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: 4.0

Date: December 9, 2013

Summary:

- Chapter introduction v.1.0 added.
- Chapter TCP v.3.0 added.
- Chapter security v.3.0 added.
- Chapter Servlet v1.0 added.
- Chapter Indirect Comunication v.2.0 added.
- Chapter REST v.1.0 added.
- Chapter Mobile v1.0 added.
- Who-did-what added

December 9, 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 talk about the TCP protocol. 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 try to connect the example to the theory behind.

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.

The task manager TCP server and its clients follow a strictly predefined structured communication based on the following conversational protocol.

- Initially a client will send a command to the server indicating that it wishes to consume the particular service offered by the server through the mentioned command.

- After receiving a command from the client, the server will respond by sending the same command back to the client indicating its readiness to provide the service mentioned in the command.

- Soon after receiving the command from the server, the client will send necessary data to the server to further process the command on the server side. In case of POST and PUT commands, the client will send a Task object, where as in case of GET or DELETE commands, the client will send an attendants name or a task id respectively.

- Finally, after processing the command the server will send back the results to the client. In case of GET command, the server will return a list of tasks (List<Task >) object to the client. On the other hand, it will simply return the result of that command (e.g. task deleted). The result could also contain an error message if any (e.g. task Id not found in the task list).

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.*

- *[Optional] The server described above can only handle one single client at a time in each run. In order to make the server more robust and handle multiple clients concurrently, one may follow the approach suggested in the page. 173 of the course textbook [DS], to create a new connection for every client request that will run on a separate thread. Therefore, develop a TaskManagerTCPServer that can handle multiple concurrent clients.*

- *[Optional] The functionality of TaskManagerTCPServer can be extended to offer more commands (e.g. OPTIONS, HEAD) similar to the HTTP protocol. In case of OPTIONS command, one may describe the list of commands offered by the server, where as for HEAD, one may only provide the number of tasks available for a given attendant instead of sending all the available tasks to the client.*

2.2 Solution

Due to the general pressure of the study we did not find the time to elaborate much on this excersize. Hence the optional parts are not implemented.

We use JAXB API to provide persistence for the taskmanager objects. JAXB is java middleware capable of marshalling and unmarshalling java objects into xml and back into java objects, in this case a 'taskmanager' object with a related collection of Task objects. To send the taskmanager object over TCP/IP, objects are transformed into bytes by ObjectInputStream and ObjectOutputStream (only objects that are serializable may be written as a bytestream).

Our TCP server first opens a serversocket on a specified port and continues to listens for clients . A client instantiates a socket specifying the serversocket's

address and port number. The client and server then communicates by passing byte streams through the agreed upon port. The transmission of bytes is done with ObjectInputStream and ObjectOutputStream methods.

Serialization

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

ObjectInputStream/ObjectOutputStream are java methods for transforming serializable java objects to and from bytestreams.

In Java, serializable objects can be sent through a connection by ObjectInputStream and ObjectOutputStream classes which transforms an object or a graph of objects into an array of bytes (for storage or transmission), and back into objects again.

Our server and client communicates by passing bytes in and out of matching inputstreams and outputstreams. Command messages (requests) are passed via the writeUTF and received by the readUTF methods as DataStreams. The Task and taskmanager objects, implementing serializable, are sent via ObjectOutputStream and received by ObjectInputStream.

The server responds to requests by sending the request back. The client waits for this confirmation before proceeding.

JAXB

Serialization and deserialization to and from xml is done with the JAXB API. Quoting 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.

```
1  @XmlRootElement(name = "taskmanager") // JAXB annotation
2  public class TaskManager implements Serializable{
3
4          @XmlElementWrapper(name = "tasks") //
5      @XmlElement(name = "task")
```

```
6       public List<Task> tasks;
7   }
```

TCP

The Transmission Control Protocol (TCP) is one of the main protocols of the transport layer of the TCP/IP suite. TCP provides reliable and ordered streams over the internet. TCP is located at the transport layer where it receives data from the application layer and sends data to the lower level internet layer IP protocol.

TCP transforms data into packets, a sequence of bytes consisting of a headder (metadata) and a body (the data).

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 will wait for a client to connect. The Client creates a socket on the same port which takes the server's InetAddress and the port it is listening on as arguments.

2.3 example run

In our example the server sends out confirmation messages to the client for each request received. The client proceeds, only after receiving an acknowledgement of its request. This mimics the way the TCP protocol uses acknowledgements (acks) to confirm packet reception. Acks are the way the TCP protocol achieves *reliability*. A client will wait for an ack and if none arrives it retransmits the package.

After establishing a socket connection between client and server the client sends a request to the server. The server responds by sending an ack to the client (in this case the ack is the request itself).

Listing 2.1: server sends an ack to a request

```
1   while(running){
2           // client request
3           String message = dis.readUTF(); // [blocking call]
4           // accept client request by returning the (request) message
5           dos.writeUTF(message);
```

The client continues, only after receiving the ack from the server.

Listing 2.2: client request and wait for ack

```
1   String message = "POST";
2   dos.writeUTF(message);  // send the request
3   response = dis.readUTF(); // wait for ack [Blocking call ]
```

```
4
5   Task t = new Task();
6   t.name = "Tout les circles";
7   t.id = "one more cup of coffe";
8   t.date = "15-09-2013";
9   t.description = "recondre";
10  t.status = "mais jai le plus grande maillot du monde";
11  t.attendants = "bjarne, lise, hans, jimmy";
12
13  // if server acknowledges
14  if(response.equals(message)){
15          ous.writeObject(t);
16  }else{
17          System.out.println("Client: the server did not acknowledge the request.");
18  }
```

The client-server uses readUTF() and writeUTF() to send requests and acks but in order to send an object they use writeObject() and readObject() to transform the object into a byte stream to send over the socket. *Only objects implementing serializable can be transmitted by writeObject().* After receiving the bytestream we cast it back into an object.

After executing a POST operation the server then proceeds to persist the taskmanager object by marshalling it with JAXB.

Listing 2.3: server POST

```
1  // expect an object from the client
2  if(message.equals("POST")){
3          Task t = (Task) ois.readObject(); //
4          serializer.allTasks.tasks.add(t);
5      serializer.Serialize();
6  }
```

Before the example run the taskmanager xml document looked like this:

Listing 2.4: 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>
```

After the run, in which the client requested the server's POST method with a new Task, the xml document looks like this:

Listing 2.5: xml after POST

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

```
4                   <task id="handin-01" name="Submit_assignment-01" date="16-12-2013" status="
                        not-executed">
5                       <description>
6                               Work on mandatory assignment.
7                       </description>
8                       <attendants>student-01, student-02</attendants>
9               </task>
10              <task id="one_more_cup_of_coffe" name="Tout_les_circles" date="15-09-2013"
11              status="mais_jai_le_plus_grande_maillot_du_monde">
12                      <description>recondre</description>
13                      <attendants>bjarne, lise, hans, jimmy</attendants>
14              </task>
15      </tasks>
16  </taskmanager>
```

Not included in this summary, but in the complete example the client also requests the GET method and the PUT method wanting to change the new task's description property. Finally, the client requests the DELETE method with the taskid 'handin'01' parameter. After the run the xml document now looks like this:

Listing 2.6: xml after PUT

```
1   <?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>
```

Table 2.1: console output tcp client

```
Client is connecting to server...
Client: requesting GET
Client: server response : GET
Client: Server accepted. Waiting for object
Client: requesting Task object
Client: Object received 'Submit assignment-01'

Client: requesting POST
Client: server response : POST
Client: server accepted. Sending object
Client: server response : Task with id: 'one more cup of coffe' saved

Client: requesting PUT
Client: server response : PUT
Client: server acknowledged. Sending updated object
Client: server response : task with id 'one more cup of coffe' updated

Client: requesting DELETE
Client: server accepted. Sending parameter
Client: server response : Task with id handin-01 deleted
Client signed out.
```

Table 2.2: console output server

```
Server is listening at port 7896
Server: A client is connected

Server: Waiting for client request
Server: client request received : GET
Server: Waiting for parameter
Server: parameter  received : 'handin-01'... Looking for object
Server: Object found 'Submit assignment-01'...Sending object

Server: Waiting for client request
Server: client request received : POST
Server: object received 'Tout les circles', persisting object...

