Information Systems Institute

Distributed Systems Group (DSG)
UE Verteilte Systeme WS 2015/16 (184.167)

Lab 2

Submission Deadline: 07.01.2016, 18:00

Contents

1 General Remarks

- We suggest reading (at least) the following materials before you start implementing:
  - Chapter 9 - Security from the book Distributed Systems: Principles and Paradigms (2nd edition)
  - Java Cryptography Architecture (JCA) Reference Guide[1]: Tutorial about the Java Cryptography Architecture (JCA).
  - Java RMI Tutorial[2]: A short introduction into RMI.
  - JGuru RMI Tutorial[3]: A more detailed tutorial about RMI.
- **Lab 2 is a group assignment** - make sure to distribute the work evenly within your group, and also make sure that **all group members** fully understand the entire solution in detail. Each group member should be capable of describing the code, also the parts which they did not implement!
- Collaboration across group boundaries (i.e., exchanging code with other groups) is **NOT** allowed. Discussions with colleagues (e.g., in the forum) are allowed but the code has to be written alone.
- Don't use any other 3rd party library except the ones we provided for you (e.g., Bouncy Castle).
- Be sure to check the Hints & Tricky Parts[4] section! For this assignment we have put lots of helpful code snippets there!
- Furthermore check the provided FAQ[5], where we discuss a lot of known issues and problems from the last years!

[1] http://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html
[2] http://docs.oracle.com/javase/tutorial/rmi/index.html
[3] http://www.dse.disco.unimib.it/ds/extra/rmiTutorial.pdf
[4] https://tuwel.tuwien.ac.at/mod/book/view.php?id=226077&chapterid=67
[5] https://tuwel.tuwien.ac.at/mod/book/view.php?id=226077&chapterid=62

2 Submission Guide

2.1 Submission

- You must upload your solution using TUWEL before the submission deadline: **07.01.2016, 18:00** - please note that the deadline is hard! You are responsible for submitting your solution in time. If you do not submit, you won't get any points!
- Upload your solution as a **ZIP file**. Please submit **only the provided template and your classes**, the `build.xml` file and a `readme.txt` (no compiled class files, no third-party libraries - except the libraries already provided in the template, no svn/git metadata, no hidden files etc.).
- The purpose of `readme.txt` is to reflect about your solution. It should contain a short summary of the status of your code so that a tutor can get the information right before the mandatory interview (see below) and can give you some tips for the next assignment.
- Your submission must compile and run in our lab environment. Use and complete the project template provided in TUWEL.
- Test your solution extensively in our lab environment. It'll be worth the time.
- Please make sure that your upload was successful (i.e., you should be able to download your solution - as the tutors will do during the interview).

2.2 Interviews

- After the submission deadline, there will be a mandatory interview (Abgabegespräch). You must register for a time slot for the interviews using TUWEL.
- You can do the interview only if you submitted your solution before the deadline!
- The interview will take place in the DSLab PC room[6]. During the interview, you will be asked about the solution that you uploaded (i.e., **changes after the deadline will not be taken into account!**). In the interview you need to explain your code, design and architecture in detail.
- Remember that you can do the interview only once!

Important: **Lab 2 consists of several (more or less independent) stages. If you want to get all points for this assignment, you will have to implement all of them. If you are satisfied with less, you can leave one or more of them unimplemented.**

[6] https://tuwel.tuwien.ac.at/mod/book/view.php?id=226077&chapterid=59

3 Installation/Registration of the Bouncy Castle Provider

For the second and third part we will use the Bouncy Castle[7] library as a provider for the Java Cryptography Extension (JCE) API, which is part of JCA. The Bouncy Castle provider (JDK 1.6 version) is already part of the provided template. Please stick to the provided version as this is the one used in our lab environment.

The provider is configured as part of your Java environment by adding an entry to the java.security properties file (found in `$JAVA_HOME/jre/lib/security/java.security`, where `$JAVA_HOME` is the location of your JDK distribution). You will find detailed instructions in the file, but basically it comes down to adding this line (but you may need to move all other providers one level of preference up):

```
security.provider.1 = org.bouncycastle.jce.provider.BouncyCastleProvider
```

Where you actually put the jar file is mostly up to you, but the best place to have it is in `$JAVA_HOME/jre/lib/ext` (which is the **jre folder within the JDK installation** - do not confuse with the JRE installation).

The installation of a custom provider is explained in the Java Cryptography Architecture (JCA) Reference Guide[8] in detail.

Note: If you get "`java.lang.SecurityException: Unsupported keysize or algorithm parameters`" or "`java.security.InvalidKeyException: Illegal key size`" exception while using the Bouncy Castle library, then check this hint[9].

