To: Gustavo Soares
DEV-2005#19-SE

GOALS

Implement a simulated telephony application using Python + Twisted.

Different implementations with different complexities are proposed to achieve the goal. Whichever implementation type is chosen, the same set of prerequisites apply.

There are also some bonus points that may be implemented if desired and time allows.

To: Gustavo Soares
DEV-2005#19-SE

PREREQUISITES

1. A clean *CentOS 7* system accessible via SSH.
2. The development may be done on a VM or container running on your own computer.
3. The produced code must be kept in a *Git* repository. It is up to you where the repository will live: in *GitHub*, *GitLab*, in a local directory in your computer, etc.

To: Gustavo Soares
DEV-2005#19-SE

BASIC IMPLEMENTATION

Implement a call center queue application that manages a set of call center operators and delivers received calls to be "answered" by them.

Description

1. Write a simulated call center application. It is called *"simulated"* because it does not actually handle real calls made by softphones or telephone devices. Instead, it handles commands typed in *stdin* and reports their results by printing messages to *stdout*.

2. The application accepts commands like *"call"*, *"answer"*, *"reject"* and *"hangup"*. Those commands try to simulate actions performed by a real softphone or telephone device. (Tip: use the *"cmd"* module of the Python standard library.)

3. The application must accept the commands:

`call <id>`	makes application receive a call whose id is `<id>`.
`answer <id>`	makes operator `<id>` answer a call being delivered to it.
`reject <id>`	makes operator `<id>` reject a call being delivered to it.
`hangup <id>`	makes call whose `id is <id> be finished`.

4. An operator has an id (like "`A`", "`B`", ...) and a state. Its state is *"available"* when idle, *"ringing"* when a call has been delivered but it has not yet been answered by it and *"busy"* when it is answering a call.

5. When the application receives a call it must print to stdout:

   `Call <call id> received`

6. The received call must be delivered to an operator which is available, the application must set the operator state as *"ringing"* and print to stdout:

   `Call <call id> ringing for operator <operator id>`

7. The application must now wait and detect when the operator answers the call, then set the operator state "*busy"* and print to stdout:

   `Call <call id> answered by operator <operator id>`

8. If the operator instead rejects the call, the application must set the operator state as "*available"* again and print to stdout:

   Call <call id> rejected by operator <operator id>

9. The application must detect when an answered call is finished, set the respective operator as *"available"* again and print to stdout:

   Call <call id> finished and operator <operator id> available

10. If no operators are available when a call is received, the call must be put at the end of a queue, left waiting until an operator is available to answer it and the application must print to stdout:

    Call <call id> waiting in queue

11. As soon as an operator becomes available, the application must deliver the call waiting at the *first* position of the queue to be answered by that operator as described in step 6.

12. If a call is not answered by an operator, the application must try delivering it to the next available operator and repeat step 6 until it is answered.

13. If a call is finished before it is answered by an operator, the application must print to stdout:

    Call <call id> missed

Tests

1. Configure the application to have 2 operators whose ids are *A* and *B*.

2. Type *"call 1"* to start a call and you should see:

   Call 1 received
   Call 1 ringing for operator A

3. Type *"answer A"* for operator *A* to answer it and you should see:

   Call 1 answered by operator A

4. Type *"hangup 1"* to finish the call and you should see:

   Call 1 finished and operator A available

5. Type *"call 2"* to start a call and you should see:

   ```
   Call 2 received
   Call 2 ringing for operator A
   ```

6. Type *"call 3"* to start another call and you should see:

   ```
   Call 3 received
   Call 3 ringing for operator B
   ```

7. Type *"answer A"* for operator *A* to answer call and you should see:

   ```
   Call 2 answered by operator A
   ```

8. Type *"answer B"* for operator *B* to answer call and you should see:

   ```
   Call 3 answered by operator B
   ```

9. Type *"call 4"* then *"call 5"* and you should see:

   ```
   Call 4 received
   Call 4 waiting in queue
   Call 5 received
   Call 5 waiting in queue
   ```

10. Type *"hangup 3"* to finish call and you should see:

    ```
    Call 3 finished and operator B available
    Call 4 ringing for operator B
    ```

11. Type *"hangup 2"* to finish call and you should see:

    ```
    Call 2 finished and operator A available
    Call 5 ringing for operator A
    ```

12. Type *"answer B"* for operator *B* to answer call and you should see:

    ```
    Call 4 answered by operator B
    ```

13. Type *"reject A"* for operator *A* to reject call and you should see:

    ```
    Call 5 rejected by operator A
    Call 5 ringing for operator A
    ```

14. Type *"call 6"* to start a call and you should see:

    Call 6 received
    Call 6 waiting in queue

15. Type *"hangup 5"* to finish call and see:

    Call 5 missed
    Call 6 ringing for operator A

16. Type *"call 7"* to start a call and you should see:

    Call 7 received
    Call 7 waiting in queue

17. Type *"hangup 7"*, *"hangup 6"* and *"hangup 4"* to finish calls and see:

    Call 7 missed
    Call 6 missed
    Call 4 finished and operator B available

To: Gustavo Soares
DEV-2005#19-SE

ADVANCED IMPLEMENTATION

Extend the basic implementation by splitting it in two processes that work together by communicating via network in a client-server architecture.

