OPICHAT — Subject

version #

ASSISTANTS C/UNIX 2022 <assistants@tickets.assistants.epita.fr>

Contents

1 Introduction **6**
1.1 Goal 6

2 Subject Rules **6**

3 OPIChat protocol **8**
3.1 Concepts and terminology 8
3.2 Syntax 8
3.3 Semantic 10
3.3.1 Request 10
3.3.2 Response 10
3.3.3 Notification 10
3.3.4 Error 10

4 Client **12**
4.1 Command line arguments 12
4.2 Handling server input 12
4.2.1 Responses 13
4.2.2 Notifications 13
4.2.3 Errors 13
4.3 Handling user input 13
4.3.1 Step 1 - Get the command 13
4.3.2 Step 2 - Get the parameters 13
4.3.3 Step 3 - Get the payload 14
4.3.4 Example 14

5 Server **15**
5.1 Command line arguments 15
5.2 Handling client input 15
5.2.1 Events 15

*https://intra.assistants.epita.fr

5.2.2 Parsing . . . 16
5.2.3 Process . . . 16
5.2.4 Respond . . . 16

6 Core features **17**
6.1 PING . . . 17
6.1.1 Parameters . . . 17
6.1.2 Error codes . . . 17
6.1.3 Examples . . . 17
6.2 LOGIN . . . 18
6.2.1 Parameters . . . 18
6.2.2 Error codes . . . 18
6.2.3 Examples . . . 18
6.3 LIST-USERS . . . 19
6.3.1 Parameters . . . 19
6.3.2 Error codes . . . 19
6.3.3 Examples . . . 19
6.4 SEND-DM . . . 20
6.4.1 Parameters . . . 20
6.4.2 Error codes . . . 20
6.4.3 Client side . . . 20
6.4.4 Examples . . . 20
6.5 BROADCAST . . . 22
6.5.1 Parameters . . . 22
6.5.2 Error codes . . . 22
6.5.3 Client side . . . 22
6.5.4 Examples . . . 22

7 Additional features **24**
7.1 CREATE-ROOM . . . 24
7.1.1 Parameters . . . 24
7.1.2 Error codes . . . 24
7.1.3 Examples . . . 24
7.2 LIST-ROOMS . . . 26
7.2.1 Parameters . . . 26
7.2.2 Error codes . . . 26
7.2.3 Examples . . . 26
7.3 JOIN-ROOM . . . 27
7.3.1 Parameters . . . 27
7.3.2 Error codes . . . 27
7.3.3 Examples . . . 27
7.4 LEAVE-ROOM . . . 28
7.4.1 Parameters . . . 28
7.4.2 Error codes . . . 28
7.4.3 Examples . . . 28
7.5 SEND-ROOM . . . 29
7.5.1 Parameters . . . 29
7.5.2 Error codes . . . 29
7.5.3 Client side . . . 29
7.5.4 Examples . . . 29

7.6 DELETE-ROOM 31
7.6.1 Parameters 31
7.6.2 Error codes 31
7.6.3 Examples 31
7.7 PROFILE 32
7.7.1 Parameters 32
7.7.2 Error codes 32
7.7.3 Examples 32

8 Testing **33**
8.1 Tools 33
8.1.1 netcat 33
8.1.2 strace 34
8.2 Fuzzing 34

Obligations

*Obligations are **fundamental** rules shared by all subjects. They are non-negotiable and to not apply them means to face sanctions. Therefore, do not hesitate to ask for explanations if you do not understand one of these rules.*

Obligation #0: **Cheating**, as well as sharing source code, tests, test tools or coding-style correction tools is **strictly forbidden** and penalized by not being graded, being flagged as a cheater and reported to the academic staff.

Obligation #1: If you do not submit your work before the deadline, it will not be graded.

Obligation #2: Your submission repository must be **clean**. Except for special cases, which (if any) are **explicitely** mentioned in this document, an *unclean* repository may contain:

- binary files;[1]
- files with inappropriate privileges;
- forbidden files: `*~`, `*.swp`, `*.o`, `*.a`, `*.so`, `*.class`, `*.log`, `*.core`, etc.;
- a file tree that does not follow our specifications.

Obligation #3: All your files must be encoded in ASCII or UTF-8 without BOM.

Obligation #4: When examples demonstrate the use of an output format, you must follow it scrupulously.