4 Project Template

The `lib` directory contains the Bouncy Castle library. The `keys` directory contains all the keys required to test your implementation. The private keys used in the communication between client and chatserver are encrypted with following password for the respective participant:

```
alice.vienna.at: 12345

bill.de: 23456

chatserver: 12345
```

The secret key used in the client-to-client communication is not password protected at all.

Please check again that you modified several port constants in the `.properties` files according to your assigned port range from Lab 0.

[7] http://www.bouncycastle.org/

[8] http://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#ProviderInstalling

[9] https://tuwel.tuwien.ac.at/mod/book/view.php?id=226077&chapterid=67

5 Lab Description

Lab 2 is divided in the following parts:

- Stage 1 (16 points): In this stage you will now implement a small distributed naming service based on **RMI** that allows us to extend the mechanism for the registration and lookup of private user addresses from the first lab. For this, you will get acquainted with basic naming[10], name server lookup, and **RMI** communication.
- Stage 2 (12 points): This stage secures the TCP communication between the client and the chatserver by implementing a secure channel and mutual authentication using **public-key cryptography**. In our case, the secure channel will protect both parties against interception and fabrication of messages. Note that the common definition of a secure channel additionally implies a protection against modification. The client communication will remain vulnerable in this regard; however, Stage 3 will show how to address this issue concerning the communication between the clients.

  The first part of setting up our secure channel is to mutually authenticate each party (i.e., every party needs to prove its identity). In this assignment we will authenticate using the well-known challenge-response protocol. To subsequently ensure confidentiality of the messages after the authentication, we will use secret-key cryptography by means of session keys. Note that the session key is shared only between one specific client and the chatserver (and not with other clients). It is generated and exchanged during the authentication phase (i.e., if and only if the authentication is successful, both parties will know how to continue communication securely). This approach ensures that the user and the chatserver are having a confidential communication and that each party is who it claims to be.
- Stage 3 (7 points): This stage will show you how to verify that a message reaches its receiver unmodified using the JCA. For the sake of simplicity, we will add this feature solely to the otherwise unsecured communication between the clients. Here, communication is not protected against interception. By using a Message Authentication Code (MAC) and prepending it to each message, the receiver can check whether the message has been modified on the way through the channel.

5.1 Stage 1 - Naming service and RMI (16 points)

5.1.1 Description

In this stage you will learn:

- the basics of a simple distributed object technology (RMI).
- how to bind and lookup objects with a naming service.
- how to implement callbacks with RMI.

5.1.2 Overview

In this stage you will extend your chatserver of the first lab by adding a naming service. Figure 1 depicts the overall updated (simplified) architecture of our system.

Since in the first lab we used a relative simple in-memory mechanism for storing private addresses of users, we now want to update this approach by using a naming service, consisting of a network of nameservers.

Similar to DNS, our nameservers span a distributed namespace as shown in Figure 1. The namespace network is hierarchically divided into a collection of domains. There is a single top-level domain, hosted by the `Root Nameserver`. Using nameservers, each domain can be divided into smaller subdomains.

[10] The naming strategy that you will implement is presented in this assignment. For more information about naming in general, please consult Chapter 5 from the book Distributed Systems: Principles and Paradigms (2nd edition).

