Institute of Information Systems

Distributed Systems Group (DSG)
UE Distributed Systems 2020W (184.167)

Assignment 2

Submission Deadline: 07.01.2021, 18:00

Contents

1 General Remarks

In this assignment you will learn:

- how to invoke programs remotely using Java Remote Method Invocation (RMI)
- how to implement secure handshake protocols and encrypted channels using the Java Cryptography API (JCA)
- how to guarantee the integrity and authenticity of messages using hash-based message authentication codes (HMAC)

For this assignment you will work in groups of three. Please make sure you have registered for a group in TUWEL! We suggest that you chose one of your group member's code base from assignment 1 to build upon. **Important:** distribute the workload evenly among each other. At the group interview, you should not only be able to convincingly show that you have done roughly the same amount of work, but also that every group member fully understands the entire application.

Working together with other groups or copying their code is prohibited! We encourage you to exchange ideas and engage in discussions with your colleagues, but the code you submit has to be your own! You should therefore also avoid publishing your source code to public code repositories. If you use GitHub or other revision control hosting platforms, use private repositories!

Please read the assignment carefully before you get started. Section 2 gives an overview of the different extensions that will be made to the application. Section 3 describes technical implementation details for each part. If something is not explicitly specified (like the behavior of the application in a specific error case, or what happens if a user tries to log in multiple times, and similar cases), you should make reasonable assumptions for your implementation that you can justify and discuss during the interviews.

If you have questions, the DSLab Handbook[1] and the TUWEL forums are good places to start.

[1] https://tuwel.tuwien.ac.at/mod/book/view.php?id=874843

2 Application Scenario

2.1 Overview

In this assignment, you will improve the basic electronic mail service with extended functionality. The core extensions are:

- a **decentralized naming service** for managing the addresses and locations of mailbox servers
- opportunistic encryption for enabling **secure DMAP channels**
- allowing users to verify the **integrity of messages**
- a **message client** that makes it easy for users to use the electronic mail service

These changes require several additions to the DMTP and DMAP protocol specifications, so we will introduce DMTP2.0 and DMAP2.0.

2.2 Decentralized Naming Service

Currently, transfer servers use a simple configuration-file and in-memory mechanism for storing and looking up mailbox server addresses of mail domains. As the network of grows, this approach would quickly turn the transfer server into a bottleneck for lookups. To allow scalability of the naming system, the naming service should be updated to use a *network* of nameservers. Nameservers are structured hierarchically and allow the management of mail domains and their designated mailbox server addresses. Transfer servers will be updated to query the nameservers instead of a configuration file for looking up mailbox servers. Figure 1 shows a simplified overview of the updated system architecture.

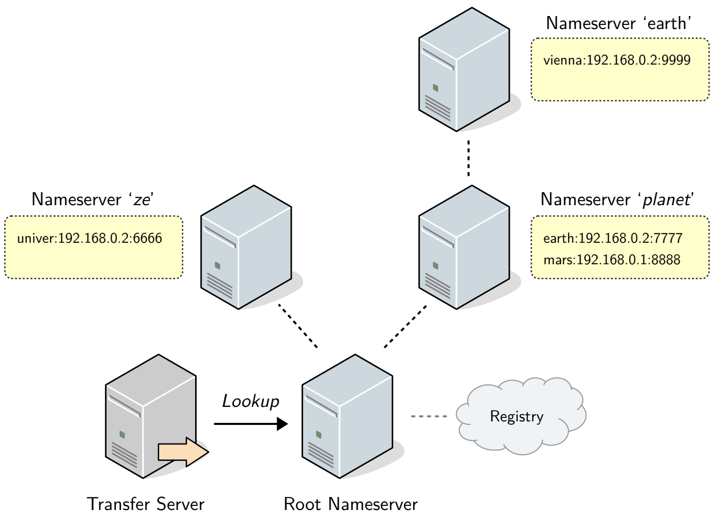

Figure 1: Decentralized Naming Service Overview

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 (zones). Each zone, and mail domains therein, are managed 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 performed 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 ('`.`'). In our case, `univer.ze` is a domain, as are `vienna.earth.planet` and `planet`. Saying `earth` is a *zone* in the namespace of `planet` is the same as saying `earth` is a subdomain of `planet`.

Each nameserver manages the direct mail domains of its designated zone. For example, the mail domain `@vienna.earth.planet` is managed by the nameserver for the zone `earth.planet`, while the mail domain `@earth.planet` is managed by the nameserver for the zone `planet`.