Server: Waiting for client request
Server: client request received : PUT

Server: Waiting for client request
Server: client request received : DELETE

Server: Waiting for client request
Server: client request received : quit
Server is closing.........................
```

2.4 Reflection

We have used this example to demonstrate and experiement with the TCP/IP suite. There are two transport protocols in the TCP/IP suite; UDP and TCP. UDP transfers text messages and TCP transfers byte streams. Unlike UDP, TCP is reliable and ordered. TCP can provide guarantees as to the ordering of its messages i.e., messages arrive in the order they were sent. It does so by giving every packet a number and then ordering the packets on arrival. TCP provides guarantees as to the delivery of the messages i.e, the message eventually arrive. It does so by using acknowledgements and retries (resend messages).

In contrast, the underlying network protocol (the IP protocol) offers only best-effort semantics, there is no guarantee of delivery and packets can be lost, duplicated, delayed or delivered out of order. IP delegates the task of providing a reliable service to other layers, namely the TCP layer atop it. This nicely demonstrates the end-to-end principle. Application specific functionality should reside at the end hosts.

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.

Our simple example mimiced the way the TCP protocol achieves reliability. By sending acknowledgements our client-server is, in principle, capable of achieving reliability. It could be made to resend a packet in case of no response from the other end. Off course things like latenzy, troughput or duplicate packages come into play here.

The code in our solution is very much a 'proof-of-concept'. The code style 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 theory behind.

As with many other examples, we did not have time to implement the optional parts of the assignment. This is much regrettable but we feel we at least have had a good introduction to the TCP/IP suite.

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 (before improvements) implements a simple security protocol. This protocol has a few short-commings. 1) The protocol is not secure against man-in-the-middle attacks and 2) how can the server trust the clients message is not a replay? Note *The protocol implies that a successful decryption authenticates the sender!*

The challenges in the assignment includes how 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. How to improve the protocol to provide security against man-in-the-middle attacks and replays.

The handed-out project uses shared private keys between participants i.e., private keys are known between [token server : server] and [token : server-client]. These keys are already distributed between the participants. The protocol, however, provides no means for the server to authenticate the client. Thus it is open to man-in-the-middle attacks and replays. A way to improve the protocol is to make the server able to authenticate the client, much liek it is done in the Needham-Schroeder protocol.

In our solution, first the client sends a request to the authentication server (token server). The token server (upon authenticating the client with their shared

secret key) sends back a encrypted response inside which is a ticket encrypted in the [token:server] private key, plus a session key for the communication between the client and the server. Note this is another insecure issue in the original protocol in that the authentication server (ITU server) have no means of authenticating the client.

```
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] private key. The client now has a ticket plus the secret key (session key) to use for communicating with the server. The ticket is encrypted in the [token server : server] shared private key (so if the respons is intercepted its content is secure) 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 (from the token-server to the client) is intercepted the encryption with the clients private key poses a challenge as do the ticket encrypted in the servers private key but, if the enemy manages to intercept and 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, and if they do not match the server denies access to its resources. Thus authenticating the client provides a protection against man-in-the-middle attacks.

Second, the client now in possesion of the session key, 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 ticket contains a timestamp used to timeout requests.

But first, the server sends a reply consisting of a message plus a nonce, encrypted with the [server : client] shared key (the session key). The purpose of the nonce is to provide proof to the server of message authenticity i.e., if the enemy intercepts and resends the message posing as the client, the server will be able to compare the nonces to authenticate the message.

```
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 // nonce, preventing replay
6 else
7         no, {message}K_sc // nonce
```

When the server receives a new message from the client it should contain the nonce transformed by some agreed upon function (in this case simply the nonce - 1). By applying a transformation to the nonce the message/client effectively authenticates it's own identity to the server. Messages from the client must come in a predetermined order, the transformation of the nonce in each message in the communication-. The use of nonces prevents replay by numbering the messages in a way known to the server and client only.

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 authenticates the message (against the transformed nonce) and the clients role against the access rights table. 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.

```
1  // from client to server
2  execute, {transformed nonce}K_sc, {data}K_sc
3  // from server to client
4  if(succesful execute)
5          yes, {another nonce}K_sc
6  else
7          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 protocol offers protection against man-in-the-middle attacks in several levels. The example uses nonces to authenticate the client and it provides the client identity, encrypted in the ticket, to the server again authenticating the client and the message as not a man-in-the-middle attack. The example still relies on a trusted third party, the principles involved still need to initially share secret keys. In the example the ITU server is the trusted source.

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 Relating To The Theory

For a recap of Worst Case Assumptions an a definition of the enemy see the appendix.

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 a key to obscure the content of messages (encrypt the message), and to decrypt the message. Encryption algorithms are based on one-way functions i.e., a function that converts an input to an output from which the input can not easily be deducted.

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 solution for organizations which 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 (idealy). Public/private algorithms (asymmetric algoritms) use trap-door functions. Trap-door functions are one-way functions that can be inversed by applying a key.

In our example we used the JAVAX.CRYPTO package to encrypt end decrypt messages. We used the Data Encryption Standard (DES) algorithm but other algorithms can be used.

Cryptography plays three roles in implementing secure systems:

Secrecy and Integrity: Scenario1 (*Secret communication by shared secret key*). A principle use a shared secret key to encrypt the messages and the receiving principle use the shared key to decrypt the messages(hence the name *symmetric cryptography*). As long as the secret key is a secret secrecy is secure. 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!*

This is the scenario that the handed-out example (before we improved on it) closest resembles. There' no secret key between the server and client and the server can not be sure the messages from the client are not replays.

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.

This is the protocol that closest resembles our finalized example. 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 a way of eliminating the need to keep sending a password over the network.

Scenario3 (*Authenticated communication with public keys*). A principle makes a public/private key pair and makes the public key available. Anyone can encrypt a message using the public key but only the principle can decrypt those messages as he is the only one with the private key.

4.5 Conclusion

Sending messages in distributed systems poses a security risk. The enemy can intercept messages, man-in-the-middle attack, and the resend the message posing as the client, replays. To protect against those we use encryption and carefully designed protocols.

Instead of authenticating by password we use a challenge. A challenge is an encryted message which can only be decrypted with the right key. Inside the encrypted message is a ticket, another challenge, encrypted in the server key. By adding the clients identity to the ticket a server can authenticate the sender of the message. By adding a sequence of nonces the server can authenticate subsequent messages from the client.

An enemy amy intercept the initial message from the client to the authentication/token server but will be unable to decrypt the message, and ticket inside. The enemy may send this message to the server anyway but the server can verify the sender's identity by comparing the identity inside the ticket to the sender.

An enemy may intercept messages from the client to the server and replay a message posing as the client but the sequence of nonces in the message enables the server to authenticate the message. If the enemey does not know the right nonce tranformation this is also a futile effort.

Though the security protocol in our example does provide protection against man-in-the-middle attacks and protection against replays it still assumes a trusted source. It assumes that the secret keys used for communication between principles and the trusted source are distributed safely. Eliminating the need for a trusted source is desirable but unatainable.

Even when principles share private keys there is a possibility of a man-in-the-middle attack. If the initial message from the client to the token server is intercepted the server will not be able to detect the attacker as it will be its identity inside the ticket.

This example used private keys. Using public/private keys would present a stronger encryption but suffer the same short-commings.

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

Group Communication

Contents

This chapter is about group 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

Our 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 message. If the sender's and receiver's 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 multicast to the group as 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.

5.3 Example Run

In our taskmanager application 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 receiver's setState method is not implemented at 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. The WriteToChannel(envelope, channel) method then serializes the envelope and wraps it in a JGroup Message object and sends that message to the group.

Listing 5.2: eventloop

```
1  // create an empty message container
2  Envelope envelope = new Envelope();
3
4  switch (command.toLowerCase().trim()) {
5          case "request":
6          System.out.println("type or paste task Xml you want to request (in single line)!");
7          System.out.print("> ");
8
9          String requestXml = in.readLine();
10
11         Task requestTask = TaskSerializer.DeserializeTask(requestXml);
12         envelope.command = command;
13         envelope.data.add(requestTask);
14
15         // here we send the message to the group. (message, Channel)
16         WriteEnvelopeToChannel(envelope, channel);
17
18                 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 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

```
1  if(DeserializeEnvelope.command.equals("request")){
2              Task taskWithId = GetTaskWithId(DeserializeEnvelope.data.get(0).id);
3
4              if (taskWithId != null) {
5                      // execute therequired 'action' on the given Task object
6                  taskWithId.required = true;
7
8              try {
9                      // persist changes
10                     provider.PersistTaskManager();
11             } catch (JAXBException ex) {
12                 System.out.println(prefix + "Failed to persist envelope Xml. Error message" + ex);
13             } catch (IOException ex) {
14                 System.out.println(prefix + "Failed to persist envelope Xml. Error message" + ex);
15             }
16
17             System.out.println(prefix + "Task with Id:" + DeserializeEnvelope.data.get(0).id + " requested!
                       " + "total number of tasks: "
18                       + provider.TaskManagerInstance.tasks.size());
19
20         } else {
21             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());
22         }
23 }
```

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'.