Obligation #5: The coding-style needs to be respected at all times.

Obligation #6: **Global variables** are forbidden, unless they are **explicitly** authorized

Obligation #7: Anything that is not **explicitly** allowed is **disallowed**.

Obligation #8: Your code must compile with the flags:

```
-std=c99 -pedantic -Werror -Wall -Wextra
```

Advice

- ▷ Read the *whole* subject.
- ▷ If the slightest project-related problem arise, you can get in touch with the assistants.

  Post to the dedicated **newsgroup** (with the appropriate **tag**) for questions about this document, or send a **ticket** to **<assistants@tickets.assistants.epita.fr>** otherwise.
- ▷ In examples, `42sh$` is our prompt: use it as a reference point.
- ▷ Do **not** wait for the last minute to start your project!

[1] If an executable file is required, please provide its sources **only**. We will compile it ourselves.

1 Introduction

1.1 Goal

The goal of the *OPIChat* project is to learn the basics of network programming, following the client/server paradigm. You will have to implement a chat server following a simple protocol whose specifications are described in this document. You will also have to implement a client able to communicate with the said server.

You will learn to manipulate different kernel objects: sockets, polling event loops and threads.

2 Subject Rules

Files to submit:

- ./Makefile
- ./src/*
- ./tests/*

Provided files:

- ./opichat_server
- ./opichat_client

Makefile: Your makefile should define at least the following targets:

- all: Produces both opichat_server and opichat_client binaries and optionally your opichat_fuzzer binary if you implemented it
- opichat_client: Produces the opichat_client binary
- opichat_server: Produces the opichat_server binary
- check: Runs your testsuite
- clean: Deletes everything produced by make

Forbidden functions: You can use all the functions of the standard C library except:

- fork(2)
- popen(3)
- system(3)
- execl(3)
- execlp(3)
- execle(3)
- execv(3)
- execvp(3)

- execvpe(3)

3 OPIChat protocol

3.1 Concepts and terminology

OPIChat is an instant messaging system following the client/server paradigm. A client is an application that will ask another application, called server, to process some requests. The server should be able to understand client's requests and respond accordingly. It acts as a way to connect multiple clients through a central authority.

The clients and the server communicate using a protocol. Its goal is to define the syntax of messages, their semantic and how they are being exchanged (i.e. when, where...).

In *OPIChat*, clients can send messages to the server. We will call such messages `requests`.

These requests contain a command, indicating the action that needs to be performed by the server.

The server will reply with another message called either `response` or `error` depending on the completion of the request. It provides information about the status of the performed action.

A server might also send unsolicited responses to connected clients to inform them of an event (for example, notify that a client just received a message from another client). Such messages will be called `notifications`.

The *OPIChat* protocol is inspired by HTTP and IRC. We introduce you to a simplified version of these protocols so you do not lose time reading complex specifications.

3.2 Syntax

Requests, responses, notifications and errors in *OPIChat* all follow the same syntax.

They are composed of five main components: payload size, status code, command, command parameters and payload:

- The payload size is the number of bytes of the payload.
- The status code is an integer indicating the type of the message or the possible errors. Every integer has its own meaning:
  - A status code of `0` indicates that the message is a `request`.
  - A status code of `1` indicates that it is a successful `response`.
  - A status code of `2` indicates a `notification` from the server to a client (for example a message from a client to another).
  - A status code of `3` indicates that the message is a response to notify an error while processing a request, such message is called an `error`.
- The command is a string identifying the action performed by the server.
- The command parameters specify special requirements for a command (for example the username of a client you might want to send a message to). There can be any number of parameters in a message, each command as its own required parameters documented in the appropriate sections. Each parameter consists of a key/value pair with the following syntax: `key=value`. Parameters can be in any order in the message.

- Finally, the payload is simply the raw content of the request or response (for example the content of a message you would send to another client).

Each of theses elements are separated by a newline and there can be any number of parameters. The end of command parameters is marked by an empty line, which therefore represents the start of the payload.

For example the following is a request sent by a user named `ING1` asking the server to send the message `2022` to the user `acu`:

```
4$
0$
SEND-DM$
User=acu$
$
2022
```

In this example and all the others in this document, the `$` character corresponds to a newline.