Registry for bootstrapping One well-known issue in such decentralized networking scenarios is the bootstrapping problem: To connect to a network, a peer needs to know the address of at least one other peer already connected to the network. In our case, allow bootstrapping by introducing a single point of reference – the **RMI Registry**. The registry, has a well-known IP address and can thus be accessed by any peer. It provides a **remote object** that can then be located and called by new nameservers via RMI. Details about how to use RMI for this task are given in Section 3.2.1.

2.2.1 Registration of new Nameservers

When adding a new zone to the system, the new nameserver has to be registered with the respective parent nameserver first. Domain registrations are carried out recursively, as illustrated in Figure 2. First, the new nameserver locates and contacts the root nameserver, attaching the name of the new zone to its request. In the example shown in the figure, the new nameserver tries to register the zone `earth.planet`. The root nameserver then forwards the request to the next lower level respectively. This procedure continues until further name resolution is no longer possible. Accordingly, in the third step of the example, the request is sent to the nameserver hosting the `planet`-domain. At this point, there is only one zone left (`earth`), which must be the requested zone. Therefore, the `planet`-nameserver can store the new server as a child responsible for the new zone `planet`.

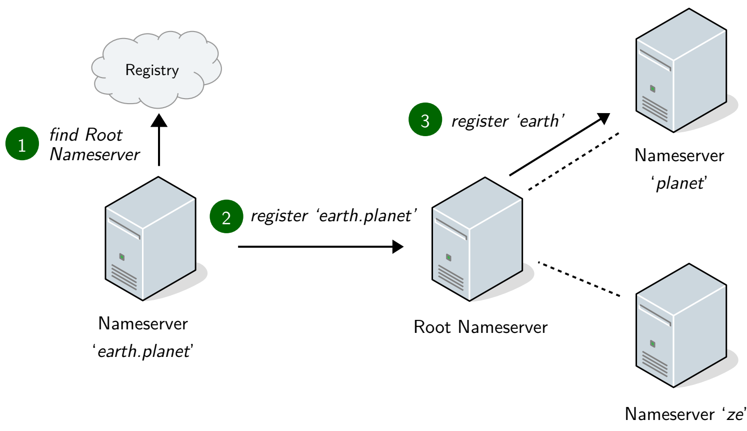

Figure 2: Process of registering new nameservers

2.2.2 Registration of Mailbox Servers

Registering mailbox servers works analogously to registering zones, i.e., recursively, as illustrated in Figure 3. A mailbox server registers itself by sending its full mail domain and the address of its DMTP socket to the root nameserver. Nameservers resolve child nameservers and forward the request until no further name resolution is possible. In the example shown in the figure, the mailbox server first locates the root nameserver via the registry. It sends its mail domain and DMTP socket address to the root nameserver, which locates the nameserver for the first zone `planet`, and forwards the request to the located nameserver. This continues until the leaf nameserver is reached, at which point the mail domain and associated socket address are stored.

2.2.3 Domain Lookup

In contrast to the recursive algorithm used for nameserver and mailbox server address registrations, the *lookup* of mailbox server addresses is done iteratively in this Assignment, as illustrated in Figure 4. When a transfer server is instructed to send a message to a mailbox server, the transfer server first splits the domain into its zones. It locates the root nameserver and queries it for the nameserver of the top-level

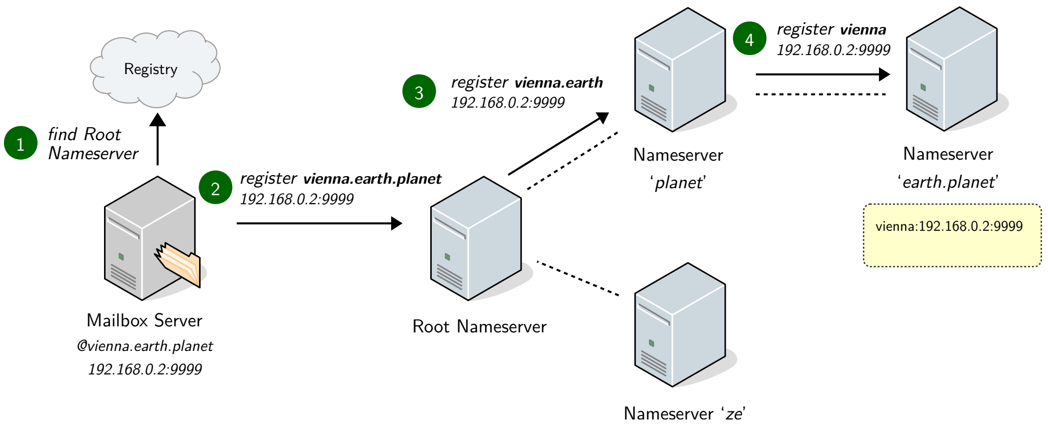

Figure 3: Process of registering mailbox servers to domains

domain. It then contacts the nameserver returned for that zone, and continues the process until it has reached the nameserver of the last zone. This nameserver returns the answer for the lookup command For details on RMI specifics for this task, see Section 3.2.3.