Description

1. Start with a working basic implementation.

2. Split its code in two parts: the command interpreter and the call center queue manager.

3. The command interpreter will act as a *client* that receives from *stdin* the very same set of commands as in the basic implementation. Each command is then sent to the server to be handled by the call center queue manager.

4. The call center queue manager will act as a *server* that receives commands from clients. Each command is then handled in the very same way as it was in the basic implementation.

5. The command interpreter code part must be turned into a TCP client using *Python* + *Twisted*. It must connect to server on port `5678` in order to send commands to it. Each valid command typed in *stdin* must be converted to a *JSON* string and sent to server.

6. Commands must be converted to *JSON* according to the table:

`call <id>`	`{"command": "call", "id": "<id>"}`
`answer <id>`	`{"command": "answer", "id": "<id>"}`
`reject <id>`	`{"command": "reject", "id": "<id>"}`
`hangup <id>`	`{"command": "hangup", "id": "<id>"}`

7. *Tip*: the methods `Cmd.cmdloop` of Python `cmd` module and `reactor.run` of Twisted cannot be used simultaneously because both block the program execution. A possibility is to run `Cmd.cmdloop` in another thread using `reactor.callInThread`.

8. The call center queue manager code part must be turned into a TCP server using *Python* + *Twisted*. It must accept connections on port `5678` in order to receive commands from connected clients. Each command must be interpreted as a *JSON* string and handled.

9. After each command is handled, the server must send a response to the client. The response is exactly the same message that would be printed to *stdout* in the basic

implementation. It must be sent as a *JSON* string like this:

```
{"response": "<message>"}
```

10. The client must receive each response, interpret it as a *JSON* string and print its *message* to *stdout* exactly as it would be in the basic implementation.

Tests

1. Configure the call center queue manager to have 2 operators.
2. Run the call center queue manager.
3. Run the command interpreter.
4. Run each and every test described for the basic implementation by typing commands in the *stdin* of the command interpreter process and verifying the results printed to its *stdout*. The results should be exactly the same.

BONUS TASKS

These are tasks that are absolutely not essential but may be done for bonus points if desired and time allows (do none, any or all).

- Implement *timeout* detection. If an operator does not *answer* or *reject* a call within 10 seconds after it is delivered to it, the application must give up, set the operator state as "*available*" again and print to stdout:

  `Call <call id> ignored by operator <operator id>`

- Improve the advanced implementation of the command interpreter: instead of running `Cmd.cmdloop` in another thread, use Twisted to read lines from *stdin* and call `Cmd.onecmd` as each line is read.

- Package the call center queue manager application as a container, the command interpreter as another and publish both to Docker Hub so that anyone can run them.

REFERENCES

1. The Python Standard Library: Support for line-oriented command interpreters
   Suggestion: learn how to implement command interpreters.
   https://docs.python.org/2/library/cmd.html

2. The Python Standard Library: JSON encoder and decoder
   Suggestion: learn how to convert to and from *JSON* in Python.
   https://docs.python.org/2/library/json.html

3. Twisted Network Programming Essentials - Jessica McKellar, Abe Fettig
   Suggested chapters: 1, 2 and 6.
   http://shop.oreilly.com/product/0636920025016.do

4. Twisted documentation: Scheduling tasks for the future
   Suggestion: learn how to run a function after a timeout using Twisted
   http://twistedmatrix.com/documents/current/core/howto/time.html

5. Twisted example: Reading lines from stdin without blocking
   Suggestion: learn how to read lines from *stdin* using Twisted
   https://twistedmatrix.com/documents/15.5.0/_downloads/stdin.py

6. An Introduction to Asynchronous Programming and Twisted
   Suggestion: learn how to use *twistd* to run Twisted programs as daemons.
   http://krondo.com/?p=2345

7. Containerize a Python App in 5 Minutes
   Suggestion: learn how to package a Python application as a container
   https://www.wintellect.com/containerize-python-app-5-minutes/