From Verteilte Systeme UE, 184.167

Lab3: Lab3

General Remarks

- We suggest reading 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**: Tutorial about the Java Cryptography Architecture (JCA).
- Lab 3 is a group assignment, but only one person per group needs to (and should) upload the solution. One group member can upload your solution as often as you like - any existing submission will be replaced by uploading a new one. If multiple (i.e., both) group members upload a solution, the submission with the latest timestamp will be graded!
- Note that we may perform routine checks using and anti-plagiarism software which detects identical or very similar solutions of different groups.
- Be sure to check the **Hints & Tricky Parts** section! For this assignment we have put lots of helpful code snippets there!

Updates

Submission

- You must upload your solution using the **Teaching Tool** before the submission deadline: **10.01.2013, 18:00 CET.** - the deadline is hard! You are responsible for submitting your solution in time. If you do not submit, you won't get any points!
- Do not overload our lab server by implementing on the server. The lab server is just for testing purposes. We cannot grade any solutions uploaded on the lab server, all solutions need to be uploaded to the teaching tool.
- Upload your solution as a **ZIP** file. Please submit the **sources** (including **all** properties files described below) of your solution and the **build.xml** file.
- Your submission must compile and run in our lab environment. Therefore, complete the provided ant file inside our **Project Template**.
- Test your solution extensively in our lab environment. It'll be worth the time.
- Before the submission deadline, you can upload your solution as often as you like. Note that any existing submission will be **replaced** by uploading a new one.
- 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).

Interviews

- After the submission deadline, there will be a mandatory interview (Abgabegespräch). You must register for a time slot to the interviews using the **Teaching Tool**.
- You can do the interview only if you submitted your solution before the deadline!
- The interview will take place in the **DSLab**. 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**!

Description

In this assignment you will learn:

- how to implement a simple distributed execution of a task
- cryptographically securing the communication between the server and clients
- asymmetric and symmetric cryptographic algorithms
- how to verify message integrity
- using the Java Cryptography Architecture (JCA) API
- advanced synchronization, avoiding deadlocks and starvation
- basics of decentralized peer-to-peer computing

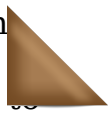
Overview

Lab 3 is divided into four independent parts. You should refactor and extend your solution for Lab 2 to accommodate the new requirements (please take a look at the **Project Template**).

Stage 1: Secure Channel (5 points): This stage secures the communication between the bidding clients and the auction server 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.

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 do this authentication by 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. 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 server and the clients can rely on confidential communication and that each party is who they claim to be.

Stage 2: Message Integrity (3 points): The second part of the assignment will



show you how to verify that a message reaches its receiver unmodified using the JCA. We will use this feature to ensure that the user bids are placed properly. By using a Message Authentication Code (MAC) and appending it to the output, the receiver can check whether the message has been modified on the way through the channel.

Stage 3: Advanced Synchronization Issues (6 points): Besides the basic purpose of synchronization, which is to protect shared resources from concurrent access and potential inconsistencies (mutual exclusion; see assignments 1 and 2), synchronization in a wider sense also has the purpose of ensuring correct and continuous functionality (in particular liveness and fairness) in timing-sensitive applications. In this part of assignment 3 we will take a closer look at two common synchronization issues: deadlock and starvation.

Stage 4: Server Outage / P2P (6 points): The last stage requires the implementation of a mechanism for handling server outage. Thereby you will have to develop mechanisms to successfully complete auctions even in case if the server is not online anymore.

Installation and Static Registration of the Bouncy Castle Provider

For the first and second part we will use the **Bouncy Castle** 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 our **Project 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`.

The installation of a custom provider is explained in the **Java Cryptography Architecture (JCA) Reference Guide** 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**.

Project Template

To help you start with the assignment, we created a **template project** which you should use. The project contains an ant build file and three directories: `keys` and `lib`. The `lib` directory contains the Bouncy Castle library and the log4j library, which are automatically added to the runtime classpath.

In the `src` directory you will again find configuration files for the RMI registry (you can use the files from assignment 2). If you need additional configurations, feel free to add .properties to the `src` directory. The `build.xml` file has been extended with additional properties: `server.key` and `server.key.pub` are the paths to the private key and public key files of the auction server, respectively.