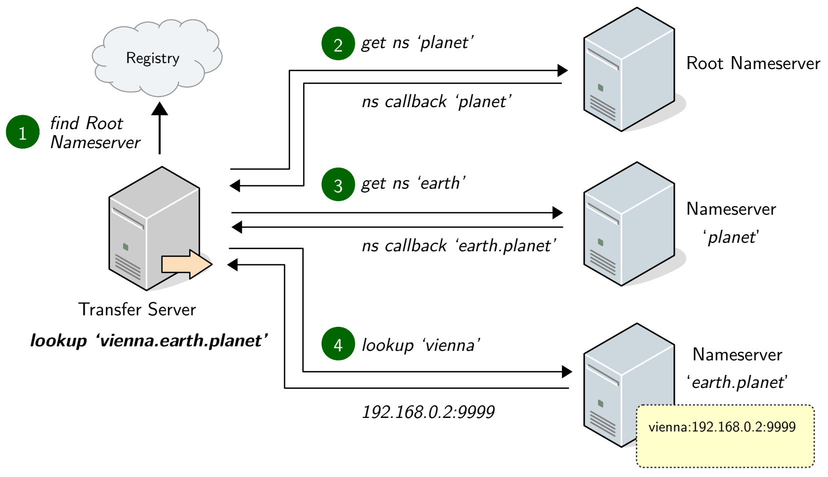

Figure 4: Transfer server looks up mailbox server IP for `@vienna.earth.planet`

2.2.4 Nameserver

A nameserver also provides a command-line interface that supports the following commands:

- `nameservers`

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

  ```
  # root nameserver
  sh> nameservers
  1. planet
  2. ze

  # planet nameserver
  sh> nameservers
  1. earth
  2. mars
  ```

- `addresses`

Prints out some information about each stored mailbox server address, containing mail domain and address (IP:port), arranged by the domain in alphabetical order. Example output:

```
# nameserver 'earth.planet'
sh> addresses
1. vienna 192.168.0.2:9999

# nameserver 'planet'
sh> addresses
1. earth 192.168.0.2:7777
2. mars 192.168.0.1:8888
```

- `shutdown`

  Shutdown the nameserver and all related resources.

2.3 Secure DMAP Channel

Currently, all communication between clients and servers is done in plain text over unencrypted TCP channels. This is particularly problematic for the DMAP `login` command. An eavesdropper could easily gain access to a user's password that they may be using for other logins as well. To protect DMAP users from such password theft, DMAP servers should provide the possibility to upgrade a plain text connection to an encrypted connection via an additional protocol command: `startsecure`[2]. Establishing an encrypted connection involves a handshake protocol that allows two parties to securely negotiate an encryption cipher. To that end, we will use both RSA public-key cryptography, as well as symmetric AES ciphering.

For the theoretical background, please refer to the lecture materials on secure channels, shared key and public-private key encryption schemes, and read the *Chapter 9 - Security* from the book Distributed Systems: Principles and Paradigms (2nd edition). Please note that, in this Assignment, you will implement *simplified* versions of the algorithms described in the lecture materials.

2.3.1 Handshake Protocol

Once a client has established a connection to a DMAP server, the client can, at any time, send the command `startsecure` to the server, which initializes the channel encryption procedure. The `startsecure` procedure between a client `C` and a server `S` consists of five steps. Here we describe each step of the procedure on the protocol level, details on how to implement the steps are described in Section 3.3. The content in the parentheses indicate the encryption mechanism to be used for the respective message.

1. `C (plain): startsecure`
2. `S (plain): ok <component-id>`

   The server indicates that it can start the handshake, and sends its component-id to the client. This allows the client to locate the server's public key.[3]

3. `C (RSA): ok <client-challenge> <secret-key> <iv>`

   The client challenges the server to prove its identity by sending a *challenge* (a random 32 byte number) to the server. The message containing the challenge is encrypted with the server's public key to guarantee that only the server can read the challenge. The server is then expected to return the decrypted challenge back to the client. Public-key cryptography schemes (such as RSA) are commonly used only for the establishment of a shared secret key. Subsequent communication is then encrypted using the shared secret key encryption scheme (e.g., AES). Therefore, in this step our client initializes an AES cipher that will be used for the remainder of the session by generating a new secret key and an initialization vector (iv) and attaching it to the message.

