Institute of Information Systems Engineering

Distributed Systems Group (DSG)
UE Distributed Systems 2019W (184.167)

Assignment 1

Submission Deadline: 14.11.2019, 18:00

Contents

1 General Remarks

In this assignment you will learn:

- the basics of TCP and UDP socket communication
- how to implement application-layer protocols
- how to manage multithreaded programs

Group work is prohibited for this assignment. 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 explains the application scenario and presents the specification of individual components. Section 3 then gives you detailed information on how to implement the components. 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=649459

2 Application Scenario

2.1 Overview

In this assignment, you will implement a basic electronic mail service. The core components are:

- a message transfer protocol
- a message access protocol
- transfer servers (responsible for forwarding messages to mailbox servers)
- mailbox servers (responsible for storing messages and making them available to users)
- monitoring server (receives usage statistics from transfer servers via UDP)

In this stage, users communicate with servers via TCP tools such as Netcat[2], or PuTTY[3]. Figure 1 shows the general system architecture.

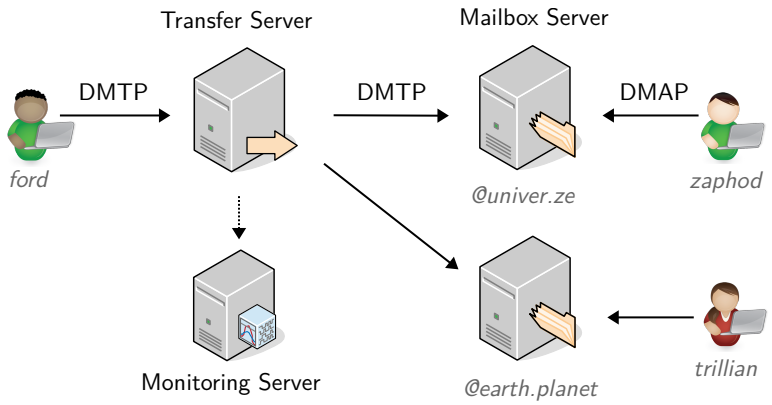

Figure 1: Electronic mail service architecture overview

2.2 Message Transfer Protocol

The *DSLab Message Transfer Protocol* (DMTP) is a plaintext application-layer protocol that specifies a way to exchange messages in a set of instructions. A message is defined by one or more recipients, one sender, a subject, and a text body (data). Both the transfer server and the mailbox server implement DMTP. It is used to exchange messages via TCP between the servers themselves, and between users and servers, as illustrated in Figure 2.

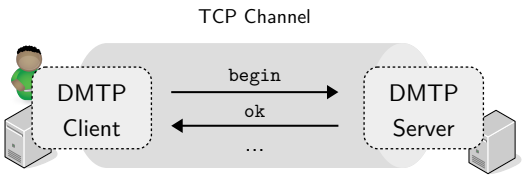

Figure 2: DMTP interaction

[2] https://en.wikipedia.org/wiki/Netcat (pre-installed on most *NIX systems as `nc`)
[3] https://www.chiark.greenend.org.uk/~sgtatham/putty/

2.2.1 Client instructions

When a client wants to send a message, it has to connect to a server that speaks DMTP via TCP, and send the instructions over the socket. Each instruction is defined as a command that can take one or more arguments. The instructions are as follows:

- `begin`

  Indicates the start of a message

- `to <address>[,<address>...]`

  Sets the recipients of the message. Recipients are a list of one or more comma separated email addresses, e.g., `to zaphod@univer.ze,trillian@earth.planet`

- `from <address>`

  Sets the sender of the message. The sender is a single email address.

- `subject <string>`

  Sets the subject of the message (single line)

- `data <string>`

  Sets the content of the message (single line)

- `send`

  Finalizes the message and instructs the server to send it

- `quit`

  Closes the socket connection

2.2.2 Server responses

A server that accepts DMTP instructions will immediately respond to each instruction with a specific answer. This communication pattern is known as synchronous request–response. Each time a valid instruction is received, the server responds with `ok`. When a client connects, the server initially sends the string: `ok DMTP` thereby telling the client that the server is ready and speaks DMTP. If the instruction caused an error, the server responds with `error <explanation>`. Repeated commands (e.g., setting the subject twice) will overwrite the previous value. If the server receives an instruction that is undefined, then it may immediately terminate the connection.

