CZ4013

Distributed Flight Information System

Author:
Pavel Jahoda (N1800740K)

General overview

The Distributed Flight Information System is based on client-server architecture. Both of these programs are written in Python. At the start of the server, the server binds a socket and waits for new connection from the client. A separete method is called to establish connection. This allows us to develop a secure client-server handshake. After the connection is established, client can send requests to the server which performs the requested service, such as looking up flights based on their source and destination. Multiple invocation semantics (at-least-once and at-most-once) are implemented.

Request and reply design

Each request from the client is at the beginning represented as an array of objects. The array consists of identifier of the requested service, number of objects, unique numbers representing each object's type (including the error message) and the objects itself. This array is then send to the marshalling methods where they are converted into an array of bytes. After the server receives the request (array of bytes), unmarshalling method is called. The unmarshalling method converts the array of bytes back to the array of objects.

Based on the identifier of the requested service, the server then performes the service and reply is send back in a similar way. When the server needs to send an error message (for example, when no flight with queried ID exists), it does so by sending an array that consists of the requested service id and special number that indicates that an error message is being send.

Marshalling

Our goal in the marshalling process is to convert objects into array of bytes. First, we convert more "complex"objects (instances of classes such as flight object) into array of data types such as strings and integers. These integers and string are then converted into numbers between 0 and 127 indicating their ascii code and then converted into an array of bytes. At the beggining of such array there is an indication of the objects type (unique number) and number of bytes which is used to represent this object in an array.

In our array of bytes, we have indication of number of objects, unique numbers indicating their type and we also have number of bytes needed to represent a particular object. All these information give us the ability to perform unmarshalling and reconstruct all the objects.

Services

The flight information system has 7 services available.

- A service that establishes connection between the client and the server
- A service that allows a user to query the flights by specifying the source and destination places
- A service that allows a user to query the flights by specifying the flight identifier
- A service that allows a user to make seat reservation on a flight
- A service that allows a user to query number of flights
- A service that allows a user to praise the information system by giving a "like"

Last two services are idempotent and non-idempotent respectively. Giving a like is an non-idempotent operation, because in our case a user can give multiple likes to the information system and the server always replies with the current number of likes it received. Therefore, if we perform the operation twice, the server will reply with two different numbers.

Invocation semantics

Experiment

The main goal of the experiment is to observe how different invocation semantics behave when using idempotent and non-idempotent service. The two invocation semantics are at-least-once and at-most-once. In the experiment, the loss of messages between a server and a client is simulated by simple simply not sending any messages with a propability 50%.

In the text below (Listing 1), we can see the output of the client's console, after we were using an at-least-once invocation semantics on a non-idempotent service. We can see that after client has given like number 1 and like number 2, the reply from the server states that the user has given like number 1 and like number 3. So what has happened?

Listing 1: Client console output

```
Invocation semantics used: at least once
timeout
Communication successfully established
Welcome to flight information system.
Press:
1 - query flights by source and destination of the flight
2 - query flights by ID
3 - make an reservation
4 - monitor flight updates
5 - Check number of flights
6 - Give a like
9 - Exit
6
```

```
You have given a like number 1
Press:
1 - query flights by source and destination of the flight
2 - query flights by ID
3 - make an reservation
4 - monitor flight updates
5 - Check number of flights
6 - Give a like
9 - Exit
6
timeout
timeout
timeout
You have given a like number 3
Press:
1 - query flights by source and destination of the flight
2 - query flights by ID
3 - make an reservation
4 - monitor flight updates
5 - Check number of flights
6 - Give a like
9 - Exit
```

From the server log below (Listing 2), we can observe that the server has indeed received three requests from the client to give a like. This has happened, because when the reply from the server got lost, the client simply retransmitted request message to give a like. Then, by the rules of at-least-once invocation semantics the server did not filter the message and and simply re-executed the method to give a like.

Listing 2: Server console output

```
starting up on localhost port 10000
waiting for data
Received data from address: ('127.0.0.1', 34299)
Received b'\x00d'
Starting using At-least-once invocation semantics
[0, 100]
waiting for data
Received data from address: ('127.0.0.1', 34299)
Received b'\x06\x01\x0b\x05\x00\x00\x00\x00'
Service 6; User gave us a like number 1
[6, 1, 11, 1]
waiting for data
Received data from address: ('127.0.0.1', 34299)
Received b'\x06\x01\x0b\x05\x00\x00\x00\x01'
Service 6; User gave us a like number 2
[6, 1, 11, 2]
waiting for data
Received data from address: ('127.0.0.1', 34299)
```

```
Received b'\x06\x01\x0b\x05\x00\x00\x00\x01'
Service 6; User gave us a like number 3
[6, 1, 11, 3]
waiting for data
```

Using the at-most-once invocation semantics we can observe how the client and the server behaves from the client and server outputs below (listings 3 and 4). From the server log below we can see that after the service was performed ("Service 6; User gave us a like number 1"), it did receive another request. However, because at-most-once invocation semantics was used, the server recognized the duplicate request and retransmited the reply instead of re-executing the request.

Listing 3: Client console output

```
Invocation semantics used: at most once
Communication successfully established
Welcome to flight information system.
Press:
1 - query flights by source and destination of the flight
2 - query flights by ID
3 - make an reservation
4 - monitor flight updates
5 - Check number of flights
6 - Give a like
9 - Exit
6
timeout
timeout
You have given a like number 1
Press:
1 - query flights by source and destination of the flight
2 - query flights by ID
3 - make an reservation
4 - monitor flight updates
5 - Check number of flights
6 - Give a like
9 - Exit
```

Listing 4: Server console output

```
starting up on localhost port 10000
waiting for data
Received data from address: ('127.0.0.1', 43936)
Received b'\x00e'
Starting using At-most-once invocation semantics
waiting for data
Received data from address: ('127.0.0.1', 43936)
Received b'\x06\x01\x0b\x05\x00\x00\x00\x00'
Service 6; User gave us a like number 1
```

```
waiting for data
Received data from address: ('127.0.0.1', 43936)
Received b'\x06\x01\x0b\x05\x00\x00\x00\x00'
waiting for data
```

Using different invocation semantics on idempotent operation did not lead to different results. However, the at-most-once invocation semantics did not re-executed the methods which could lead to significant server optimization if we deal with computationaly difficult operations.