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*. E.g. IP multicast is space un-coupled but time coupled.

Various techniques for indirect communication exist:

- **Group communication:** A message is sent to an address -the intermediary- and from here 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 the intermediary fails all communication is down.

- **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:

- **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.

5.4.2 Ordering

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

Ordering of event matters. E.g. a question should appear before its answer. A flight reservation can be made only if the seats are available. The challenge is that two computers cannot rely on a shared definition of time and must communicate only by sending messages.

So we talk about logical time: we know the order of events within the same process and we know that any message is sent before it is received. Lamport called this *partial ordering* or the *happended-before* relation.

Lamport phrased the term *logical clock*, a way to timestamp events achieving partial order. A Lamport timestamp is a way to determine the order of events. A process increments its counter before each event in that process. When a process sends a message, it includes its counter value with the message. On receiving a message, the receiver process sets its counter to be greater than the maximum of its own value and the received value.

```
1  e1 -> then L(e1) -> L(e2)
```

Note e1 - e2 are partially ordered events since the reverse is *not* true. If the Laport time of one event suggests it happened before another it is not to say the actual event happened before the other.

The concept of *vector time* takes one step further toward ordering of events and provides that:

```
1 e1 -> e2 <-> L(e1) -> L(e2)
```

Vector time does so by letting each process hold a record (Vn) of the time of its own events and that of all other processes' events as well. This way the ordering of events in a distributed system can be determined in that the time of an event, ti, is the number of events that happened in process i before the event.

In other words, a Vector holds the time of the past events of all processes. The order of two given events is easy to determine once the time of previous events is known.

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 like 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. JGroups 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.

Ordering of messages is important. Lamport phrased the term *happened-before* and provided the *Lamport timestamp* in which each process encrements its counter before sending a message. The receiver compares its own counter to that of the received message to determine the order. This is known as *partial ordering*.

Vector clocks does one better providing total ordering by having each process keep a vector of timestamps for all processes, a sort of Lamport timestamp for each process. This is known as total ordering.

Concurrency Control

Contents

This chapter is about concurrency control. In distributed systems where there are several different nodes which needs to perform operations on some shared ressources, there is a need for controlling the access to these ressources in order to maintain consisteny and ensure correctness of the operations.

We shall describe the result of the mutual exclusion exercise. In section 6.2 we demonstrate our solution. In section 6.3 we show an example run of the solution. In section 6.4 we discuss the underlying theory and relate that to the solution. In section 6.5 we round up the chapter.

6.1 lab description

The main objective of this weeks assignment is to understand some of the primary challenges of concurrency control. In order to understand those challenges, we will use Task Manager group communication, bulding on the last weeks lab exercise using JGroups.

The sample code is provided in order to help you to understand how tasks can results into different states if it is executed concurrently at different task manager servers. This weeks sample code is more or less similar to the last weeks sample code and uses JGroups for group communication among the Task manager servers. Even though each task manger uses an individual task manager Xml file, the tasks in the xml files are the same. It offers two simple operations on the tasks as described below.

- ***execute:*** *accepts id of a task and executes the task by assigning the status attribute to executed and required attribute to false. After the execution of the task, the task manager server will inform all other task manager servers in the group to execute the task matching to id.*
- ***request:*** *accepts id of a task and assigns the required attribute to true for the task matching to the id. Then, the task manager server will inform to all other task manager servers in the group for making the same changes to the task at their respective locations.*

The basic idea is to initiate to execute and request operations on a task with same Id from two different instances of task manager concurrently. If these

operations are carried out concurrently at different task manager servers, then the task would results in different states based on the sequence of two operations carried out at the respective task manger servers. For example, at one task manager, if execute is followed by request, then the final state of task will set the attribute required=true, on the other hand at an another task manger server, if the request followed by execute then the task will have the attribute required=false.

The Assignment
As part of the this weeks assignment, your task is to Implement a mutual exclusion algorithm (as explained in chapter 15 of DS text book) protecting the change of state of a task. Furthermore, analyze your mutual exclusion algorithm for:

- *Safety (ME1)*
- *Liveness (ME2)*
- *Ordering(ME3) [optional]*

as described on page 650 of DS text book. You should also try to evaluate the performance of your mutual exclusion algorithm according to the criteria specified on page 651 of DS text book (bandwidth, client delay and synchronization delay).

6.2 Solution

The solution to this assignment is primarily based on the solution of the former chapter, Indirect Communication, where there are two primary classes for solving this assignment; TaskManagerServer and TaskReceiver2.

The TaskManagerServers primary function in the relationship between two the classes, is to send request to other TaskManagerServers, while TaskReceiver2 receives the requests on behalf of the TaskManagerServers and executes them.

The solution uses the same JGroup system for sending and receiving messages, so this chapter will not so much focus on the setup of the two classes, but rather the process of how requests are handled in this mutual exclusion solution, which uses the simple concurrency control protocol as described on the blog: `https://blog.itu.dk/SMDS-E2013/schedule/lab-exercise-07/`

The General idea of the simple concurrency control protocol is; that each Server must be able to lock the tasks which they wish to execute, send a request to all other servers and wait until it has been given authorization to execute the task.

In order for the Servers to be able to lock the tasks they wish to execute, each Server has a hashtable which contains the taskId as key and then the amount

of Grantlocks it needs, which is the number of members in the group -1, before the task is executable, as value.

When we need to process the requests and execute the tasks we are in need of 4 different tokens: RequestLock, GrantLock, DenyLock and Commit. We havent made the tokens, however we use strings which we pass around to the different receivers using the envelope. The envelope now contains two string fields; command and lock, in order to tell the receiver what to do.

When a Server wants to execute or request a task, it first checks if it already has the task listed in its hashtable. If the task is already there, then the user will be informed that the task is already enlisted. Else we add the task to Servers own hashtable with the number of grantlocks needed, and then sends a request to all the other members of the group.

When the group members receives a request they first check if they have the task enlisted in their own hashtable. If they do not have the task enlisted, then they will send a grantlock back to the server which requested it. If the server do have the task enlisted in its hashtable then it will send a denylock back to the requesting server.

If the server receives a denylock it will immediately delete the specified task from the hashtable. However if the server receives a grantlock, then it will countdown the number of grantlocks needed for the specific task, and if the countdown hits zero, then the server sends a Commit to all members of the qroup.