Such request will cause the server to send a response to the original client indicating the status of the operation. Upon successful completion, the response will look like this:

```
0$
1$
SEND-DM$
User=acu$
$
```

While processing the request, the server will also generate a notification for user `acu` to notify them of a message:

```
4$
2$
SEND-DM$
User=acu$
From=ING1$
$
2022
```

3.3 Semantic

The status codes allow us to know the type of the message.

3.3.1 Request

Requests are indicated by a status code of `0`. They are sent by a client to the server to ask them to run a special action.

A server will always send a response or an error back to the sender to indicate the status of the action.

Parameters and payloads of a request are command-specific.

3.3.2 Response

Responses are indicated by a status code of `1`. A response is sent by the server after receiving and successfully processing a request from a client. The response will always be sent to the client that sent the original request. The response must include the same command and the same parameters as the request it is originated from.

The payload of a response is command-specific.

3.3.3 Notification

A notification is indicated by a status code of `2`. Some requests might generate notifications in addition to a response. They are messages sent by a server to a client without prior request of the said client.

The command of the notification must be the same as the request that triggered it and it must include all parameters of the requests. A notification might add additional parameters in addition to the ones of the request.

The payload of the notifications is command-specific.

3.3.4 Error

An error is indicated by a status code of `3`. They are sent instead of a response to indicate the failure to process a request (invalid request syntax, invalid parameters, internal server failure…).

Errors must have the same command and parameters than the request that triggered them.

The payload of an error is a small sentence indicating the error. The error payloads are error-specific, each command defines its own set of errors.

Be careful!

Two errors in particular have to be treated separately: the invalid syntax and missing parameter errors.

Upon receiving a message that does not respect the syntax described in the previous section, if the command of the request is not recognised, or if the received message is not a request, a server must reply with an error with the `INVALID` command and the payload `Bad request$`.

When receiving a request with a missing required parameter, a server must reply with an error with the payload `Missing parameter$`.

4 Client

Your client must be able to read `stdin` for commands that will be formatted into a valid request before being sent to the server.

It must also read messages from the server and log relevent messages.

Since regular IO on `stdin` or on sockets is blocking, you must use threads to be able to read `stdin` and the socket asynchronously.

You will have a total of 2 threads:

- One for reading `stdin`, generating requests and sending them to the server.
- Another one to receive data from the server, parse it and process it.

To manage threads you will use `pthreads(7)`.

4.1 Command line arguments

Your client will take 2 arguments:

```
./opichat_client <host> <port>
```

Where `<host>` is either the domain name or the IP address of the server you want to connect to and `<port>` is the TCP port you want to connect to.

Invalid command line arguments or failure to connect to the server must result in your client exiting with status code `1` logging a pertinent message on `stderr`.

4.2 Handling server input

You client must continuously read potential data from the server. To handle the problem of blocking IO, you must read data on the socket on a separate thread.

A client might receive responses, notifications or errors from the server. You will have to parse these messages and apply necessary treatment.

This section defines how the client should handle such messages.

Be careful!

If your client receives a message from the server that is not a valid response, notification or error, your client must disconnect from the server and exit with status code 1.

4.2.1 Responses

Upon receiving a successful response from the server (status code `1`) with a non empty payload, a client must log `"< "` followed by the raw payload of the response on `stdout`. Responses with empty payloads must be ignored.

4.2.2 Notifications

Each command defines its own set of notifications. The way to handle a notification is command-specific, they are therefore defined in the appropriate section of this document.

4.2.3 Errors

Upon receiving an error from the server, a client must log `"! "` followed by the raw payload of the error on `stderr`.

4.3 Handling user input

Your client must read data on `stdin` that will allow you to format requests.

Since reading `stdin` is a blocking operation, you must run it in a second thread.

There are three main parts to process `stdin` to be able to format requests:

4.3.1 Step 1 - Get the command

When starting up, your client will ask you the command that you want to send to the server.

It will print `Command:$` on `stdout` and will read a line on `stdin` to retrieve the command.

If the provided command does not exist, your client must print `Invalid command$` on `stderr` and go back to step 1 to ask for the command again.

Once you get a valid command, this is the command that will be sent by your client and you can go to step 2.

4.3.2 Step 2 - Get the parameters

This step must be executed only if the chosen command requires parameters, otherwise, you must skip to step 3.

Your client will print `Parameters:$` on `stdout` and read a line on `stdin`, the line will have the same format as the command parameters: `<name>=<value>$`.