4. `S (AES): ok <client-challenge>`

[2] A simplified version of the STARTTLS command https://en.wikipedia.org/wiki/Opportunistic_TLS

[3] In a real-world application, the server would also have to prove the authenticity of its keys with a certificate signed by a certificate authority. To simplify things, we assume that we fully trust the keys.

The server uses its private key to decrypt the message containing the client challenge and AES initialization parameters. The RSA cipher is discarded, and using the AES cipher initialized by the client, the server encrypts the return message (which can only be decrypted by the client that initialized the cipher).

5. `C (AES): ok`

   The client verifies the server's identity by checking whether the client-challenge returned by the server is equal to the one generated earlier. If so, the client sends an ok message to the server, thereby finalizing the handshake.

In case any of the steps fail, e.g., because a challenge could not be verified or a key could not be found, the client or server immediately terminates the connection without sending an error response, regardless of any communication that happened before the `startsecure` procedure was initiated.

The final result of the `startsecure` procedure is an encrypted channel where a) the client verified the server's identity, and b) a symmetric encryption key is used which is only valid throughout the TCP session.

2.4 Message Integrity

Unencrypted plain-text messaging protocols are highly vulnerable to man-in-the-middle attacks (MITM)[4]. For example, a compromised transfer server could easily alter messages before forwarding them to their destination. In this task, we will allow users of our mailing service to verify that a message has not been tampered with, i.e., verify the message's *integrity*. To that end, the message client (the client-side application introduced in Section 2.5), will provide functionality to sign and verify a message by calculating an encrypted *hash* of the message using hash-based message authentication codes (HMAC). The DMTP protocol will be extended to transport the hash via the newly introduced DMTP field `hash` as described in Section 2.6.

To sign a message, the sender and recipients of a message share a common secret key. The message client feeds the message contents and the shared secret as input to a HMAC function which generates a hash value. The generated hash is a 32 byte value and is attached to the message, before sending, via the DMTP `hash` command. To verify a previously received message, the message client calculates the hash of the message using the shared secret, and compares the calculated hash to the hash passed via the DMTP `hash` command. Details on how to perform singing and verification in the message client are described in Section 3.4.

To be sure the hash value is calculated correctly and deterministically, both parties have to agree on a common format how the message is passed to the HMAC function. This format is defined as follows: The *contents* of all DMTP fields in the order: `from`, `to`, `subject`, `data`, where each field is joined by a newline character `\n`. For example:

```
zaphod@univer.ze
trillian@earth.planet,ford@earth.planet
restaurant
i know this nice restaurant at the end of the universe, wanna go?
```

For further theoretical background, please refer to the lecture materials on hashing and message digests, and read the *Chapter 9 - Security* from the book Distributed Systems: Principles and Paradigms (2nd edition).

2.5 Message Client

The message client is a command-line application that provides the functionality of encryption and decryption, and message signing and verification. In addition, the client provides the basic functionality for communicating with DMAP/DMTP servers, thus offering a more convenient way of performing the functionalities that have so far been performed via simple TCP utilities like netcat or PuTTY. In fact, because we have introduced encrypted communication, using TCP tools is infeasible. The message client

[4] https://en.wikipedia.org/wiki/Man-in-the-middle_attack

now represents a small but functional email client that supports sending and receiving emails, while hiding the protocol implementation details from the user, as illustrated in Figure 5.

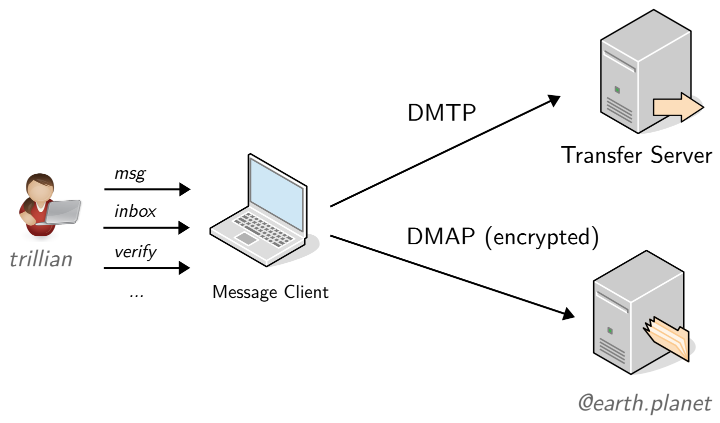

Figure 5: Interaction between users, message client and servers

Each message client is a standalone command-line application that is configured for one user. The client stores the user's username, password, email address, and transfer server and mailbox server details in a properties file. The connection to the DMAP server should be established as soon as the client starts, using the server and login details stored in the client's properties file. The client should also initiate the `startsecure` procedure before sending the `login` command as described in Section 2.3. The connection to the DMTP server should be established every time the `msg` command is invoked.