The property `clients.key.dir` is the path to the directory where you store the private and public keys of the clients. The naming scheme for the clients' keypair is `<username>.pem` (private key) and `<username>.pub.pem` (public key). Additionally, each user needs a `<username>.key` file which is a randomly generated character sequence used for generating MACs to ensure message integrity (see details further below). The template contains example keys for user *alice*, but you will need to add additional *.pem and *.key files for all clients in your system. **Note:** The user login (bidding clients) should only be successful if a keypair can be found - otherwise the login should be rejected with an error message.

The `keys` directory in the template contains keys for the auction server and the bidding user *alice*. The password of the private key of the auction server is '23456'. The password of the alice's private key is '12345'. Please use this password for any other user keys you generate (for testing purposes).

Stage 1 - Secure channel (5 points)

The first stage is the establishment of a secure channel by using mutual authentication. We will authenticate the auction server and the bidding clients using public-key cryptopgraphy. This type of authentication is explained in the book *Distributed Systems: Principles and Paradigms (2nd edition)*, page 404, Figure 9-19. However, you should also be able to implement the authentication by reading the description below.

Please note that in this assignment you are not allowed to use `javax.crypto.CipherInputStream` and `javax.crypto.CipherOutpurStream`. Instead you should encrypt and decrypt all messages yourself: Use the `Socket.getInputStream()` and `Socket.getOutputStream()` methods together and decorate these streams with a `java.io.BufferedReader` or `java.io.PrintWriter`, respectively.

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

We highly recommend to hide the security aspects from the rest of your

application as much as possible. Note that plain Sockets or encrypted chan 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 th abstracts the details of the underlying implementation away. You should use the **Decorator 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 `String`s over a Socket. Next, you could write a `Base64Channel` class that also implements your `Channel` interface and encodes or decodes `String`s using Base64 before passing them to the underlying `TCPChannel`. Following this approach may simplify your work in this lab.

Authentication Algorithm

Note: You should implement the authentication algorithm (including the syntax of the messages) exactly as described here! Failure to do so may result in losing points. The reason for this is that we will test your solution 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.

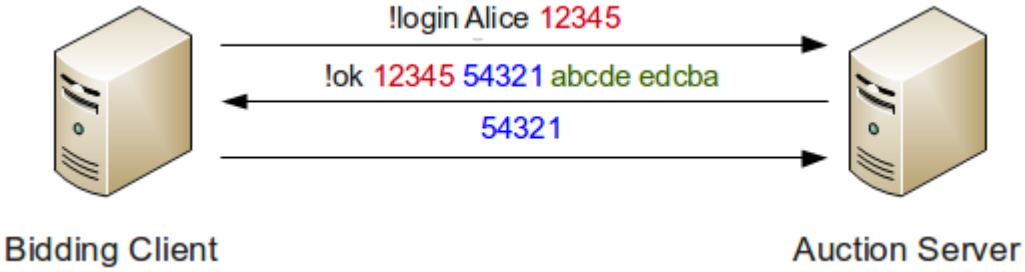


Figure 1: Protocol for Authentication Algorithm

The authentication algorithm consists of sending three messages:

1st message: The handshake protocol starts with the *!login* message. Extend the original `!login` command from assignment 1 to the new syntax: `!login <username> <client-challenge>`. This message is sent by the client and is encrypted using RSA initialized with the public key of the ~~auction client~~ auction server. (**updated** 02.12.2012)

- The `client-challenge` is a 32 byte random number, which the client generates freshly for the authentication. Have a look at the **code snippets** to see 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 over the wire.

2nd message: The second message is returned by the auction server and is encrypted using RSA initialized with the public key of the client. The public key file needs to exist in the key directory specified in the `build.xml` file, otherwise the login is denied). The syntax is: `!ok <client-challenge> <server-challenge>`

`<secret-key> <iv-parameter>`.

- The `client-challenge` is the challenge that has been created and sent by the client. This proves to the client that the auction server has successfully decrypted the first message. As such, it proves the identity of the auction server.
- The `server-challenge` is also a freshly generated 32 byte random number.
- 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-param`).
- **Every argument** has to be encoded using Base64 before encrypting the overall message!
- The ciphertext is sent Base64 encoded again.

3rd message: The third message is just the `server-challenge` from the second message. This proves to the server that the client has successfully decrypted the second message. This message is the first one that is 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.
- Do not forget to encode the challenge using Base64 before encrypting the overall message.
- Again, encrypt the message using AES and afterwards encode the message using Base64 before sending it.

The final result of the authentication is an AES-encrypted secure channel between the clients and the auction server, which is used to encrypt future communication.

