Report

Distributed Systems, ITU, fall 2013.
group: nsth, jobn, mbyb

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

Version no: 3.0

Date: November 15, 2013

Summary:

Chapter introduction v.1.0 added.

Chapter TCP v.2.0 added.

Chapter security v.1.0 added.

Chapter Indirect Comunication v.2.0 added.

Chapter REST v.1.0 added.

October 2013

Contents

Introduction

Contents

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, a stepingstone for the further 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 background for a discussion of the topics. The projects therefore only serve as proof of concept and as a basis for analysis and discussion of concepts.

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

1.1.1 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

```
<cal>
 <tasks>
  <task id="handin-01" name="Submit assignment-01" date="16-12-2013"
        status="not-executed" required="false">
        <attendants>student-01, student-02</attendants>
        <role>student</role>
  </task>
  -
  -
  -
 </tasks>
</cal>
```

1.1.2 Task

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

Listing 1.2: Task

```
@XmlRootElement(name = "task")
public class Task implements Serializable {
    @XmlID
    @XmlAttribute
    public String id;
    @XmlAttribute
    public String name;
    @XmlAttribute
    public String date;
    @XmlAttribute
    public String status;
    @XmlAttribute
    public String description;
    @XmlElement
    public String attendants;
    @XmlElement
    public String role;
    @XmlAttribute
    public Boolean required;
}
```

1.2 report structure

Chapter 1 is an introduction to the report and to some basic concepts in distributed systems. We briefly outline the definition of a distributed system, the foundations of the report.

In chapter 2 we discuss the java Servlet structure. A servlet is Java's answer to asp or php active server pages. They provide dynamic content between a client and a server.

In chapter 3 we create a small client-server application for communicating over TCP/IP.

In chapter 4 we create a RESTfull web-service. The taskmanager is made available as resources. Clients access task via HTTP (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 security protocol and a role-based security control mechanism and we encrypt the messages sent between nodes in the system.

In chapter 6 we examine indirect communication. Java's JGroups is used to establish an example to serve as the background for a discussion of the theories behind indirect/group communication.

In chapter 7 we look at concurrency control. We implement a simple protocol for controlling concurrent access to the taskmanager resource.

In chapter 8 we dive into mobile applications with android.

Chapter 9 conclutes the report and summarizes what we've learned.
-

-

-

1.3 Baseline model

Before we move on, a brief summary of some challenges in distributed systems is outlined.

A distributed system is defined by three properties:

- Nodes run concurrently (thus, if they operate on the same resource we risk unintended behaviour.)
- There's no notion of time (since two nodes would not be able to agree on the time due to differences in location and processor speed etc.)
- The potential crash of one node should not influence the rest (i.e., nodes must be able to come and go without affecting the remaining nodes)

These properties have an impact on the requirements for distributed systems. Here is a list of challenges that follows from the basic definition of distributed systems.

Heterogeneity Each node in a distributed system may have been developed in different languages on different OS', by different developers, on different hardware etc.

E.g. Even though protocols exist for communicating on the internet the heterogeneity of distributed systems makes it difficult to provide guarantees as to a given systems performance under any circumstances.

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. Due to challenge 1 this is a challenge in itself.

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

The solution is to use encrypted messages.

Scalability As nodes can come and go freely, the size of the system can become very large. Preferably, the cost of adding another node should be constant. (This has an impact on the algorithms used in distributed systems.)

vailabilityand Failure Handling We can *detect* failures by using checksums against the data. We can *mask* failures by catching the failure and resending the message. We can *tolerate* failures: e.g. by letting the browser timeout instead of waiting indefinitely tying down resources.

Some systems must guarantee the ability to recover from a failure,: e.g. by performing a server rollback or by building in redundancy, e.g., DNS servers keep copies at several locations. In case one server crashes the tables are always available elsewhere.

Concurrency How to deal with concurrent requests to resources? Ideally, we avoid race conditions and dead-locks. And we also preferably serve clients in the order they arrive.

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

There are several transparency issues.

Table 1.1: transparency

Access-transparency	access to local and remote resources must go through the same interfaces thus providing a transparent interaction.
Location-transparency	The node accessing a resource should not need to know it's exact location.
Concurrency-transparency	Multiple nodes can access a resource simultaneously. Thus providing transparent of system load.
Replication-transparency	multiple instances of a resources can be utilized without the nodes noticing.
Failure-transparency	In case of failure in the resource (hardware or software) the node should be able to complete its task. Perhaps achieved by building in redundancy?
Mobility-transparency	the resource can be moved without the nodes noticing. THe exact location of the resource is tranparent to the nodes.