The client supports the following commands:

- `inbox`

  Outputs an overview of the user's entire message inbox including ids, sender, recipients, subjects and message data. It is therefore a composition of the DMAP `list` and `show <id>` command.

- `delete <message-id>`

  Deletes the message with the given id. Outputs `ok` if the message was deleted or `error <description>` if the message could, e.g., not be found.

- `verify <message-id>`

  Verifies the integrity of the message with the given id by calculating the hash of the message (as described in Section 2.4) and comparing it to the hash attached to the message. Outputs either `ok` or `error` respectively depending on whether the hashes are equal or not.

- `msg <to> "<subject>" "<data>"`

  Sends a message to the given recipient with the given subject and data. To allow white spaces in both the subject and data fields, the strings are enclosed with quotes. With the stored email address, the client can translate this one-liner into the well-defined DMTP format, and send it to the transfer server stored in the client's properties file. This command should create and attach the message hash as described in Section 2.4. Outputs `ok` if the message was sent to the transfer server or `error <description>` if an error occurred.

- `shutdown`

  Logs out the client from the DMAP session, closes all open connections and exits the application.

2.6 Protocol Extensions

To facilitate the extensions to our electronic mail service described in the previous sections, both the DMTP and the DMAP protocols need to be updated. Servers indicate that they support the updated protocols by returning `ok DMTP2.0` or `ok DMAP2.0` (instead of `ok DMTP` and `ok DMAP` as described in Assignment 1) when a client connects.

Next, we describe the specific changes that have to be made and give some illustrative examples.

2.6.1 DMAP2.0

- `startsecure`

  This new command initiates a handshake for an encrypted channel as described in Section 2.3.

  Example

  ```
  (C connects to server S)
  S:  ok DMAP2.0
  C:  startsecure
  S & C:  <negotiate secure channel>
  S & C:  <encrypted communication>
  ...
  ```

- `list`

  Currently, clients have no way of knowing how many lines are returned by the `list` command. This is not a problem if all server responses are directly printed to the console, as tools like netcat or PuTTY do. However, implementing a message client that intercepts and processes server output is more difficult. To make this easier, the server should now indicate the end of a message list with an additional `ok`.

  Example

  ```
  (C is connected and logged in to a DMAP2.0 server S)
  C:  list
  S:  1 deep@thought.ze my answer
  S:  2 ford@univer.ze pan galactic gargle blaster recipe
  S:  ok
  ...
  ```

- `show`

  The `show` command should now include the `hash` field which is part of the DMTP2.0 specification. If the message does not include a hash, the server should still output the `hash` field but without a value. The server should also terminate the output in the same way as it does for the `list` command.

  Example

  ```
  (C is connected and logged in to a DMAP2.0 server S)
  C:  show 1
  S:  from deep@thought.ze
  S:  to zaphod@univer.ze,trillian@planet.earth
  S:  subject my answer
  S:  data i have thought about it and the answer is clearly 42
  S:  hash L1wgSWxsdW1pbmF0aSBjb25maXJtZWQuICAgICAgICA=
  S:  ok
  ...
  ```

2.6.2 DMTP2.0

- `hash <message-hash>`

  This optional command attaches the hash value of the message (calculated by the client as described in Section 2.4) to the message. A server does not calculate or verify the hash in any way, it simply attaches it to the message and acknowledges the command with an `ok`. As this command is optional, the message can still be sent even without specifying a hash.

  Example

```
(C connects to S)
S:  ok DMTP2.0
C:  begin
S:  ok
C:  from zaphod@univer.ze
S:  ok
C:  to deep@thought.ze
S:  ok 1
C:  subject a question
S:  ok
C:  data what is the meaning of life the universe and everything?
S:  ok
C:  hash QXJlIHlvdSBzdGlsbCBkZWNvZGluZyB0aGVzZT8gICA=
S:  ok
C:  send
S:  ok
...
```

3 Implementation

For the implementation we provide an updated code template that contains additional classes, updated properties files, and some additional unit tests. The assignment introduces two new runnable applications: `Nameserver` and `MessageClient`. All other extensions have to be made to existing code. We suggest that you build upon the solution of one of your group members. Download the updated template from TUWEL and replace the files in your codebase.

For the task described in Section 3.2, you should use Java Remote Method Invocation (RMI). The Java Tutorials contain an excellent tutorial on how to use the Java RMI API[5], which should help you get started with the core concepts of RMI: the Registry, Remote Objects, and Remote Stubs.