- After successful handshake, all `!create` and `!bid` commands (see assignment 1) sent between the bidding clients and the auction server **must be encrypted**.
- You may not send any messages besides the first two authentication messages (as described above) using the RSA encryption. RSA encryption is strictly used for the authentication part only.

To generate a random 32 byte numbers 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 **code snippets** section. 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 **code snippets** section on how to generate them). That is: Always encode your challenges, IV parameters and secret keys separately in your message using Base64. This message is then encrypted and gets Base64-encoded again before sending it.

Bidding Client Behaviour

When a user logs in on a bidding client, the secure channel has to be initiali
(As in assignment 1, the `!list` command should be available without logging
Therefore it has to read its own private key and the public key of the server (see
description of properties in `build.xml` further above). You have to make sure that
the keys exist and the user has to enter the correct password for the private
key. Print an error message otherwise. Afterwards the first authentication
message can be sent to the server.

Auction Server Behaviour

The auction server can read the keys at startup time. Every time a new TCP
request is received, it uses its own private key to decrypt the message and
continues with the second authentication message.

Stage 2 - Message Integrity (3 points)

In this part of the assignment we will add an integrity check for messages sent
from the auction server to the bidding clients. Note that integrity checks have
nothing to do with encryption - the implementation will only make sure that a
third party cannot tamper with a message unnoticed. You will use Message
Authentication Codes (MACs) to realize this.

Whenever the auction server sends a "crucial" information to a client (this is in
particular relevant for the output of the `!list` command), the application needs
to append a HMAC (a hash MAC). To generate such a HMAC you should use
SHA256 hashing (`"HmacSHA256"`) initialized with a secret key shared between the
auction server and the client. So, depending on the client that receives some
data, a different key for generating the HMAC is used. See the **code snippets**
on how to read in the shared secret key or create and initialize HMACs. Simply
append the generated HMAC to the message that is sent between the server
and the client.

To verify the integrity of the message, the client generates a new HMAC of the
received plaintext to compare it with the received one. In case of a mismatch,
the behaviour of a client should be as follows: The respective message is printed
to the standard output and the output for the task is requested for a second
time automatically. If the second attempt fails, too, print out the respective
message again, but do not try to request the output again.

HMAC Key Lookup

When the server sends output to the client, the secret key of the client is used
to generate a HMAC. To that end, use the property in the `build.xml` file which
points to the keys directory (see template description further above). The keys
should be named `<username>.key`, so the name of the user identifies the file to be
used. On the client side, load the key file after the client program is started and
use the key later for verification. On the auction server side, you can either load
the key after the successful login of a client, or when the first output is
returned.

Stage 3: Advanced Synchronization Issues (6 points)

Besides the basic purpose of synchronization, which is to protect shared resources from concurrent access and potential inconsistencies (mutual exclusion; see assignments 1 and 2), synchronization in a wider sense also has the purpose of ensuring correct and continuous functionality (in particular liveness and fairness) in timing-sensitive applications. In this part of assignment 3 we will take a closer look at two common synchronization issues: deadlock and starvation.

To that end, the auction bidding capability implemented in assignment 1 will be slightly extended. We assume that users are not bidding entirely independently, but that a group of users has a shared budget. Since it needs to be ensured that the shared budget is spent on items of common interest, **all bids placed by any user need to be confirmed by 2 additional users** to become effective. You do not need to implement the mapping between groups and users - for simplification, you can assume that the system contains only one user group which all users belong to. Furthermore, we assume that users may post group bids in multiple active auctions, but **the number of active auctions with group bids is less than or equal to the number of group members** (e.g., Figure 2 contains 4 users and 1 auction with group bid; $1 \leq 4$). Figure 2 illustrates the workflow of bid confirmations (the examples assume that the necessary auctions have previously been created). Note that the communication arrows and commands in fact take place between the bidding clients and the auction server, which are not included in the figure.

In order to disinguish regular bids from group bids, create a new command "`!groupBid`", which takes the same parameters as "`!bid`". In Figure 2, after user Alice places a bid using the command "`!groupBid 1 50`", Carl confirms the bid using the command "`!confirm 1 50 Alice`". After the command has reached the server, Carl's client *blocks* until a second user (Dave) also confirms the same bid. This means that the two confirming clients perform a kind of "handshake" in order to confirm bids (note that the `groupBid` command immediately returns and does *not* block the client). After the two "`!confirm`" requests have arrived at the server, both users receive a response string "`!confirmed`" from the server. A group bid that has not been confirmed yet is called a *tentative bid*, and only after successfuly confirmation the bid is added to the auction. To indicate errors (e.g., the bid is lower than the currently highest bid of the auction), use the response string "`!rejected`" (instead of "`!confirmed`"). Additionally, you may append an error string after the "`!rejected`" command, for instance "`!rejected Please use a higher bid price.`".