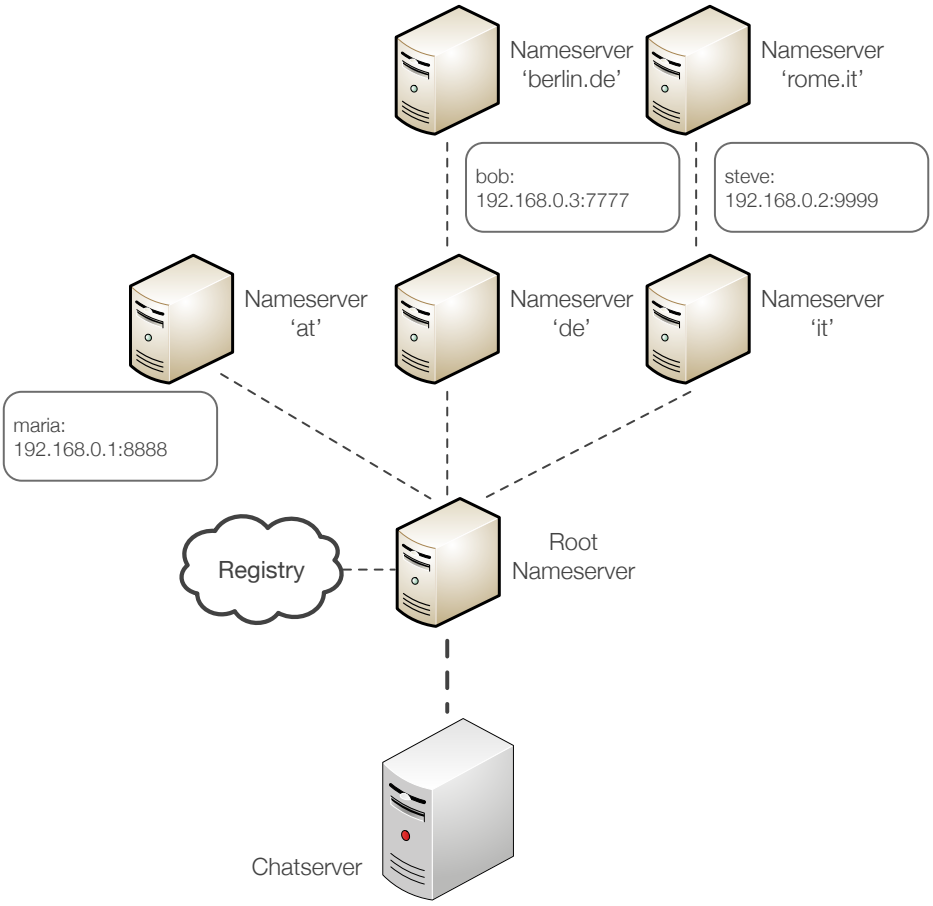

Figure 1: Naming Service: Overview

Each domain is hosted by exactly one `Nameserver`. Likewise, nameservers only host exactly one zone. A nameserver can communicate with the nameservers on the next lower level. In our system nameservers never need to contact their parent nameservers, i.e., name resolution is only done top-down.

In our scenario, a domain name is composed of a sequence of zones, where each zone consists of alphabetic characters (case insensitive). If a domain consists of multiple zones, these zones are separated by single dots (`'.'`). That is, in our example, `'berlin.de'` is a domain as are `'rome.it'` and `'at'`. Saying `'berlin'` is a zone in the namespace of `'de'` is the same like saying `'berlin'` is a subdomain of `'de'`.

Domain registrations are carried out in the following recursive style (1.-3.), depicted in Figure 2: First, the new nameserver contacts the root nameserver, attaching the name of the desired domain to its request. In Figure 2, the new nameserver tries to register for the domain `'berlin.de'`. (2.) The root nameserver forwards the request to the next lower level in charge. This procedure continues until name resolution is no longer possible. Accordingly, in the third step of the example, the request is sent to the nameserver hosting the `'de'`-domain (3.). At this point, there is only one subdomain left (`'berlin'`), which must be the requested zone. Therefore, the `'de'`-nameserver can store the new server as a child, responsible for the new zone `'berlin'`.

One well-known issue in such scenarios is the bootstrapping problem: To connect to a network, the respective participant needs to know the address of at least one other participant already connected to the network. In our case, to start the registration process, a new server first needs to find the root nameserver. For this, the root nameserver stores its remote object in a registry, where every new nameserver can perform a lookup (1.). Luckily, RMI already provides a registry service for you, so you

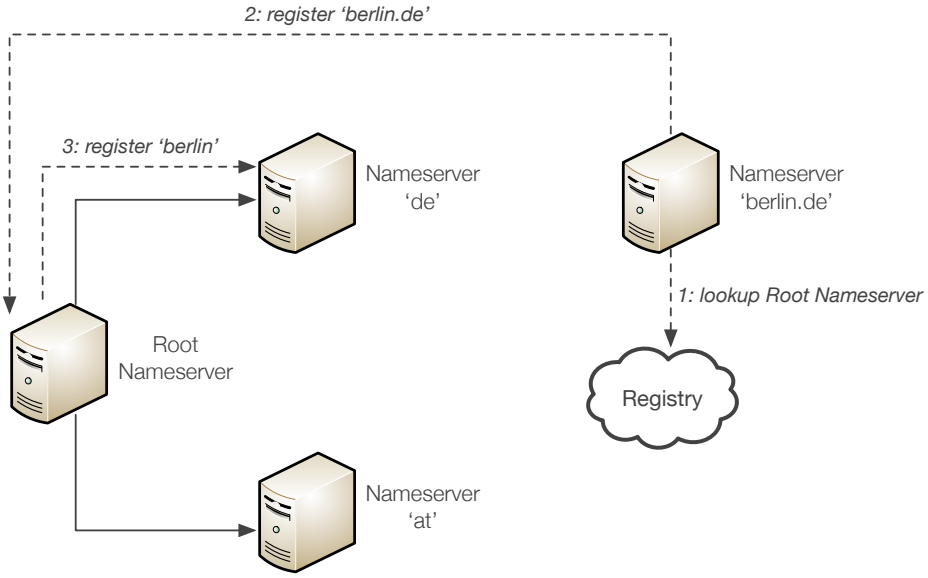

Figure 2: Naming Service: Registration

do not have to write the respective code on your own.