Accepting messages The response to the `to` command also contains the number of recipients that were accepted by the server (see Example 1, line 6). Whether or not a message is accepted depends on the behavior of the server type. Transfer servers generally accept any recipients. Mailbox servers only accept messages that contain *known* addresses of the domain they manage (see Section 2.5 for details). In the case that the `to` field of the message contains an unknown recipient, the server should respond with an error message (see Example 3).

2.2.3 Examples

Example 1 Here is a sample conversation between a DMTP client `C` and a server `S`. Notice that the client tried to execute the `send` instruction before the message was completed (line 11), and the server responded appropriately, but did not terminate the connection.

```
(C connects to S)                                                    1
S:  ok DMTP                                                          2
C:  begin                                                            3
S:  ok                                                               4
```

```
C:  to deep@thought.ze                                                    5
S:  ok 1                                                                  6
C:  subject a question                                                    7
S:  ok                                                                    8
C:  data what is the meaning of life the universe and everything?         9
S:  ok                                                                    10
C:  send                                                                  11
S:  error no sender                                                       12
C:  from zaphod@univer.ze                                                 13
S:  ok                                                                    14
C:  send                                                                  15
S:  ok                                                                    16
C:  quit                                                                  17
S:  ok bye                                                                18
(S terminates connection to C)                                            19
```

Example 2 In this example, the client sends an illegal command[4] to the server which returns an error message and then immediately closes the connection. This is typical behavior implemented by many protocols.

```
(C connects to S)                                                         1
S:  ok DMTP                                                               2
C:  BREW                                                                  3
S:  error protocol error                                                  4
(S terminates connection to C)                                            5
```

Example 3 In this example, the transfer server `T` attempts to send a message to a mailbox server `M`, which contains a recipient that `M` does not know.

```
(T connects to M)                                                         1
M:  ok DMTP                                                               2
T:  begin                                                                 3
M:  ok                                                                    4
T:  to ford@earth.planet,trillian@earth.planet                            5
M:  error unknown recipient ford                                          6
...                                                                       7
```

2.3 Message Access Protocol

Similar to DMTP, the *DSLab Message Access Protocol* (DMAP) is a plain text protocol that allows users to access stored messages via a set of instructions.

2.3.1 Client instructions

The instructions are as follows:

- `login <username> <password>`

  Attempts to authenticate the user with the given username and password. The username corresponds to the local-part of the email address (what comes before the @).

- `list`

  Lists all emails of the current user in the format `<message-id> <sender> <subject>`.

- `show <message-id>`

  Shows the message with the given ID.

[4] https://en.wikipedia.org/wiki/HTCPCP

- `delete <message-id>`

  Deletes the message with the given ID.

- `logout`

  Ends the current session (but does not close the connection)

- `quit`

  Closes the socket connection

2.3.2 Server responses

The general communication pattern and rules for DMAP are the same as for DMTP explained in Section 2.2.2.

2.3.3 Examples

Example 1 This example illustrates a standard DMAP client–server interaction session. A user first logs in and lists all their messages. An attempt to delete a message with an unknown id causes the server to respond with an error. After successfully deleting the first message, the message is no longer shown in the list.

```
(C connects to S)                                                    1
S:  ok DMAP                                                          2
C:  login zaphod 12345                                               3
S:  ok                                                               4
C:  list                                                             5
S:  1 deep@thought.ze my answer                                      6
S:  2 ford@univer.ze                                                 7
C:  delete 3                                                         8
S:  error unknown message id                                         9
C:  delete 1                                                         10
S:  ok                                                               11
C:  list                                                             12
S:  2 ford@univer.ze                                                 13
C:  logout                                                           14
S:  ok                                                               15
C:  quit                                                             16
S:  ok bye                                                           17
(S terminates connection to C)                                       18
```

Example 2 The show command simply outputs the message in the DMTP format.

```
(C connects to S)                                                    1
S:  ok DMAP                                                          2
C:  login zaphod 12345                                               3
S:  ok                                                               4
C:  show 1                                                           5
S:  from deep@tought.ze                                              6
S:  to zaphod@univer.ze,trillian@planet.earth                        7
S:  subject my answer                                                8
S:  data i have thought about it and the answer is clearly 42        9
C:  logout                                                           10
S:  ok                                                               11
C:  quit                                                             12
S:  ok bye                                                           13
(S terminates connection to C)                                       14
```