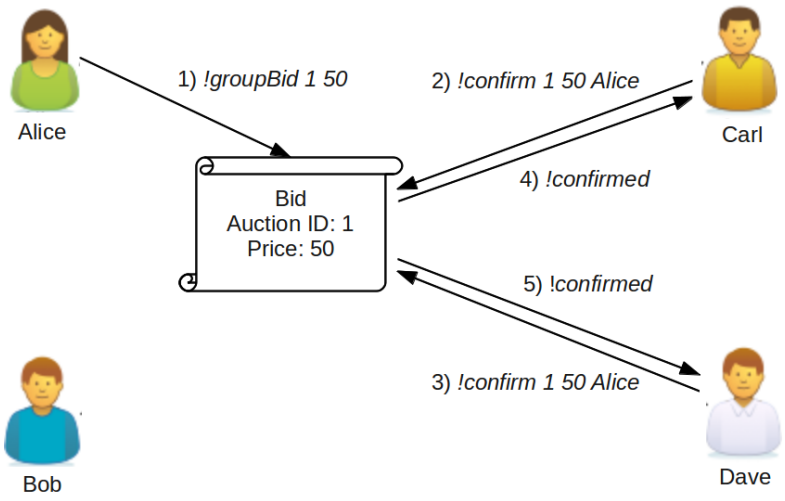


Figure 2: Bid Confirmation (Regular Case)

Preventing Deadlocks

Because the client is blocking during a `!confirm` operation, and given the requirement that each user can only be logged in on one client, each user can only perform *one confirmation* at a time. In most cases this will work fine, but there are situations in which the system might run into problems. Consider the configuration in Figure 3: in steps 1) to 4) the users Alice, Bob, Dave, and Carl create a bid. Next, in steps 5)-8) each of the bids is confirmed by another user. If the last confirmation by Dave were accepted (step 8)), the system would end up in a *deadlock*, which means that no participant is able to perform any more actions, and there is no (regular) way to escape from this situation. This, of course, should be avoided at any cost, and therefore the system returns a "`!rejected`" message in step 9). Note that this configuration is just one example, although other deadlock situations are also possible in this scenario.

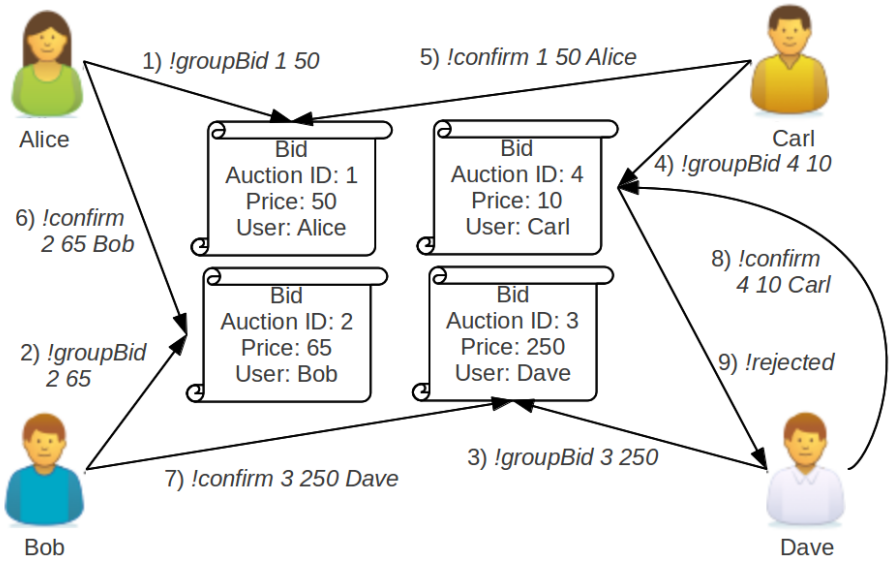


Figure 3: Bid Confirmation (Deadlock)

The situation outlined in Figure 3 has similarities with the "*Dining*

Philosophers", a popular problem that is often used to illustrate synchroniza issues. Before you solve the synchronization issue for your auction system, advise you to study the Dining Philosophers and related problems (e.g., Sleeping Barber problem, Cigarette Smokers problem, etc.) in detail (e.g., using the material from the Distributed Systems lecture; there are also a lot of useful resources on the Web).