In contrast to the recursive algorithm used for nameserver and user address registrations, the lookup of user addresses will be handled in an iterative manner (see Figure 4 for an example): for further information see Section 5.1.4.

5.1.3 Nameserver

Arguments

The nameserver application reads the following parameters from the `ns-X.properties` config file, where `X` is the ID of the nameserver:

- `root_id`: the name the root nameserver is bound to or shall be bound to in case the nameserver you are currently starting is the root nameserver.
- `registry.host`: the host name or IP address where the registry is running.
- `registry.port`: the port where the RMI registry is listening for connections.
- `domain`: the domain that is managed by this nameserver. The root nameserver is the only nameserver that does not have this property. Therefore you can easily check if the nameserver you are currently starting is either an ordinary nameserver or a root nameserver.

You can assume that the parameters are valid and you do not have to verify them.

Implementation Details

At this point we will first describe how to use the already mentioned `java.rmi.registry.Registry` service. This service can be used to reduce coupling between the chatserver (looking up) and nameservers (binding): the real location of the nameserver object becomes transparent. In our case, the root nameserver starts a registry using the `LocateRegistry.createRegistry(int port)` method, which creates and exports a `Registry` instance on localhost. Use the provided property (named `registry.port`) to get the port the registry should accept requests on. Furthermore use the `registry.host` property to read the host the registry is bound to.

After obtaining a reference to the `Registry`, this service can be used to bind a RMI remote interface (using the `Registry.bind(String name, Remote obj)` method). Note that the root nameserver is the only server that needs to start a registry: the remote objects of all other nameservers will be received by using our own distributed, DNS-like namespace.

Since nameservers have to communicate with two different types of applications (namely the chatserver and other nameservers), it is recommended to create different remote interfaces for this purpose. Remote Interfaces are common Java Interfaces extending the `java.rmi.Remote` interface. Methods defined in such an interface may be invoked remotely. The template already contains the respective remote interfaces (`nameserver.INamerserver` and `nameserver.INameserverForChatserver`) including all the method signatures.

To register, a nameserver first has to read in the same properties file from the classpath as the root nameserver. This contains all information required to lookup the RMI registry of the root nameserver (again using one of the methods provided by the `LocateRegistry` class). Besides the name of the designated domain, nameservers should also provide appropriate callback objects when registering so that both the parent nameserver and the chatserver can communicate with the new nameserver.

So what are these callback objects all about? As already mentioned, the root nameserver is the only server that stores its remote object in a registry. But since nameservers need a possibility to communicate with the servers on the next lower domain level, there must be a different approach to get their remote objects: this approach is provided by callback objects. Such callback objects are similar to the object the root nameserver stores in its RMI registry. But in contrast to this object, callback objects are passed to the chatserver directly using special remote methods that accept remote references as a parameter (`INameserver.registerNameserver(..)`). This way, every nameserver that registers a new zone to a server can also store the server's callback object. Method calls on these objects will result in invocations on the remote server and can therefore be used for communication.

Keep in mind that the desired domain may already be in use by a different server or that a zone the request has to be routed to may not exist. In these cases, throw meaningful exceptions and pass them back to the actual requestor.

Synchronize the data structures you use to manage your subdomains. Even though RMI hides many concurrency issues from the developer, it cannot help you at this point. You may consult the Java Concurrency Tutorial[11] to solve this problem. However, the data does not need to be persisted after shutting down the nameserver.

You may assume that other nameservers always remain accessible, that is, you do not have to deal with zone failures.

Use the nameserver's console for logging any ongoing events, e.g. what nameservers get registered, what zones are requested by the chatserver etc. An exemplary output is shown in the following:

```
17:31:13 : Registering nameserver for zone 'berlin'
17:33:45 : Nameserver for 'berlin' requested by chatserver
```

Finally, the nameserver accepts the following interactive commands:

- `!nameservers`

  Prints out each known nameserver (zones) in alphabetical order, from the perspective of this nameserver.
  E.g.:

  ```
  # root nameserver
  >: !nameservers
  1. at
  2. de

  # de nameserver
  >: !nameservers
  1. berlin
  ```

[11] http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

- `!addresses`

  Prints out some information about each stored address, containing username and address (IP:port), arranged by the username in alphabetical order. E.g.:

  ```
  # nameserver 'at'
  >: !addresses
  1. maria 192.168.0.1:8888

  # nameserver 'berlin.de'
  >: !addresses
  1. bob 192.168.0.3:7777
  ```

- `!exit`

  Shutdown the nameserver. Do not forget to unexport its remote object using the static method `UnicastRemoteObject.unexportObject(Remote obj, boolean force)` and in the case of the root nameserver also unregister the remote object and close the registry by invoking the before mentioned static `unexportObject` method and registry reference as parameter. Otherwise the application may not stop.