If the line is a syntactically valid parameter, your client must add this parameter to the requests that will be sent and read another line to get the next parameter.

If the line is not a syntactically valid parameter, your client must print `Invalid parameter$` on `stderr`, ignore the line, and read another line to get the next parameter.

If the line is empty, this means the end of parameters, all parameters read until then must be included in upcoming requests and you can go to step 3.

4.3.3 Step 3 - Get the payload

Your client must print `Payload:$` on `stdout` and read a line on `stdin`.

After reading the line, the client must create a request with the said line as the payload (excluding the final line feed), the command from step 1 and the parameters from step 2. The client must then send the request to the server, and if the command is SEND-DM, SEND-ROOM or BROADCAST go back to step 3 to send a new request with the same command/parameters and a new payload. If it is an other command, you must go back to step 1.

If the line is `/quit`, instead of sending a request, your client must discard all commands and parameters previously initialized and go back to step 1 to prepare a new set of command/parameters for upcoming requests.

4.3.4 Example

```
42sh$ ./opichat_client 127.0.0.1 8000
Command:
LOGIN
Payload:
ING1
Command:
< Logged in
SEND-DM
Parameters:
User=acu

Payload:
Spider
Payload:
Best
Payload:
Project
Payload:
/quit
Command:
```

Let us suppose the user acu was connected at the same time, here is what his session might look like.

```
42sh$ ./opichat_client 127.0.0.1 8000
Command:
LOGIN
Content:
acu
Command:
< Logged in
from ING1: Spider
from ING1: Best
from ING1: Project
```

5 Server

Your server will wait for incoming client connections, read their requests and respond accordingly.

The server must be able to handle multiple clients simultaneously so you will have to use polling with `epoll` to solve the problem of blocking IO.

To get started with `epoll`, see `epoll(7)`.

5.1 Command line arguments

The server will take arguments similar to the client:

```
./opichat_server <ip> <port>
```

Where `<ip>` and `<port>` corresponds to the IP address and TCP port your server will bind to.

If the command line arguments are invalid or if the server failed to bind, your server must exit with status code 1 and log a pertinent message on `stderr`.

5.2 Handling client input

5.2.1 Events

Your server must continuously listen for incoming connections from clients. When a client is connected, it can send data to the server.

You therefore have to handle 2 types of events: client connections and client data. You must be careful about blocking IO because a call to `accept(2)`, `recv(2)` or `read(2)` without any connections or data will block until there are connections or data to process, thus blocking you from handling multiple clients at the same time.

To handle this problem you must use `epoll` to create an event loop and be able to know when a socket has data to process.

When receiving a connection, your client must accept it before being able to receive data from the client.

Tips

Do not forget that a client might send you multiple requests on the same socket, or your server might not receive the entire request with a single call to `recv(2)`, you will have to handle such cases and not consider them invalid. You can for example keep previously received data from the connection in the `epoll_event` struct to be able to retrieve it and update it between each event of the same connection.

5.2.2 Parsing

When receiving raw data from a client, your server must then parse it into a request message.

Be careful!

Unlike the client, you cannot just exit if a received message is invalid. On the server, clients might send you malformed messages, you will have to handle such messages by responding with the appropriate error. If the said error is a `Bad request$` your server must disconnect the client after sending the error.

5.2.3 Process

After successfully parsing a request, the server must process it.

This includes modifying the internal state of the server, generating and sending appropriate notifications, and generating the response, which might be an error if the request could not be handled properly.

5.2.4 Respond

Once the server has generated the response, it must serialize it into raw data and send it to the client that sent the original request.

6 Core features

This section defines the commands you have to implement.

Tips

Unless stated otherwise and if not specified, payloads and parameters can be ignored.

6.1 PING

The `PING` command does not require any payload in the request. The server must simply respond with `PONG$` in the payload.

6.1.1 Parameters

The `PING` command does not require any parameter.

6.1.2 Error codes

No specific error codes are defined for the `PING` command.

6.1.3 Examples

Request

```
0$
0$
PING$
$
```

Response

```
5$
1$
PING$
$
PONG$
```

6.2 LOGIN

The `LOGIN` command is used by a client to assign itself a user name.

The user name is sent in the payload of the request and consists solely of alpha-numerical characters.