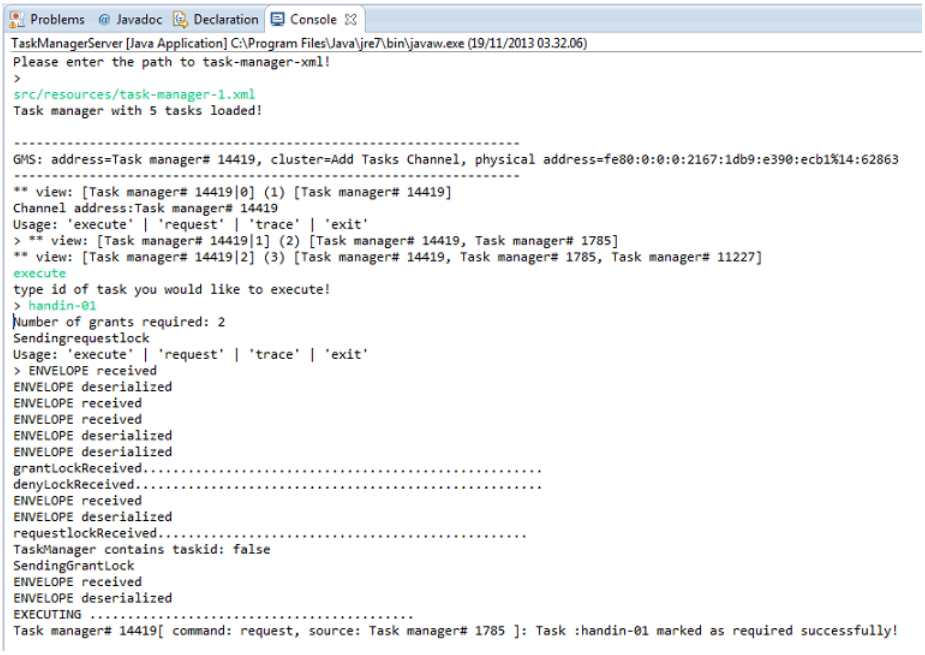
When a Server is given a Commit, it will execute the command on the specified task and save its xml-file. And thus is the simple concurrency control implemented.

We ran into some troubles concerning how to send a message to only one member of the group. There where also some degree of confusion as when to send a message to one or all members of the group.

6.3 Example Run

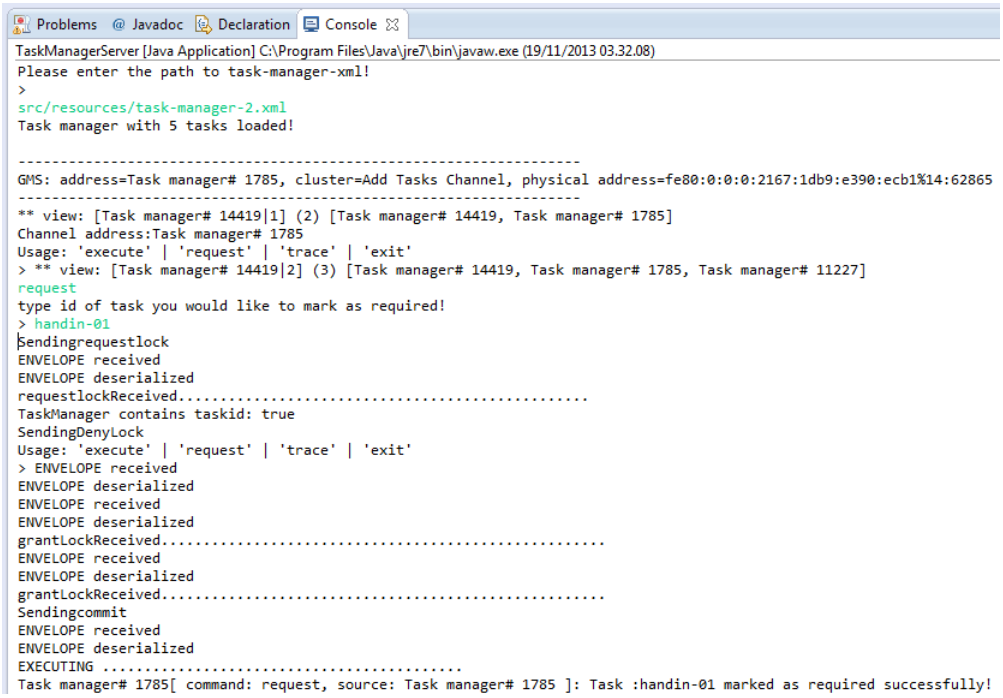
In this run we will run 3 TaskManagerServers each with its own xml task file. After the servers has been initialized we will ask two of the servers to do an execute and a request operation on the same task, and then hopefully one of them will be able to run its command, while the other will be denied.

Table 6.1: Server 1



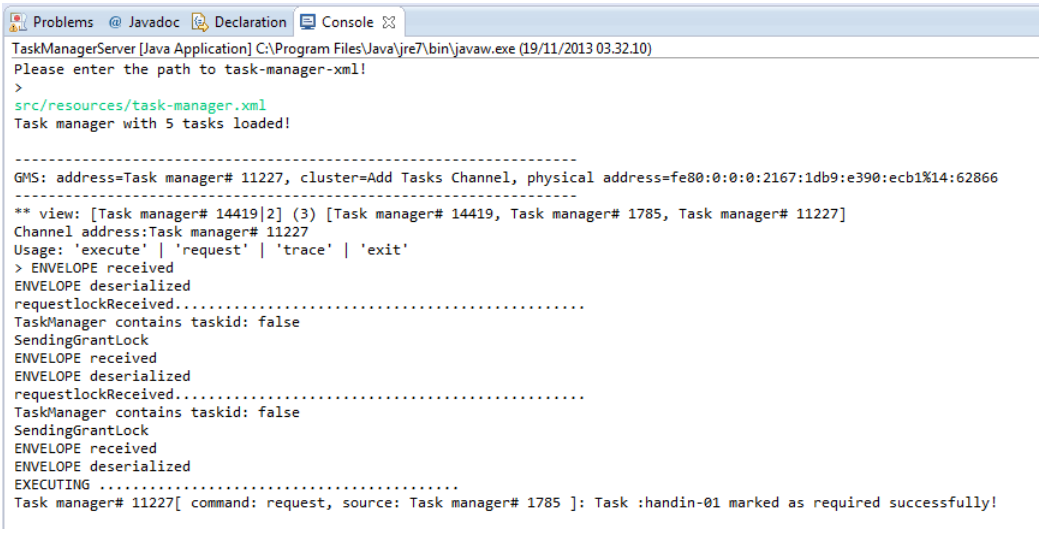
In the first server we run the execute command. The server sends a request to the other servers and receives a grantlock from server 3, which is not running any commands, and a denylock from server 3, which is running the request command. After receiving the denylock, the server then deques the task and waits for new orders. Then after a while the server is asked for grantlock from server 2 which it gives, because it no longer has the task in the hashtable. After that it then receives a commit from server 2, and performs the request command on the specified task.

Table 6.2: Server 2



In the second server we run the request command. The server send a requestlock to the other servers, but before it receives any feedback, it gets a request from server 1. Since the server has the same task in the hashtable, it sends back a denylock. After sending the denylock, it receives a grantlock from both servers, and then because the requirement for grantlocks is fulfilled, it sends a commit to all servers including itself, telling them to execute its command on the specified task.

Table 6.3: Server 3



In the third server no command is run. The server receives a requestlock from the other servers and since it doesnt have any commands running, it simply sends back a grantlock. After a while it then receives a commit from server 2 and performs the command on the specified task.

We sometimes run into the problem, that two servers might both get a denylock, if they send the requestlock on the same time. This means that no command will be run on the specified task, which is undesirable. The problem can be solved by using timestamps and queue the request.

6.4 Motivation- and theory

6.4.1 Coordination: Mututal Exclusion

Mututal exclusion algorithms are used in systems where there is some common resources, which are accessed from several different users eg. servers. The algorithms for distributed mutual exclusion rely on message passing which can either be; permission based or token based.

For all distributed algorithms there are the following requirements:

- **ME1:** At most one node may be in the critical section at a time (Safety)
- **ME2:** Requests to enter/exit eventually succeed (liveness)
- **Optional ME3:** Requests are served in happened-before order (ordering)

Following the requirements there is also a set of mutex quality measurements:

- **Bandwidth:** messages per entry/exit

- **Client delay:** messages required for entry+exit
- **Synchronisation delay:** messages required from resource is exited, until it is next entered

There are several different ways to make a permission based mutual exclussion. It can be:

- **Centralised:** A single coordinator handles entry.
- **Decentralised:** A majority of coordinators must say ok before entry.
- **Distributed:** Everyone must say ok before entry.

One of the more prominent token based algorithms is:

- **Token Ring:** A token is constantly passed around when a node needs the token it, accesses it, when it gets it, and passing it on when releasing it.

There are also other mutual exclussion algorithms, which are based on elections, which are very good at handling crashes.

The simple concurrency control protocol. Which we used to make this assignment, is most similar to distributed permission, as the node needs permission from all other nodes. Though they are very similar, the concurrency control lacks in some areas.