5.1.4 Chatserver Updates

Arguments

There are three new configuration properties in the `chatserver.properties` file, which the chatserver will need in order to use the domain service.

- `root_id`: the name the root nameserver is bound to.
- `registry.host`: the host name or IP address where the registry is running.
- `registry.port`: the port where the RMI registry is listening for connections.

Implementation Details

At startup, the chatserver now also reads the previously mentioned additional properties to obtain the information where the RMI registry is located. Afterwards it is able to retrieve the remote reference of the root nameserver using the `root_id` property.

The classes and methods you will need for all these steps have already been explained above: `LocateRegistry.getRegistry(String host, int port)` and `Registry.lookup(String name)`.

After obtaining the root nameserver, the chatserver waits for commands sent by the client. Now to facilitate the new naming service, we have to update our approach from the first lab when handling the following commands sent by the client.

- **Register the private address of a user**: When a client wants to register the private address of a user, the chatserver now uses the distributed naming service as illustrated in Figure 3.

  First, when receiving the request, the chatserver forwards it to the root nameserver by invoking the `INameserverForChatserver.registerUser(String name, String address)`. The root nameserver registers the user address in a recursive style, by forwarding the request to the next lower level in charge. This procedure continues until name resolution is no longer possible. Considering the example in Figure 3, we want to register the address for `'bob.berlin.de'`. The root nameserver first forwards the request to the `'de'` nameserver, which further forwards the request to the nameserver responsible for the `'berlin.de'` domain, which must be the requested zone. Therefore, the `'berlin.de'`-nameserver can now store the user and address. In any error case (e.g., there exists no nameserver for a requested zone) use the provided exceptions, and provide meaningful messages for the user. Furthermore, you can assume that nameserver zones and usernames will use a different naming schema, which avoids overlapping names.

- **Lookup the private address of a user**: Next, when a client wants to lookup the private address of a user, the chatserver also uses the naming service as illustrated in Figure 4.