Upon succesfull completion, the server must send `Logged in$` in the response.

6.2.1 Parameters

The `LOGIN` command does not take any parameters.

6.2.2 Error codes

Bad username

This error is sent when the provided user name is invalid (i.e. it contains non alpha-numerical characters or empty user name).

- Payload: `Bad username$`

Duplicate username

This error is sent when the provided user name is already used by another client.

- Payload: `Duplicate username$`

6.2.3 Examples

Request

```
3$
0$
LOGIN$
$
acu
```

Response

```
10$
1$
LOGIN$
$
Logged in$
```

6.3 LIST-USERS

The `LIST-USERS` command lists all currently connected and logged in users.

Users are outputted in the payload of the response by outputting each user name followed by a new-line. Client that are not logged in are not listed.

User names are ordered by connection date in ascending order.

6.3.1 Parameters

The `LIST-USERS` command does not require any parameter.

6.3.2 Error codes

No specific error codes are defined for the `LIST-USERS` command.

6.3.3 Examples

Request

```
0$
0$
LIST-USERS$
$
```

Response

```
15$
1$
LIST-USERS$
$
acu$
Hoppy$
ING1$
```

6.4 SEND-DM

The `SEND-DM` command asks the server to send a direct message to the user specified in parameter. Such direct message is represented by a notification containing the same payload as the request.

Upon completion, the server must send a response with an empty payload.

6.4.1 Parameters

User

Represents the user name of the recipient of the message. Required in request, response and notification.

From

Represents the user name of the sender of the message. Required in notification.

If the sender is not logged in, the `From` parameter must take the value `<Anonymous>`.

6.4.2 Error codes

User not found

This error is sent when the specified user could not be found.

- Payload: `User not found$`

6.4.3 Client side

Upon receiving a `SEND-DM` notification from the server, a client must log the message on `stdout` with the following format:

```
From <user>: <payload>$
```

By replacing `<user>` by the user name of the sender and `<payload>` by the payload of the notification.

6.4.4 Examples

`ING1` sends the direct message `2022` to `acu`:

Request

From `ING1` to the server

```
4$
0$
SEND-DM$
User=acu$
$
2022
```

Response

From the server to `ING1`

```
0$
1$
SEND-DM$
User=acu$
$
```

Notification

From the server to `acu`

```
4$
2$
SEND-DM$
User=acu$
From=ING1$
$
2022
```

6.5 BROADCAST

The broadcast command is used to send a direct message to every user currently connected, logged in or not, excluding the original sender. As for the `SEND-DM`, the message is represented by a notification containing the same payload as the request.

Upon completion, the server must send a response with an empty payload.

6.5.1 Parameters

From

Represents the user name of the sender of the message. Required in notification.

If the sender is not logged in, the `From` parameter must take the value `<Anonymous>`.

6.5.2 Error codes

No specific error codes are defined for the `BROADCAST` command.

6.5.3 Client side

Upon receiving a `BROADCAST` notification from the server, a client must log the message on `stdout` with the following format:

```
From <user>: <payload>$
```

By replacing `<user>` by the user name of the sender and `<payload>` by the payload of the notification.

6.5.4 Examples

Request

```
4$
0$
BROADCAST$
$
2022
```

Response

```
0$
1$
BROADCAST$
$
```

Notification

```
4$
2$
BROADCAST$
From=ING1$
$
2022
```

7 Additional features

This section defines more advanced commands that will for example allow you to create chat rooms.

7.1 CREATE-ROOM

The `CREATE-ROOM` command is used by a client to create a room.

The room name is sent in the payload of the request and consists solely of alpha-numerical characters.

The user making the request is marked as the owner of the room.

Upon completion, the server must send a response with `Room created$` in the payload.

7.1.1 Parameters

The `CREATE-ROOM` command does not take any parameters.

7.1.2 Error codes

Bad name

This error is sent when the provided room name is invalid (i.e. it contains non alpha-numerical characters or empty room name).

- Payload: `Bad room name$`

Duplicate name

This error is sent when trying to create a room when another room already owns the same name.

- Payload: `Duplicate room name$`

7.1.3 Examples

Request

```
8$
0$
CREATE-ROOM$
$
FlagRoom
```

Response

```
13$
1$
CREATE-ROOM$
$
Room created$
```

7.2 LIST-ROOMS

The `LIST-ROOMS` command lists all active rooms.

