Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano

Prof. Matteo Camilli, Elisabetta Di Nitto, and Matteo Rossi

20133 Milano (Italia)
Piazza Leonardo da Vinci, 32
Tel. (39) 02-2399.3400
Fax (39) 02-2399.3411

Software Engineering 2 – Written Exam 2 (WE2)

June 14, 2024

Last Name, First Name

Id number (Matricola)

Number of paper sheets you are submitting as part of the exam

Notes

A. You must write your name and student ID (matricola) on each piece of paper you hand in.
B. You may use a pencil.
C. Incomprehensible handwriting is equivalent to not providing an answer.
D. The use of any electronic apparatus (computer, cell phone, camera, etc.) is strictly forbidden, except for an ebook reader.
E. The exam is composed of 2 parts, one focusing on requirements, and one focusing on design. Read carefully all points in the text.
F. Total available time for WE2: 1h and 30 mins

System Description: ChirpNet

ChirpNet is a social network that looks like Twitter (or X). After registration and authentication, users can *follow* other existing users to see messages posted by them. The *following* relation is asymmetric, meaning that if *A* follows *B*, *B* does not necessarily follow *A*. Users can publish new messages to their followers. Every message is a string with the following format: *"From: <sender>; message body"*. Users can view messages posted by the people they follow through their own *home timelines* that show recent posts (last few days) ordered by date and time. Users can also send direct messages to other users. In this case, the sender posts a message with a body starting with the special prefix *@receiver*, where *receiver* is the username of the recipient of the direct message. The direct message is not delivered if the recipient does not exist. Users can see the list of direct messages by browsing a dedicated inbox, rather than the home timeline.
Finally, there are *bots*. Bots are like users, but they are controlled by software rather than humans. Bots can have followers and they can post messages like nominal users. Furthermore, they answer (arbitrarily) pre-defined commands when users send them a direct message. For instance, let us consider a weather forecast bot called *meteo*. If user *musk* sends it a direct message such as `"From: musk; @meteo Milano tomorrow"`, the bot may answer back with `"From: meteo; @musk Milano tomorrow – isolated thunderstorms"`.
On ChirpNet, bots can be made available by users who hold special *"developer" privileges*. These privileges are granted exclusively by administrators, who evaluate user requests submitted through specific forms. The development of the bots is not handled by the platform. Developers can create their own bots and then use the platform to register and make them available in ChirpNet. When a user adds a new bot to the platform, ChirpNet generates an API key, that is, a unique identifier for that bot. The API key, along with a password defined by the user at bot registration time, must be used to authenticate the bot during the interaction with ChirpNet.

Part 1 Requirements (7 points)

RASD_Q1 (4 points)

Consider the following selected goals related to the interaction with bots.

- **G1**: users with developer privileges want to make a new bot available for other users.
- **G2**: users want to interact (send and receive messages) with existing bots created by other users.

1) Define all *relevant* phenomena involved in the achievement of the two goals. Group the phenomena by category: world-only, shared world-controlled, and shared machine-controlled.

2) Define a *minimal* set of requirements and domain assumptions that are *necessary* to achieve them.

RASD_Q2 (3 points)

Use a UML Class Diagram to describe the domain of ChirpNet.

Part 2 Design (7 points)

DD_Q1 (3 points)

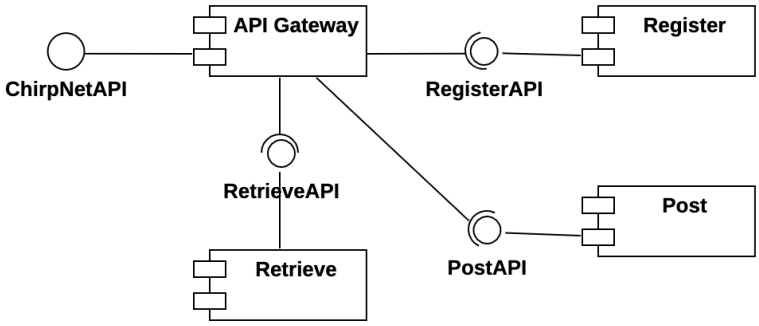
Define the programming interface offered by ChirpNet to realize the *world-controlled* shared phenomena you defined in question RASD_Q1.