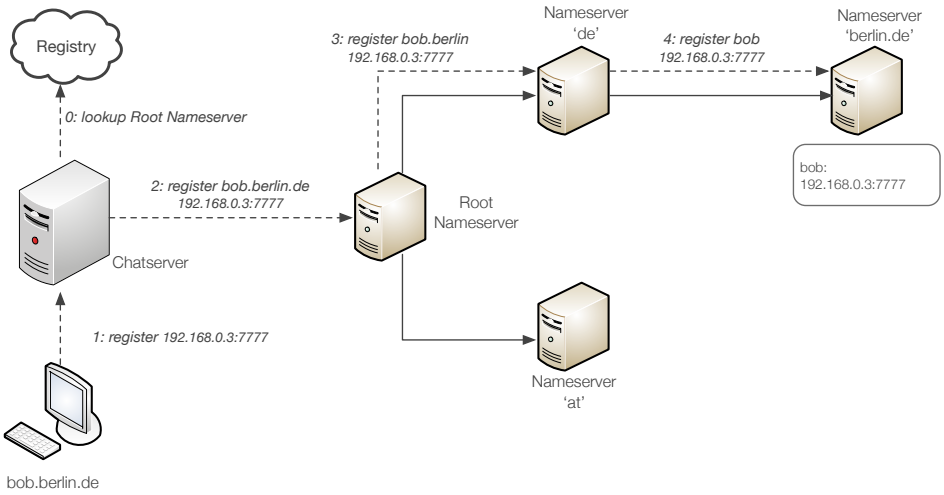

Figure 3: Naming Service: User-Address registration

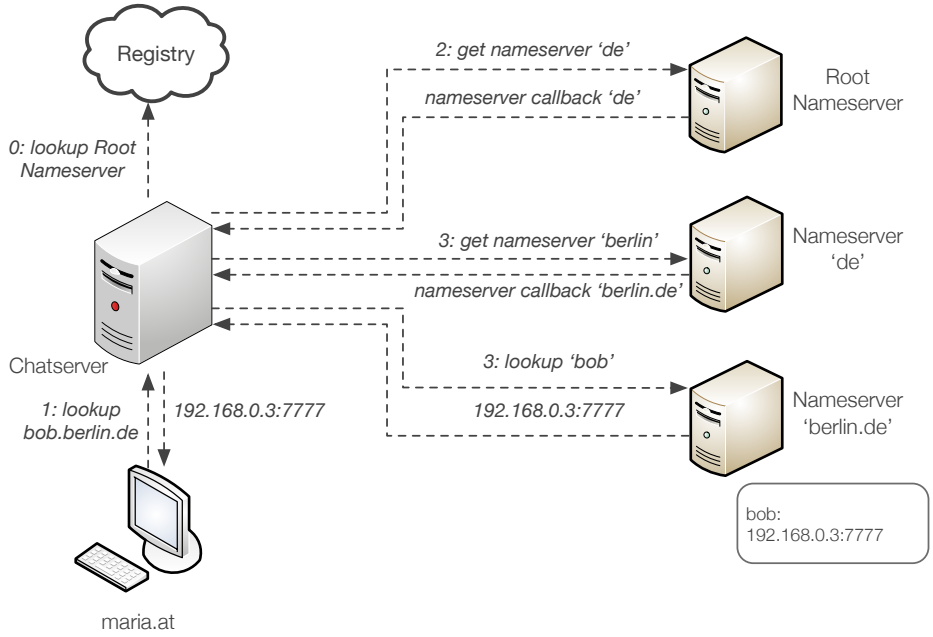

Figure 4: Naming Service: User-Address lookup

The processing of lookup requests thus follows an iterative approach: First, the chatserver gets the remote object of the `'de'`-domain from the root nameserver (using the `INameserverForChatserver.getNameserver(String zone)`). Then the chatserver asks the `'de'`-nameserver for the remote object of the `'berlin'`-zone. Since this nameserver is the last subdomain in the resolution, it must be the one that handles the respective user. Therefore, the chatserver can finally ask the nameserver for the private address of the user by invoking the `INameserverForChatserver.lookup(String username)`. If a domain or the user can not be found, provide a meaningful message for the user.

5.2 Stage 2 - Secure channel (12 points)

The first stage of establishing a secure channel (for the TCP communication between the client and the chatserver) is a mutual authentication. We will authenticate a client and the chatserver using public-key cryptography. This type of authentication is explained in the book *Distributed Systems: Principles and Paradigms (2nd edition)*, page 404, Figure 9-19. However, we also describe the principles in detail below.

Important:
Please note that in this assignment you are not allowed to use `javax.crypto.CipherInputStream` and `javax.crypto.CipherOutputStream`. Instead you should encrypt and decrypt all messages yourself.

In order to get rid of any special characters which are unsuitable for transmission as text you should **encode the already encrypted messages using Base64 before transmission** (see code snippets[12]). However, do not forget to Base64 decode the messages after receiving, otherwise the decryption of the messages will of course fail.

Note: This is very important in order to avoid encoding issues when converting `byte[]` to String!

It is recommended to avoid unnecessary conversions. In other words, first you should apply all modifications to the `byte[]` and create the resulting String in the end.

We highly recommend to hide the security aspects from the rest of your application as much as possible. Note that plain sockets or encrypted channels have many commonalities, e.g., they are both used to send and receive messages. Therefore, it may be a good idea to define a common interface that abstracts the details of the underlying implementation away. It is recommended to use the Decorator[13] pattern to add further functionalities step by step: For example, you could write a `TcpChannel` that implements your `Channel` interface and provides ways to send and receive objects over a socket. Next, you could write a `Base64Channel` class that also implements your `Channel` interface and encodes or decodes objects using Base64 before passing them to the underlying `Channel`. Following this approach may simplify your work in stage 2 and 3.

5.2.1 Authentication Algorithm

Note: You should implement the authentication algorithm (including the syntax of messages) exactly as described here! Failure to do so may result in losing points. The reason for this is that we will probably test your assignment using a modified client, which relies on the protocol being **exactly** as described. Do yourself and the tutors a favor and implement the protocol as described.

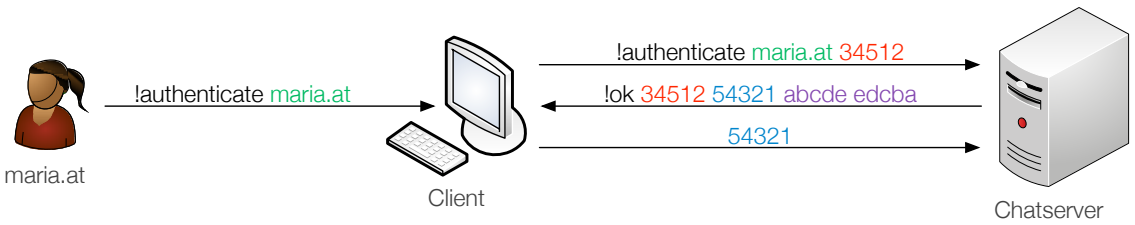

Figure 5: Secure Channel: Overview

The authentication algorithm consists of sending three messages (Figure 5):

[12] https://tuwel.tuwien.ac.at/mod/book/view.php?id=226077&chapterid=67
[13] http://en.wikipedia.org/wiki/Decorator_pattern

1st Message: The first message is a `!authenticate` request (analogous to the `!login` command from Lab 1). The syntax of the message is: `!authenticate <username> <client-challenge>`. This message is sent by the client and is encrypted using RSA initialized with the chatserver's public key.