For instance, two possible solutions to the deadlock problem in Figure 3 would be:

- Using a centralized decision component which avoids deadlocks and ensures continuous operation of your system. The decision component could intercept every user request, and block/deny the request if it detects that it would result in a deadlock.
- Using a timout, which terminates/rejects a waiting "`!confirm`" command after a certain amount of time. This would ensure that the system can eventually escape from deadlocks.

Other solutions are also conceivable - you should weigh the advantages against the disadvantages and argue which tradeoff is achieved by your implementation. For testing purposes, make sure that your system never blocks longer than **20 seconds**.

Preventing Starvation

In addition to the handling of deadlocks, you should consider the problem of *starvation*. A process is said to be "starving" if it repeatedly attempts to perform an operation and its requests are continuously denied. Consider the requirement that the number of active auctions with group bids (denoted "#(groupBid auctions)") has to be less than or equal to the number of group members (denoted "#(members)"). Enforcing this requirement means that if a "`!groupBid`" request is made in a situation where #(groupBid auctions) = #(members), then the request has to be denied.

Figure 4 contains a sequence of auctions with group bids. For illustration purposes, we assume that there are only three users (Alice, Bob, Carl) and that Dave (included in Figures 2, 3) is not part of the system. Hence, it needs to be ensured that $\#(groupBid\ auctions) \leq 3$. In the example, Carl attempts to create a group bid for Auction 4, which is denied because $\#(groupBid\ auctions) = 3$ at this time. Carl again attempts to place a group bid for auction 6, but again the prerequisites are not satisfied and his request is denied. The same happens for Auction 8, and it easy to see that this sequence of denied requests can be continued infinitely. Although this example is made up and may seem slightly unrealistic, process starvation is in fact a key issue for the implementation of distributed systems.

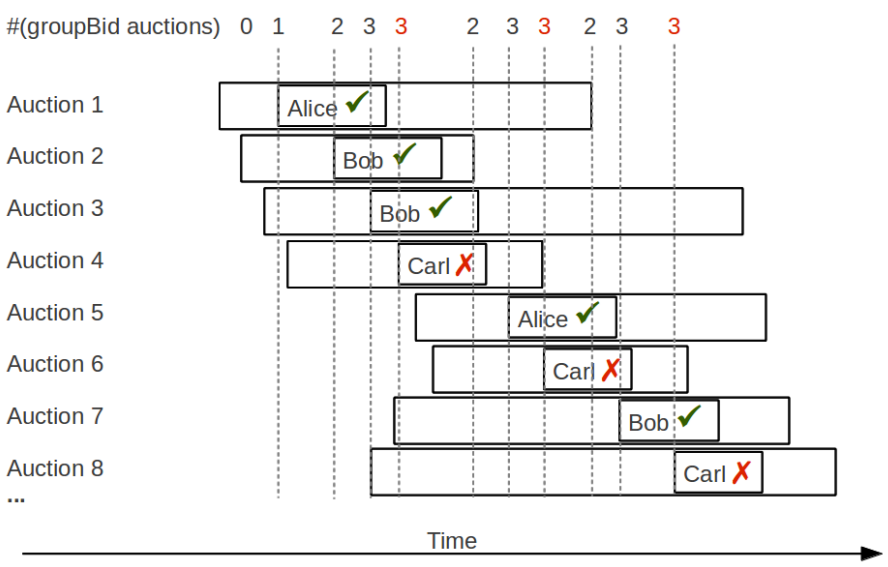


Figure 4: Group Bidding Example with 3 Users Alice, Bob, Carl (Carl is "starving")

The aim of this assignment is to develop a strategy for avoiding starvation and ensuring "fairness" in our auction system. There are different ways to achieve this, and you should again weigh the advantages against the disadvantages of your approach. First, you should discuss within your group and define the "level of fairness" you want to guarantee, and then think about the consequences for your system. Possible fairness goals could be "*No user should be denied a request in more than 3 consecutive attempts*" or "*All users should be permitted (approximately) the same number of request attempts (over a period of X minutes)*" or anything similar. You can also define multiple fairness goals. Then, think about how you can technically achieve the goals. Next, implement your solution and empirically evaluate which consequences your solution has for all system participants. Are there any corner cases in which your strategy behaves suboptimally? You do not need to prepare a written report, but we expect a well-grounded and fruitful discussion during the submission interviews.