Fulfilling requirements:

- **ME1(Safety):** Since all nodes must give a grantLock before a node may enter the critical section, it is pretty certain to say that it is save.
- **ME2 (Liveness):** In terms of Deadlock we are certain that only one Node is capable of entering the critical state so there should be no problems in exiting the critical state. However when a Node requests permission from the other nodes and one of the nodes crashes, then that node will never be able enter the critical state, and all other nodes will be prevented from entering, which is a clear deadlock! In terms of starvation the concurrency control is also very poor. When a node is given a denylock, the operation is simply removed, which means that it will never be executed. That problem leads to ultimate starvation. Also because of this, the concurrency control isnt fair either, and doesnt even pass the other fairness requirement, which is happened-before ordering, which is also stated in the run.
- **ME2 (Ordering):** As stated in Liveness the concurrency control doesnt support happened-before ordering, meaning that if there are two requests on the same task, then the request with the "smallest" timestamp will execute first.

Quality of the simple concurrency control protocol:

- **Bandwidth:** The concurrency control uses 3(n-1) messages to enter and exit the critical stat. First have to send requests to all other nodes, then all the nodes receiving the requests has to send back a reply, and finally when the operation is going to be executed, the node again has to send a message to all the other nodes, in order for them to executing operation as well. This is terribly slow!
- **Client delay:** In terms of client delay the concurrency control again uses 3(n-1).
- **Synchronization delay:** Since the concurrency control has not implemented the option to queue operation requests, but just sends denylock, which ensures that the request is terminated, then the time for a the resource to be entered into CS again will take 2(n-1). This again states how bad this system is!

6.5 Conclusion

We have implemented the simple concurrency control protocol as described in the assignment description, with one deviation being that we use strings instead of tokens. The SCCP successfully satisfy the requirements of the assignment, however there are many drawbacks to the chosen solution. The first problem we run into is the deadlock, where one node waits for a request from a crashed node. In terms of livenes and ordering it would be pretty simple to implement a queue of requests for each node, and to add a timestamp to the requests, which would fix the problems with starvation and ordering. The queue would also improve the Synchronization delay from 2(n-1) to 1. So to sum up; the solution meets the requirements, but could certainly be better!

Mobile client

Contents

7.1 Lab Description

The main objective of this assignment is to understand some of the some of the basic primitives and challenges of mobile computing.
You are required to expose Task managers Task entity as Google Endpoint (similar to the Todo item in the sample code) hosted on the Google App engine web application project, so that it can consumed by the mobile devises. Furthermore, also develop the functionality to send push notifications to the registered mobile devises whenever a new task is created on the app engine background. On the mobile application side, develop functionality for listing the tasks from the app engine background and also to display the push notification messages received from from the app engine backend whenever a new task is created on the app engine backend.

7.2 Solution

The solution implemented makes use of the Android Development Tool Bundle; a collection of programs and tools assisting creation of software run on the Android platform. Using the Eclipe-integrated software suite the majority of the code needed to complete the assignment can be auto generated. During the development the Google plugin for Eclipse is also heavily used.
Communication between the parts of the distributed solution is handled by Google endpoints. These are java classes that exposes methods for receiving and modifying data objects. Each endpoint is expected to expose a single data entity, as defined by the javax.persistence namespace. The endpoint framework manages the communication between endpoints by converting method calls to REST requests and method return type to REST responses (see section 3.4). Methods used by the endpoints are identified by use of method signature attributes defined in the com.google.api.server.spi.config namespace.
The solution consists of two eclipse projects: a backend and a client. The backend is deployed to the Google App Engine; a cloud service allowing for web services communicating via Google endpoints to be deployed and run on Googles dedicated servers. The client is an Android application communicating

with the backend by use of Google endpoints.
In order to create the backend, a Task entity class is defined. This is a simple data object exposing the same data fields as the contents of task-manager-xml.xml. From this entity an endpoint exposing the entity can be generated. This endpoint provides functionality for receiving, adding, updating and deleting Tasks from the persistence managed by the service. Two additional endpoints are automatically set up to assists the program execution; one for transmitting device data, when a device registers for updates from the service, and one for sending and receiving message data. Persistence is managed by the framework, and accessed through the EntityManager class.
The Android clients control flow is entirely event driven. An application on this platform runs a number of activities, and these activities respond to events such as the screen being touched, or the device turning on and off. The application we create only run a single activity, RegisterActivity. This activity allows the user to register the device, so that it can receive notifications from the server.
We modify the TaskEndpoint class, so that insertion and updating of tasks triggers a broadcast message to all devices registered. The message contains a short list of the latest tasks to be added to the collection, printed out in a simple text format. We use a DeviceInfoEndpoint to gain access to the collection of devices that have registered to the service via RegisterActivity. We then simply iterate over this collection, and send the message to each device. In order to send the message we have to make use of the services API key; a unique number identifying the service on the Google App Engine.

7.3 Example Run

In order to debug and run the client we are setting up a virtual Android machine that can execute the code.

A number of requests are send manually to the TaskEndpoint to create new Task objects on the server. Figure 7.1 shows the request to the server and the response. Consequently, a number of tasks are created on the site. Figure 7.2 shows a list of the tasks created.