- The client-challenge is a 32 byte random number, which the client generates freshly for each request (see code snippets[14] to learn how to generate a secure random number). Encode the challenge separately using Base64 before encrypting the overall message.
- Initialize the RSA cipher with the `"RSA/NONE/OAEPWithSHA256AndMGF1Padding"` algorithm.
- As stated above, do not forget to encode your overall ciphertext using Base64 before sending it to the chatserver.

2nd Message: The second message is sent by the chatserver and is encrypted using RSA initialized with the user's public key. Its syntax is: `!ok <client-challenge> <chatserver-challenge> <secret-key> <iv-parameter>`.

- The client-challenge is the challenge that client sent in the first message. This proves to the client that the chatserver successfully decrypted the first message (i.e., it proves the chatserver's identity).
- The chatserver-challenge is also a 32 byte random number generated freshly for each request by the chatserver.
- The last two arguments are our session key. The first part is a random 256 bit secret key and the second is a random 16 byte initialization vector (IV) parameter.
- **Every argument** has to be encoded using Base64 before encrypting the overall message!
- The ciphertext is sent Base64 encoded again.

When the client receives this message, it checks if the received `<client-challenge>` matches the sent one. In case the challenge does not match, the client prints an error message for the user and stops the handshake.

3rd Message: The third message is just the chatserver-challenge from the second message. This proves to the chatserver that the client successfully decrypted the second message (i.e., it further proves the client's identity). This message is the first message sent using AES encryption.

- Initialize the AES cipher using the `<secret-key>` and the `<iv-parameter>` from the second message. Details about these parameters are out of scope of this lab - you will learn about them in a Cryptography lecture.
- Use the `"AES/CTR/NoPadding"` algorithm for the AES cipher.
- Again, encrypt and encode the message before sending it.

When the chatserver receives this message, it checks if the received `<server-challenge>` matches the sent one. In case the challenge does not match, the chatserver prints an error message for the user and closes the connection.

The final end product of the authentication is an AES-encrypted secure channel between the client and the chatserver, which is used to encrypt all future communication. You **may not** send any message unencrypted (between a client and the chatserver) in this assignment. You **may not** send any messages besides the first two authentication messages (as described above) using the RSA encryption (i.e., the RSA encryption is strictly for the authentication part).

To generate a random 32 byte number to be used for challenges and random 16 bytes numbers to be used for IV parameter, you should use the `java.security.SecureRandom` class and its `nextBytes()` method as shown in the Hints & Tricky Parts section[14]. Base64 encoding them is required because this method could return bytes which are unsuited to be inserted in a text message. Same holds true for random secret keys in the AES algorithm (see the Hints & Tricky Parts section[14] on how to generate them).

[14] https://tuwel.tuwien.ac.at/mod/book/view.php?id=226077&chapterid=67

Important: Always encode your challenges, IV parameters and secret keys separately in your message using Base64. The message is then encrypted and gets Base64-encoded again before sending it.

5.2.2 Client Application Behavior

When a client application is started, it doesn't know which user will try to log in. Therefore the client application will need to process the `!authenticate` request before it is sent to the chatserver to find out which user is trying to log in and read his private key (used for decrypting the `!ok` message). Make sure a private key for this user does exist, otherwise, print an error message. The processing of the `!authenticate` command also includes appending the client-challenge. When implementing this stage, the `!login` command from the previous lab is obsolete.

There are two new configuration properties in the `client.properties` file, which the client application will need for the authentication phase:

- the `keys.dir` property denotes the directory where to look for the user's private key (named `<username>.pem`),
- and the `chatserver.key` property defines the file from where to read the public key of the chatserver.

5.2.3 Chatserver Application Behavior

The chatserver application should read its private key during the startup time. The user's public keys are read when the chatserver receives an authentication request. The user is said to be online when the authentication phase is successfully completed.

There are also two configuration properties in the `chatserver.properties` file, which the chatserver will need for the authentication phase:

- the `keys.dir` property denotes a directory where to look for user's public keys (named `<username>.pub.pem`),
- and the `key` property telling where to read the private key of the chatserver.

5.3 Stage 3 - Message Integrity (7 points)

Note: You should implement the syntax for private messages exactly as described here, for the same reasons as discussed above.