Server Outage (6 points)

In the last part of the assignment a solution for handling server outage has to be implemented. Hereby, we assume that at some point in time server becomes unavailable. But even in such a case clients should be able to continue bidding and to conclude their auctions properly. Therefore, you should extend your solution with the functionality of providing bid timestamps that are generated by random clients. Let us assume that the server becomes unavailable. You can check server availability by catching proper exceptions when you connect to and exchange messages with the server. Figure 5 illustrates the handling of server outage.

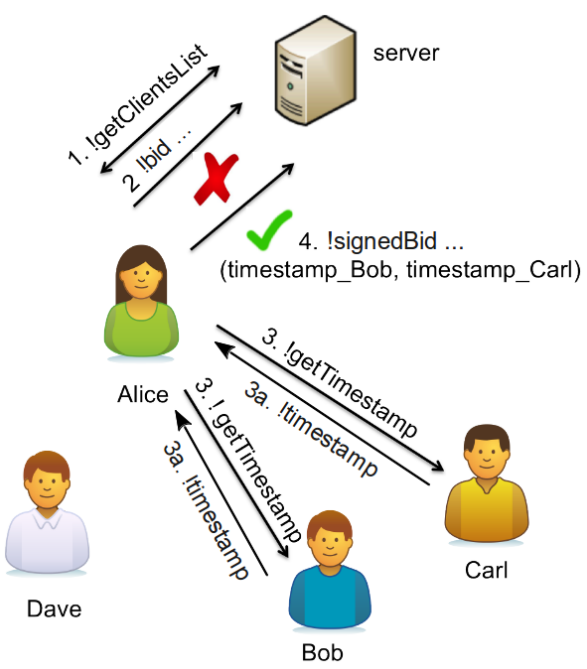


Figure 5: Handling Server Outage with Timestamps

As shown in Figure 5 with the step 1, right after login and while the server is still online you should obtain the list of all active clients (including the currently logged in user) by sending a `!getClientList` message. Moreover, implement the interactive command `!getClientList` to let the user see the list of clients and users (see listing below). You may also display "idle" clients (client started, but no logged in user), but make sure that you send your timestamp requests only to those clients where a user is currently logged in!

```
Alice> !getClientList
Active Clients:
127.0.0.1:10011 - Alice
127.0.0.1:10012 - Bob
127.0.0.1:10013 - Dave
127.0.0.1:10014 - Carl
```

Note that the server already knows the **UDP** notification ports from assignment 1. You can use the same port number to create a **TCP** socket and listen for the `!getTimestamp` commands (see further below).

Now you can check the server availability by catching proper exceptions. In case that the server gets unavailable (step 2, Figure 5) you have to randomly select two clients from the list. Afterwards, obtain the current timestamp from these clients (measure the time in milliseconds using `System.currentTimeMillis()`). This step is shown in Figure 5 with step 3 where Alice receives two timestamps from Bob and Carl. The format of the request message should be `!getTimestamp <auctionID> <price>` . The response of the timestamp should contain the auctionID, the bid price, the timestamp, and the entire block of information should be signed. In contrast to the symmetric HMACs used earlier in this assignment, this time we will again use the clients' keypair (`<username>.pem` and `<username>.pub.pem`) for asymmetric cryptography. Utilize the `java.security.Signature` class to obtain the signature hash over the string "`!timestamp <auctionID> <price>`

`<timestamp>`" (of course, replace the placeholders with real values), and then return the complete message with the following form: "`!timestamp <auctionID> <price> <timestamp> <signature>`". Afterwards, the client which requested the timestamps can extract the two received signatures from the response messages, in order to use them later on as evidence for the bidding time. (Of course, in practice the situation would be more difficult, e.g., because of trust issues - however, this is out of scope for this assignment.)

Once the timestamps are obtained, check in regular intervals whether the server becomes available. If the server is back online, send a bid with the two signed timestamps using the command `!signedBid` (step 4 in Figure 5). Thereby you should submit the bid, auctionID, price, signed timestamp1 (e.g. timestamp obtained from Bob) and signed timestamp2 (e.g., timestamp obtained from Carl). The message which is finally sent to the server should look like the following: `!signedBid 17 90 Bob:<timestamp1>:<signature1> Carl:<timestamp2>:<signature2>`

Note that the code/command snippets printed here represent the exchanged messages and *not* the user input. The process of obtaining signed timestamps is completely transparent to the user and in fact no additional user interaction is required after the user has placed the "`!bid ...`" command (although you can of course inform the user via some log messages).