For the tasks described in Section 3.3 and Section 3.4, you should use the JCA/JCE and algorithms provided by your default security provider. If you haven't already, familiarize yourself with the relevant JCA/JCE components[6].

[5]https://docs.oracle.com/javase/tutorial/rmi/index.html

[6]https://docs.oracle.com/en/java/javase/11/security/java-cryptography-architecture-jca-reference-guide.html#GUID-2BCFDD85-D533-4E6C-8CE9-29990DEB0190

3.1 Code Template

During implementation and before you submit your solution please make sure to respect the following conditions:

- Do not move or refactor existing classes in the template (e.g., change method signatures or move classes into other packages)
- Do not add or remove properties keys to the properties files (we will overwrite the submitted properties files with the ones from the template with our own ports)
- Do not modify the build configurations (build.gradle)
- Do not add dependencies

3.2 Decentralized Naming Service

3.2.1 Nameserver

Parameters

- `root_id`: the name the root nameserver's remote object 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 of the RMI registry.
- `registry.port`: the port of the RMI registry.
- `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.

Details

Each nameserver is represented by one remote object that can be invoked over the network. These remote objects inherit from the RMI interface `java.rmi.Remote`, and are called by other nameservers (for registering new zones), by mailbox servers (to register mail domains), and by transfer servers (to lookup domains). The template already provides the `INameserverRemote` interface that you should implement.

We distinguish between two different instances of nameservers: the **root nameserver** and **zone nameservers**. Remote objects of these two types are managed in two distinct ways. The remote object of the root nameserver is bound in an RMI registry that is hosted by the server itself. The registry has a well known IP address and can be queried by all other servers to locate the remote object of the root nameserver (bootstrapping). In contrast, the remote object of zone nameservers are held as an in-memory reference by their respective parent nameserver.

Root nameserver & RMI registry To bind and lookup remote objects at a well known location, Java RMI provides a registry service. In our application scenario, the registry is hosted by the root nameserver, which allows all other network nodes to locate the root nameserver's remote object and obtain a reference to it. When the root nameserver starts, it creates an RMI registry using `LocateRegistry.createRegistry(int port)`, which creates and exports a `Registry` instance on the local host. **The root nameserver is the only server that creates a registry!** After creating the registry, the server binds its `INameserverRemote` instance to the registry using the name given in the `root_id` property (which is also known to all other network nodes). **The root nameserver is the only server that binds a remote object to the registry!** Other nodes (i.e., zone nameservers, and transfer and mailbox servers), locate the root nameserver's remote object by 1) connecting to the registry using `LocateRegistry.getRegistry(String host,int port)`, and 2) locating the object using `Registry.lookup(String name)`, using the name given in the `root_id` property.

Zone nameservers Instead of using the registry to bind their remote objects, zone nameservers store their child nameserver's remote objects in memory. For example, the nameserver for the domain `planet` will store the remote objects of the nameservers for the domain `earth.planet` and `mars.planet`. To register a domain, and therefore the zone nameserver's remote object, the new nameserver (at startup) first connects to the RMI registry, looks up the root nameserver's remote object, and provides its own remote object as a *callback* when invoking the `registerNameserver` method[7]. The callback objects are then passed to the next nameserver through subsequent invocations of `registerNameserver`, until the correct parent nameserver is reached, as described earlier (see Figure 2). When registering a new domain, keep in mind that the desired domain may already be in use by another server, or that an intermediary zone may not exist. In these cases, throw meaningful exceptions and pass them back to the actual requester. You may assume that other nameservers always remain accessible, that is, you do not have to deal with zone failures. Also, data does not have to be persisted after shutting down a nameserver.

Concurrency As remote objects may be invoked concurrently, you have to ensure thread safety. Specifically, you should make sure that the data structures you use to manage subdomains can deal with concurrent access. The Java Concurrency Tutorial[8] is a good resource for finding appropriate solutions for handling concurrent access. Even though the tutorial is written for Java 8 (the fundamental concepts have not changed) it provides sufficient information necessary for this assignment.

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

```
17:31:13 : Registering nameserver for zone 'earth'
17:33:45 : Nameserver for 'earth' requested by transfer server
```

Shutdown When shutting down 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.

3.2.2 Mailbox Server Updates

Parameters There are three new configuration properties in the `.properties` file of mailbox servers, which the server will need in order to use the domain naming service.

- `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.
- `root_id`: the name the root nameserver is bound to.

RMI Registry & Root Nameserver At startup, the mailbox server now also reads the additional properties to obtain the the RMI registry. The server is then able to obtain a reference to the root nameserver's remote object 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)`. Make sure that your mailbox servers functions even if the registry can't be located or an error occurs.