1.4 Failure model

The mentioned challenges are at 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 model distinguishes between failure in the channel and failure in the processes.

Table 1.2: failures

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

Providing guarantees in distributed systems is a matter of cost versus quality. It may not be feasable to provide a guarantee under any circumstances. Therefore the challenges are handled on a system to system basis.

Many, many more models and challenges faces the developer of distributed systems. These were just the sad tip of the iceberg.

TCP server and Serialization

Contents

In this chapter we describe the result of the TCP lab exercise. in section 2.1 we describe the assignment. In section 2.2 we describe our solution. in section 2.3 we provide an example run of the solution. In section 2.4 we reflect on the theory behind the assignment. In section 2.4.1 we sum up what we've learned and round up the chapter.

2.1 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.

2.2 Solution

We use the JAXB API to provide persistence. JAXB unmarshall's the xml document into a 'taskmanager' object with a related collection of Task objects and 'marshall's' it back into a xml document.

Our server opens a serversocket on a specified port and listens for clients. A client instantiates a socket specifying the server's address and port number. The

client and server communicates by passing byte streams through the port. The transmission of data is done with ObjectInputStream and ObjectOutputStream methods.

Serialization

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 these annotated objects into an xml tree-structure and back into an object graph.

TCP

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 and the server won't perform any other tasks before a client connects. The Client creates a socket on the same port but this time with the servers InetAddress as a second argument.

Information in OO programs are typically stored as data structures whereas data in the messages used to communicate in distributed systems are binary. So, no matter the communication protocols used the data structures need to be flattened before transmitting and then reassembled at the other end.

Byte Stream

In Java, annotating data structures with the 'Serialization' keyword allows the ObjectInputStream and ObjectOutputStream classes to transform a graph of objects into an array of bytes for storage or transmission, and back into objects.

Our 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 DataStreams. The Task and taskmanager objects are then sent as ObjectOutputStream and received as ObjectInputStream.

2.2.1 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?.

2.3 example run

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

Listing 2.1: xml before run

```
<?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>
	</tasks>
</taskmanager>
```

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

Listing 2.2: xml after POST

```
<?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 2.3: xml after PUT

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

```
2 <taskmanager>
3         <tasks>
4                 <task id="one_more_cup_of_coffe" name="Tout_les_circles" date="15-09-2013" status=
                      "mais_jai_le_plus_grande_maillot_du_monde">
5                   <description>recondre les roix</description>
6                   <attendants>bjarne, lise, hans, jimmy</attendants>
7                 </task>
8         </tasks>
9 </taskmanager>
```

The TCP server code of interest looks like this.

Listing 2.4: 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 2.5: 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)
```

2.4 Reflection

In this exercise we worked with the TCP protocol. TCP is a transport layer level protocol. The TCP protocol provides communication between the application layer and the network layer?

The TCP protocol is part of the TCP/IP protocol suite. A protocol suite is a stack of protocols each responsible for a single logical task, e.g. transport, packaging etc. Each layer communicates with the layer directly above and below, only. Messages are passed down the stack at one side and they bubble up through the stack at the other side.

At the bottom of the TCP/IP suite are the physical layer (hardware). On top of the physical layer the network layer provides an interface to the transport layer, application services (and their 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.

Using protocols gives us network independance i.e., different application types and languages can use the same network to communicate as long as they comply to the same protocol(s). Note *even so, the heterogeneity of distributed systems makes it difficult to provide guarantees.*

In our example application we use the Transmission Control Protocol (or TCP protocol) to communicate with the network layer through a socket connection. Our client sits at the application/presentation layer utilizing the TCP transport protocol to communicate with other nodes through the network layer.

Nodes communicate through a port. Both client and server addressed messages to the same port. The transport address (the IP number) was, in this case, composed of the computer's localhost address and a 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 i.e., messages arrive in the order they were sent. And TCP provides guarantees as to the delivery of the messages be using a retry.

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. Note *This seem logical as the network layer (IP protocol) can not control the other end of the connection.*

2.4.1 Good and bad

A source of concern is the structure of our code. This could only have been done better but since this was a course in Distributed Systems, and not in code writing norms, we chose to leave the code as is and concentrate on the distributed systems.

The main lesson learned is about protocols and layers, each layer 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 which UDP does not and this makes TCP well suited for server-client architectures in distributed systems.

