Lab 12

Push technology

[Compulsory]

Authors: Peter Larsson-Green and Johannes Schmidt 2018-2022

In this lab we introduce a firebase database as an example of push-technology.

12.1 Introduction

Push technology, a.k.a *server push*, is a way of communication where messages can be sent from server to client without the client explicitly initiating a communication. The server can *push* messages to the client. The only precondition is that the client is *subscribed* to the server's pushes. This way of communication is different from the classical server-client model such as e.g. HTTP: the client opens a connection, sends a request, the server answers the request, the connection is shut down (i.e., the server can not send any data to the client on its own, the client always needs to start the process).
Google's firebase database synchronizes automatically among all clients that are subscribed. It is thus a piece of push technology. Any changes in the database done by any client are automatically pushed to all subscribed clients. There is a python module `firebase-admin` that renders it very easy to use a firebase database from within python. We want to benefit from this and implement a chat client in functionality very similar to lab 8. The difference lies in the implementation, we don't use sockets and TCP/IP, but a firebase database. A side effect is that we don't need to implement a chat server, as this part will be done by the firebase database automatically.

12.2 Set-up

Before we get started with python, we need to set up a firebase database. Here is how you can achieve this:

1. You need to have a google account. If you don't have one, you need to create one (be it just for the purpose of this lab).
2. Go to `https://firebase.google.com/`, click *sign in* in the upper right corner, and login with your google account.
3. Click *Get started*.
4. Click *add project*, enter a Project name of your choice, click *continue*.
5. Uncheck *Enable Google Analytics for this project*, click *Create project*.
6. ...wait, then click *continue* and your project is created.
7. On the left menue click on *Realtime Database* (this might just be an icon, but if you hover over the icons, the text will be displayed).
8. Click *create database*, then *Next*.
9. Select *Start in test mode*, click *Enable* and your database is created with the settings we need.

In order to be able to subscribe to the database from python, we need two things:

1. your database's url
2. a service account key, as .json file

The url should be displayed at the top of your database (e.g. something like *https://johannes-test-2021-default-rtdb.firebaseio.com/*).
The service account key can be obtained as follows. Click on *Project settings* (menue on the left, possibly reduced to icons), then click the tab *Service accounts*. Select *Python*, then click *Generate new private key*. In the pop-up window click on *Generate key* and the .json file is downloaded. Save it into the folder of your python code (most convenient). Possibly give it a shorter name than the suggested default name.

You are now ready to use the database in python. You only need to install the module `firebase-admin`. You can do this form the command line with

the command `pip install firebase-admin`, or possibly adding the option `--user` if you can not obtain admin rights.

Note: to use the module in python, you need to replace the dash (-) with the underscore (_), e.g., `import firebase_admin`.

12.3 Your task

Implement a gui chat client using a firebase database. The entries in the database should simply represent chat messages. When the program starts, it subscribes to the database and consequently receives the whole database's contents with a giant push from the server. The messages should be displayed appropriately. Any further pushes of entries should be received and displayed accordingly. Sending a chat message translates to adding an entry to the database.

Numerous hints

1. You need to import `firebase_admin` and from there also `db`:

   ```
   import firebase_admin
   from firebase_admin import db
   ```

2. You create a database object linked to your firebase database with

   ```
   cred = firebase_admin.credentials.Certificate(your_json_file_name)
   firebase_admin.initialize_app(cred, {'databaseURL':your_database_url})
   ref = firebase_admin.db.reference('/')
   ```

   where `your_json_file_name` is the name of the .json file you downloaded above, and `your_database_url` is your database's url you retrieved above.

3. You can add an entry with

   ```
   ref.push(newData)
   ```

   We suggest, however, that you put the messages under a child tag called `messages`, e.g. with

   ```
   ref.child('messages').push(newMessage)
   ```

We further suggest to represent a message itself by a dictionary with the two keys `name` and `text`, e.g. with

```
newMessage = {'name': 'Johannes', 'text': 'hello world'}
ref.child('messages').push(newMessage)
```

4. You subscribe to pushes regarding the `messages` tag with

   ```
   messages_stream = ref.child('messages').listen(streamHandler)
   ```

   where `streamHandler` is a call-back function the server calls when it wants to push something.

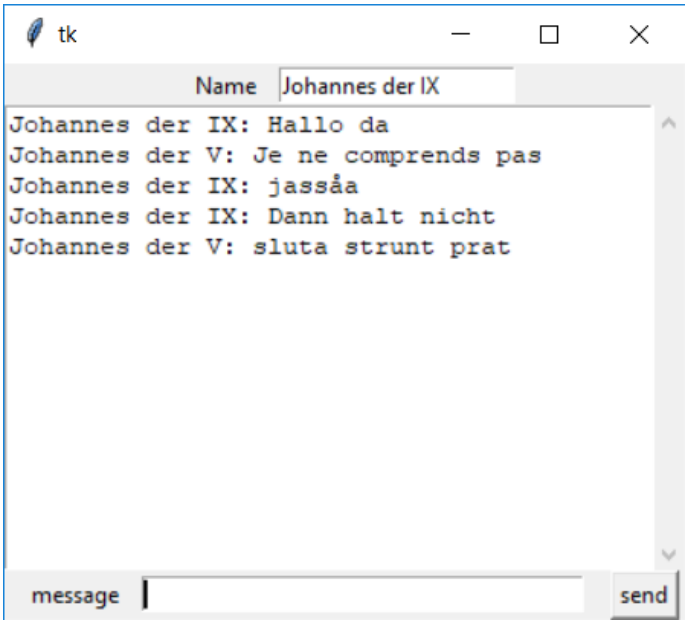
5. You unsubscribe by calling `messages_stream.close()`. Note that this can take sometimes many seconds (so your program might freeze for a while).

6. You receive a server push by a call to `streamHandler()`. From the following example (that you could use as is) you can see what data `incomingData` contains and what its meaning is.

   ```
   def streamHandler(incomingData):
       if incomingData.event_type == 'put':
           if incomingData.path == '/':
               # This is the very first reading just after subscription:
               # we get all messages or None (if no messages exists).
               if incomingData.data != None:
                   for key in incomingData.data:
                       message = incomingData.data[key]
                       handleMessage(message)
           else:
               # Not the first reading.
               # Someone wrote a new message that we just got.
               message = incomingData.data
               handleMessage(message)
   ```

   The function `handleMessage(message)` should extract `name` and `text` from the message and display it appropriately in the gui.

7. You can inspect (and edit) the contents of your database online: when you are in your project, click on *Realtime Database* (possibly reduced to an icon) and it should be displayed.



Figur 12.1: A screen shot of Johannes' gui chat client in action.

8. You can use the code skeleton from lab 8 and modify it according to your needs. For instance, you need not poll for messages. Instead you have your call-back function `streamHandler(incomingData)` to receive server pushes.

9. Figure 12.1 shows a simple example of such a gui chat client. It does not contain a subscribe / unsubscribe button. You are free to add such a button of course.

10. Test your chat client by launching multiple clients simultaneously.

12.4 Examination

Submit your python program on Canvas and present it orally to your lab assistant.