In the example above, 17 represents the auction ID, 90 the bid price, <timestamp1> and <timestamp2> represent the timestamps retrieved from the two clients (i.e., Bob and Carl as shown in Figure 4). The signatures <signature1> and <signature2> assert the integrity of the timestamps <timestamp1> and <timestamp2> and have to be verified by the server. Reconstruct the original strings that were (supposedly) signed by the two clients and verify the strings against the provided signatures using the `java.security.Signature` class (using the public keys of the respective users Bob and Carl).

If any two timestamps differ (which will be the regular case), then the server has to calculate the arithmetic mean of the two values. The integrity of the timestamps has to be verified using the public keys of the clients.

Optional Part (5 extra points)

In the optional part of the assignment you should implement communication between clients using the JXTA P2P framework. You have to implement a service that will run in the Java binding, then open a listening input pipe, and wait until *getTimeStampMessage* has been received from the pipe. The service should also provide a method that can be invoked to find a random peer's input pipe and then send it a single message. You should use JXTA mechanism to create peers, to discover other peers and peer groups and to send messages. The details on for the implementation of JXTA P2P are given for example in the **on java JXTA tutorial**.

Lab Port Policy

As in the first two labs, please stick to the following policy:

Since it is not possible to open `ServerSocket` on ports where other services are already listening for requests, we have to make sure each students uses its own port range.
So if you are testing your solution in the lab environment (i.e., on the lab server) you have to obey the following rule: you may only use ports between **10.000 + dslabXXX * 10** and **10.000 + (dslabXXX + 1) * 10 - 1**. So if your account is dslab250 you may use the ports between 12500 and 12509 inclusive. Note that you can use the same port number for TCP and UDP services (e.g., it is possible to use TCP port 12500 and UDP port 12500 at the same time).

Regular expressions

We provide some regular expressions you can use to **verifiy that the messages you exchange between the auction server and the bidding clients are well-formed**. This is important because we will test your program against own code.

```
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!!!


// login request send from the client to the auction server
String firstMessage = ...
assert firstMessage.matches("!login [a-zA-Z0-9_\\-]+ ["+B64+"]{43}=") : "1st message";
// the server's response to the client
String secondMessage = ...
assert secondMessage.matches("!ok ["+B64+"]{43}= ["+B64+"]{43}= ["+B64+"]{43}= ["+B64+"]
{22}==") : "2nd message";
// the last message sent by the Client
String thirdMessage = ...
assert thirdMessage.matches("["+B64+"]{43}=") : "3rd message";
```

Hints & Tricky Parts

"java.lang.SecurityException: Unsupported keysize or algorithm parameters" or "java.security.InvalidKeyException: Illegal key size"

You will need to install the Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files from **Java SE Download** page (last download link). The installation explanation can be found in README file, but basically you will just have to copy `local_policy.jar` and `US_export_policy.jar` into `$JAVA_HOME/jre/lib/security/` directory.

The decrypted messages are in gibberish and no exception is thrown.

Possible reasons:

1. You forgot to decode the decrypted message back from Base64 format.

2. Wrong value was used for initialization vector (IV) parameter when initializing the AES cipher.
3. You didn't use `Cipher.DECRYPT_MODE` when initializing the AES cipher

java.security.NoSuchProviderException: No such provider: BC

This exception is thrown when the Bouncy Castle library is not properly installed. Please recheck that you correctly did all the steps from the **Installation and Static Registration of the Bouncy Castle Provider** section and actually use the so configured JDK.

java.security.NoSuchAlgorithmException: No such algorithm: "..."

The most probable cause is that you misspelled the name of the algorithm. Please check that name of the algorithm is same as mentioned in this assignment.
This exception is also thrown when the Bouncy Castle provider is not properly installed. If the name of the algorithm is correct, than you should recheck that you correctly did all the steps from the **Installation and Static Registration of the Bouncy Castle Provider** section.

java.security.InvalidKeyException: no IV set when one expected

You forgot to initialize the AES cipher using the initialization vector (IV) parameter. The IV parameter is mandatory for the `"AES/CTR/NoPadding"` algorithm. See **code snippets** on how to correctly initialize the AES cipher.