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 method ;-)

REST Service

Contents

3.1 Lab description

The purpose of this week?s lab exercise is to develop a REST web service (and also a client) for management of tasks in the Task manager. As part of the exercise,

1. Develop a web service which exposes operations for the management of the tasks from the taskmanager.xml in a RESTful way. To be more specific, you will use the following HTTP methods explicitly for the operations on the tasks.

   (a) HTTP GET: to get tasks as resources
   (b) HTTP POST: to create a new task
   (c) HTTP PUT: to update a task
   (d) HTTP DELETE: to delete a task

2. Develop a client application to test the functionality of task manager RESTful service.

In order to develop the above RESTful service and client, you can use The Java API for RESTful Web Services (JAX-RS). A very good tutorial on developing a REST service in java using JAX-RS specification can be found at REST with Java (JAX-RS) using Jersey - Tutorial. The tutorial also clearly explains about how to expose data entities as resource URIs and you can take inspiration from the article.

3.2 Solution

Our solution makes use of the Jersey API in order to implement a RESTful web service. The Jersey framework provides a series of tools allowing for handling of Http-requests by mapping Http-methods to Java methods. This mapping is carried out for our part mainly by use of dedicated Java attributes.
The attributes we are using are:

- @Path: modifies class and method declarations to allow for the address of resources to determine which class or method is targeted.
- @GET, @POST, @PUT and @DELETE: modifies method declarations to signify which Http-methods are allowed to map to the given Java-method.
- @Produces: modifies method declarations and determines how the textual output from the method is interpreted when converted to a http reply message. The Java type MediaType encapsulates valid test formats, such as Html, Xml or Json.
- @Consumes: modifies method declarations and defines, corresponding to @Produces, which format of text is accepted in the method body of the Http-method when mapping to this Java-method.
- @FormParam: modifies arguments in declarations and allows for mapping from named parameters in the Http-method to named String parameters in the Java-method.

With the use of these attributes, a number of classes and methods can b defined to accept input from all relevant Http-method formats. Based on the form of the request the Jersey framework finds the Java-method suitable for handling the request. Ambiguities are reported at compile time.
As part of the Http specification it is only necessary to implement the GET method to qualify for a valid Http server. We have implemented GET, POST, PUT and DELETE as part of the exercise. All four methods are only implemented to produce html output. The logic behind the methods are rather simple: a List of Task objects are extracted from persistence, its content is returned in the GET method, while the remaining methods modify the lists contents and and writes it back to persistence.

Persistence of Tasks is carried out in a manner nearly identical to the previous exercise; a Task-class encapsulating the data content of the provided Xml-schema is made serializable by Xml-binding, and a wrapping class, TaskSerializer, serializes a collection of tasks with a similar binding. The purpose of the construct have been to correspond exactly to the provided schema without modification.

Several difficulties have been encountered during the exercise:

- General problems with implementing and handling the Jersey framework. Were resolved.
- Difficulties getting the Jersey-defined Java servlet to run on the Apache Tomcat server used for test runs. Were not resolved.
  We have had general problems with running code on the server throughout the entire course, and in this case we have not been able to resolve the problem. When attempting to run the code the server responds with "404

resource not found". As a consequence it has not been possible to run the code and test the client, but the core logic of manipulating the task collection is so simple that we are pretty confident in asserting that it is correct.

- Problems reading from persistence when running the program on the server. Were not resolved.
  It seemed these problems where caused by the execution being run from a different system position when run on the server. During development we had no problems reading from the .xml-document included in the eclipse-project when running the program as a stand-alone Java application. This leads us t believe that the relative file path needed to define access to the file were pointing to an invalid position when the default base path where defined from the server. We were not able to isolate the correct position to place the document at in order for the server to gain access to it, other than to use an absolute filepath, which would break the build when ported to another machine.

3.3 Test run

Due to the problems stated in section 3.2 we have not been able to produce a test run.

3.4 Reflection

This exercise exemplifies an implementation of a RESTful web service through Http, and an underlying handling of the central Http methods "GET", "POST", "PUT" and "DELETE".
The purpose of the REST architecture model is to enforce scalability of web based systems, and definition of uniform and generalized interfaces for communication between web components. The service implemented follows the principal rules of RESTful services:

- The server is separated from any client implementation, as communication happens only by Http.
- The server is stateless; it stores no session state and no information about which requests have been made. During the individual request the server only knows about the state of the resources it exposes before and after the request has been executed.
- The rules on limitations of side effects for the different methods have been followed:
  - The GET method should have no side effects; that is it should not change the representation of any resources exposed by the server.