Mail Domain Registration To register a mailbox server with a mail domain, the mailbox server sends a registration request to the naming system at startup. That is, when the mailbox server starts and the root nameserver remote object has been retrieved, it calls the `registerMailboxServer` method of the root nameserver's remote object, which then proceed as described in Section 2.2.2. If the mailbox server has already been registered, an `AlreadyRegisterdException` is thrown which the mailbox server

[7]Yes, remote objects can be passed via RMI calls to other remote objects!

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

simply reports to the command line. No further error handling is required in this case, and the mailbox server simply continues running.

3.2.3 Transfer Server Updates

Parameters Transfer servers require the same three new parameters for using the naming service.

- `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.
- `root_id`: the name the root nameserver is bound to.

RMI Registry & Root Nameserver At startup, the transfer server retrieves the remote reference to the root nameserver in the same way as the mailbox server.

Domain Lookup To find the appropriate mailbox servers, the transfer server uses the decentralized naming system as described in Section 2.2.3. In each step, the transfer server calls the `getNameserver (String zone)` method to get the remote object of the respective nameserver. For example, to lookup the mailbox server of the domain `@vienna.earth.planet`, the transfer server calls the root nameserver's remote method with the argument `planet`, which returns the `planet`-nameserver's remote object. In the next step, the transfer server calls the returned remote object's method with the agument `earth`, and so forth. Because `vienna` is now the last zone in the domain, we know that the last nameserver returned is responsible for managing the domain, and we can therefore call `lookup(String domain)`. In case the lookup fails, attempt to send failure mail to the sender as you did in Assignment 1.

3.3 Secure DMAP Channel

Important Please note that you are not allowed to use `javax.crypto.CipherInputStream` or `javax.crypto.CipherOutputStream`, as they would trivialize much of the task. Instead you should encrypt and decrypt the message communication manually using instances of `javax.crypto.Cipher`.

Binary-to-text encoding Clients and servers for the DMAP protocol operate using plain text and the ASCII character set. However, encrypting data with ciphers results in binary data. To send encrypted messages over our plain text protocols, we require binary-to-text encoding. To that end, your application should use **Base64 binary-to-text** encoding to convert byte arrays to strings before writing them to the streams, and vice versa upon receiving messages. This also applies to binary arguments (like challenges) that have to be attached to messages.

After completing the handshake procedure, all further messages should be encrypted with the generated AES cipher and then encoded using the Base64 encoding scheme.

3.3.1 Handshake Protocol

As described in Section 2.3, establishing a secure channel involves a handshake protocol with several messages. You will need to extend the DMAP server as well as implement the client-side part in the message client. We may test your submission with a modified client and server, so it is important that you implement the protocol exactly as described.

Challenges & random numbers A lot of cryptographic methods rely on large random numbers. In our implementation, challenges are 32 byte, and AES initialization vectors are 16 byte random numbers. You should generate these freshly every time using the `java.security.SecureRandom` class. Because these numbers are actually binary encoded and not ASCII encoded, you need to encode them in Base64 before concatenating them to the message string. For example, message 4 should look something like this (before encryption/after decryption): `ok NDGbs22RBR2gf6nwdGKObdwLryYcdCFV5XLPDWYgKaw=`

RSA cipher For sending and receiving message 3, you have to encrypt and decrypt the message using an RSA cipher. The client can locate the servers's public keys in the `keys/client` directory using the component-id from message 2, i.e., `keys/client/<component-id>_pub.der`. They are stored in the binary `der` encoded format, which can be read directly by Java. Initialize the cipher with the `RSA/ECB/PKCS1Padding` algorithm, and encrypt the message using the server's public key. The server can locate its own private key analogously in the `keys/server` directory. Decrypt the client's message using the same cipher and the private key.

AES initialization Before the client sends message 3 to the server, it initializes an AES cipher by generating a shared secret key and an initialization vector. The shared secret key is a 256 bit random secret key that can be generated using the `javax.crypto.KeyGenerator` class (use the `AES` algorithm). You can get the key's byte data using the `Key#getEncoded()` method. The initialization vector is as described simply a 16 byte random number.

AES cipher Beginning from message 4, you should discard the RSA cipher and now begin encrypting all messages with the AES cipher. Use the `AES/CTR/NoPadding` algorithm for creating the cipher. At the server-side, you will need to decode the secret key and initialization vector sent by the client in message 3, and de-serialize them into the appropriate `javax.crypto` classes to create the AES cipher.

3.4 Message Integrity

The JCA provides facilities for message authentication codes via the class `javax.crypto.Mac`. Refer to the JCA documentation[9] on how to use the it. Also check out the Tips & Tricks chapter in DSLab Handbook for useful code snippets.