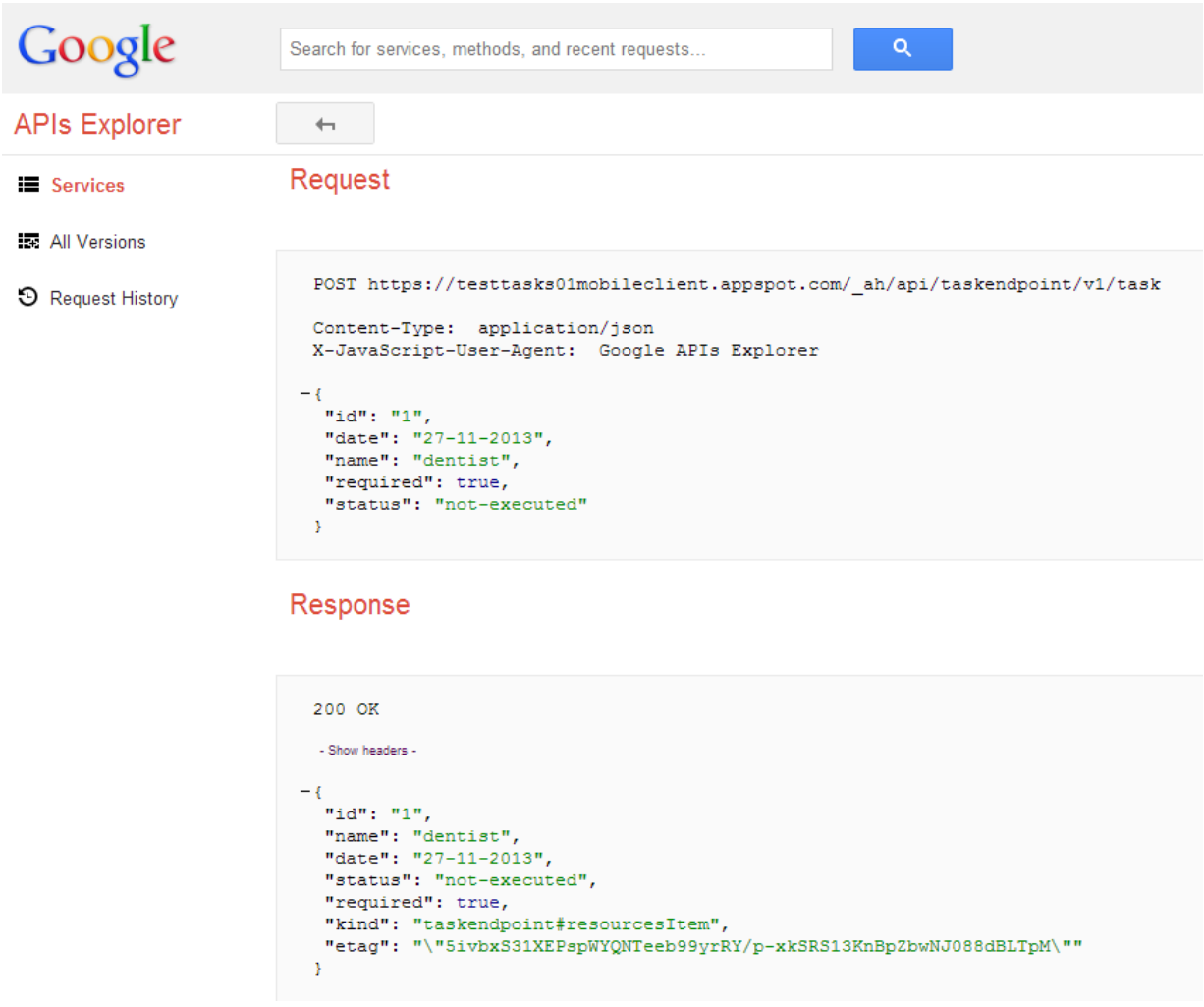


Table 7.1: creating a task

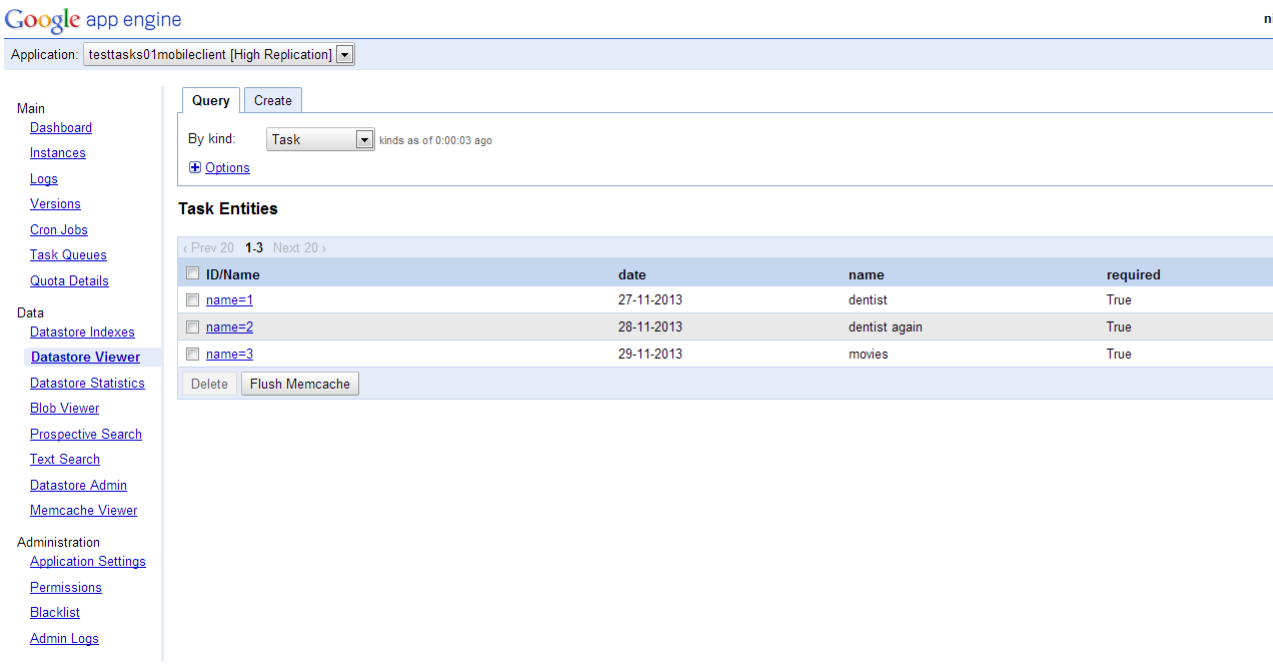


Table 7.2: list of tasks created

The client is registered to receive updates whenever a new task is added to the Task collection. Figure 7.3 shows the message informing that registration was successful.

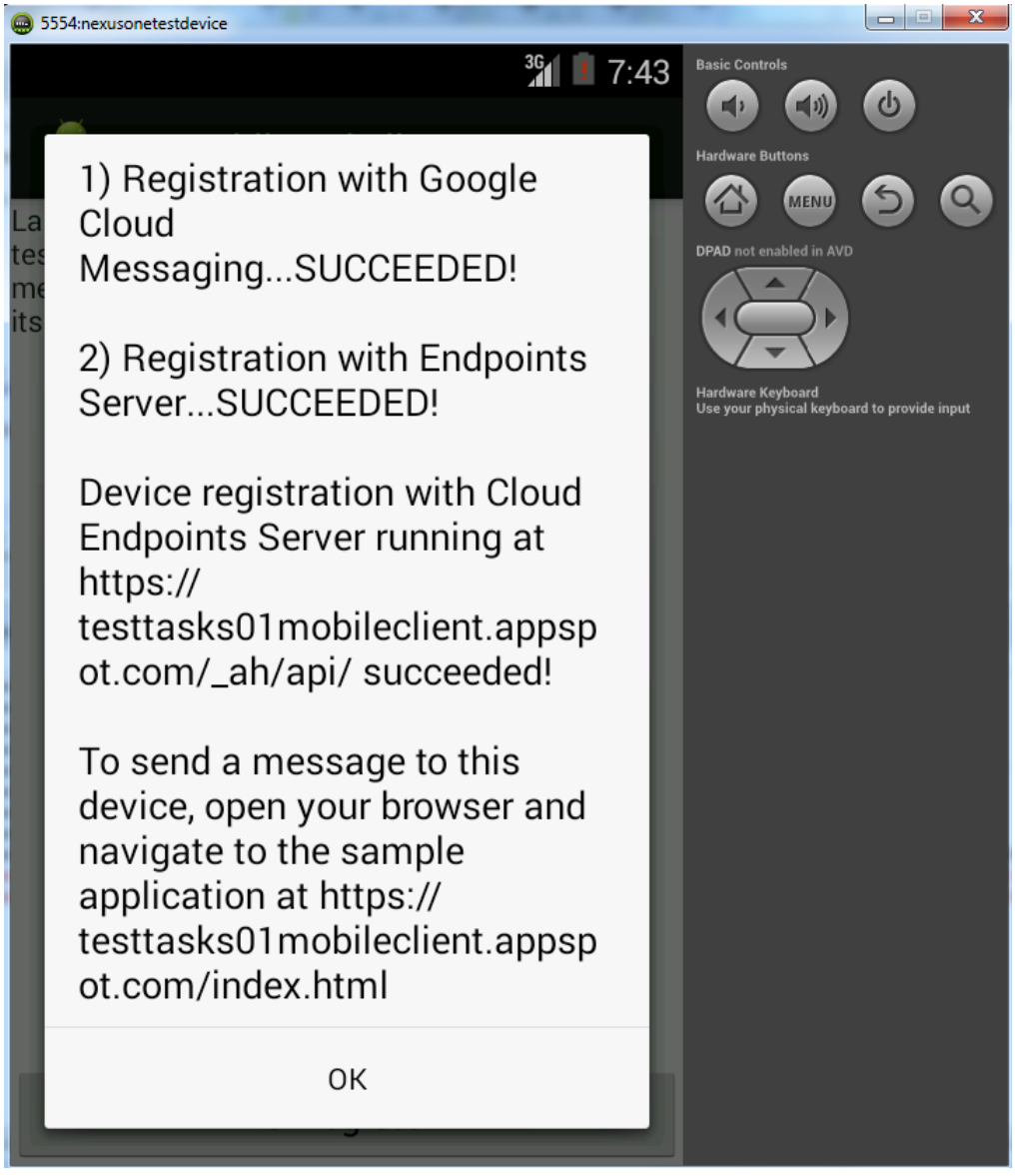


Table 7.3: registration succeeds

To test that communication happens properly a test message is sent manually. Figure 7.4 shows the message being sent at the server site, and figure 7.5 shows the client receiving the message.