Rooms are outputted in the payload of the response by outputting each room name suffixed by a newline.

Room names are ordered by creation date in ascending order.

7.2.1 Parameters

The `LIST-ROOMS` command does not require any parameter.

7.2.2 Error codes

No specific error codes are defined for the `LIST-ROOMS` command.

7.2.3 Examples

Request

```
0$
0$
LIST-ROOMS$
$
```

Response

```
24$
1$
LIST-ROOMS$
$
CISCO$
LABSR$
MIDLAB$
SM14$
```

7.3 JOIN-ROOM

The `JOIN-ROOM` command is used by a client to subscribe to a room.

A user subscribed to a room will receive all messages sent to this room.

The room to join is indicated in the payload of the request.

Upon completion, the server must send `Room joined$` in the payload.

7.3.1 Parameters

The `JOIN-ROOM` command does not require any parameter.

7.3.2 Error codes

Room not found

This error is sent when the specified room could not be found on the server.

- Payload: `Room not found$`

7.3.3 Examples

Request

```
8$
0$
JOIN-ROOM$
$
FlagRoom
```

Response

```
12$
1$
JOIN-ROOM$
$
Room joined$
```

7.4 LEAVE-ROOM

The `LEAVE-ROOM` command is used by a client to unsubscribe from a room.

The room to leave is indicated in the payload of the request.

Upon completion, the server must send `Room left$` in the payload.

7.4.1 Parameters

The `LEAVE-ROOM` command does not require any parameter.

7.4.2 Error codes

Room not found

This error is sent when the specified room could not be found on the server.

- Payload: `Room not found$`

7.4.3 Examples

Request

```
8$
0$
LEAVE-ROOM$
$
FlagRoom
```

Response

```
10$
1$
LEAVE-ROOM$
$
Room left$
```

7.5 SEND-ROOM

The `SEND-ROOM` command asks the server to send a message to a room and therefore, all users who joined it except the original sender.

Upon completion, the server must send a response with an empty payload.

7.5.1 Parameters

Room

Represents the destination room name. Required in request, response and notification.

From

Represents the user name of the sender of the message. Required in notification.

If the sender is not logged in, the `From` parameter must take the value `<Anonymous>`.

7.5.2 Error codes

Room not found

This error is sent when the specified room could not be found on the server.

- Payload: `Room not found$`

7.5.3 Client side

Upon receiving a `SEND-ROOM` notification from the server, a client must log the message on `stdout` with the following format:

```
From <user>@<room>: <payload>$
```

By replacing `<user>` by the user name of the sender, `<room>` by the name of the room and `<payload>` by the payload of the notification.

7.5.4 Examples

`ING1` sends the message 2022 to the `FlagRoom`:

Request

From `ING1` to the server

```
4$
0$
SEND-ROOM$
Room=FlagRoom$
$
2022
```

Response

From the server to `ING1`

```
0$
1$
SEND-ROM$
Room=FlagRoom$
$
```

Notification

From the server to every user who joined `FlagRoom`

```
4$
2$
SEND-ROOM$
Room=FlagRoom$
From=ING1$
$
2022
```

7.6 DELETE-ROOM

The `DELETE-ROOM` Deletes the room specified in the payload.

The deletion is possible only if the user making the request is the owner of the room.

Upon completion, the server must send `Room deleted` in the payload.

7.6.1 Parameters

The `DELETE-ROOM` command does not require any parameter.

7.6.2 Error codes

Room not found

This error is sent when the specified room could not be found on the server.

- Payload: `Room not found$`

Unauthorized

This error is sent if the user is not the owner of the specified room

- Payload: `Unauthorized$`

7.6.3 Examples

Request

```
8$
0$
DELETE-ROOM$
$
FlagRoom
```

Response

```
13$
1$
DELETE-ROOM$
$
Room deleted$
```

7.7 PROFILE

The `PROFILE` command lists different information about the user making the request.

The payload of the response follows the following syntax:

```
Username: <name>$
IP: <ip>$
Rooms:$
<room>$*
```

`<name>` corresponds to the user name of the client making the request.

`<ip>` is the IP address of the client making the request.

`<room>` is the name of a room joined by the client making the request. Every room joined by the user must be listed by room creation date in ascending order and suffixed by a newline.

