# ID2203 Tutorial 1

Cosmin Arad
icarad@kth.se

Handout: January 30, 2008
Due: February 6, 2008

## 1 Introduction

The goal of this tutorial is to get you accustomed to the TBN framework, and how to use it for implementing distributed algorithms. This tutorial is accompanied by a document describing the TBN framework, and by an archive containing helper source code that you will use in all assignments of the course.

First, you should read the TBN Tutorial that accompanies this document and try out the HelloWorld! application described in there.

Every assignment will require you to implement one or more distributed programming abstractions, sometimes relying on lower-level abstractions, that you'll have implemented in previous assignments. Each abstraction will be implemented by one[1] TBN component. To test your implementations you will typically need to run a number of processes implementing those abstractions.

Each process will consist of a number of TBN components. Besides the components implementing your distributed programming abstractions, you'll use a TBN CommunicationComponent to send messages between the processes, a TBN TimerComponent to set one-time timers, a DelayComponent and a DropComponent for simulating fair-loss links, and an ApplicationComponent as a user interface.

Information about your network of processes is described in a topology file, in XML format. An example topology file is given in Figure 1.

Each process or node is given a numeric identifier, an IP address, and a port number on which its communication component will listen for incoming connections. A directed link between two processes is characterized by the

[1] sometimes two

```
<topology xsi:noNamespaceSchemaLocation="topology.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <nodes number="3">
        <node id="0" ip="127.0.0.1" port="12340" />
        <node id="1" ip="127.0.0.1" port="12341" />
        <node id="2" ip="127.0.0.1" port="12342" />
    </nodes>
    <links>
        <link src_id="0" dst_id="1" latency="1000" undirected="true"/>
        <link src_id="0" dst_id="2" latency="3000" loss_rate="0" undirected="true"/>
        <link src_id="1" dst_id="2" latency="2000" loss_rate="0" />
        <link src_id="2" dst_id="1" latency="0" loss_rate="0.5" />
    </links>
</topology>
```

Figure 1: An example topology file.

id of the source node, the id of the destination node, the link's latency in milliseconds, and the link's loss rate in percentage (a loss rate of 0.3 means that on average 30% of the messages sent on that link are lost). A link descriptor containing the attribute `undirected="true"` is an undirected link and has the same latency and loss characteristics in both directions. Not all pairs of processes need to be connected by a link.

For testing an assignment you'll write a topology file describing your network of processes, and you'll launch all processes described in the topology file passing to them the path to the topology file and their numeric identifiers, as arguments. The main program, writes these properties in the ApplicationComponent's configuration file and then builds the system. To understand how an assignment should be run read the following files that have been provided in the accompanying assignments kit: `Assignment1.java`, `ApplicationComponent.java`, and `TopologyDescriptor.java`.

When the aplication component is initialized, it reads from a properties configuration file (that is specified in the architecture description file, and passed to its `init` method, by the system builder) the local node's id and the path to the topology file. It parses the topology file and builds a topology descriptor object that contains all the outgoing links of the local process.

When the application component is started, it triggers an `InitEvent`, passing the topology descriptor to other components that need it, typically the delay component and the drop component. The application component may also start a thread for reading input from the standard input and possibly triggering events into the system.

The drop component and delay component are meant to filter all messages that the local process sends. They will handle the `InitEvent` triggered

by the application component and will thus receive a topology descriptor and become aware of the outgoing links of the local process. The delay component should be layered upon the communication component and the drop component should be layered upon the delay component. All component that send messages to remote processes should publish the messages into a channel to which the drop component is subscribed. Based on a message's destination node, the drop component either drops the message or triggers it forward to the delay component which delays it (using the timer component), according to the link specification in the topology descriptor, and then sends it to the destination using the communication component. Using the drop component and the delay component in this manner, together with the communication component with the UDP transport, you provide a fair-loss links abstraction. Just using the delay component and the communication component with the TCP transport, you provide a perfect links abstraction.

## 2 Assignment

Using TBN and the helper components and source code provided in your assignments kit you should implement a very simple message flooding protocol, like the one presented in lecture 1. You will use the CommunicationComponent and the TimerComponent from the library jars in your assignments kit. Use the DelayComponent, the DropComponent, the ApplicationComponent, and the Assignment1 main program provided as source code in your assignments kit.

You have to write a `FloodComponent` that behaves according to the following specification. It handles the `InitEvent` triggered by the application component and stores the topology descriptor (specifying all the process' neighbors) in an internal variable. It handles `FloodInitEvent` events triggered by the application component. A `FloodInitEvent` has a `message` attribute of type `String`. Upon receiving a `FloodInitEvent`, the flood component sends a `FloodMessage` to all its neighbors. A `FloodMessage` has a `message` attribute of type `String`.

Upon receiving a `FloodMessage` that it sees for the first time, the flood component sends it to all its neighbors. When the flood component has received the same `FloodMessage` from every neighbor, it triggers to the application a `FloodDoneEvent`.

Implement the flood component and experiment with it 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.

The assignment is due on February 6th. You have to send your source code and written report by email before the next tutorial session. During the tutorial session you will present the assignment on a given topology description. You can work in groups of maximum 2 students. Be prepared to answer questions about your process's system architecture and explain the behavior of the flood algorithm on the types of topologies suggested in the exercises. Any questions are welcome on the mailing list.

**Exercise 1** Use topologies with bidirectional links and no loss. Experiment your implementation with 2 different topologies and check whether the order in which the flood is done (finished, marked by a `FloodDoneEvent`) at each process is correct. The order in which the flood is done at each process depends on the link delays. To observe this order, run all processes on the same machine and print the current system time when handling the `FloodDoneEvent` in the application component.

**Exercise 2** Experiment with lossy links and observe that the flood algorithm does not terminate for all processes (by triggering a `FloodDoneEvent`) if a message is lost. Explain why.

**Exercise 3** Experiment with unidirectional links in only one direction for some pairs of processes and observe that the flood algorithm does not terminate at all processes. Explain why.