Report

Distributed Systems. Report
nsth_jobn_mbyb

Nikolai Storr-Hansen, nsth
Jonas Bredvig Bendix Nielsen, jobn
Mikkel Bybjerg Christophersen, mbyb

Version no: 1.0

Date: November 4, 2013

Author: Nikolai (nsth)

Summary:

Chapter TCP written.

Description:
tcp/ip protocol/stack. Server-Client communication,

October 2013

Contents

Introduction

1.1 Project Presentation

This report is an examination of some of the principles seen in the application of distributed systems. The report's modus operandi is to test the underlying theory on small independant applications. Projects in the report are based on a taskmanager application. The basic concept provides a means of creating, editing, deleting and persisting tasks.

The report consists of 8 chapters each eamining a single topic in distributed systems. The taskmanager application provides an empirical dimension and is the steepingstone for the analysis of individual topics.

Due to the pressure of other concurrent courses at ITU we have not been able to provide a single coherent project examining all issues, instead we use the individual projects from the course lab exercises as the empirical background for a discussion of the challenges of distributed systems. The projects therefore only serve as proof of concept and as a basis for analysis and discussion of concepts.

In the fisrt chapters the taskmanager application is used to set up a client-server based application. Later chapters provide other functionality such as group communication and security, web services and serialization etc.

The idea is to try out some of the techniques commonly seen in the implementation of distributed systems and thereby demonstrate our understanding of the theories behind as well as (some of) the challenges faced by distributed systems.

1.2 Taskmanager

The taskmanager application is founded on an xml document containing tasks. The xml document is serialized using JAXB into an object tree. Note in the google app engine project we use JPA for serializing in order to accommodate google app engine specific requirements for persistence. The taskmanager.xml file structure looks like this:

Listing 1.1: taskmanager.xml

```
1 <cal>
2  <tasks>
3   <task id="handin-01" name="Submit assignment-01" date="16-12-2013"
4         status="not-executed" required="false">
5         <attendants>student-01, student-02</attendants>
6         <role>student</role>
7   </task>
8   -
9   -
```

```
10    –
11   </tasks>
12  </cal>
```

1.2.1 Task

The taskmanager application holds a list of Task objects. A task has the following properties:

ID (a unique identifier), name (the name of the task), date (the date of creation), status (status of the task. E.g executed or non-executed.), required (E.g. true or false.), role (role, is an access control technique in where the role of a task signifies the access rights level of that task.), Attendants (a list of attendants for the task)

Listing 1.2: Task

```
1  @XmlRootElement(name = "task")
2  public class Task implements Serializable {
3      @XmlID
4      @XmlAttribute
5      public String id;
6      @XmlAttribute
7      public String name;
8      @XmlAttribute
9      public String date;
10     @XmlAttribute
11     public String status;
12     @XmlAttribute
13     public String description;
14     @XmlElement
15     public String attendants;
16     @XmlElement
17     public String role;
18     @XmlAttribute
19     public Boolean required;
20 }
```

1.2.2 report structure

Chapter 2 is dedicated to the basics of setting up a distributed system; nodes run concurrently, theres no notion of time and the potential crash of one node should not influence the rest. A server client structure is implemented and we discuss the intricacies of java Servlets and JDOM.

In chapter 3 we'll take a look at 'Jersey' web services. A server client communication is established.

In chapter 4 we create a RESTfull service. The taskmanager is made available as resources. Clients access task via HTTP but in the way that the HTTP protocol was originally intended to be used.

In chapter 5 we discuss security in distributed systems. We implement a simple role-based security control mechanism and we encrypt the messages sent between nodes in the system.
- -

1.3 Baseline model

Before we continue, a summary of the challenges in distributed systems is outlined:

1.3.1 Heterogeneity

Each node in a distributed system may have been developed in different languages on different OS.

1.3.2 Openness

Nodes communicate through interfaces. A system is open if it publishes a public interface. It is paramount that the protocols are followed closely to mask the differences as seen above.

1.3.3 Security

Confidentiality: (authorization) Integrity (protection against corruption i.e, the message you receive is genuine.) Availability: (?)