7.7.1 Parameters

The `PROFILE` command does not require any parameter.

7.7.2 Error codes

No specific error codes are defined for the `PROFILE` command.

7.7.3 Examples

Request

```
0$
0$
PROFILE$
$
```

Response

```
50$
1$
PROFILE$
$
Username: acu$
IP: 127.0.0.1$
Rooms:$
CISCO$
FlagRoom$
```

8 Testing

8.1 Tools

8.1.1 netcat

`netcat` can be used to send raw requests to your server.

It can act as a client and you will have to format manually the output:

```
42sh$ nc 127.0.0.1 8000 | cat -e
0
0
PING

5$
1$
PING$
$
PONG$
```

or also:

```
42sh$ echo -ne '0\n0\nPING\n\n' | nc 127.0.0.1 8000 | cat -e
5$
1$
PING$
$
PONG$
```

It is a really useful to manually test all kind of requests. You can for example use it interactively to test if your server can handle incomplete requests.

You could also pipe echo into `netcat` to try to send arbitrary bytes.

8.1.2 strace

As you know, strace is a tool that allows you to trace syscalls in a program.

It is one of the most useful tool to debug your client and server since you will be able to see exactly which syscall fails with which parameters.

8.2 Fuzzing

Fuzzing is a testing technique that aims to provide unexpected/pseudo-random/corrupted data to a program and monitor for potential bugs such as memory leaks or invalid memory access (SEGFAULT, buffer overflows…). The goal is to test the robustness of your server to make sure it will not crash on malformed requests and to watch for potential security vulnerabilities that may be caused by invalid memory access.

Many fuzzing techniques exist.

Some require an external program to provide data to your program through the expected input method (files, sockets, stdin…). This is very useful to test a server for example. This approach is similar to a functional testsuite.

An other method consists to build an executable that will link to your code and will provide data directly by calling your functions with corrupted data. This approach is more similar to unit testing, where you call directly your code instead of testing it from outside.

We strongly recommend you to create a fuzzer for *OPIChat*. In a real life scenario, such a system might be provided malformed input that may exploit some vulnerability in your code. It is essential to watch for those vulnerabilities in such a critical application. In addition to looking for vulnerabilities, it is also a very efficient and easy way to make sure your code will behave normally on all kind of input, which is way more efficient than writing separate tests for every possible malformed input.

Be careful!

A fuzzer won't replace a regular testsuite, the main goal of the fuzzer is to monitor for crashes and/or invalid memory usage, not to check if the actual output is the expected one.

You want to test any part of your code that will be exposed to unexpected, user controlled data. It will likely be mainly your parser and request processor.

You are free to use the method and libraries of your choice to implement your fuzzer but we recommend you to use libFuzzer, a fuzzing library from the LLVM project.

`libFuzzer` is only available on `clang`, to build the fuzzer, you simply need to provide a fuzz target, which is a function that defines what to do with some data that will be provided by the fuzzing engine.

Of course the target depends on your implementation but it might look something like this:

```c
#include <stdlib.h>
#include <stdint.h>

#include "message/message.h"

extern int LLVMFuzzerTestOneInput(uint8_t *data, size_t size)
{
```

(continues on next page)

(continued from previous page)

```
    struct message *message = parse_message((char *)data, size);

    if (message)
        free_message(message);

    return 0;
}
```

Going further...

Such a fuzz target will only test your parser, adapt it to match the fuzz target with your implementation and fuzz other parts of your code. You can for example fuzz your request processor.

Once the fuzz target is defined, you can simply compile it and link it with the relevant object files compiled from your code and add the `-fsanitize=fuzzer` flag in your linking and compile flags. This flag will add the main function from `libFuzzer` that will continuously run your fuzz target by providing it with corrupted data.

Be careful!

Do not forget to change your compiler from `gcc` to `clang` if necessary. Also make sure to compile all the object files of your fuzzer with both `-fsanitize=fuzzer` and `-fsanitize=address` compile flags, otherwise, you will not benefit from the libFuzzer coverage instrumentation, or from the memory leaks/invalid memory access from `fsanitize=address`.

Going further...

It is then possible to provide a corpus of data to `libFuzzer` that will give it examples of correctly formatted input that it will try to modify while fuzzing. To use such a corpus, you can refer to the documentation of `libFuzzer`.

It is my job to make sure you do yours.