures Kompics

An implemented port is positive on the

A required / used port is negative on

```
ScheduleTimeout st = new ScheduleTimeout(δ);
st.setTimeoutEvent(new MyTimeout(st));
trigger(st, timer);
```

Ports

vn (negative, −)

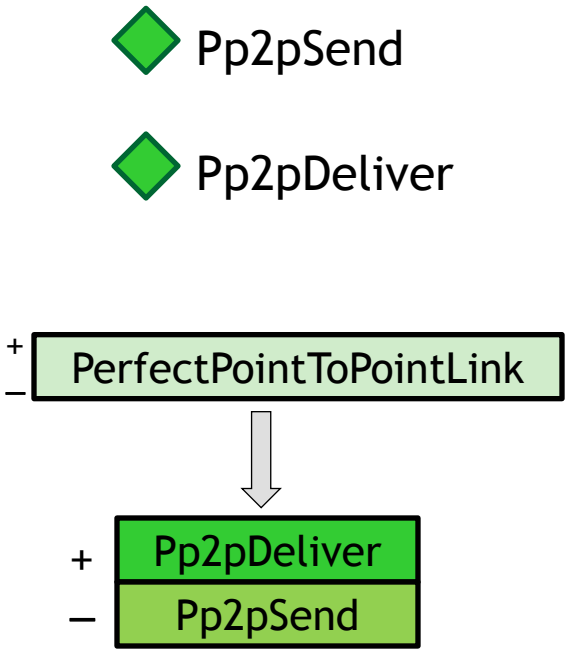
(positive, +)

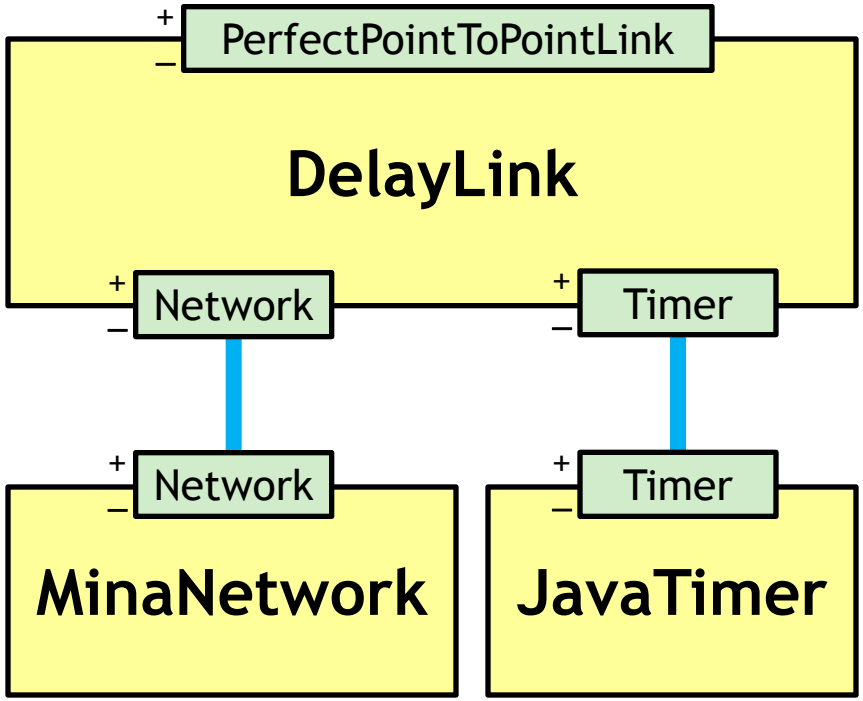
- Algorithms
- Implements **SomeModul**
- Uses **SomeModule**
- *startTimer*(δ, MyTimeout);

nponents

- **egative<SomePort>**
- **Positive<SomePort>**
- Trigger ScheduleTimeout event on Timer port