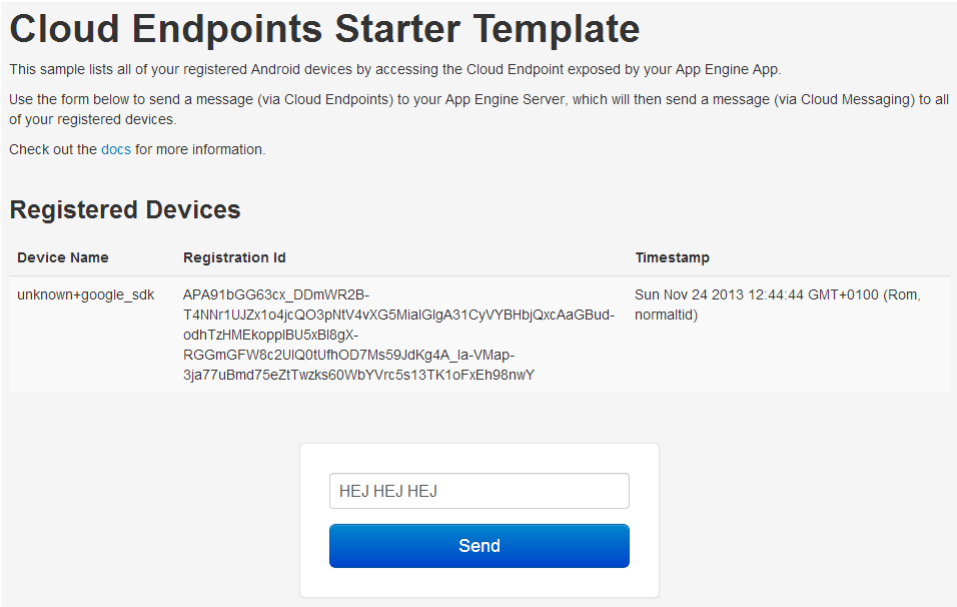


Table 7.4: message being sent

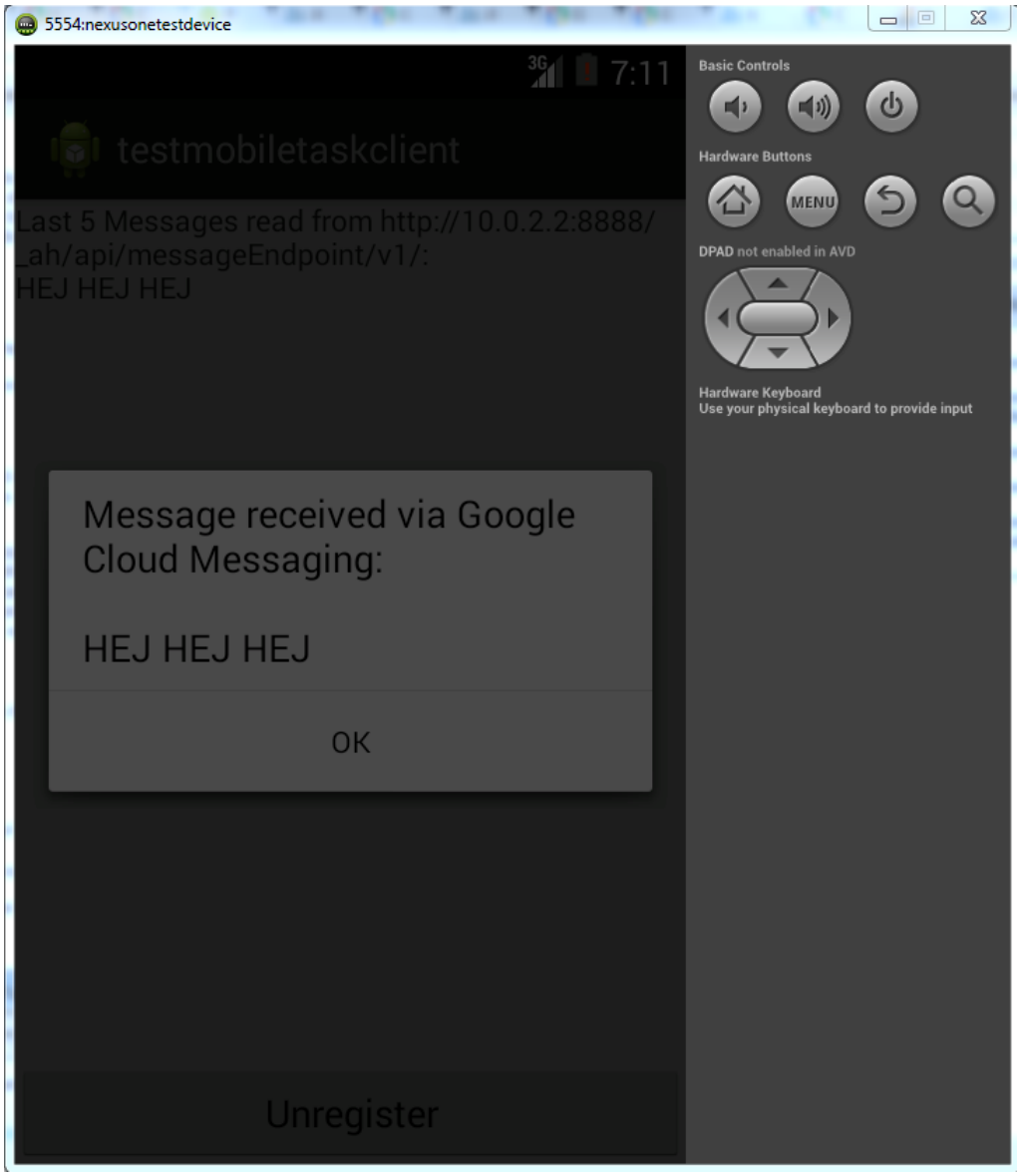


Table 7.5: message being received

When a new Task is being created at the server site, the client receives the update message as shown in figure 7.6.