This is followed, as GET never calls TaskRestServer.writeObjects(), an therefore does not change the content of persistence.

- GET, PUT and DELETE should not cause different results in the content of exposed data if they are called with the same content several times, as opposed to being called only once.
  The is followed by GET for the reasons stated above.
  PUT requests provide a task, and adds it to the collection only if it can find an existing task with the same id. It then deletes the existing task, ensuring that only the first in a sequence of identical calls can change the contents of the collection.
  DELETE removes a task from the collection with the id provided. As it is defined as an invariant of the collection that only one task with a given id can exist (both POST and PUT enforces this invariant), only the first call in a sequence will find any task to remove. Following calls will therefore have no effects.

- The data exposed by the server (tasks) is exposed, transported and modified through an Xml-representation; the implementation of tasks as a Java class is irrelevant to the client.

- Difference in URI identifies resources being addressed. The current solution only supports one resource, but the design would allow for additional URIs to be defined.

3.5 Conclusion

We have implemented a service that follows the ruled and guidelines for a RESTful web service.
The service uses Http for communication, and exposes a collection of tasks for retrieval and modification through the methods "GET", "POST", "PUT" and "DELETE". The tasks are exposed as an Xml-representation.
The implementation of the Jersey framework and the handling of the different requests seems to be working correctly.
We have not been able to run the program correctly on the server used for the assignment, an as such have not been able to fully test the solution. Reading from and writing to the persistent storage of tasks have not been resolved satisfactory.

Security

Contents

4.1 Lab Description

The main objective of this week's assignment is to understand and analyze security models and protocols and furthermore implement a simple security protocol for task manager. As part of the assignment, you will study and develop a simple role based access control mechanism for tasks based on an authentication using ITU credentials. Moreover, you will also use one of the crypto algorithms to ensure security among all/some parts of communication.

4.2 Solution

The handed-out project implements a simple security protocol. There's no way for the server to authenticate the messages from the client and vice versa. The challenge in the assignment is to implement an improved protocol that, among other things, allows a client and a server to exchange a shared secret key which then facilitates authenticated communication between them. Thus providing slightly better security.

The project uses shared private keys between participants i.e., the keys are already distributed between the participants. Token server and client share a private key and so does token server and server and also server and client. The challenge is how to implement a way for the client and server to share a secret key (session key) with which they can communicate in a secure way.

First, the client sends a request to the authentication server (token server). The token server sends back a encrypted response inside which is a ticket encrypted in the token-server key plus a session key for the communication between the client and the server.

```
1 // from client to token server
2 { credentials }K_tc
3 // from token service to client
4 { {role, timestamp, identity, session key}K_ts, session key}
```

The client is able to decrypt the response with the already distributed [token server-client] shared private key. The ticket is encrypted in the [token server-server] shared private key and contains the [client-server] shared key (the session key) plus the clients identity, plus a timestamp and the clients role(note the server later uses the role to authenticate the client against an access rights table restricting access to it's resources).

If this message is intercepted the encryption with the clients private key poses a challenge to the enemy but crucially, if the enemy manages to send the message to the server posing as the client, it is of no matter as the server then compares the identity in the ticket against the sender, if they do not match the server denies access to its resources.

Second, the client sends a 'authenticate' request to the server along with the ticket encrypted in the servers private key. The server decrypts the message and authenticates the client against the identity in the ticket. The session key has now been distributed between the server and client and will be used for communication between them. The server sends a reply consisting of a message and a nonce encrypted with the server-client shared key (the session key). Further communication between client and server is to be authenticated against that nonce.

```
1  // from client to server
2  authenticate, {role, timestamp, identity, session key}K_ts
3  // from server to client
4  if(authenticated)
5          yes, { nonce }K_sc
6  else
7          no, {message}K_sc
```

When the server later receives a message from the client it contains that nonce transformed by some agreed upon function (in this case simply the nonce - 1). By applying the transformation to the nonce the client effectively authenticates it's identity to the server.

Using a nonce like this adds another layer of security. If an enemy intercepts the message it will need to first deal with the session key after which it will still need to know which transformation of the nonce is the correct one in order to continue communicating with the server.

Only now do the client send a resource request to the server. the request message consists of a command plus the transformed nonce encrypted in the client-server shared key plus the data on which to act, also encrypted in the shared key.