Example 3 Users have to be logged in to execute commands, otherwise the server will respond with an error. Failed login attempts are also responded to with appropriate error messages.

```
(C connects to S)
S:  ok DMAP
C:  list
S:  error not logged in
C:  login zaph 12345
S:  error unknown user
C:  login zaphod 123
S:  error wrong password
C:  quit
S:  ok bye
(S terminates connection to C)
```

2.4 Transfer Server

The **transfer server** is responsible for forwarding messages via our message exchange protocol to mailbox servers. It is both a DMTP server and a DMTP client. When a user connects to a transfer server, the server acts as a DMTP server. When the transfer server connects to a mailbox server, the transfer server takes the role of a DMTP client. Unlike mailbox servers, transfer servers accept messages from any recipient domain.

Domain lookup A key responsibility of the transfer server is to find the appropriate mailbox server(s) for a message. Once a message has been received via DMTP, the transfer server checks a domain registry to find the IP address and port for each domain supplied in the `to` field, and forwards the messages to those servers. That means a message may be sent to multiple different mailbox servers. Once the appropriate mailbox servers are located, the transfer server simply connects to the mailbox server's DMTP port and re-plays the received message via DMTP. Details about how domain lookup is implemented are described in Section 3.

Message delivery failures In the case that a message delivery fails, e.g., because DMTP communication to a mailbox server caused an error, or a domain lookup failed because no associated mailbox server was found, the sender of the message needs to be notified. To that end, the transfer server generates an error message (in the DMTP format) and attempts to send the error message to the mailbox of the sender of the original message. The `from` field of the error message should identify the transfer server (use the address mailer@[*IP*], where *IP* is the address of the transfer server[5]). If the error message delivery fails too, the transfer server gives up and also discards the error message.

2.5 Mailbox Server

The **mailbox server** is responsible for receiving and storing mails, and providing means for users to access them via the DMAP server protocol. Each mailbox server is associated with exactly one domain, e.g., `example.com` or `univer.ze` and manages email addresses ending with these domains. Only mails to recipients ending with the managed domain are stored.

A mailbox server receives emails by also providing the DMTP server protocol. However, unlike transfer servers, mailbox servers do not forward messages to other mailbox DMTP servers, as they do not have the capability to lookup mail domains. Mailbox servers should be able to handle multiple DMTP as well as DMAP connections at once. An important feature of the mailbox servers is to store messages for users even if they are not currently logged in.

[5] Although rare, this format is RFC compliant https://en.wikipedia.org/wiki/Email_address#Domain

Accepting messages As described in Section 2.2, mailbox servers only accept messages when they contain known recipients. However, a mailbox ignores recipients that do not belong to the domain it manages. So, for example, if the mailbox server manages the domain `earth.planet` and the user `trillian`, then the server would accept a message with the `to` field: `to zaphod@univer.ze,trillian@earth.planet`, and respond with `ok 1` (one recipient was accepted). If the `to` field contains an address with a domain the mailbox server manages, but the user is unknown to the server (i.e., no mailbox exists), the server should respond as described in Section 2.2, Example 3.

Message IDs When the mailbox server has received and accepted a message via DMTP server, the message must be assigned an id so it can be accessed by the recipient. How the message IDs are assigned is not explicitly specified, but when accessing a message with `show <id>`, it should not be possible for a user to read emails from other users!

2.6 Monitoring Server

The purpose of the **monitoring server** is to receive and display usage statistics of the outgoing traffic of transfers servers. Specifically, it records the amount of messages sent from specific servers and users. This is useful for analyzing message throughput of individual servers and users to, e.g., detect server abuse.

The monitoring server receives usage statistics via a UDP socket. For each mail sent by a transfer server, it sends a UDP packet to the monitoring server containing the transfer server's host and port, and the sender's email address. The packet contains the information as plain text and should be formatted as follows:

```
<host>:<port> <email-address>
```

For example, if the sender `zaphod@univer.ze` sends an email through the transfer server at host `127.0.0.1` and port `1337`, the packet should contain the string `127.0.0.1:1337 zaphod@univer.ze`

The monitoring server provides an interactive command-line interface to access the data. Specifically, it provides the following commands:

- `addresses`

  Lists all addresses and the number of messages they have sent (aggregated over all servers). Example output:

  ```
  zaphod@univer.ze 12                                                    1
  trillian@earth.planet 4                                                2
  ```

- `servers`

  Lists all servers and the number of messages sent over these servers. Example output:

  ```
  10.0.0.1:1337 321                                                      1
  10.0.0.2:1338 512                                                      2
  ```

3 Implementation

For the implementation we provide a basic code template with some dependencies that you should use. Please find more information on how to use it (and what not to modify) in TUWEL[6]. This section will kickstart you into programming, and should answer most of your questions regarding implementation details. We will first go through behavior similar to all three applications.