Interface definition shall include:

- Name of the operation and brief description
- Arguments and corresponding types
- Return type
- Mapping to the corresponding shared phenomena

DD_Q2 (4 points)

Consider the following high-level UML Component Diagram, which shows a first draft of selected components and interfaces of ChirpNet.



The *API Gateway* is the component offering the interface defined at point DD_Q1. The *Register* component is used to save/collect relevant information about users and bots, while *Post* and *Retrieve* are two components in charge of issuing new posts and retrieve posts, respectively.

1) Focus on the interaction between users and bots and complete the diagram with any *component* and/or *interface* needed to support this interaction. Justify your choices.

2) Define a UML sequence diagram to illustrate an example of a direct message from a user to a registered bot and corresponding answer.

Note: The sequence diagram must be consistent with the component diagram, the interface defined in DD_Q1 and the shared phenomena defined in in RASD_Q1.

Solution
RASD_Q1
Phenomena
World only
W1: Developer defines a set of commands for a bot
W2: Developer develops a bot
W3: Developer deploy a bot
W4: Bot performs the computation it has been developed for

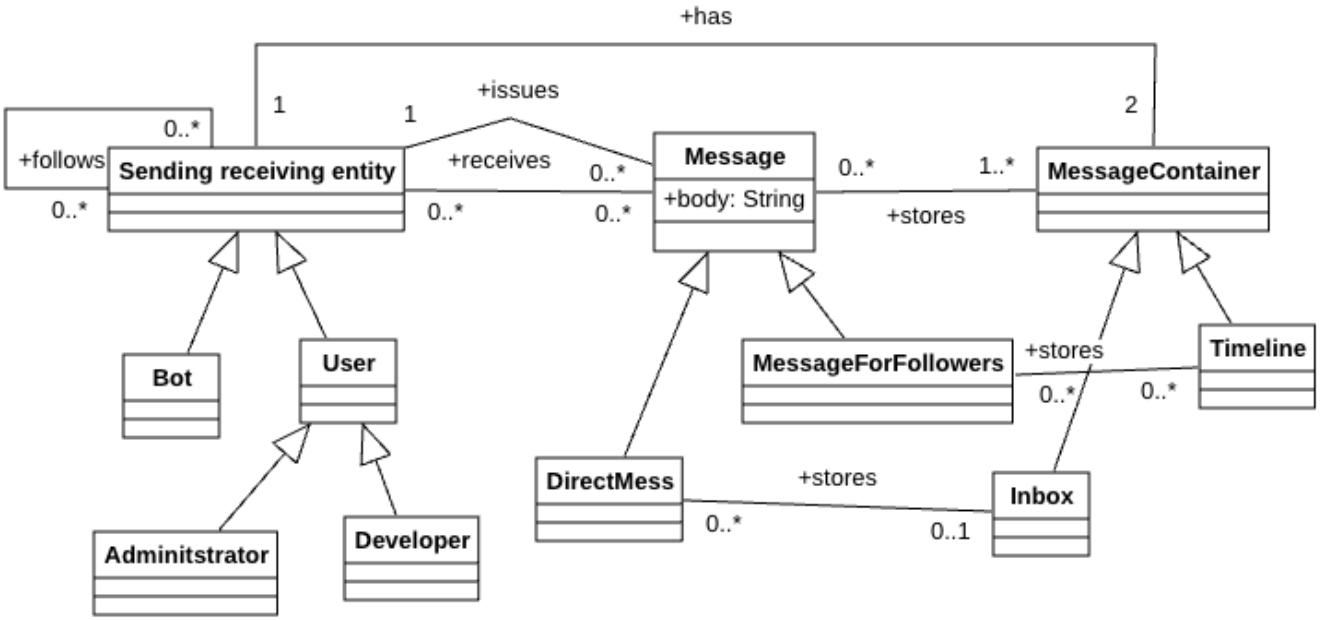
Shared world-controlled
SWC1: Developer selects the register bot functionality
SWC2: Developer registers a bot by inserting its name, password and the set of commands the bot offers
SWC3: User issues a message "From: userName; @botName command parameter"
SWC4: botName issues message "From: botName; @userName reply"
SWC5: botName or user requests the messages available in its inbox