The server validates the transformed nonce and the clients role against the access rights table, and finally it checks the message timestamp. It replies by sending a message and yet a transformed nonce (the nonce could possibly be a new transformation of the same nonce?) on which further client-server

communication is based.

```
// from client to server
execute, {transformed nonce}K_sc, {data}K_sc
// from server to client
if(succesful execute)
        yes, {another nonce}K_sc
else
        no, {message}K_sc
```

In this rather convoluted way the server and client now has obtained a shared private key on which to communicate (more) securely. The example still relies on a trusted third party, the principles involved still need to initially share secret keys. So in the end there remains the need for some trusted source, some authentication service.

4.3 Example run

Table 4.1: example run

Token server

```
[TS]: started at 8008
[TS]: credentials received: wrSHHkaitp7EGMfzVyaUsQlmiidESt65wzkQmPWc21++5DNeWTOOGw==
Authentication succeeded for user: nsth
[TS]: response sent
```

Client

```
[C]: server ticket and session key extracted successfully from TS response: HF6FgZRHhOon4xRqzmNyiKtp/j5dnx0VY1i9TM/BpQDapoeEiJSoIduLEkSx4ojMdlvqrqHpW5E=
[C]: session key is: yeah man n all dat
[C]: token is: hZDzbcsRdzXexV9Nid2KrIi5ojmZzt6gO21Xq1aN6HYo7Cb5bw3u49VkE6vuKXJollOKupxwVEHR
JT/GSNiF8pg3z11P7ok52LSegO8+xgc=
[C]: contacting task server...
[C]: message received from task server is: 7gPfZbH31+Q=
[C]: decrypted message from task server is( nonce is ): 251
```

Server

```
[S]: Taskmanager loaded with :6 tasks!
[S]: server started at: 8010
[S]: received client Request: HF6FgZRHhOon4xRqzmNyiKtp/j5dnx0VY1i9TM/BpQDapoeEiJSoIduLEkSx4ojMdlvqrqHpW5E=
[S]: request decrypted: role[student] timestamp[2013-11-11T16:14:26Z] principle[nsth] sessionKey[yeah man n all dat]
The nonce is 7gPfZbH31+Q=
[S]: received client Request: f66OW9GxJBv4rlctW9mU6OfJ70jWQQDbtA72Yyf4oUombH9UFC4DU4515ERgjOjH
altered nonce: 250, task id: qualify-for-examine
[S]: the task with Id:qualify-for-examine executed successfully!
```

4.4 Theory

The solution to security issues in distributed systems is to encrypt messages. All encryption algorithms are based on using a secret (called a key). Encryption algorithms use the key to obscure the content of messages (encrypt the

message), and to decrypt the message using the same key. There are two types of keys and thus two types of encryption algorithms:

shared secret keys: In which both the sender and the receiver knows the secret, note *this is a typical army or corporate solution in that these organizations are able to distribute the secret key, secretely.*

public/private key pairs: In which a principle publishes a public key which can be used to encrypt messages. It doesn't matter if an enemy intercepts the key since only the coresponding private key can decrypt the message.

Cryptography plays three roles in implementing secure systems:

Secrecy and Integrity: Scenario1 (*Secret communication by shared secret key*). The principle uses a shared secret key to encrypt the messages and the receiver uses the key to decrypt the messages. As long as the secret key is a secret secrecy is kept (optionally add a checksum to provide integrity). There are two problems with this protocol: 1) How to share the secret key in the first place? and 2) how can the receiver trust the message is not a replay? Note *This protocol implies that a successful decryption authenticates the sender!*

Authentication: Scenario2 (*Authenticated communication with a server*). One way to provide authenticated communication is by involving a trusted source (say ... a server somewhere). The server hold secret keys for all participants. A principle requests the trusted server for access to a resource (say.. a server somewhere). The trusted source uses the principles key to authenticate the principle and then issues a response encrypted in the principles secret key (this is called a challenge. See below). The response contains a 'ticket' encrypted in the second principles secret key and a new secret key used for further communication between those two principles.

Note this protocol requires a trusted third party. A trusted third party is not always a possibility and the distribution of secret keys requires a secure channel which is not always possible either.

In this scenario a cryptographic challenge is used to eliminate the need for a principle to authenticate itself to the server. The reply sent back from the server is encrypted in the principles key, thus presenting a challenge that only the principle can overcome. This is how we eliminate the need to keep sending a password over the network?

Scenario3 (*Authenticated communication with public keys*). Assuming the communicating principles have distributed a public key this technique allows the principles to establish a shared secret key...