The solution is to use encrypted messages.

1.3.4 Scalability

Preferably, the cost of adding another node should be constant.This has an ipact on the algorithms used in distributed systems.

1.3.5 Availability and Failure Handling

Detect failures using checksums. Mask failures by catching the failure and resending the message Tolerate failures: e.g. by letting the browser timeout instead of waiting indefinitely tying down resources. Recover from failure: e.g. by performing a server rollback. Redundancy: i.e., DNS servers keep copies at several locations in case one server crashes the tables are always available elsewhere.

1.3.6 Concurrency

How to solve concurrent requests?

1.3.7 Transparency

How to make the system seem one though it is in fact comprised of several subsystems perhaps located on different continents.

1. Access transparency: access to local and remote resources must go through the same interfaces.
2. Location transparency: The node accessing a resource should not need to know its location.
3. Concurrency transparency: Multiple nodes can access the resource simultaneously.
4. Replication transparency: multiple instances of a resources can be utilized without the nodes noticing.
5. Failure transparency:In case of failure in the resource (hardware or software) the node should be able to complete its task.
6. Mobility transparency: the resource can be moved without the node noticing
7.
8.

1.4 Failure model

The above challenges are a t the root of a basic failure model for distributed systems.

There are three places of potential failure: the sender, the channel and the receiver. The basic failure model in distributed system distinguish between failure in the channel and failure in the processes.

1. Omission: Channel failure. An outgoing message never arrives at the destination
2. Send-Omission: Process failure. The message never leaves the process.
3. Receive Omission: Process Failure. The message arrives but theprocess can't find it.
4. Crash: Process failure. A process halts. Others might not recognize this
5. Fail-stop: Process failure. A process halts. Others see this.
6. Arbitrary: Process or channel failure. All or any of the above.

Task Manager Servlet

Java's answer to php. We use Servlets to produce html markup and to process data that was submitted. Servlets run on the HTTP protocol (any protocol will do).

In our application

generating either html or processing requests.

TCP server and Serialization

Contents

In this chapter we describe the result of the TCP lab exercise. in section 3.1 we describe the assignment. In section 3.2 we describe our solution. in section 3.3 we provide an example run of the solution. In section 3.4 we reflect on the theory behind the assignment. In section 3.4.1 we sum up what we learned and round up the chapter.

3.1 lab description

The purpose of this weeks lab exercise is to develop a simplified version of web server for task manager that runs on TCP protocol.

In this lab exercise, you are required to develop the code for the following functionality.

Develop java serialization classes for task-manager-xml using Java Architecture for XML Binding (JAXB) APIs (by annotating java classes) to handle deserialization/serialization from/to task-manager-xml in the server.

Develop TaskManagerTCPServer and TaskManagerTCPClient classes to implement the above described functionality of server and client that communicate on the TCP protocol.

3.2 Solution

In our solution we first deserialize the xml document into a 'taskmanager' object and Task objects. Then we create appropriate input- and output streams for communicating with client.

Serialization and deserialization to and from xml is done with the JAXB API. To quote Oracle; *JAXB provides methods for unmarshalling (reading) XML instance documents into Java content trees, and then marshalling (writing) Java*

content trees back into XML instance documents.

JAXB uses annotations to achieve this. We annotate the Task and Taskmanager Java objects and JAXB then knows how to convert the objects into an xml tree-structure and back into an object graph.

The TCP server in our solution first connects to a serversocket and then waits for incoming requests on the port bound to the socket. Note this is a blocking call. The Client creates a socket on the same port but this time with the servers InetAddress as a second argument.

The server and client communicates by passing messages in and out of the matching inputstreams and outputstreams. Command messages are passed via the writeUTF and received by the readUTF methods (as strings?). The Task and taskmanager objects are sent via writeObject and received by readObject methods (as byte-streams ?).

3.2.1 http protocol

The server responds to the clients request by sending the request back. The client then proceeds accordingly. This mimics the HTTP protocol request-response pattern in detail?.