Shared machine-controlled
SMC1: ChirpNet returns an API key for a bot
SMC2: ChirpNet passes a message "From: username; @botName command parameter" to botName
SMC3: ChirpNet passes an answer "From: botName; @userName reply" to userName

Requirements
R1: ChirpNet shall allow developers to register bots by inserting bot name, password and the set of commands the bot offers.
R2: ChirpNet shall compute and return API keys upon user registration.
R3: ChirpNet shall allow bots to authenticate using their API keys and passwords.
R4: ChirpNet shall allow users and bots to issue messages with the syntax "From: sender; @recipient message body".
R5: ChirpNet shall keep track of active bots.
R6: ChirpNet shall identify recipients of a message by interpreting the content of the message itself.
R7: ChirpNet shall include direct messages in the recipients' inboxes.
R8: ChirpNet shall allow users and bots to obtain the messages available in their respective inboxes.

Domain assumptions
DA1: Users are aware of the bot names and commands they want to interact with
DA2: Developers register into ChirpNet the bots they want to make available to the public
DA3: Bots are available
DA4: Bots answer to all commands they offer without crashing or hanging

RASD_Q2



DD_Q1

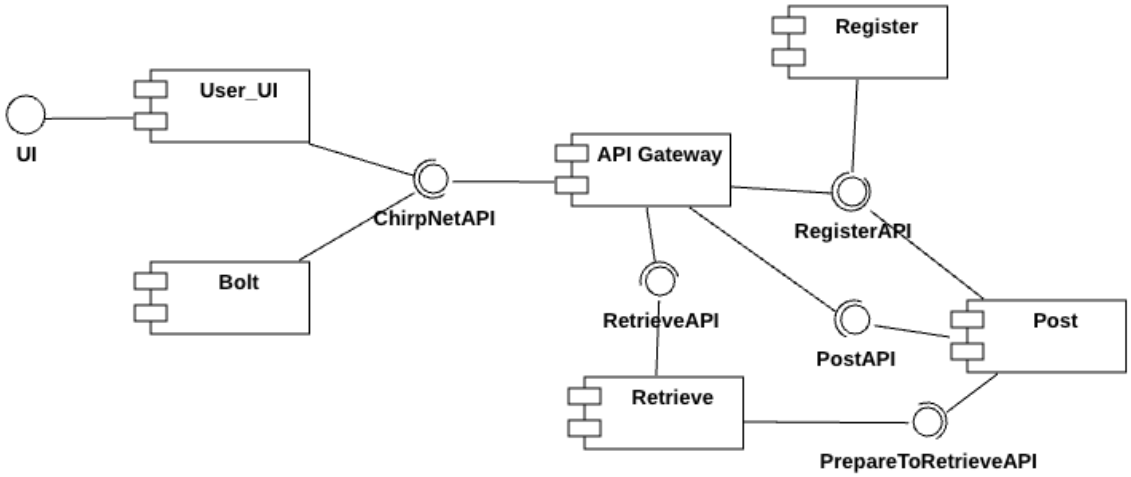
`registerBot(botName: String, botPassw: String, commands: Document): API_key`
This is the operation offered by ChirpNet to allow developers registering bots. It receives the bot name, password and a document defining the bots commands. It returns an API_key. This maps to SWC1 and SWC2.