All three servers should be implemented as runnable Java applications. In particular, your applications should compile with Java 11 and without additional dependencies beyond those provided in the template.

[6] https://tuwel.tuwien.ac.at/mod/book/view.php?id=649459&chapterid=1048

Each server application has its own interface which already has a stub implementation in the template. The stubs also contain a `main` method that can be executed through the `ant` commands. The first and only argument passed to the `main` method is the application's *component id*, e.g., *monitoring* or *transfer-1*. We use the factory pattern and the `ComponentFactory` class to instantiate application components. At startup, the server applications read parameters from their respective `<component-id>.properties` configuration files. You can assume that the parameters are valid and do not have to verify them. The class `dslab.util.Config` can be used to read these files from the classpath.

The next step is to create a `java.net.ServerSocket` instance (or two instances, in case of the mailbox server), and start listening for new connections. For the monitoring, you also need to start a `java.net.DatagramSocket` to send and receive UDP packets. If you haven't already, study the Java sockets tutorial and familiarize yourself with the concepts[7].

Handling client connections When you receive a new socket connection via `ServerSocket.accept()`, you can use the socket's `InputStream` and `OutputStream` to communicate with the client. Refer to the Java socket tutorial on how to read and write from a socket[8]. As you will be using these streams a lot, it makes sense to encapsulate them into a separate object and provide methods to read and write strings via a common interface.

Threading Because all I/O operations (including accepting connections via `ServerSocket.accept()`) on the default socket API are blocking, we need concurrency mechanisms to be able to accept new connections and serve multiple client requests simultaneously. Each new socket connection should therefore be handled in a separate thread. The Java concurrency tutorial is a great source of information on how to write multi-threaded Java programs[9]. Because threads are a relatively expensive resource, we recommend the use thread pools to re-use threads for tasks that are short-lived, e.g., client connections. You can read up on Java's `java.util.concurrent.ExecutorService` interface and thread pool implementations in the respective chapter of the concurrency tutorial[10]. For long-lived threads, such as those running the `ServerSocket` accept loop, instantiating dedicated threads may be preferable. Concurrency is an important concept for distributed systems, and regardless of your implemented solution, during the interviews you have to be able to explain in detail how your threading strategy works and why you have implemented it that way, including its benefits and drawbacks.

Application shell To provide a way to interact with the application itself, for example for administrators to initiate a command to shut down the application, or to implement the monitoring server's commands, you can use the `Shell` class provided by the *Orvell*[11] dependency. The `Shell` is a `Runnable` and a good candidate for blocking the main thread which executes the application's `main` method. It reads from `System.in` and translates console input into commands. You can find an example in our example repository in TUWEL.

Connecting to servers Because your application uses standard TCP and a plain-text protocol, you can use TCP command-line utilities to talk to the servers. For *nix system users, we recommend using netcat, which is already installed on most systems as `nc`. For Windows users, we recommend using PuTTY and running `putty -raw` from the command line. Read the DSLab Handbook Section Tools[12] for further details.

Application shutdown Each application provides a shutdown method that should be invoked when `shutdown` is typed in the console window of the running application. When your application shuts down, you should properly close all resources. Do not simply force your application to end via `System.exit` (this

[7] http://docs.oracle.com/javase/tutorial/networking/sockets/index.html
[8] http://docs.oracle.com/javase/tutorial/networking/sockets/readingWriting.html
[9] http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html
[10] http://docs.oracle.com/javase/tutorial/essential/concurrency/executors.html
[11] http://hyde.infosys.tuwien.ac.at/trausch/orvell
[12] https://tuwel.tuwien.ac.at/mod/book/view.php?id=649459&chapterid=1054

is also why you should not use the `exit` shell command). Closing the `ServerSocket`(s) is a good place to start. It does not disconnect any clients, but the server stops accepting new connections. You should then proceed to terminate all open `Socket` connections. Close any other I/O resources you may be using and shut down all your thread pools. If you use the shell, throw a `StopShellException` which will break the read loop and make `Shell.run()` return gracefully. If your application does not terminate after exiting the `main` method, you may still have some threads running which did not properly terminate.