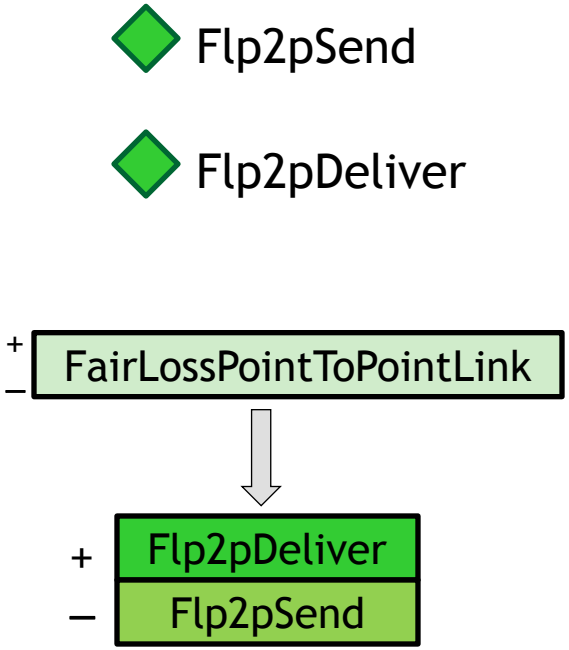
PerfectPointToPointLink

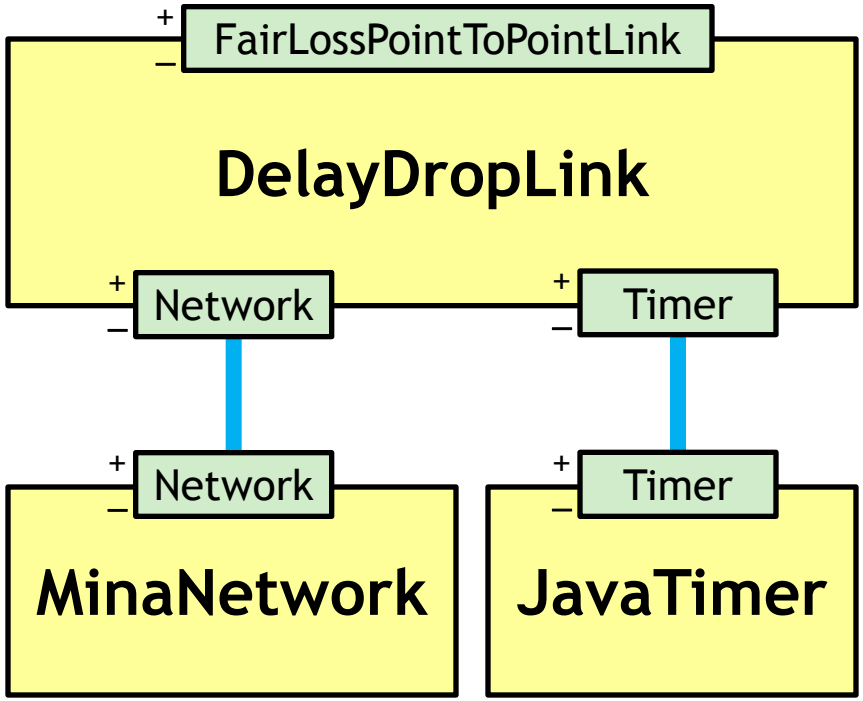




- Messages sent over perfect links should extend the Pp2pDeliver event

FairLossPointToPointLink





- Messages sent over fair-loss links should extend the Flp2pDeliver event

Assignments framework

How we run distributed systems and create experiment scenarios

Experimentation framework

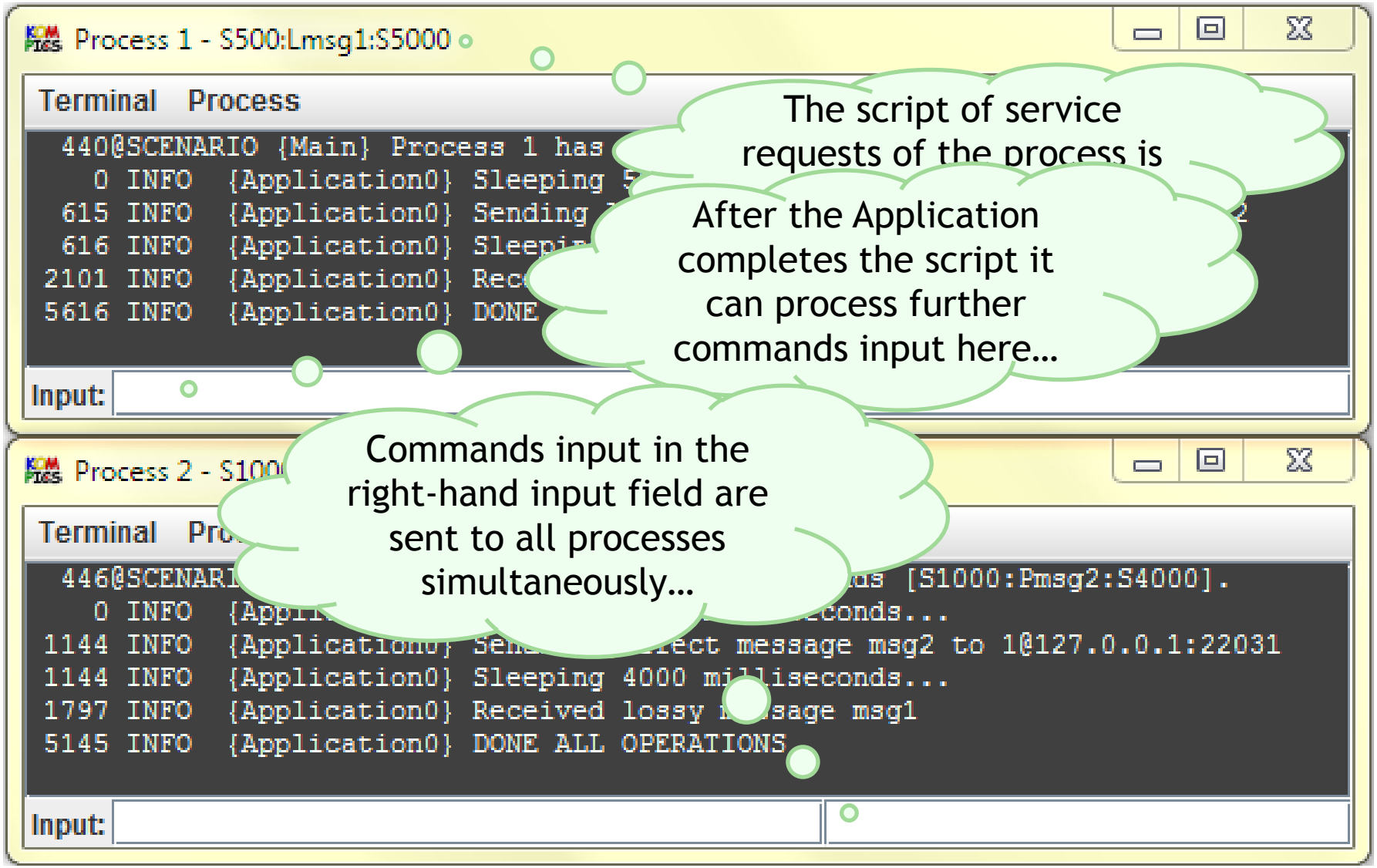
- Define network topologies
  - processes and their addresses: <id, IP, port>
  - properties of links between processes
    - latency (ms)
    - loss rate (%)
- Define execution scenarios
  - the sequence of service requests initiated by each process in the distributed system
- Experiment with various topologies/scenarios
  - Launch all processes locally on one machine

Distributed System Launcher

- Read complete documentation at
  - http://kompics.sics.se/trac/wiki/DistributedSystemLauncher

```
public final class Experiment1 {
  public static final void main(String[] args) {
    Topology topology1 = new Topology() {{
       node(1, "127.0.0.1", 22031);
       node(2, "127.0.0.1", 22032);
       link(1, 2, 1000, 0).bidirectional();
    }};
    Scenario scenario1 = new Scenario(Main.class) {{
       command(1, "S500:Lmsg1:S5000");
       command(2, "S1000:Pmsg2:S4000");
    }};
    scenario1.executeOn(topology1);
  }
}
```

