Royal Institute of Technology

MSc. Software Engineering of Distributed Systems

ID2212 Network Programming with Java

Project

Andrei Shumanski
andreish@kth.se
00460707761992

Trigonakis Vasileios
vtri@kth.se
00460707694420

Stockholm 2009

TABLE OF CONTENTS

1. SHORT TASK SPECIFICATION.

The task that we had to implement was to create a state-full application that takes advantage of the self-healing and managing features of Niche Platform. A platform for developing, deploying and execution of self- managing distributed systems and applications . For more information about Niche Platform you can visit the Niche Web-page, at http://niche.sics.se. It includes a component-based programming model (Fractal), API, and an execution environment.

The specific application that we implemented is a counter application, replicated into more than one component. By this way, we can ensure that if one of the components fails, there will be still at least one that can maintain the state and continue functioning, until our configuration with Niche (our management elements) “sense” the failure, deploy another component and initialize it to the correct state.

The main challenge we met was the one of coherence. In order our application to ensure that in any time, there is one working component that can give us the correct state (counter value), we had to ensure that all the components are synchronized to the same state, or more precisely that more than one component should have the same, correct, state. This task was not trivial, because the infrastructure that Niche works on, is a distributed system, so have no guarantees about the sequence of the message delivery.

Also, our goal was to create and implement an algorithm that can keep the components synchronized not only in case of a component failure, but when components miss some increase counter commands, so they get desynchronized. For this purpose, our application can work in two different modes; a strict one, that guarantees that no state change command will be lost, and a more relax mode, were there is a small chance, if the computation is too intense and all the components keep omitting commands, that we can lose some commands.

2. INFORMATION ABOUT HW PLATFORM

Computer: laptop Toshiba A300 - Core 2 Duo 2200, 4096 Mb RAM, 400 GB HDD

Operating system: openSuse Linux 11.2 (Linux kernel 2.6.31)

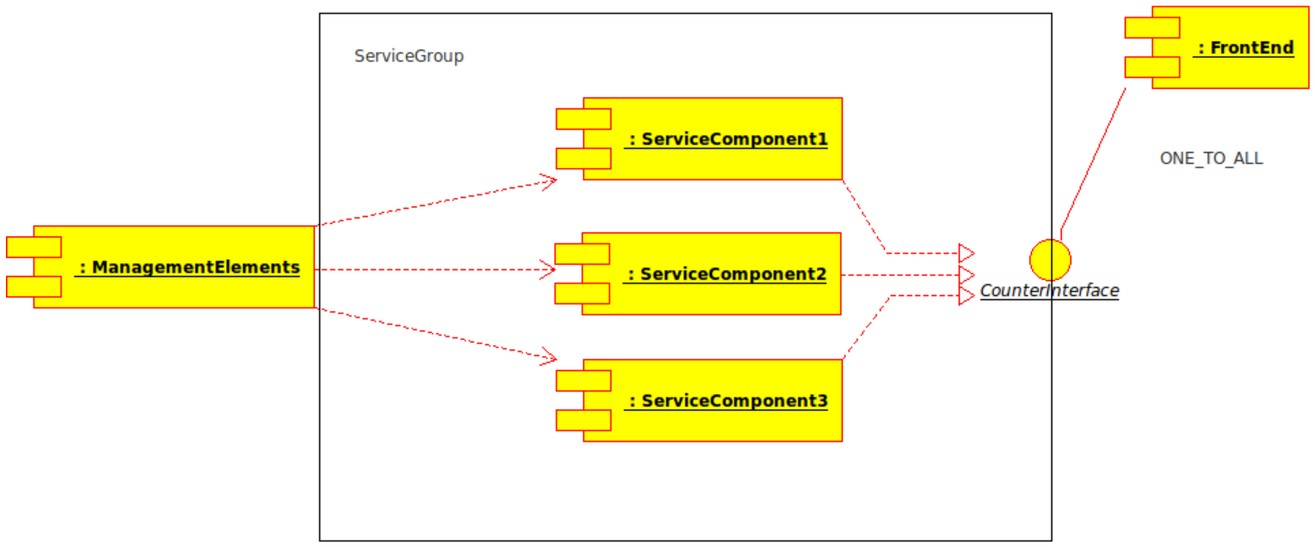
3. A LIST OF JAVA SOFTWARE

1. Java 1.6.0_17-b04
2. Niche version 0.2.2
3. Apache Ant version 1.7.1
4. Apache HTTP server version 2.2.13
5. PHP version 5.3.0