Digital signatures: Verification of the senders identity. Digital signatures uses a 'digest' of the message (a compressed form of the message). A digest is similar to a checksum. A digest function produces a digest of a message and the inverse function produces the message. The digest acts as a signature and accompanies the message.

4.5 Conclusion

Though the improved security protocol in our example provided a means of sharing a secret key between principles, it still assumes a trusted source. And it assumes that the secret keys used for communication between principles and the trusted source are distributed safely. Therefore this protocol has som issues that could be improved upon...

This example used private keys but using public/private keys pose the same challenge. We would still need to trust the distributer. When transmitting messages over a network ...

the nonce transformation function need to be known by server and client. How can they share that secret securely?

Indirect Communication

Contents

This chapter is about indirect communication. In distributed systems there's often a need for multicasting messages. It is much more efficient to send a single message to a group of processes instead of sending that message to a group as a series of one-to-one messages. IP multicast is an example of a simple group communication protocol which does not provide guarantees for atomicity, lossless messaging or ordering of messages. These properties are however provided by middleware like JGroups.

We shall describe the result of the JGroup exercise. In section 5.2 we demonstrate our solution. In section 5.3 we show an example run of the solution. In section 5.4 we discuss the underlying theory and relate that to the solution. In section 5.5 we round up the chapter.

5.1 lab description

The primary focus of this week?s 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 it?s 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.*

5.2 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 a group of receivers. The receiver's task is to receive the requests and execute them.

The sender instantiates a 'channel' object. This is similar to a socket and one of the main concepts in the JGroup API. messages are sent over the 'channel' to the group or to individual processes. A 'Message' object sent over the channel takes the sender's and receiver's addresses as well as a marshalled message. If the sender and receivers' 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 the envelope to the group. i.e., a request for a given operation on a given task is put into an envelope object with the task's ID. The envelope is marshalled and then multicast to the group in a JGroup Message.

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

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.

5.3 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. Note the receivers setState method is not implemented as the time of writing!!!

Listing 5.1: 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 group 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 and sends that message to the group.

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

Listing 5.2: 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 interface or extend receiver-Adapter and simply override the 'receive(Message)' method. In our simple application we content our selves to do the later.

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

Listing 5.3: 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());
        }
}
```

5.4 Motivation- and theory

Remote invocation paradigms (RPC, RMI) imply a coupling between the participants that is often not desirable in distributed systems. i.e., the sender need to know the receiver.

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. One process sends to several subscriber processes.*

Uncoupling the communicating entities reveal two properties of indirect communication:

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:

- **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 ...?

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

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

- **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.

5.4.1 Group Communication Programing Model

A group has a conceptual group-membership. Processes may join or leave the group transparently. 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))

- **Integrity**. The message received is the same as the one sent, and no messages are delivered twice.
- **Validity**. Any outgoing message is eventually delivered.
- **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 ordering to group membership. Among other things this includes:

Table 5.1: JGroup properties

Lossless transmission of a message to all recipients (with retransmission of missing messages.)
Fragmentation of large messages into smaller ones and reassembly at the receiver's side
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)
Atomicity: a message will be received by all receivers, or none of them.
Knowledge of who the members of a group are.
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 etc.

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 group), a sender address and a message.

5.5 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 configurations of the protocol stack.

Concurrency Control

TO DO!

6.1 Example run

```
Please enter the path to task-manager-xml!
>
src/resources/task-manager.xml
Task manager with 5 tasks loaded!

------------------------------------------------------------------
GMS: address=Task manager# 45413, cluster=Add Tasks Channel, physical address=fe80:0:0:0:24ec:d1c5:cd48:fd6e%13:59099
------------------------------------------------------------------
** view: [Task manager# 45413|0] (1) [Task manager# 45413]
Channel address:Task manager# 45413
Usage: 'execute' | 'request' | 'trace' | 'exit'
> ** view: [Task manager# 45413|1] (2) [Task manager# 45413, Task manager# 18390]
execute
type id of task you would like to execute!
> review-01
Number of grants required: 1
Usage: 'execute' | 'request' | 'trace' | 'exit'
> ENVELOPE received
ENVELOPE deserialized
ENVELOPE received
ENVELOPE deserialized
denyLock..................................................... (another process holds the resource.)
ENVELOPE received
ENVELOPE deserialized
requestlock.................................................    (no other process holds the resource.)
TaskManager contains taskid: false
ENVELOPE received
ENVELOPE deserialized
EXECUTING ..........................................
Task manager# 45413[ command: request, source: Task manager# 18390 ]: Task :review-01 marked as required successfully!
```