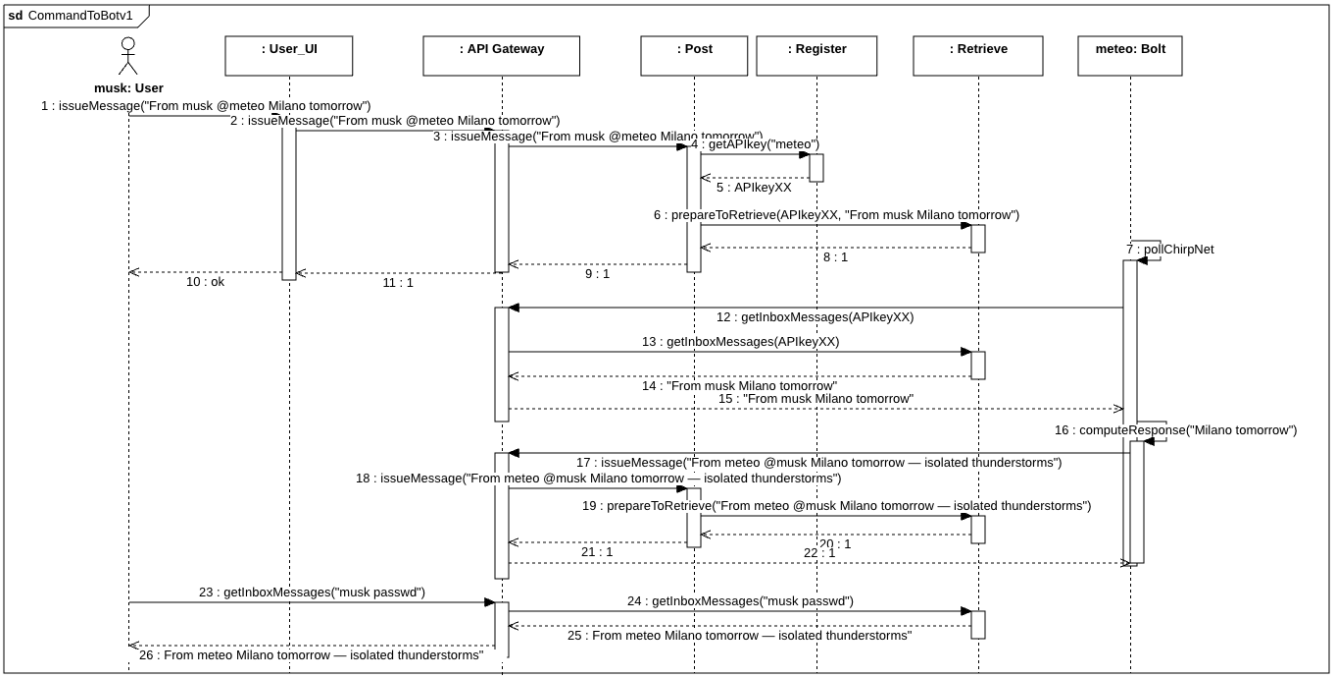
`issueMessage(messageString: String): int`
This is the operation offered by ChirpNet to allow users and bots to issue a message. In case of bots the message is a reply to a request, but this is not relevant to ChirpNet. The parameter is a string with the proper syntax. It returns positive integer if successful, -1 otherwise. This maps to SWC3 and SWC4.

`getInboxMessages(senderInfo: String): String[]`
`getInboxMessages(senderInfo: API_key): String[]`
The first operation allows users, recognized through a string which combines username and password, to retrieve messages from their inbox. The same operation does the same receiving an API_key as parameter. In both cases, the return value is an array of strings where each string is a message. This maps to SWC5.

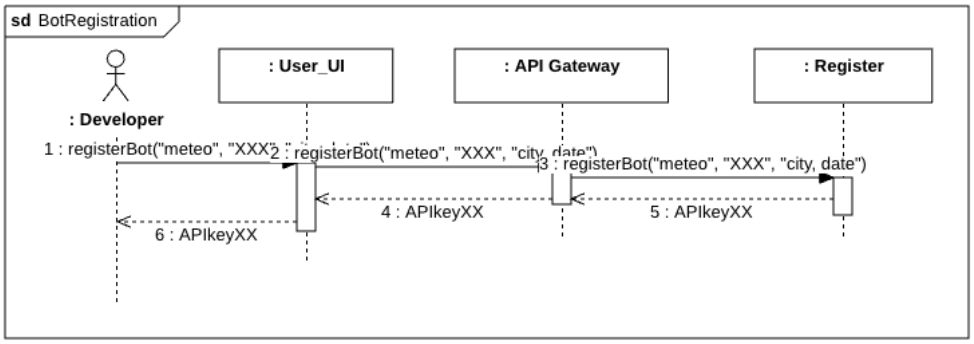
DD_Q2
We can envisage multiple solutions. We show here two of them with significantly different characteristics. For clarity, we have assigned to all interfaces specific names that characterize their role.

Version 1: the main point concerns the paradigm that defines the interaction between ChirpNet and a bot. In this first version we assume that bots poll periodically the system, through the ChirpNetAPI offered by the API Gateway, to obtain messages directed to them. Given the different roles of the components Post and Retrieve, we should assume that the two interact with each other so that posted messages can be retrieved. In the following component diagram, we assume that, as soon as any message is received, Post passes it to Retrieve through the PrepareToRetrieveAPI that we have introduced between the two. The sequence diagram below clarifies how the whole system works.