4. DETAILED DESCRIPTION.

SYSTEM ARCHITECTURE

In the following diagram, you can see the architecture of our system:



Elements of our architecture:

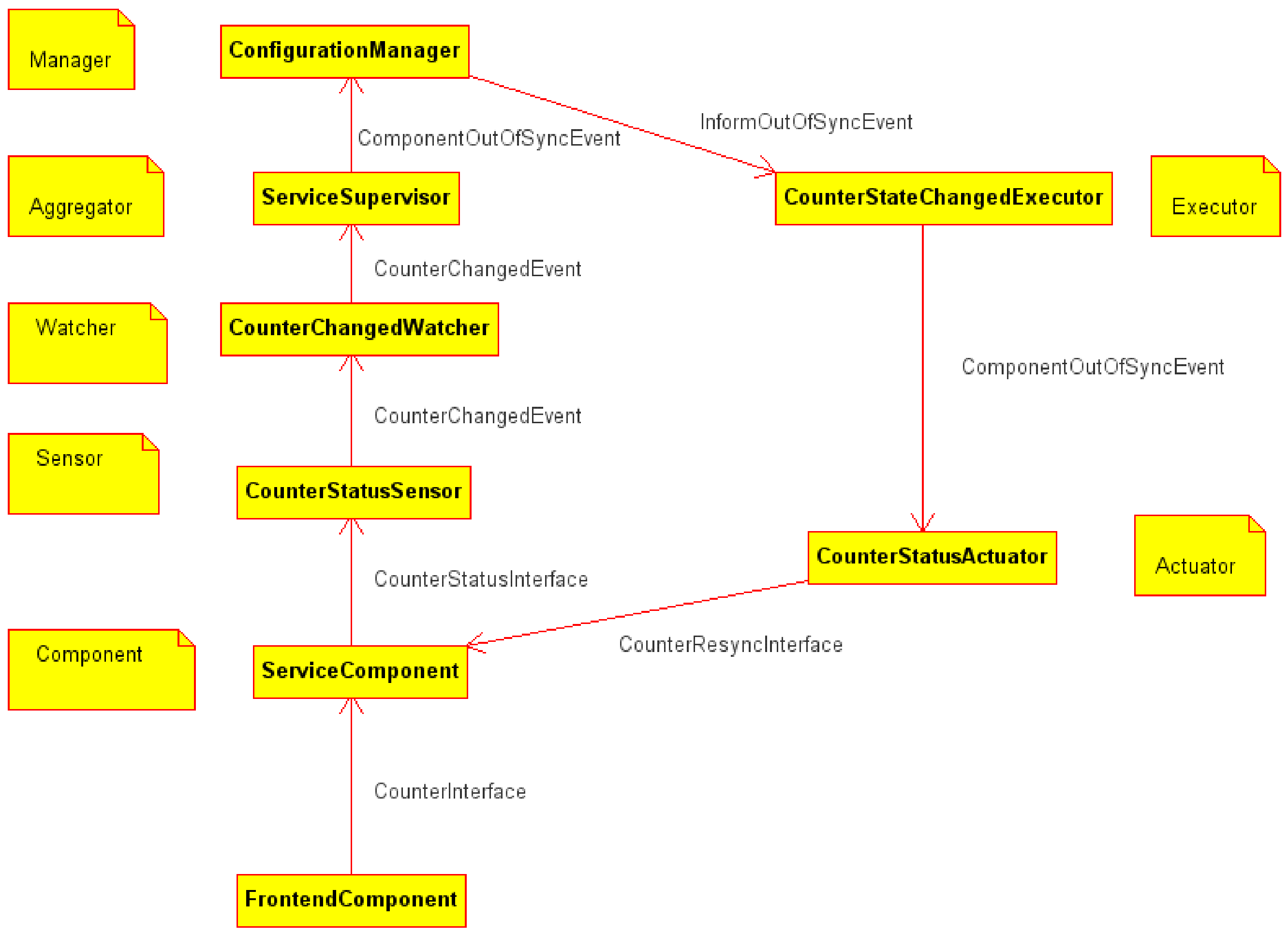
ServiceComponents : are the components that implement the functionality of our system. Specifically, in our implementation, the ServiceComponents implement the counter functionality. As you can see in the diagram, they implement the CounterInterface, which has the increaseCounter method by which the components are "ordered" to change their state. All of them are included in a Group, so the can be handled by the other components as they are one only component.