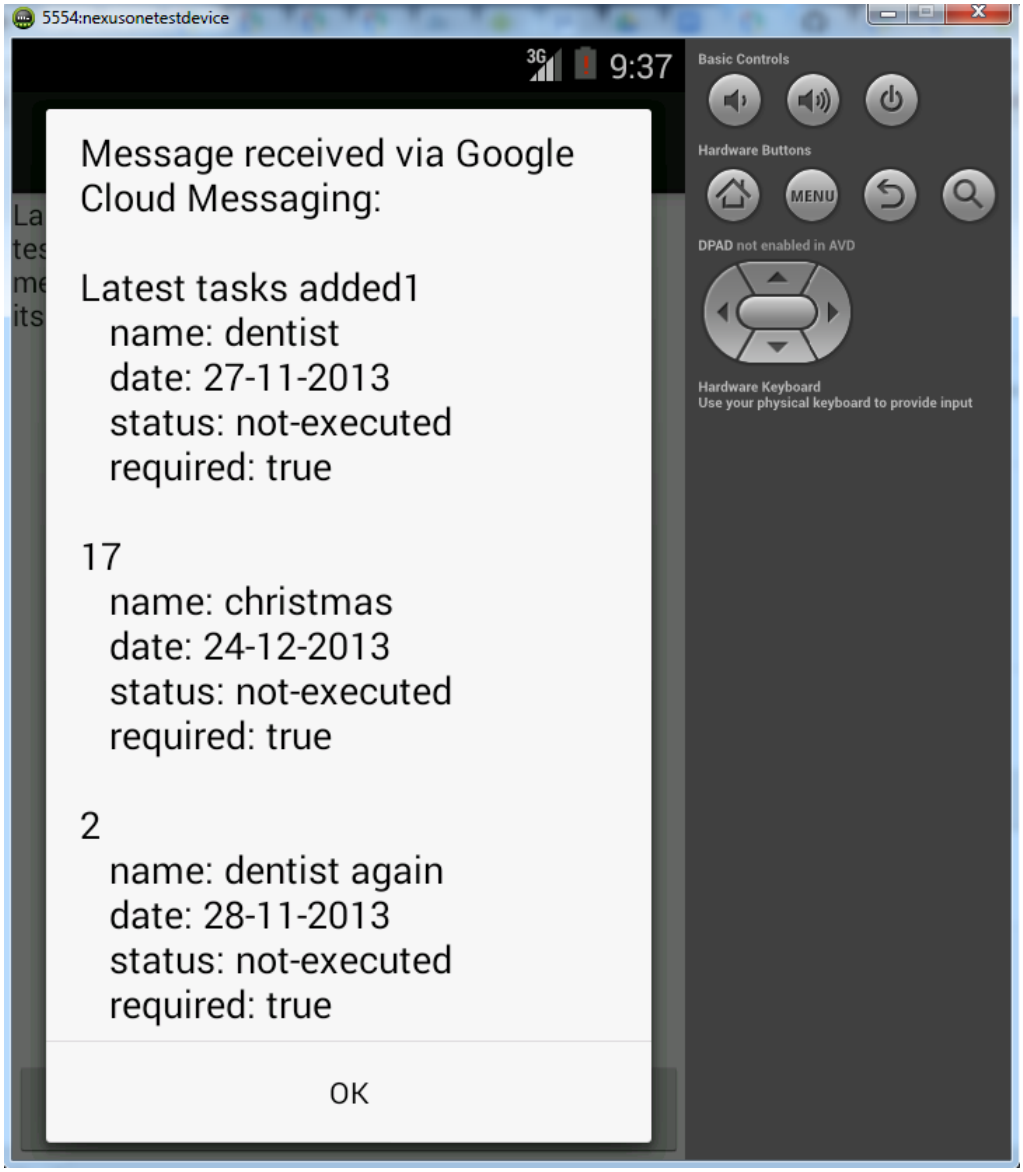


Table 7.6: task update being received

7.4 Reflection

mobile applications are characterized first and foremost by volatility and instability. Volatility refers to connections that are assumed to fail more often than not, while instability refers to high-level formats and standards that are not expected to remain relevant to a given solution for an extended period of time. The Google endpoint technology used in the solution attempts to overcome the instability problem by relying on the well established and stable Http interface for communication, and thus centralizing definitions of data formats and protocols.
The volatility issue is never directly assessed in the exercise, as the mobile device involved is not a physical device but a simulation. Common solutions to volatile connections involves the device continuously scanning its surroundings for suitable connection providers, and preparing to switch between providers frequently and easily.
The Android client designed exemplifies several characteristics of the Android architecture:

- The platform does not require compilation to a common code format, but supports embedding of other code languages, here among java. This could allow for a mobile application to be designed for several mobile platforms at once, without having to deal with a lot of platform-specific code.

- The system is entirely event driven. Activity objects are instantiated as an abstraction of running processes on the platform, and each activity listens for a number of relevant events, and reacts to them. No functional entry point (main method) exists for the application, only a designated main activity that is instantiated and started at program startup. In our specific solution the main activity only start a single activity; the register activity.

- The application switches between a number of states during execution. These include starting, running, pausing, restarting and stopping. The activities on the application reacts to these changes in state by listening to the relevant events.

In accordance with another problem common to mobile devices, namely limited memory, the Google App Engine manages all object persistence in the solution. Given the data centred design of the task manager application, proper persistence performance can be expected to be a relevant characteristic.

Conclusion

Contents

8.1 TCP

We have used this example to demonstrate and experiement with the TCP/IP suite. There are two transport protocols in the TCP/IP suite; UDP and TCP. UDP transfers text messages and TCP transfers byte streams. Unlike UDP, TCP is reliable and ordered. TCP can provide guarantees as to the ordering of its messages i.e., messages arrive in the order they were sent. It does so by giving every packet a number and then ordering the packets on arrival. TCP provides guarantees as to the delivery of the messages i.e, the message eventually arrive. It does so by using acknowledgements and retries (resend messages).

In contrast, the underlying network protocol (the IP protocol) offers only ?best-effort? semantics, there is no guarantee of delivery and packets can be lost, duplicated, delayed or delivered out of order. IP delegates the task of providing a reliable service to other layers, namely the TCP layer atop it. This nicely demonstrates the end-to-end principle. Application specific functionality should reside at the end hosts.

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.

Our simple example mimiced the way the TCP protocol achieves reliability. By sending acknowledgements our client-server is, in principle, capable of achieving reliability. It could be made to resend a packet in case of no response from the other end. Off course things like latenzy, troughput or duplicate packages come into play here.

The code in our solution is very much a 'proof-of-concept'. The code style 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 theory behind.

8.2 REST

8.3 Security

Sending messages in distributed systems poses a security risk. The enemy can intercept messages, man-in-the-middle attack, and the resend the message posing as the client, replays. To protect against those we use encryption and carefully designed protocols.

Instead of authenticating by password we use a challenge. A challenge is an encryted message which can only be decrypted with the right key. Inside the encrypted message is a ticket, another challenge, encrypted in the server key. By adding the clients identity to the ticket a server can authenticate the sender of the message. By adding a sequence of nonces the server can authenticate subsequent messages from the client.

An enemy amy intercept the initial message from the client to the authentication/token server but will be unable to decrypt the message, and ticket inside. The enemy may send this message to the server anyway but the server can verify the sender's identity by comparing the identity inside the ticket to the sender.

An enemy may intercept messages from the client to the server and replay a message posing as the client but the sequence of nonces in the message enables the server to authenticate the message. If the enemey does not know the right nonce tranformation this is also a futile effort.

Though the security protocol in our example does provide protection against man-in-the-middle attacks and protection against replays it still assumes a trusted source. It assumes that the secret keys used for communication between principles and the trusted source are distributed safely. Eliminating the need for a trusted source is desirable but unatainable.

Even when principles share private keys there is a possibility of a man-in-the-middle attack. If the initial message from the client to the token server is intercepted the server will not be able to detect the attacker as it will be its identity inside the ticket.

This example used private keys. Using public/private keys would present a stronger encryption but suffer the same short-commings.

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

8.4 Concurrency

8.5 Group Communication

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 like 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. JGroups 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.

Ordering of messages is important. Lamport phrased the term *happened-before* and provided the *Lamport timestamp* in which each process encrements its counter before sending a message. The receiver compares its own counter to that of the received message to determine the order. This is known as *partial ordering*.

Vector clocks does one better providing total ordering by having each process keep a vector of timestamps for all processes, a sort of Lamport timestamp for each process. This is known as total ordering.

8.6 Mobile Client

Security model

A.1 Security Model

A goal of security in distributed systems is to restrict access to information and resources to just those with authorization. Another is non-repudiation, securing that a process (or user) cannot deny participating in an activity.

This can be achieved by securing the processes and the channels used to interact and by protecting the objects they use against unauthorized access.

A basic model for the analysis of security risks include:

The enemey: we postulate an enemy who can send messages and receive messages to and from any two processes.

Threats include threats to processes and to communication channels:

Processes Even if protocols include an address in the messags sent there's no guarantee the sender is who he claims to be. A server cannot be sure the sender is authorized to access a certain resource if it cannot deermine the source of the message. Likewise, a client cannot be sure the result of a invokation is actually comming form the server and not a man-in-the-middle.

Channels The conceptual enemy can copy, alter or inject messages as they travel the network.

Security falls into 3 broad classes - Leakage: acquisition of info by unauthorized recipients, Tampering: Altering of info, Vandalism: interference without gain to the perpetrator.

Attacks can be classified into the way a channel is misused -

- Eavesdropping: Copying a message.
- Masquerading: portraying to be someone else
- Message tampering: Altering the contents of messages
- Replaying: re-sending a message
- Denial of service: Flooding the channel with messages denying access to a resource

Worst-Case Assumptions

Designing a secure system is a matter of constructing a list of threats and then designing the system against it. Note *the size and characteristics of such a list*

is weighed against time and quality requirements (read cost).

to create the list we have a set of worst-case assumptions and guidelines:

- interfaces are exposed:
- networks are insecure
- limit the lifetime and scope of secrets: (the longer the lifetime the grater the risk of compromization)
- code and algorithms are available to the enemy. (better expose the algorithm hoping to discover holes in it and then creating better ones.)
- attackers have acces to large resources. (computing power is cheap. Assume strong enemies.)
- Minimize trust base: the portionof a system responsible for security should be kept small.

A.2 The Enemy

The enemy can Read, Remember, Intercept, Interrupt, Modify, Fabricate (eavesdrop, masquerade, tamper, replay, denial of service) and guess/get secrets if she/he has sufcient time/power.

Who-did-what

Table B.1: who did what

Task manager servlet:	Mikkel
TCP Task Manager Server & XML Serialization:	Nikolai
REST Task Manager:	Mikkel
Secure Task Manager:	Nikolai
Collaborative Task Manager with jGroups: (indirect communication)	Nikolai
Concurrency Control:	Jonas and nikolai
Mobile Client for Task Manager:	Mikkel and nikolai