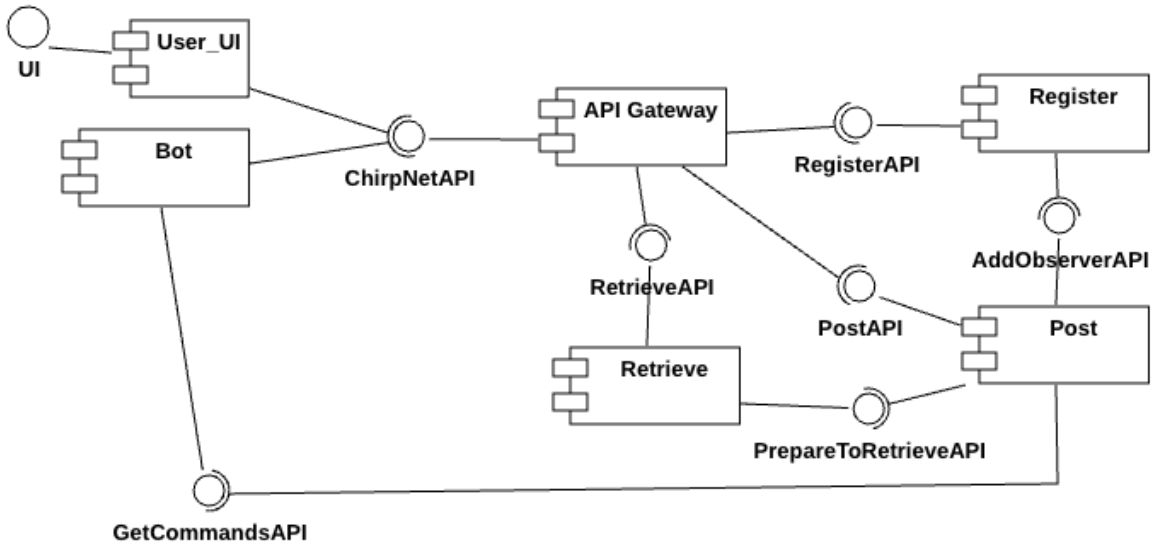
To keep the diagram above less crowded, the registration of a bot is shown in the following separated sequence diagram.