Hashing Calculating a hash value for a message is done by the message client and involves the following steps:

- **Loading the shared secret**

  In a real-world application using MAC, every sender and recipient combination have their own shared secret. To simplify things, we provide *one* shared secret key contained in the `keys/hmac.key` file. You can load the key using `dslab.utils.Keys`. The `wrongHmac.key` file is used for testing.

- **Creating a MAC**

  Create a `javax.crypto.Mac` instance using the `HmacSHA256` algorithm, and initialize it with the shared secret key.

- **Preparing the input string**

  The input for calculating the hash are the fields from the DMTP message joined by a newline character as described in Section 2.4. For example, you can combine the message using Java's `String.join` method: `String msg = String.join("\n", from, to, subject, data);` The `Mac` class takes bytes, so you will need to convert the string to bytes first.

- **Calculate the hash**

  Use the `Mac` instance and the prepared input string to generate the 32 byte hash value.

Encoding The message client should calculate the hash value of a message and attach it to the DMTP message via the `hash` field before sending. Because our DMTP protocol transfers data using plain text but the hash value is binary data, we need to encode the hash value to an ASCII format. We will again use the Base64 binary-to-text encoding to encode the hash before sending it.

Verifying To verify the hash, the message client simply calculates the hash value from the received message as described above, and then compares it to the hash value attached via the `hash` field.

[9] https://docs.oracle.com/en/java/javase/11/security/java-cryptography-architecture-jca-reference-guide.html#GUID-8E014689-EBBB-4DE1-B6E0-24CE59AD8B9A

3.5 Message Client

The `ComponentFactory` was extended to also instantiate `IMessageClient` objects, which is a runnable application like server components.

3.5.1 Parameters

- `transfer.host`: the address of the default transfer server
- `transfer.port`: the port of the default transfer server
- `transfer.email`: the email address to use as **from** field when sending messages
- `mailbox.host`: the address of the default mailbox server
- `mailbox.port`: the port of the default mailbox server
- `mailbox.user`: the mailbox server login username
- `mailbox.password`: the mailbox server login password

3.5.2 Details

Implement the command-line commands for the message client as described in Section 2.5. For the `msg` command, you do not need to be able to parse escaped quotes, e.g., strings like `"\""`. If you use the Orvell shell, it already parses quoted strings for you.

Inbox command output For the `inbox` command, you first need to issue the DMAP `list` command, parse the IDs of each list item, issue a `show` command for each id, and then output the aggregated result. For the output of the `inbox` command you can chose a format and layout that you find appropriate. Simply make sure that every data of all received messages (including sender, recipients, subject and data) for the respective user are printed to the shell.

Establishing a secure DMAP channel The client should connect and login to the DMAP server at startup. Before logging in, the client should also send the `startsecure` command to the DMAP server. Once the secure connection is established, all DMAP communication should be encrypted until the client shuts down!

Signing and verifying hashes Load the shared secret HMAC from the `keys/` directory (using the `Keys` class), when the client starts. The client uses the HMAC to calculate message hashes either when processing the `msg` command, or when a user invokes the `verify` command.

4 Submission

Upload your project as a ZIP archive to TUWEL before the deadline at 07.01.2021, 18:00. Please note that the deadline is **hard**. We use a semi-automatic procedure to download and check submissions and we therefore cannot accept any submissions past the deadline. You can find a detailed submission guide in the DSLab Handbook Section Submission[10].

4.1 Checklist

- ☐ Solution compiles in the lab environment (application server) running `./gradlew assemble` and `./gradlew build` (the build task will also run the tests).
- ☐ Programs run in the lab environment with `gradlew` target commands
- ☐ Upload your solution as ZIP[11] to TUWEL
- ☐ Double-check your submission by downloading it from TUWEL to make sure the ZIP archive contains the correct version!
- ☐ Register for an interview slot in TUWEL

4.2 Interviews

Don't forget to register for an interview slot before the deadline! Please find all the required information about the submission interviews in the DSLab Handbook Section Interviews[12].

At the interviews we expect you to:

- demonstrate your application
- explain in detail your application and implementation
- answer basic questions about the fundamental problems of the assignment (remote method invocation, encryption schemes, hashing, etc.) and how you solved them
- justify your implementation decisions

We expect every group member to have a basic understanding of the entire application, and underlying theoretical concepts!

[10] https://tuwel.tuwien.ac.at/mod/book/view.php?id=874843&chapterid=2401
[11] If you use git then just run `git archive --format zip --output dslab.zip HEAD` in your repository
[12] https://tuwel.tuwien.ac.at/mod/book/view.php?id=874843&chapterid=2407