Process window



To terminate an experiment press

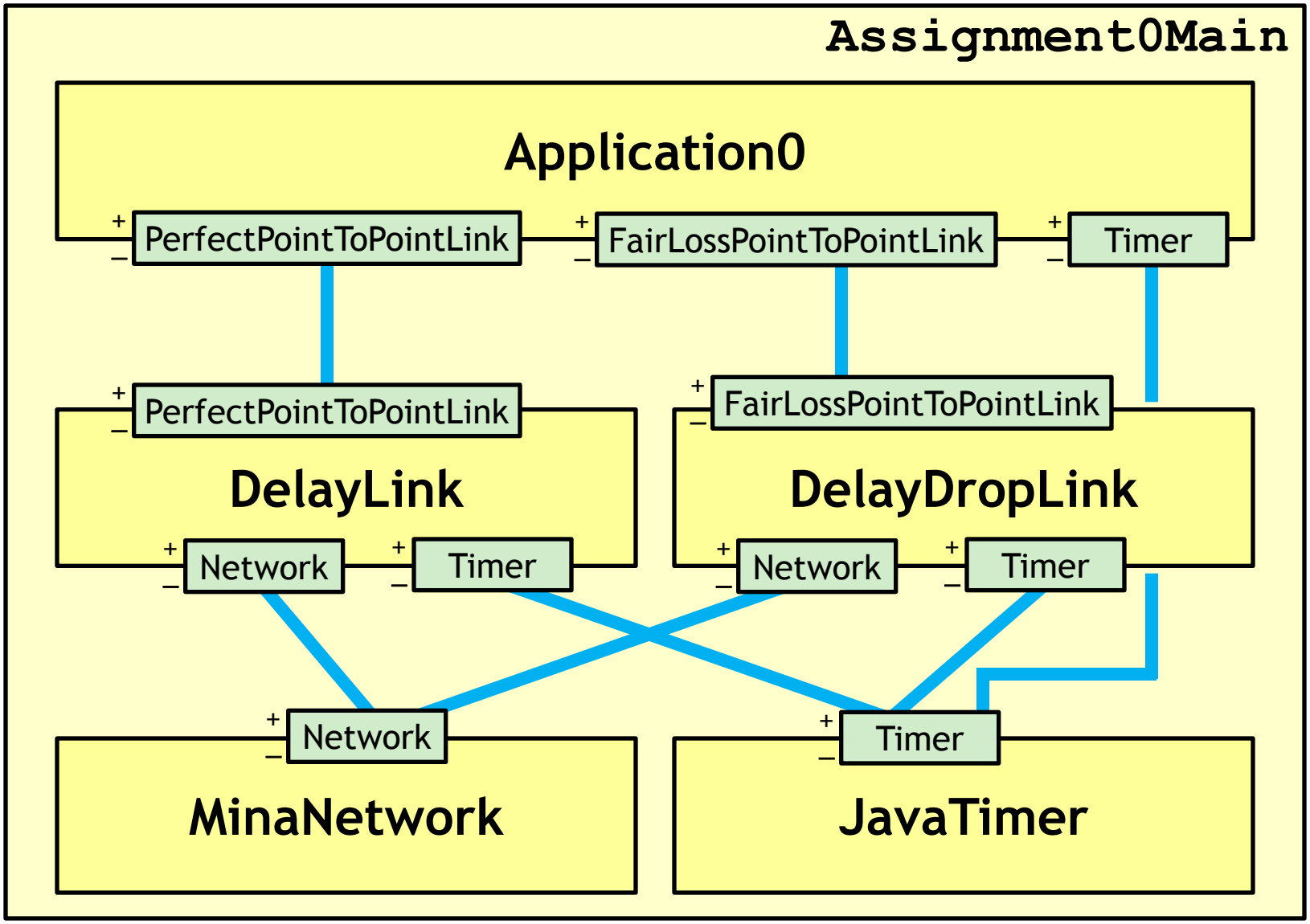
Ctrl+K

Assignments Kit

- Source code fo
- Source code fo
  - skeleton for a typ
  - processes send messages to all their neighbors
- Kompics and third party JARs
  - add them to your projects as libraries
  - **`log4.properties`**: control logging output
- Miscellaneous
  - sample Eclipse project files
  - sample **`pom.xml`**, if you want to use Maven

Be careful not to suppress exceptions and remember what you suppressed. When you debug something you can switch all loggers to DEBUG.

Architecture for Assignment0



First assignment

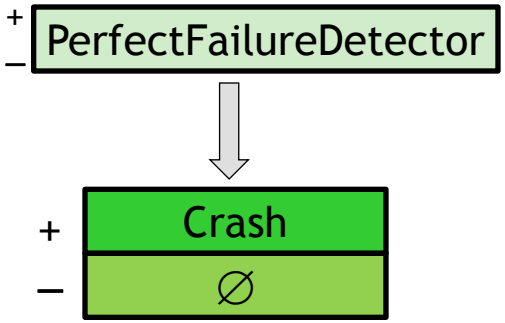
Failure detectors

First assignment: failure detectors

- Implement and experiment with
  - Perfect Failure Detector
    - Synchronous model
  - Eventually Perfect Failure Detector
    - Partially synchronous model
- Failure detectors discussed in lecture 4
- Read chapters 1 and 2 of the textbook!
- For requirements see Assignment1.pdf
- Use the algorithms in Assignment1.pdf!
  - Not the ones in the textbook!