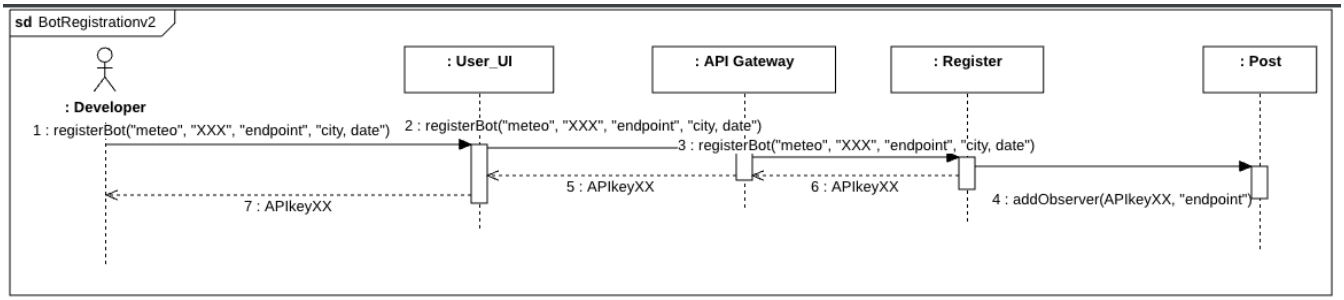


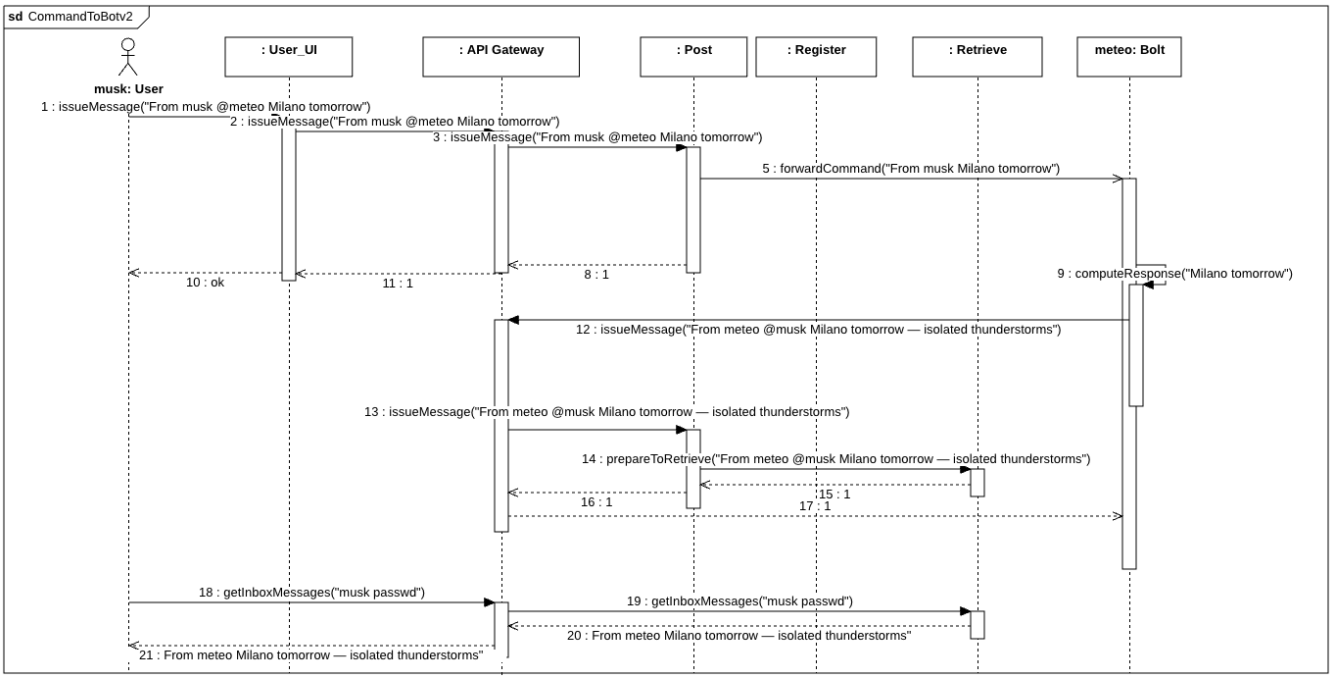
Version 2: In this second version we implement the interaction with the bot according to a push approach: upon registration, the developer specifies also the endpoint of the bot. To this end, the registerBot operation is modified as follows:

```
registerBot(botName: String, botPassw: String, endpoint: String, commands: Document): API_key
```

The endpoint is passed by the Register to the Post component through the AddObserverAPI defined between these two components. Every time a new message for the bot arrives, the Post component transfers it directly to the proper bot endpoint were the interface GetCommandsAPI is offered. The sequence diagrams clarify how the whole system works. As shown in the second diagram, the interaction with User_UI remains a pull one. We could move also in this case into a push solution, but we would need to assume that a UI component associated to each individual user is always active to receive any direct message.






sd CommandToBotv2
musk: User
: User_UI
: API Gateway
: Post
: Register
: Retrieve
meteo: Bolt
1 : issueMessage("From musk @meteo Milano tomorrow")
2 : issueMessage("From musk @meteo Milano tomorrow")
3 : issueMessage("From musk @meteo Milano tomorrow")
5 : forwardCommand("From musk Milano tomorrow")
8 : 1
9 : computeResponse("Milano tomorrow")
10 : ok
11 : 1
12 : issueMessage("From meteo @musk Milano tomorrow — isolated thunderstorms")
13 : issueMessage("From meteo @musk Milano tomorrow — isolated thunderstorms")
14 : prepareToRetrieve("From meteo @musk Milano tomorrow — isolated thunderstorms")
15 : 1
16 : 1
17 : 1
18 : getInboxMessages("musk passwd")
19 : getInboxMessages("musk passwd")
20 : From meteo Milano tomorrow — isolated thunderstorms"
21 : From meteo Milano tomorrow — isolated thunderstorms"