Concurrent access & thread safety Many parts of your code will be subject to concurrent access and will require you to think about thread safety. Provide thread safety where necessary, but avoid simply declaring each method as `synchronized`. Also, using a concurrent collection from the Java standard library (such as `ConcurrentHashMap`), does not automatically solve all concurrency issues. During the interviews you should be able to explain in detail how concurrent access affects your implementation, and what you did to control it.

3.1 Transfer Server

3.1.1 Parameters

- `tcp.port`: the port used for instantiating the `ServerSocket` (for incoming DMTP connections)
- `monitoring.host`: the name (or IP address) where the monitoring server is running
- `monitoring.port`: the UDP port where the monitoring server accepts packets

3.1.2 Details

Message forwarding Because forwarding a message involves a domain lookup and creating a socket connection to the mailbox server, the forwarding process may take a while. This process should not block the user from sending messages to the server. Instead, the transfer server should handle the message forwarding concurrently, and not in the same thread as the client connection. Therefore, you need to implement a thread synchronization mechanism between the threads that create messages and a thread (or multiple threads) that forward messages. This is an instance of the well known producer–consumer problem[13], and there are several ways to solve it. It is part of the task to think about an appropriate solution. However, you should avoid simply creating new thread for every message.

Domain lookup As stated earlier, each domain is associated with exactly one mailbox server. For this task, the domains lookup will be very simple and involve a static configuration. Specifically, the domains and their designated mailbox servers are encoded in the `domains.properties` files. Simply read from the properties file when starting the transfer server and parse the socket addresses.

Usage monitoring The transfer server sends usage statistics to the monitoring server via UDP. Every time a message was successfully sent to a mailbox server, send a datagram packet to the monitoring port as described in Section 2.6. When instantiating the local `DatagramSocket` for sending packets, omit the port to let the system randomly select an open port.

3.2 Mailbox Server

3.2.1 Parameters

- `domain`: the domain the mailbox server manages (e.g., `univer.ze`)

[13] https://en.wikipedia.org/wiki/Producer-consumer_problem

- `tcp.dmtp.port`: the port used for incoming DMTP connections
- `tcp.dmap.port`: the port used for incoming DMAP connections
- `users`: reference to the properties files that hold the user and password data

3.2.2 Details

Users and passwords Users and passwords are stored for each mailbox server in separate properties files. A reference to the respective file is stored in the `users` property of the mailbox server's properties file. The template provides a file with example users. The `Config` class does not provide ways to parse users directly, instead you have to parse user data yourself. Each user corresponds to the local-part of the corresponding email address. For example, if the mailbox server manages the domain `univer.ze`, the user `zaphod` has the email address `zaphod@univer.ze`.

Mailboxes You do not need to persist messages received by the mailbox server on disk, but you need some in-memory data structure to store messages and associate them with their intended recipients. Whatever data structure you chose to implement, it will be subject to concurrent access. Thread safety is therefore very important here.

3.3 Monitoring Server

3.3.1 Parameters

- `udp.port`: the port used for instantiating the `DatagramSocket`

3.3.2 Details

The monitoring server only operates via UDP and does not require a TCP `ServerSocket`. And because UDP is a connection-less protocol, we do not need the same concurrency mechanisms (i.e., thread-per-connection). It is sufficient to read datagram packets in a loop in one thread and update your data structures that hold the usage statistics directly.

The monitoring server should only provide a command-line interface for administrators to issue some commands. Implement the commands as described in Section 2.6, including an additional `shutdown` command. We encourage you to use the Shell to save some time on boilerplate code.

4 Submission

Upload your project as a ZIP archive to TUWEL before the deadline at 14.11.2019, 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[14].

4.1 Checklist

- ☐ Solution compiles in the lab environment (on a lab PC or server) running `ant clean compile`
- ☐ Programs run in the lab environment with `ant` target commands
- ☐ Upload your solution as ZIP[15] 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[16].

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 (communication patterns, thread synchronization, etc.) and how you solved them
- justify your implementation decisions

[14] https://tuwel.tuwien.ac.at/mod/book/view.php?id=649459&chapterid=1051
[15] If you use git then just run `git archive --format zip --output dslab.zip HEAD` in your repository
[16] https://tuwel.tuwien.ac.at/mod/book/view.php?id=649459&chapterid=1057