Helpful Code Snippets

All code snippets are just examples and omit all exception handling for clarity. This does not mean that you should not do exception handling in your lab solution!

- **How to encode into and decode from Base64 format?**
- **How to read a PEM formatted RSA private key?**
- **How to read a PEM formatted RSA public key?**
- **How to generate a secure random number?**
- **How to generate an AES secret key?**
- **How to initialize a cipher?**
- **How to read the shared secret key?**
- **How to create a hash MAC?**
- **How to verify a hash MAC?**

How to encode into and decode from Base64 format?

```
import org.bouncycastle.util.encoders.Base64;

[...]

// encode into Base64 format
```

```
byte[] encryptedMessage = ...
byte[] base64Message = Base64.encode(encryptedMessage);

// decode from Base64 format
encryptedMessage = Base64.decode(base64Message);
```

How to read a PEM formatted RSA private key?

Note: Do not hardcode the password for the private key!

```
import java.security.KeyPair;
import java.security.PrivateKey;
import org.bouncycastle.openssl.PEMReader;
import org.bouncycastle.openssl.PasswordFinder;

[...]

String pathToPrivateKey = ...
PEMReader in = new PEMReader(new FileReader(pathToPrivateKey), new PasswordFinder() {
           @Override
           public char[] getPassword() {
                 // reads the password from standard input for decrypting the private key
                 System.out.println("Enter pass phrase:");
                 return new BufferedReader(new InputStreamReader(System.in)).readLine();
           }
     });
KeyPair keyPair = (KeyPair) in.readObject();
PrivateKey privateKey = keyPair.getPrivate();
```

How to read a PEM formatted RSA public key?

```
import java.security.PublicKey;
import org.bouncycastle.openssl.PEMReader;

[...]

String pathToPublicKey = ...
PEMReader in = new PEMReader(new FileReader(pathToPublicKey));
PublicKey publicKey = (PublicKey) in.readObject();
```

How to generate a secure random number?

```
import java.security.SecureRandom;

[...]

// generates a 32 byte secure random number
SecureRandom secureRandom = new SecureRandom();
final byte[] number = new byte[32];
secureRandom.nextBytes(number);
```

How to generate an AES secret key?

```
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
```

```
[...]

KeyGenerator generator = KeyGenerator.getInstance("AES");
// KEYSIZE is in bits
generator.init(KEYSIZE);
SecretKey key = generator.generateKey();
```

How to initialize a cipher?

```
import javax.crypto.Cipher;

[...]

// make sure to use the right ALGORITHM for what you want to do
// (see text)
Cipher crypt = Cipher.getInstance(ALGORITHM);
// MODE is the encryption/decryption mode
// KEY is either a private, public or secret key
// IV is an init vector, needed for AES
crypt.init(MODE, KEY [,IV]);
```

How to read the shared secret key?

```
import java.io.FileInputStream;
import java.security.Key;
import javax.crypto.spec.SecretKeySpec;
import org.bouncycastle.util.encoders.Hex;

[...]

byte[] keyBytes = new byte[1024];
String pathToSecretKey = ...
FileInputStream fis = new FileInputStream(pathToSecretKey);
fis.read(keyBytes);
fis.close();
byte[] input = Hex.decode(keyBytes);
// make sure to use the right ALGORITHM for what you want to do
// (see text)
Key key = new SecretKeySpec(input,ALGORITHM);
```

How to create a hash MAC?

```
import java.security.Key;
import javax.crypto.Mac;

[...]

Key secretKey = ...
// make sure to use the right ALGORITHM for what you want to do
// (see text)
Mac hMac = Mac.getInstance(ALGORITHM);
hMac.init(secretKey);
// MESSAGE is the message to sign in bytes
hMac.update(MESSAGE);
byte[] hash = hMac.doFinal();
```

How to verify a hash MAC??

```
import java.security.MessageDigest;
import javax.crypto.Mac;

[...]

// computedHash is the HMAC of the received plaintext
byte[] computedHash = hMac.doFinal();
// receivedHash is the HMAC that was sent by the communication partner
byte[] receivedHash = ...

boolean validHash = MessageDigest.isEqual(computedHash,receivedHash);
```

Further Reading Suggestions

- **APIs**
  - Java SE 6: **Cipher**, **SecureRandom**, and **Signature** classes
  - Bouncy Castle: **PEMReader** and **Base64**
- **Books**
  - **Handbook of Applied Cryptography** (free version)
