# ID2203 Tutorial 1 - Failure Detectors

Cosmin Arad
`icarad(@)kth.se`

Tallat M. Shafaat
`tallat(@)kth.se`

Handout: February 1, 2012
Due: February 8, 2012

## 1 Introduction

This tutorial is accompanied by slides explaining the Kompics framework, Kompics jar-files, examples and helper source code. To begin with, you should read the Kompics tutorial and slides and try out the described examples.

Every assignment will require you to implement one or more distributed programming abstractions, sometimes relying on lower-level abstractions, that you will have implemented in previous assignments. To test your implementations you will typically need to run a number of processes implementing those abstractions.

Each algorithm implementation will consist of a number of Kompics components. Besides the components implementing your distributed programming abstractions, the following components provide helpful functionality that will be used by your components.

- **DelayLink:** For sending and receiving messages with delays. This component provides an implementation of the Perfect Links algorithm given in the course book.
- **DelayDropLink:** For sending and receiving messages with delays, while some messages get dropped. This component provides an implementation of the Fair-loss Links algorithm given in the course book.

## 2 Assignment

The goal of this tutorial is to get accustomed to implementing and understanding failure detectors for the synchronous and partially synchronous system models.

On pages 5 and 6 you have the algorithms for two failure detectors. Algorithm 1 implements a perfect failure detector (PFD). Algorithm 2 implements an eventually perfect failure detector (EPFD). Alternatively, you can use the PFD and EPFD algorithms from the textbook (page 51 and page 55). Both algorithms use perfect point-to-point links. Using Kompics, you will implement the two failure detectors, reusing the network component, the timer component, and the delay link component. Component architecture diagrams are given in the Tutorial 1 slides.

Algorithm 1 gives a PFD. Periodically, it sends heartbeat messages and expects to receive heartbeat messages from other processes. If it detects that a process has crashed, the PFD component triggers a CrashEvent. Thus, you will need a HeartbeatMessage and a CrashEvent. Tutorial 1 slides give a detailed account of the events and ports you need to implement.

Algorithm 2 gives an EPFD. Periodically, it sends heartbeat messages and expects to receive heartbeat messages from other processes. If it suspects that a process has crashed, the EPFD component triggers a SuspectEvent. If it later detects that some process that is suspected is in fact still alive (by receiving a heartbeat from the suspected node), the EPFD component triggers a RestoreEvent. Thus, you will require a SuspectEvent and a RestoreEvent.

The PFD and EPFD components use the timer port to schedule their periodical checks and sending of heartbeat messages. According to the algorithms, you need two events extending Timeout: a HeartbeatTimeoutEvent and a CheckTimeoutEvent. You can check the usage of ApplicationContinue in the Assignment0 example to see how to implement your timeouts.

Note that if you implement the algorithms from the textbook the events you need to implement are Timeout, HeartbeatRequest, and HeartbeatReply.

For this assignment, the algorithms assume that each process knows and can communicate with all other processes in $\Pi$, which you can pass via the `Init` event. This corresponds to a fully connected topology. To build such a topology, after defining characteristics for some links, you can use the `defaultLinks(delay, lossRate)` method in your topology object. For example Figure 1 corresponds to a fully connected topology, in which all the links have `1` second delay, except the three explicitly specified links (from 1 to 2, 2 to 1, and 2 to 3), which have `3` seconds delay.

```
Topology topology1 = new Topology()
{
   {
      node(1, "127.0.0.1", 22031);
      node(2, "127.0.0.1", 22032);
      node(3, "127.0.0.1", 22033);

      link(1, 2, 3000, 0).bidirectional();
      link(2, 3, 3000, 0);
      defaultLinks(1000, 0);
   }
};
```

Figure 1: An example topology.

Both algorithms use some configuration parameters. For PFD, the parameters are $\gamma$ and $\delta$, which represent time periods and are given in milliseconds. $\gamma$ is the interval between sending two consecutive heartbeats, and $\delta$ is the upper bound on the transmission delay. Beware that the $\delta$ in PFD has to be larger than every link delay specified in your topology file. The parameters for EPFD are *TimeDelay* and $\Delta$, which represent time periods and are given in milliseconds. *TimeDelay* is the initial value of *period*, i.e. the interval for sending out heartbeats and checking the received heartbeats. Finally, $\Delta$ defines how much the *period* is increased when a process finds out a suspected process is not really crashed.

For example, if $\gamma = 1000$ and $\delta = 4000$, it means that PFD shall send heartbeats every 1 second ($\gamma = 1000$) and check received heartbeats every 5 seconds ($\gamma + \delta = 5000$). For EPFD, if $period = TimeDelay = 1000$, initially, EPFD will send heartbeats and check received heartbeats every 1 second. Suppose you have a bidirectional link between process 0 and process 1 with delay 2000. As process 0 does not receive a heartbeat from process 1 earlier than 2 seconds, 0 will suspect 1. Later 0 revises its suspicion when it receives a hearbeat from 1 and increases its *period* to 2000 (increments it by 1000, as $\Delta = 1000$). You can define the parameters in your Main component, and pass them to the PFD/EPFD component via an init event.

The PFD component triggers events of type CrashEvent, specifying the Address of the crashed process. The EPFD component triggers events of type SuspectEvent and RestoreEvent, specifying the Address of the process that is suspected or for which the suspicion is revised, respectively. In your application component, you have to handle these events and print out the details. The value of *period* should also be output on screen with each suspect or restore event.

