Simplified File Transfer Protocol

Examining Networking Capabilities of Haskell, Java, and the Event-driven Python Library Twisted

Coy Humphrey and Dustin Pfeiffer

Project Outline

Our goal was to create an FTP server in different languages to compare the networking capabilities of each language, as well as the ease of programming and understanding. Our project is inspired by the File Transfer Protocol, but it is not an exact copy. As in FTP, our protocol involves a telnet port and separate data ports for file transfer.

Our protocol includes the following commands:

- `get` *filename* - Used to get files from the server. The port command must be called before calling `get`. The server will attempt to connect to the client on the port specified by the earlier `port` command. If the connection is successful, the server will send the entire file through the new connection and close the connection when the file is finished.
- `put` *filename* - Used to send files to the server. The server will respond with a string containing the port number to connect to over the telnet socket.The client should open a connection to the server on this port and send the contents of the file. When the file has been sent the connection should be closed.
- `port` *portNum* - Used to specify the port the server should connect to during the next `get` command.
- `dir` - Used to see the contents of the directory the server is running in. The server sends this information as a string over the telnet socket.
- `exit` - Used to end the connection. The server responds by closing the telnet socket.

We chose to implement our project in Java, Haskell, and Python (using a library called Twisted).

Why choose these languages?

We chose Java because we already had an understanding of Java and we feel it represents a prototypical imperative language. Similarly, we chose Haskell because we were learning it in class, and we feel it provides a good representation of functional programming languages. We chose Twisted to explore the use of event-driven languages in network applications.

Handling Clients

Java

```
SimpleFTP.java
for (;;) {
    Socket clientSocket = serverSocket.accept();
    (new Thread(new ClientHandler(clientSocket))).start();
}
```

In Java, we had a main loop that would constantly accept clients and start a new thread to handle each new client.

Haskell

```
loop serv_sock = do
    (h,host,_) <- accept serv_sock
    forkIO $ handler h host ""
    loop serv_sock
```

The Haskell implementation follows the same structure as the one in Java. Notably, while Haskell does not have an infinite for loop as in Java, it can still loop endlessly by using tail recursion. Additionally, Haskell can start a new thread using only a function, rather than creating a Thread object as in Java.

Python

```
endpoints.serverFromString(reactor, "tcp:" + sys.argv[1]).listen(AnswerFactory())
reactor.run()
```

Twisted provides the functionality we had to hand-code in Java and Haskell under the hood. The above code will listen on the tcp port specified by the first command line argument. When a client connects, they are directed to a factory. Factories are a Twisted abstraction. They contain some state as well as a Protocol, another Twisted abstraction. The AnswerFactory, seen above, defines the AnswerProtocol, which is used to accept lines from the client, parse them, and perform appropriate actions depending on the line received.

Handling Files

Java

```
ClientHandler.java
while ((int d = in.read()) != -1){
    fileout.write(d);
}
```

Our Java implementation is naive method of reading and writing files. It reads and writes a single byte at a time without buffering until it reaches the end of the file. This affects speed of the transfer, but is still suitable for our goals in this project.

Haskell

```
withFile file ReadMode (\handle -> do
    contents <- B.hGetContents handle
    B.hPut sock contents)
```

Our Haskell implementation takes advantage of Haskell's laziness. Conceptually, we read in the contents of the entire file, then write everything to the socket. Haskell will handle any buffering that needs to be done using the Lazy Byte String.

Python

```
def connectionMade(self):
    fs = FileSender()
    fs.beginFileTransfer(self.factory.fp, self.transport)
```

Twisted provides a class called FileSender for sending files. Before discovering this class, we considered reading the file and dumping its data over the stream. However, that defeats the purpose of Twisted's event-driven approach. The program will block when data is not able to be uploaded, and prevent other events from being run. The FileSender class is designed using Twisted's event system, and will only send data when it is able to, allowing other events to run when data cannot be sent.