3.3 example run

In this example run we star out with an xml document containing one Task. Before the example run the xml document looks like this:

Listing 3.1: xml before run

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <taskmanager>
3          <tasks>
4
5                  <task id="handin-01" name="Submit assignment-01" date="16-12-2013"
6                          status="not-executed">
7                          <description>
8                                  Work on mandatory assignment.
9                          </description>
10                         <attendants>student-01, student-02</attendants>
11                 </task>
12         </tasks>
13 </taskmanager>
```

During the run the client requests the POST method with new Task. After the run the xml document looks like this:

Listing 3.2: xml after POST

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
```

```
<taskmanager>
        <tasks>
                <task id="handin-01" name="Submit assignment-01" date="16-12-2013" status="
                    not-executed">
                    <description>
                            Work on mandatory assignment.
                    </description>
                    <attendants>student-01, student-02</attendants>
                </task>
                <task id="one_more_cup_of_coffe" name="Tout les circles" date="15-09-2013"
                status="mais jai le plus grande maillot du monde">
                    <description>recondre</description>
                    <attendants>bjarne, lise, hans, jimmy</attendants>
                </task>
        </tasks>
</taskmanager>
```

The client then requests the PUT method wanting to change the new 'one more cup of coffe' Task description from 'recondre' to 'recondre les roix'. Finally, the client requests the DELETE method with the taskid 'handin'01' parameter. After the run the xml document now looks like this:

Listing 3.3: xml after PUT

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<taskmanager>
        <tasks>
                <task id="one_more_cup_of_coffe" name="Tout les circles" date="15-09-2013" status=
                    "mais jai le plus grande maillot du monde">
                    <description>recondre les roix</description>
                    <attendants>bjarne, lise, hans, jimmy</attendants>
                </task>
        </tasks>
</taskmanager>
```

The TCP server code of interest looks like this.

Listing 3.4: servers initial request-response

```
while(running){
        System.out.println("\nServer: Waiting for client command");
        // client request
        String message = dis.readUTF(); // blocking call
        System.out.println("Server: client command received : " + message);
        // accept client request by returning the (request) message
        dos.writeUTF(message);
```

Listing 3.5: server POST method

```
if(message.equals("POST")){
    // receive Task object from client
        Task t = (Task) ois.readObject();

        System.out.println("Server: object received '" + t.name +"', persisting object...");
        // Add Task object to takmanager collection
        serializer.allTasks.tasks.add(t);
        // persist to xml document<
        serializer.Serialize();
        // acknowledge to client
        dos.writeUTF("Task with id: '" +t.id + "' saved");
}
```

The corresponding TCP client code looks like this.

```
InetAddress serverAddress = InetAddress.getByName("localhost");
// Open a socket for communication.
Socket socket = new Socket(serverAddress, 7896);
// Request message to server
String message = "POST";
System.out.println("Client: requesting " + message);

dos.writeUTF(message);  // send the request
response = dis.readUTF(); // blocking call
System.out.println("Client: server response : " + response);

Task t = new Task();
t.name = "Tout les circles";
t.id = "one more cup of coffe";
t.date = "15-09-2013";
t.description = "recondre";
t.status = "mais jai le plus grande maillot du monde";
t.attendants = "bjarne, lise, hans, jimmy";

// if server acknowledges
if(response.equals(message)){
        System.out.println("Client: server accepted. Sending object");
        // send the Task object
        ous.writeObject(t);
}else{
        System.out.println("Client: the server did not acknowledge the object. Aborting.");
}