### 2.1 Exercises

Implement the PFD and EPFD components and experiment with them as instructed in the following exercises. Describe your experiments in a written report. For each exercise include the topology descriptors used, and explain the behavior that you observe.

**Exercise 1** Verify the completeness of the failure detectors by killing one or more processes and wait for crash/suspect events to be triggered in the other processes.

**Exercise 2** For EPFD, initialize *TimeDelay* to a value smaller than your link delays, and observe how it is adjusted to accommodate larger transmission delays.

**Exercise 3** Use FairLoss links for the EPFD and observe how TimeDelay is adjusted to accommodate larger transmission delays when the messages are lost.

**Exercise 4** For this exercise, assume a crash-recovery model. Simulating a crash-recovery in Kompics is explained here:
http://kompics.sics.se/trac/wiki/DistributedSystemLauncher#Crash-recoverymodel

If you want to execute some procedure, say `myrecovery` in the Application component after recovery, you can do this by adding a command for recovery. Add a command, for instance ‘R’ and process it in the Application component. For example, in your Executor, say you have `command(1, "S500:Lmsg1:S1000:X").recover("R:S500:Pmsg3:S500:X", 2000)`

This means that process 1 will be recovered 2 seconds after processing “S500:Lmsg1:S1000:X”. When process 1 recovers, your Application component will receive a command “R:S500:Pmsg3:S500:X”. In the `doCommand` method of the Application component, you can add an if case which checks if the command starts with an ‘R’. If it does, you can call your `myrecovery` method.

For this exercise, use only EPFD and do a crash-recovery for one node, say p. Give enough delay between crash of p and recovery of p, such that the EPFD at all other nodes have triggered a suspicion event for p. After p recovers, observe how a restore event is triggered at all other nodes and the fact that other processes believe that the crashed and recovered process p did not actually crash. Therefore, sometimes, in a crash-recovery model, one needs to distinguish between different incarnations of the same process.

### 2.2 Questions (optional)

If you find the following questions interesting and challenging you may include your answers to any of them in your report. There is no promised reward for answering these questions other than your self-satisfaction and increased knowledge.

**Question 1** What is the earliest and latest time that a crash may be detected in a perfect failure detector (with respect to delta and gamma)?

**Question 2** Can you improve the algorithm for PFD, such that it improves the worst case failure detection time?

**Question 3** Can you improve the algorithm for PFD, such that it requires a single timer?

**Algorithm 1** Perfect Failure Detector

**Implements:** PerfectFailureDetector ($\mathcal{P}$).
**Uses:** PerfectPointToPointLinks (pp2p).

```
upon event ⟨ Init ⟩ do
    alive := $\Pi$;
    detected := $\emptyset$;
    startTimer ($\gamma$, HEARTBEAT);
    startTimer ($\gamma + \delta$, CHECK);
end event

upon event ⟨ Timeout | HEARTBEAT ⟩ do
    for all $p_i \in \Pi$ do
        trigger ⟨pp2pSend | $p_i$, [HEARTBEAT] ⟩;
    end for
    startTimer ($\gamma$, HEARTBEAT);
end event

upon event ⟨ Timeout | CHECK ⟩ do
    for all $p_i \in \Pi$ do
        if ($p_i \notin$alive) $\wedge$ ($p_i \notin$detected) then
            detected := detected $\cup$ { $p_i$ };
            trigger ⟨crash | $p_i$⟩;
        end if
    end for
    alive := $\emptyset$;
    startTimer ($\gamma + \delta$, CHECK);
end event

upon event ⟨ pp2pDeliver | src, [HEARTBEAT]  ⟩ do
    alive := alive $\cup$ { src };
end event
```

**Algorithm 2** Eventually Perfect Failure Detector

**Implements:** EventuallyPerfectFailureDetector ($\Diamond\mathcal{P}$).
**Uses:** PerfectPointToPointLinks (pp2p).

```
upon event ⟨ Init ⟩ do
    alive := $\Pi$;
    suspected := $\emptyset$;
    period := TimeDelay;
    startTimer (TimeDelay, HEARTBEAT);
    startTimer (period, CHECK);
end event

upon event ⟨ Timeout | HEARTBEAT ⟩ do
    for all $p_i \in \Pi$ do
        trigger ⟨pp2pSend | $p_i$, [HEARTBEAT] ⟩;
    end for
    startTimer (TimeDelay, HEARTBEAT);
end event

upon event ⟨ Timeout | CHECK ⟩ do
    if (alive $\cap$ suspected) $\neq \emptyset$ then
        period := period + $\Delta$;
    end if
    for all $p_i \in \Pi$ do
        if ($p_i \notin$alive) $\wedge$ ($p_i \notin$suspected) then
            suspected := suspected $\cup$ { $p_i$ };
            trigger ⟨suspect | $p_i$⟩;
        else if ($p_i \in$alive) $\wedge$ ($p_i \in$suspected) then
            suspected := suspected \ { $p_i$ };
            trigger ⟨restore | $p_i$⟩;
        end if
    end for
    alive := $\emptyset$;
    startTimer (period, CHECK);
end event

upon event ⟨ pp2pDeliver | src, [HEARTBEAT]  ⟩ do
    alive := alive $\cup$ { src };
end event
```