In this part of the assignment we will add an integrity check for TCP messages exchanged between the clients. However, the communication won't get encrypted - the implementation will only make sure that a third party cannot tamper with a message unnoticed. To do this, it relies on Message Authentication Codes (MACs).

Whenever a client sends a TCP request to another client and whenever a client responds, the application needs to compute a hash MAC (HMAC). To generate such a HMAC you should use SHA256 hashing (`"HmacSHA256"`) initialized with a secret key shared between the clients in the network. See the Hints & Tricky Parts[15] on how to read the shared secret key or create and initialize HMACs. After the HMAC is generated, it should be encoded using Base64. Prepend the original message with the HMAC (e.g., `<HMAC> !msg <message>`).

To verify the integrity of the message, the receiver generates a new HMAC of the received plaintext to compare it with the received one. In case of a mismatch, the behavior of the receiving client should be as follows: the respective message is printed to the standard output and the sending client is informed about the tampering (sending `<HMAC> !tampered <message>`), using the same channel the message was received. When the sending client receives this report or notices a message from a receiving client itself was changed, the client prints the incident to standard output and notifies the user accordingly.

In order to read the secret key the `client.properties` define a `hmac.key` property, which denotes from where to read the secret key.

[15] https://tuwel.tuwien.ac.at/mod/book/view.php?id=226077&chapterid=67

6 Lab Port Policy

See Lab Port Policy[16] for Lab 1.

7 Regular expressions

We provide some regular expressions you can use to verify that the messages you exchange between the three applications are well-formed. This is important because we will test your program against own code. We recommend to use Java assertions for this. The provided build file enables them automatically.

```
final String B64 = "a-zA-Z0-9/+";

// --- stage II ---
// Note that the verified messages still need to be encrypted (using RSA or AES,
    respectively) and encoded using Base64!!!

// authenticate request send from the client to the chatserver
String firstMessage = ...
assert firstMessage.matches("!authenticate␣[\\w\\.]+␣["+B64+"]{43}=") : "1st␣message";

// the chatserver's response to the client
String secondMessage = ...
assert secondMessage.matches("!ok␣["+B64+"]{43}=␣["+B64+"]{43}=␣["+B64+"]{43}=␣["+B64+"
    ]{22}==") : "2nd␣message";

// the last message send by the client
String thirdMessage = ...
assert thirdMessage.matches("["+B64+"]{43}=") : "3rd␣message";

// --- stage III ---
// Private messages being exchanged between clients before the final Base64 encoding
String hashedMessage = ...
assert hashedMessage.matches("["+B64+"]{43}=␣[\\s[^\\s]]+");
```

[16] https://tuwel.tuwien.ac.at/mod/book/view.php?id=226077&chapterid=66

8 Further Reading Suggestions

- APIs:
  - IO: IO Package API[17]
  - Concurrency: Thread API[18], Runnable API[19], ExecutorService API[20], Executors API[21]
  - Java TCP Sockets: ServerSocket API[22], Socket API[23]
  - Java Datagrams: DatagramSocket API[24], DatagramPacket API[25]
- Tutorials:
  - JavaInsel Sockets Tutorial - Section 21.6[26], 21.7[27]: German tutorial for using TCP sockets.
  - JavaInsel Datagrams Tutorial - Section 11.11[28]: German tutorial for using datagram sockets (note that this chapter is from edition 7 because it has been removed in newer versions).

[17] http://java.sun.com/javase/7/docs/api/index.html?java/io/package-summary.html
[18] http://java.sun.com/javase/7/docs/api/index.html?java/lang/Thread.html
[19] http://java.sun.com/javase/7/docs/api/index.html?java/lang/Runnable.html
[20] http://java.sun.com/javase/7/docs/api/index.html?java/util/concurrent/ExecutorService.html
[21] http://java.sun.com/javase/7/docs/api/index.html?java/util/concurrent/Executors.html
[22] http://java.sun.com/javase/7/docs/api/index.html?java/net/ServerSocket.html
[23] http://java.sun.com/javase/7/docs/api/index.html?java/net/Socket.html
[24] http://java.sun.com/javase/7/docs/api/index.html?java/net/DatagramSocket.html
[25] http://java.sun.com/javase/7/docs/api/index.html?java/net/DatagramPacket.html
[26] http://openbook.galileocomputing.de/javainsel9/javainsel_21_006.htm#mjcd64e398cec5737d9a288a4b4df04e2b
[27] http://openbook.galileocomputing.de/javainsel9/javainsel_21_007.htm#mj1ba27dc5fdf53f527163767f188e1d2e
[28] http://openbook.galileocomputing.de/java7/1507_11_011.html#dodtp497f87ed-dd23-48d4-80c7-7e11b3ec99d6