FrontEnd : this component is the one that uses the CounterInterface in order to call the increaseCounter method and change the state of the ServiceComponents. As you can see on the diagram, the call that FrontEnd is bound to the ServiceGroup and not to a specific ServiceComponent. Also, it is bound with a one to all relationship, which means that every call is targeted toward all the ServiceComponents. Moreover, the fact that the FronEnd is bound with the Group helps when a ServiceComponent fails and has to be replaced. By this way we do not have to programmatically rebind the components.

ManagementElements : they are not shown in detail in this diagram, since they will be presented in a following one. They monitor the functional part of our system and they are responsible for keeping the ServiceComponents synchronized and deploy a new ServiceComponent in case that one fails.

APPLICATION STRUCTURE

The following diagram represents structure of the application:



1. FrontendComponent tells Service component to increase it's counter using CounteInterface. Frontend component sends unique transaction Id with each increase counter request.
2. To simulate failure we have some small probability that Service component will not increase it's counter. If Service component increases it's counter it tells the counter to CounterStatusSensor via CounterStatusInterface.
3. CounterStatusSensor just triggers a CounterChangedEvent to CounterChangedWatcher.
4. CounterChangedWatcher promotes received event to ServiceSupervisor aggregator.
5. ServiceSupervisor contains the logic of the application. It analyzes the received counters and if some of the ServiceComponents is out of order triggers ComponentOutOfSyncEvent to ConfigurationManager.

6. ConfigurationManager triggers InformOutOfSynEvent to CounterStateChangedExecutor to re-synchronize components.
7. Executor sends ComponentOutOfSyncEvents CounterStatusActuators
8. Actuators inform ServiceComponents through CounterResyncInterface about synchronization.
9. ServiceComponents receive counter number to synchronize and transaction Id. If component did not receive message with the same transaction Id it synchronizes and stores this transaction Id in the list to ignore this transaction in the future.

ALGORITHM(S)

As we said, we implemented two different algorithms for the synchronization of the ServiceComponents; one strict and one more loose, that can also be adjusted to be close to strict.

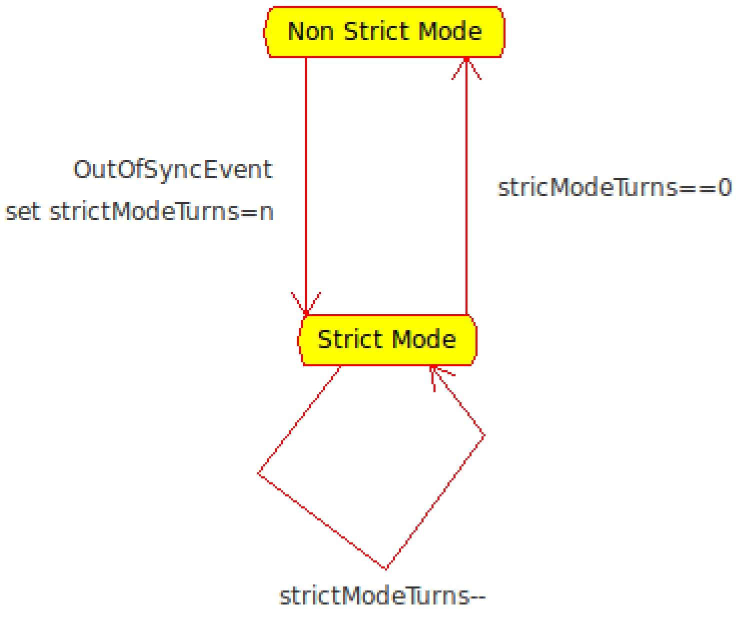
STRICT ALGORITHM

In the strict algorithm, the management components send resynchronization messages to the ServiceComponents every time that they "sense" a newer system state. By this way, we ensure that if the system change state, this state will be adopted by all the components (as soon as they function) after 2 "messages". One message until the management elements sense the new state and one more until the components are informed.