response = dis.readUTF();
System.out.println("Client: server response : " + response)
```

3.4 Reflection

In this exercise we worked with protocols at the transport layer level, well, the TCP protocol to be precise.

The TCP protocol is part of a protocol suite. A protocol suite is a stack of protocols each responsible for a single task. Each layer communicates with the layer above and below, only.

At the bottom of the TCP/IP suite are the physical layer(s), and on top of those the network layer provides an interface to the transport layer, application services (and protocols) are built on top of the transport layer, based on TCP e.g. HTTP, SMTP, POP, FTP etc. More layers can be added to provide additional functionality e.g. security etc.

The benefit in using protocols is network independance i.e., many different application types and languages can use the same network to communicate as long as they use they implement the same protocol(s).

In our application the transmission Control Protocol (or TCP protocol) communicated with the network layer through a socket connection. Our client existed in the application/presentation layers utilizing the TCP transport protocol which communicated with the network layer.

In our application messages were addressed to a communication port. The transport address (the IP number) was in this case composed of the computers localhost address and the port number.

There are two transport protocols in the TCP/IP suite; UDP and TCP. UDP transfers text messages (IP packets) and TCP transfers byte streams (as IP packets). Furthermore, TCP is reliable and ordered i.e., TCP provides guarantees as to the ordering of its messages and it provides guarantees as to the delivery of the messages.

In contrast, the underlying network protocol (the IP protocol) offers only best-effort semantics, theres no guarantee of delivery and packets can be lost, duplicated, delayed or delivered out of order. This seem logical as the network layer (IP protocol) can not control the other end of the connection .

3.4.1 Good and bad

Theres nothing too complicated about the project as such but a small practical thing to notice is that the order and type of method calls between the server and client can be confusing. It is vital that a writeUTF is picked up by a similar readUTF at the other side i.e, you can not readObject from a writeUTF. This

took some time and error detection to get right.

The main lesson learned is about the protocol layers, each providing a specialized task. Each layer communicates with the layers above and below. Network layers send IP packets to (IP)addresses accross the network. The transport layer's UDP and TCP protocols sends IP packets to processes instead of addresses. TCP provides guarantees about reliability, ordering etc that UDP does not.

Task Manager

Task Manager

Group Communication

Contents

A) the lab description of the respective topic (to make the report selfcontained)
B) 1-2 pages pointing out how much they solved and which issues they encountered
C) A print of an example run
D) 1-2 pages where they relate what they did to the relevant theory in the curriculum
E) max 1 page conclusion, concluding what was solved well (perhaps even makes you proud :-)- and what could be done differently/better and why

In this chapter we shall describe the result of the JGroup exercise. In section 4.2 we demonstrate our solution. In section 4.3 we show an example run of the solution. In section 4.4 we discuss the underlying theory and relate that to the solution. In section 4.5 we round up the chapter.

4.1 lab description

The primary focus of this weeks assignment is to understand the basic concepts of group communication.

Your primary task is to add group communication functionality to task manager application using JGroups toolkit. You can assume a scenario as shown in the following figure, where multiple instances of task manager applications are running and providing functionality for their clients to create, read, update and delete tasks, as described in the previous lab exercises.

Each of instance of a task manager server is running with its own set of tasks (i.e. their own copy of task-manager-xml) and want to communicate their incremental state changes (i.e updates to their own tasks such as add, delete, update to a task) with the other instances of task manager application. On top of that, the task manager application also needs to support state synchronization among the instances, to bring all the task manager instances to same state, i.e to have same tasks among all the instances. The task manager application achieves this functionality by creating a Task Group using JGroups toolkit and

all the instances of task manager connect to the Task Group by using JChannel. Also note that a task manager instance may choose join or leave the task group at any point of time.

The Assignment

You are required to add/implement the following functionality to the task manager JGroup application.

Extend the task manager xml with a required attribute on the task element, which accepts boolean values (true/false) indicating whether the task is required to be executed later or not.

Implement the following operations on task manager JGroup application.

***execute**: accepts id of a task and all instances of task managers in the group execute the task matching to id, by assigning the status attribute to executed and required attribute to false.*

***request**: accepts id of a task and all instances of task managers in the group assigns the required attribute to true for the task matching to the id.*

***get**: accepts a name of a role as input and then all instances of task managers in the group will output their tasks matching to the role specified in their task manager xml.*

4.2 Solution

The solution consists of two main classes. A sender and a receiver. The sender's task is to take user requests and relay them to the receivers. The receiver's task is to... receive the requests.

The sender instantiates a 'channel' object. This is similar to a socket and this is the main concept in the JGroup API. messages are sent over the channel to the group or to individual processes. The Message object itself takes the sender's and receiver's addresses as well as the marshalled message. If the sender and receiver addresses are null the message is multicast to the group members

Our sender class first packages a user request in an envelope object (which can hold a request command as well as a Task object) and then serializes the envelope object before multicasting a JGroup Message object containing this serialized envelope to the group i.e., a request for a given operation on a given task is put into an envelope object with the tasks ID. The envelope is marshalled

and then multicast to the group as a JGroup Message.

The receivers subscribe to the same JGroup channel as the sender has instantiated. The group receivers will unmarshall the message back into an envelope object giving access to the original request and the Task object.

The Message object in JGroups consists of a destination address, a sender address and a message, in this case the serialized envelope. JGroup delivers multicast as well as one-to-one communication. By providing the Message object with a receiver address that message is sent to that receiver only and not to the group.

4.3 Example Run

In our taskmanagerapplication we first create and connect to the group 'channel'. Then we fetch the state which in turn invokes the receivers 'setstate' method. This is to ensure a synchronized state across the group.

Listing 4.1: group setup