Parsing Input

After a client connects to the server, they are able to send commands. The server parses these commands and performs an appropriate action. The code to do the parsing was remarkably similar between the three languages. Each implementation involves splitting the command string into words, then checking the first word for the command. An appropriate function is called with the remaining words passed in as arguments. The cleanest of these implementations was done in Python and is shown below.

```
def lineReceived (self, line):
    cmd = trim_split (line)
    if len(cmd) < 1:
        return
    functions = {
        "get"      : self._get_,
        "port"     : self._port_,
        "dir"      : self._dir_,
        "put"      : self._put_,
        "exit"     : self._exit_,
    }
    if functions.has_key (cmd[0]):
        functions[cmd[0]](cmd[1:])
    else:
        self.transport.write ("Invalid command\n")
```

The Java and Haskell implementations use case of and switch statements respectively.

Threading Differences

Java

```
ClientHandler.java
class GetHandler implements Runnable{
    InetAddress addr;
    int portNum;
    String file;

    public GetHandler(InetAddress addr, int portNum, String file){
        this.addr = addr;
        this.portNum = portNum;
        this.file = file;
    }

    public void run(){
        ...
    }
}
...
(new Thread(new GetHandler(client.getInetAddress(), port, words[1]))).start();
```

In the Java program, any method that we wanted to run in a thread had to be wrapped in an object that implements the Runnable interface. Additionally, arguments could not be directly passed to the method. Instead we had to add fields to the method's wrapper class and create a constructor that accepted these fields.

The code above shows how we run our Get method in its own thread. We created a GetHandler class as a wrapper, and passed a GetHandler object into the Thread constructor.

Haskell

```
forkIO $ doGet host port (cmd !! 1)
```

Threading in Haskell is much cleaner. Haskell provides a function called forkIO which, given a function call, runs the function in its own lightweight thread.

The above code shows how we run our Get command in its own thread.

Known Bugs

Our `put` protocol involves opening a separate socket, sending the port number of that socket to the client, then closing the socket after receiving all of the data from the client. Doing this in Java and Haskell was easy because we had direct access to lower level Socket details. In Twisted, however, these details were abstracted away and we could not figure out how to access them.

In Java and Haskell, getting the port to send to the client required only a single function call. We could not find a comparable function call in Twisted, and so we had to use a hack instead. In our Python implementation we use Python's normal socket library to open a socket. We then keep track of the port of that socket, close the socket, and tell Twisted to start a factory on that port. This introduces a race condition, where it's

possible that the port reported by our hack may be taken by another process in the time between closing the dummy socket and starting the factory listening on the port.

We were also unable to find out how to close a factory's underlying server socket from within the factory. Because of this, every time `put` is used, a new server socket is opened, but is never closed.

Code Length

Language	File	Lines	Total
Java	SimpleFTP.java	152	
	ClientHandler.java	42	194
Haskell	SimpleFTP.hs	82	82
Python	SimpleFTP.py	102	102

Java was the most verbose of the languages. It consists of four classes over two files. The SimpleFTP class contains the main method. ClientHandler contains two inner classes to handle the `get` and `put` commands. The files, however, are riddled with `try/catch` blocks, which make the code that does the work difficult to find.

Haskell was the shortest program. However, what it gains in brevity, it loses in clarity. It is by far the most dense, and the code must be read closely to gain a good understanding of how it works under the hood.

Twisted, with just over a hundred lines of code, can still be difficult to understand because neither of us are very familiar with event-driven programming.

Conclusion

Our goal with this project was to see if there was an optimal language for creating simple networking programs. What we found, however, was that it really comes down to personal preference. Even within our own group, there are disagreements on which code is the most readable or the easiest to write. Connecting to another computer is not difficult, provided the programming language is known beforehand. The one conclusion we came to was that Python was the most difficult to code because we had little experience in Python, and none in event-driven programming.

Even the amount of code necessary is comparable. The Haskell and Python implementation came out roughly equal in line count, and the Java, though the count is about twice as much as the others, has a lot of lines taken up `try/catch` headers and brackets. With these blocks, however, it gains the extra functionality of basic error checking. If stripped down to just the necessary function calls, the three programs would be about the same length.