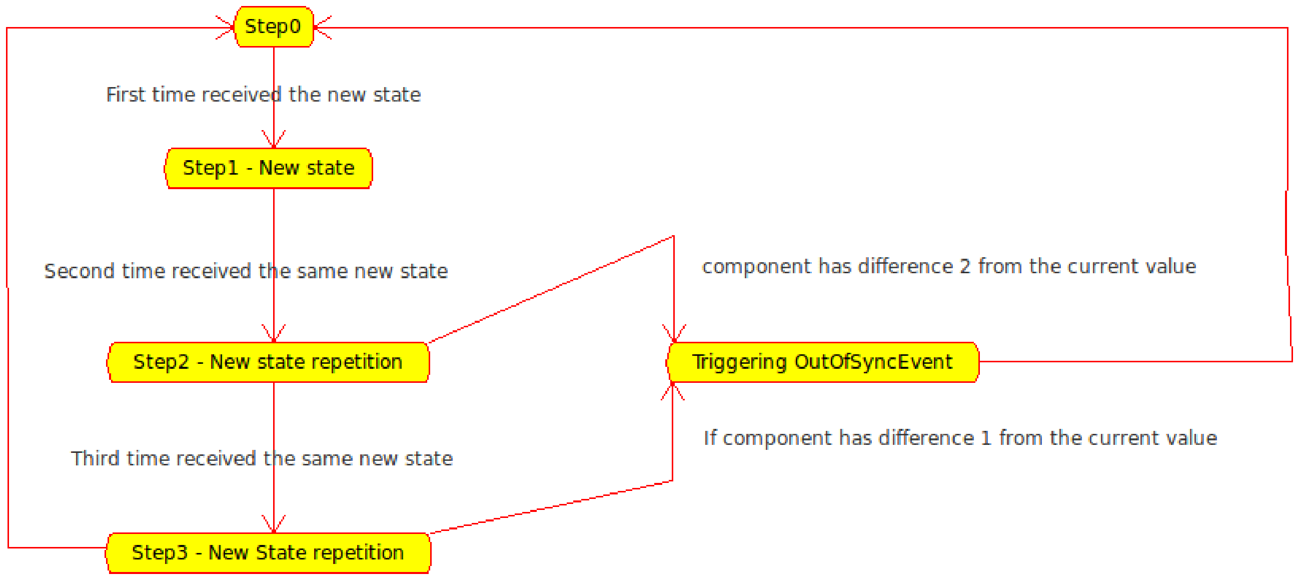
NON-STRICT ALGORITHM

The non-strict algorithm is actually a mixture of a non-strict algorithm with the strict one. Firstly, we work with the non-strict algorithm but we realized that it is impossible to make it lossless, so we created the strict algorithm and changed the non-strict into the mixed one.

The following state diagram describes the mixed approach:



The non-strict mode is a optimistic algorithm, which adds no overhead to the systems functionality if components work properly. We will describe the algorithm with a state diagram for 3 components, but it can be easily generalizes to more.


Step0
First time received the new state
Step1 - New state
Second time received the same new state
component has difference 2 from the current value
Step2 - New state repetition
Triggering OutOfSyncEvent
If component has difference 1 from the current value
Third time received the same new state
Step3 - New State repetition

The rational behind our algorithm is to keep the system with one component in the correct state only for one round. This is achieved by:

Step0: we are waiting for a “new round” of messages.

Step1: the first message of the new round arrives, no need to check for consistency since it is the 1st occurrence of the new state.

Step2: we check if the there is one component with value distance of 2 from the newly arrived value. If this is the case, we know that even if we receive the new update by this node, this will not be the arrived value.

Step3: we check if the there is one component with value distance of 1 from the newly arrived value. We expect that this is not the case, since here we should normally receive the same value by all of the components.

HANDLING MESSAGES IN THE WRONG SEQUENCES

Since Niche Platform guarantees only the message delivery and not the sequence of the messages, receiving the messages in a wrong order was one of the most difficult problems to handle. We handled this problem, by introducing a unique id to every increase call, which id was "escorting" the synchronization messages also. The ServiceComponents are keeping a list of these id, so they can assure that they will not be synchronized to a "future" value. Without this security measure, the following example could be possible:

A component is running slow, so it is informed by the management elements that it should synchronize to the value N, while it is still on the step N-1. The component synchronizes, and the handles the N step, going to value N+1, which means producing a value!

5. A CLEAR INSTRUCTION HOW TO RUN THE APPLICATION.

Counter application needs installed and running Niche. So first of all it is necessary to install Niche. The detailed instruction how to do this is available in the document "Niche quick start guide". The document can be downloaded from the web site http://niche.sics.se

To run Counter application you need to run Niche using *jadeboot* ant target and run at least three nodes using *jadenode* ant target. After this you can deploy Counter with *testG4A-NicheCounter-Deploy* ant target. After the deplyment you can run application with *testG4A-NicheCounterStart* task.