```
1  // channelTasks = ChannelHelper.getNewChannel(localIp, addPort);
2  channel = new JChannel();
3
4  // Receiver (taskprovider, channel)
5  channel.setReceiver(new TaskReceiver(provider, channel));
6
7  // Instantiate a Group. If this is the first connect, the group will be created.
8  channel.connect("Add Tasks Channel");
9
10 // State transfer. getState(target instance, timeout). null means get the state from the coordinator/the first
       instance.
11 channel.getState(null, 10000);
12
13 // the busines end.
14 eventLoop();
15
16 // when exiting the eventLoop we exit the group channel
17 channel.close();
```

After the initial creation and state transfer steps we enter the eventLoop. Here we receive requests, transform the request into a Message object and send messages to the group. A request is wrapped in an 'Envelope' object together with possibly a Task. The WriteToChannel(envelope, channel) method then serializes the envelope and wraps it in a JGroup Message object, which contains sender and receiver addresses, and sends theat message to the group.

Note overloads of the channel.send() method exists which allows us to send the message to a single address instead of the group.

Listing 4.2: eventloop

```
// create an empty message container
Envelope envelope = new Envelope();

switch (command.toLowerCase().trim()) {
        case "request":
        System.out.println("type or paste task Xml you want to request (in single line)!");
        System.out.print("> ");

        String requestXml = in.readLine();

        Task requestTask = TaskSerializer.DeserializeTask(requestXml);
        envelope.command = command;
        envelope.data.add(requestTask);

        // here we send the message to the group. (message, Channel)
        WriteEnvelopeToChannel(envelope, channel);

                break;
```

A receiver can either implement JGroup.receiver or extend receiverAdapter and simply override the 'receive(Message)' method. In our application we do the later.

After receiving a Message the application deserializes the message into an Envelope object. The envelope contains the request and possibly a Task object on which to perform the requested action.

Listing 4.3: receiver

```
if(DeserializeEnvelope.command.equals("request")){
            Task taskWithId = GetTaskWithId(DeserializeEnvelope.data.get(0).id);

            if (taskWithId != null) {
                    // execute therequired 'action' on the given Task object
                taskWithId.required = true;

            try {
                    // persist changes
                    provider.PersistTaskManager();
            } catch (JAXBException ex) {
                System.out.println(prefix + "Failed to persist envelope Xml. Error message" + ex);
            } catch (IOException ex) {
                System.out.println(prefix + "Failed to persist envelope Xml. Error message" + ex);
            }

            System.out.println(prefix + "Task with Id:" + DeserializeEnvelope.data.get(0).id + " requested!
                    " + "total number of tasks: "
                    + provider.TaskManagerInstance.tasks.size());

