Twisted Primer

Due April 27th, 2016
CSE332 Programming Paradigms, Spring 2016

Overview Write a reverse network proxy. Your proxy should include a `home` and a `work` component. The `home` component would run on your home computer, outside the firewall. The `work` component would run on your work computer, "behind" the firewall. The `work` component makes connections to a service (also behind the firewall) available without opening any ports in the firewall.

Grading will be assessed as follows:

1. (5 points) `Work` creates a "command" connection to `home`.
2. (5 points) `Home` accepts TCP connections from a client program, establishing a "client" connection.
3. (5 points) `Home` requests a "data" connection from `work` and `work` establishes that connection.
4. (5 points) `Work` establishes a TCP connection with a third-party server.
5. (5 points) `Work` links the server connection to the "data" connection.
6. (5 points) `Home` links the "data" connection to the "client" connection.

That's 30 points total. In addition to inspecting your code for the above items, we will test your code by trying an SSH connection through your proxy. A successful SSH session will indicate full credit (with the usual caveat about our reading your code ☺).

```
cmc@nanook:~/homework/twisted$ ssh localhost -p 9001
cmc@localhost's password:
Last login: Thu Apr 21 12:24:52 2016 from [removed]
[cmc@student00 ~]$
```

Notice what's happening above. I'm SSH'ing to my localhost on my laptop, port 9001. But then I log in to `student00`! In this example, I have `work.py` running on a computer in my office, and `home.py` running on my laptop off campus. `Work` is creating a reverse connection to me, while `home` listens on port 9001 for a client. When I direct my SSH client to port 9001 locally, `home` sends the data along to `work`, and `work` sends it along to port 22 on `student00`.

Discussion The difficulty in this assignment is in the concept of the reverse proxy, not the syntax of the `twisted` library. You will need to establish a series of TCP connections in a particular order, and your `work` and `home` programs will need to work together. Therefore, this time we'll talk more about concepts – you can find plenty of examples of the syntax online, such as the links under Resources below.

The first thing to understand is that there is nothing special about who starts a TCP connection. Data flows from one "end" of the connection to the other, regardless of who is the listener and who is the client. There is nothing mystical about being a "server."

But firewalls tend to block connections in only one direction. That is, a firewall protecting Notre Dame will prevent off-campus computers from initiating TCP connections with `student00.cse.nd.edu`, for example. This is important for security reasons to block worms, password brute forcing, etc. But, the firewall does not block outgoing connections. If you are an authorized user on `student00`, you can initiate a connection with any visible off-campus machine.

Once that connection is established, data may flow in either direction. So, if I can make a port on my laptop visible off-campus (e.g., at home), then I can write a program on `student00` that connects to my laptop. Then I can send commands from my laptop back to `student00`. In a highly-secure environment, this is preferable because it means that no ports need to be opened on the machine providing the services – you could build a secure database, but never have to leave that database vulnerable on the network via an open port.

The program that enables this functionality is called a "reverse proxy." A proxy being a program which passes an incoming connection to another host. And "reverse" meaning that the proxy initiates all connections, rather than listening for connections.

So to build a reverse proxy, you need two programs. We're calling these `work` and `home`. `Work` is the program which you can think of as running on a computer at work, "behind" the firewall. `Home` is the program running at home, "outside" the firewall.

The way these operate together is that `home` starts listening for connections on two ports: a "command" port and a "client" port. `Work` then connects to `home` over the command port. That forms the command connection.

Later, a client program, such as your SSH client, connects to `home` over the client port. The objective is to get this SSH client to actually connect to a third-party service behind the firewall. That third-party service is visible to `work`, but not visible to `home`. `Home` needs to ask `work` to forward the client's data to the third-party service.

What `home` will do is send a request across the command connection to `work`. The request will contain some text such as "begin data connect". At the same time, `home` will start listening on a port we'll call "data." `Work` will receive the "begin data connect" message over the command connection, and then initiate a connection to the data port on `home`. There are now three connections: 1) a command connection between `work` and `home`, 2) a data connection between `work` and `home`, and 3) a client connection between the client program and `home`.

Still missing is a connection to the third-party service. So, `work` will establish this connection, creating a fourth connection: 4) a service connection between `work` and the third-party service.

All that's left is to link the connections so that the client can reach the third-party service. `Home` will start by writing all data it receives from the client connection to the data connection. `Work` will write all data it receives from the data connection to the service connection. Likewise, `work` will write all data is receives from the service connection to the data connection, and `home` will write data from the data connection to the client connection. The process is completely transparent to the client program and third-party service. As far as the third-party service is concerned, the client program is connecting from `work`.

In class I will elaborate on many of the details and pitfalls!

Learning Objectives This assignment brings together several different topics we have discussed over the semester. Twisted is an event-driven library, so you will deal with events/controllers/handlers as you have before. Except the environment is quite different, so you will see how the concepts are similar for network protocols. You will also expand on networks knowledge gained from the CherryPy assignment – networks knowledge is extremely important in a world of mobile devices.

You will practice object-oriented concepts, as the different "factories" and "protocols" you write will need to be coordinated very carefully. You will demonstrate a security technology, see how different paradigms can be used to solve a real-world problem, and even build a practical tool which you may need to use in the future.

Turn in Package your source code into a `zip` or `tar.gz` archive and submit it to `prog.paradigms.secN.sp16@gmail.com`.

Resources

`https://twistedmatrix.com/trac/`
`https://twistedmatrix.com/documents/current/core/howto/servers.html`
`https://twistedmatrix.com/documents/current/core/howto/clients.html`

A Note This is the last programming assignment we will hand out. You have been a great class and it has been my pleasure to work with you.