Suggestions for PFD

Crash

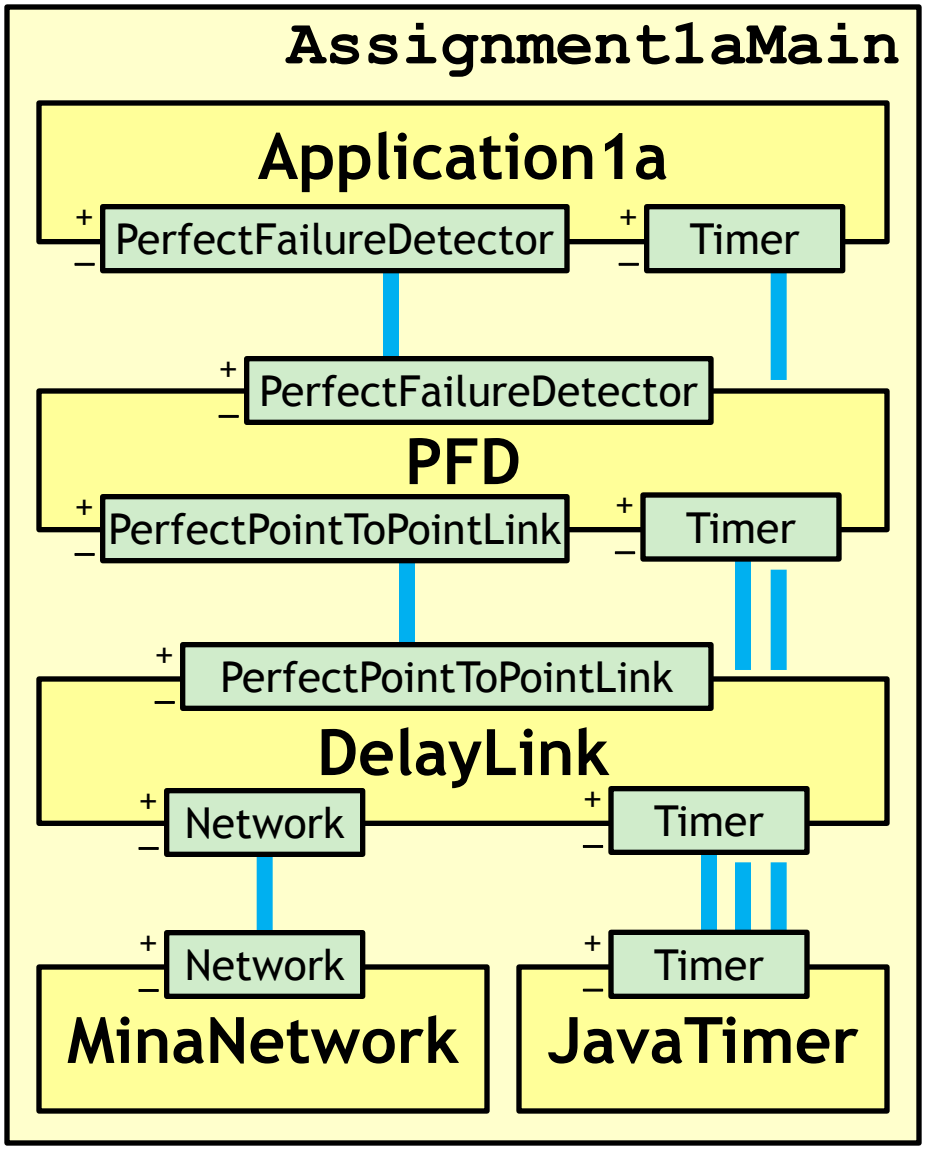


CheckTimeout ⊆ Timeout

HeartbeatTimeout ⊆ Timeout

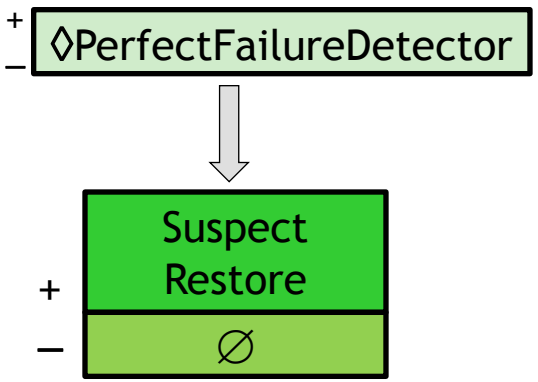
HeartbeatMessage ⊆ Pp2pDeliver

PfdInit ⊆ Init



Suggestions for EPFD

Suspect Restore

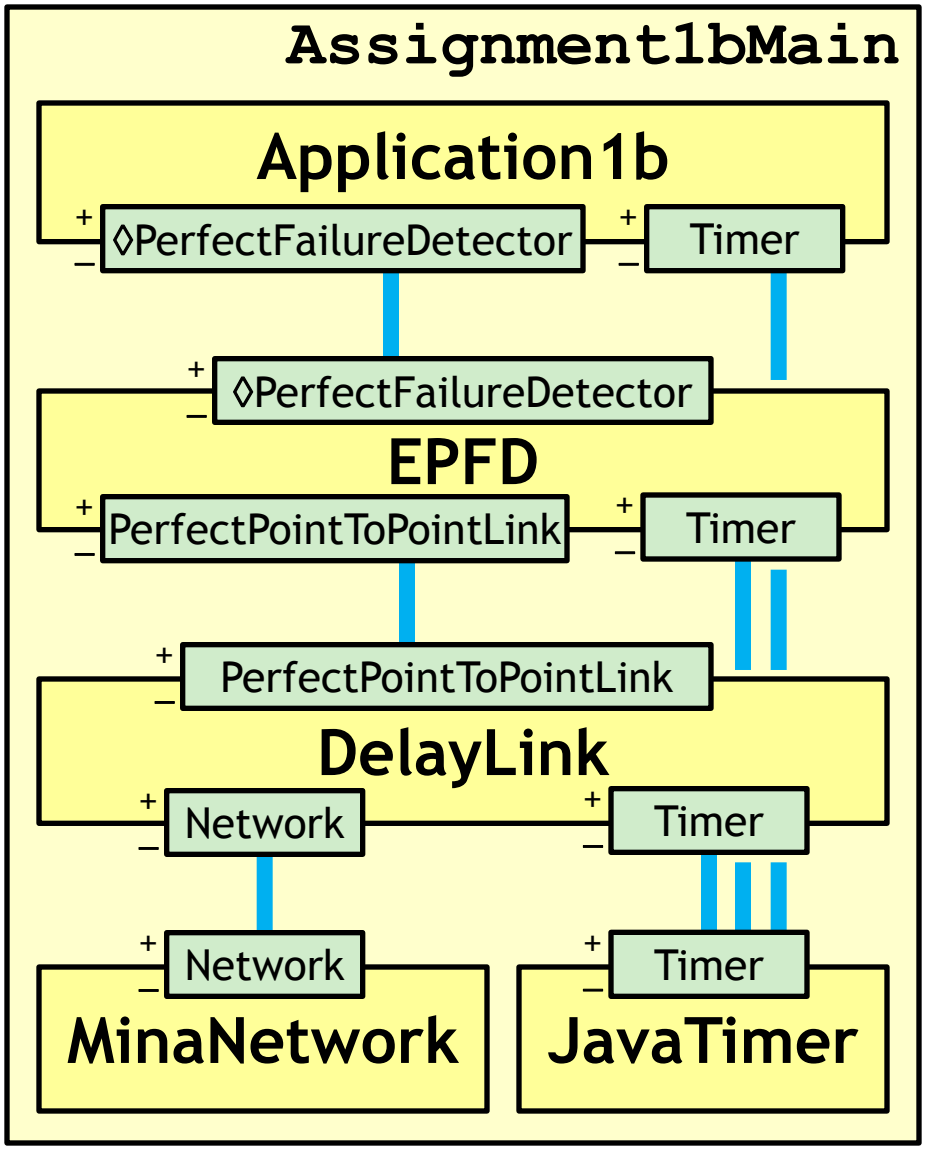


CheckTimeout ⊆ Timeout

HeartbeatTimeout ⊆ Timeout

HeartbeatMessage ⊆ Pp2pDeliver

EpfdInit ⊆ Init



Deliverables & Submission

- Deliverable 1: Source code ZIP archive
  - events, ports, components
  - exercise experiments (topologies and scenarios)
- Deliverable 2: PDF report
  - experiments observations and conclusions
  - answers to questions
- Submit deliverables to id2203.2010@gmail.com
  - Subject line: [assignmentX]LastName1LastName2
  - Before next tutorial starts (13:00 on deadline date)
  - Late submissions do not count

Presentations

- Present your implementation and experiments in class during next tutorial

- Both group members
  - Answer a few questions
  - You can present earlier (after some lecture)
  - You cannot present after the deadline

Forum

- Please use the forum to ask questions

http://castor.sics.se/forum

- Try to avoid direct emails

Good luck!