        } else {
            System.out.println(prefix + "Task with Id:" + DeserializeEnvelope.data.get(0).id + " can not
                    be found and hence NOT requested!" + " Total number of tasks: " + provider.
                    TaskManagerInstance.tasks.size());
        }
}
```

4.4 Theoretical Motivation

Remote invocation paradigms (RPC, RMI) imply a coupling between the participants that is often not desirable in distributed systems. The sender need to know the receiver at the time of communication. A common means of decoupling the system is by using a form of indirect communication. Indirect communication is defined as 'entities communicating through an intermediary'. Note there an be potentially many receivers of a single subject/object.

Uncoupling the communicating entities reveal two properties:

1. Space uncoupling: the sender does not know the receivers' identity and vice versa.
2. Time uncoupling: the sender and receiver(s) does not need to exist at the same time.

This is why indirect communication is desirable in environments where change is anticipated. Note in reality systems are not always both space and time uncoupled.

Various techniques for indirect communication exist:

1. Group communication: A message is sent to an address and then this message is delivered to all members of the group. The message delivery is guaranteed a certain ordering.Sender is not aware of the identity of the receivers. Note this does make the system vulnerable to single-point-failures if ...?

2. Publish-subscribe systems: publishers publish structured events to an event service and subscribers express interest in particular events through subscriptions.

3. Message-queue systems:A process (many processes) sends a message to a (usually FIFO) queue. A single receiver then removes them one by one.

4. Shared memory systems: Processes access DSM by reads and updates to what appears to be ordinary memory within their address space. However, an underlying runtime system ensures that processes executing at different computers observe the updates made by one another.

4.4.1 Group Communication Programing Model

A group has a conceptual group-membership. Processes may join or leave the group. The essential feature of group communication is that it issues only a

single multicast operation to send a message to each member of the group (a group of processes).

This is, off course, more effective than sending a message once to each member of the group.

The most basic form of group communication, IP Multicast, provides some guarantees as to the delivery of messages, namely:
(Reliability: (see Ch 2))

1. Integrity. The message received is the same as the one sent, and no messages are delivered twice.
2. Validity. Any outgoing message is eventually delivered.
3. Agreement. If the message is delivered to one process, then it is delivered to all processes in the group.

But IP multicast offers no guarantees as to reliability i.e., no ordering of messages and packages may be lost.

JGroups is build as an IP multicast application using a transport protocol (UDP TCP, or JMS (Java Message Service)) and, crucially, it delivers reliability and group membership. Among other things this includes:

1. lossless transmission of a message to all recipients (with retransmission of missing messages.)
2. fragmentation of large messages into smaller ones and reassembly at the receiver's side
3. ordering of messages, e.g. messages m1 and m2 sent by P will be received by all receivers in the same order, and not as m2, m1 (FIFO order)
4. atomicity: a message will be received by all receivers, or none of them.
5. Knowledge of who the members of a group are.
6. Notification when a new member joins, an existing member leaves, or an existing member has crashed.

JGroups consists of 3 main parts: (1) the Channel used by developers to build reliable group communication applications, (2) the building blocks which are layered on top of the channel and provide a higher abstraction level and (3) the protocol stack, which implements the properties specified for a given channel.

JGroups is highly configurable and developers can put together a protocol stack that suits their particular needs ranging from fast but unreliable to slower but reliable protocol stacks. i.e., a system might be composed, depending on the protocol stacks used, in such a way that it offers lossless transmission but not ordering of messages or lossless transmission, atomicity and ordering of messages.

In JGroups the channel abstraction is the group membership itself. When connecting to a channel a process joins the group. A Message abstraction is the means be which a process sends a message to the group. the message consists of a receiver address, a sender address and a message.

4.5 Conclusion

Group communication as provided by IP multicast suffers from omission failures, messages may be lost. It also offers no ordering of messages. These properties can be provided by application layers e.g. JGroups. JGroups is an overlay on the basic IP multicast offering amongst other things reliability and ordering in group communication.

JGroups delivers reliable multicast i.e., where IP multicast is unreliable, it might drop messages, deliver messages multiple times, or deliver messages out of order, JGroups offer a reliable multicast. It offers atomicity; all members receive the message or none does. It offers ordering; the messages will be received by all receivers in the same order. And it offers lossless transmission by retransmission of messages.

On a closing note, the flexibility offered by JGroups in composing a protocol stack was beyond the available time. Therefore we did only try out JGroups most basic stack setup, i.e., IP multicast over UDP. Given time it would have been nice to be able to try out some different configuration of the protocol stack.

Control

client