Introducing Network Programming into a CS1 Course

Michael H. Goldwasser
Dept. Mathematics and Computer Science
Saint Louis University
221 North Grand Blvd
St. Louis, Missouri 63103-2007
goldwamh@slu.edu

David Letscher
Dept. Mathematics and Computer Science
Saint Louis University
221 North Grand Blvd
St. Louis, Missouri 63103-2007
letscher@slu.edu

ABSTRACT

Incorporating advanced programming concepts into an introductory programming course can provide motivation, yet care must be taken to avoid overwhelming students. We describe our experiences teaching network programming in a CS1 course using Python. The simplicity of the built-in libraries allows a fair amount of networking to be introduced in a week-long module of the course. In this short time, students write both multi-threaded clients and servers.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—*computer science education*; C.2.0 [**Computer Systems Organization**]: Computer-Communication Networks—*General*; D.1.0 [**Software**]: Programming Techniques—*General*

General Terms

Languages, Design

Keywords

Client, Server, Python

1. INTRODUCTION

Many CS1 instrutors wish to expose their students to one or more advanced programming topics to increase breadth. In several recent offerings of our introductory course at Saint Louis University, we included a segment on basic network programming lasting a week or week and a half.

Network applications have become ubiquitous and are a natural example for almost any computer science course. While many instructors feel that teaching any non-trivial network programming is beyond the typical CS1 student, students have no reason to believe that network programming should be particularly hard. With properly chosen examples, it can be straightforward.

ITiCSE'07, June 23–27, 2007, Dundee, Scotland, United Kingdom.

Network programming is far from a standard inclusion in most CS1 courses. In fact, the Computing Curricula 2001 report does not have it included in any of its introductory sequences [1]. Only a few of the most popular CS1 texts include chapters on networking. While it need not be a standard topic in all introductory programming courses, it should not be automatically avoided.

A variety of courses in Java have taken advantage of built-in networking libraries and several popular texts [2] have chapters on the topic. However, time is limited in most courses and typical network coverage includes fetching webpages and other client applications [4]. More advanced topics, such as the development of client and server applications is often skipped due to time constraints. A Java RSS Reader has been suggested as a "Nifty Assignment," though in the context of CS2 [3].

Using standard Python libraries we have explored both client and server software. This includes writing a basic web server, chat clients and servers and network games. While these applications may not be as robust as ones written in a later network programming course, they introduce the basic concepts and meet our goals of exposing students to advanced topics in a manageable way.

In this paper, we discuss a four-day module introducing network programming, demonstrating complete source code used for several sample applications.

2. STUDENT BACKGROUND

Certainly, we do not expect to use network programming at the beginning of an introductory course. We want students to first master the more general programming skills. In our case, students had the necessary background for this module after about three-fourths of the semester. Basic string processing is important in managing messages. Since the network libraries rely on inheriting application specific utility classes from standard libraries, it is important that students have implemented their own classes and have a reasonable understanding of inheritance. Once connections are established, sending and receiving communications across the network is similar to reading and writing from files. So it is advantageous if students have interacted with files as an abstraction. In our final examples, we make use of Python's dictionary class for lookups.

3. DAY 1: WRITING A BASIC CLIENT

With their experience using network applications on a daily basis, our computer savvy students were used to the idea that network activity begins by connecting to some ma-

```
from socket import socket

connection = socket( )
server = 'time.nist.gov'
connection.connect( (server, 13) )

fields = connection.recv(1024).split( )
date = fields[1]
time = fields[2]

print 'Date is %s, time is %s (UTC)' % (date,time)
```

Figure 1: The simple time client

```
from SocketServer import TCPServer, BaseRequestHandler

class EchoHandler(BaseRequestHandler):
  def handle(self):
    message = self.request.recv(1024)
    self.request.send(message)

# may need to customize host and port for your machine
echoServer = TCPServer( ('my.net.addr', 9000), EchoHandler)
echoServer.serve_forever( )
```

Figure 2: The echo server.

chine on the network using a specified protocol, such as http or ssh. They might not use the same terminology that we would, but these are part of their everyday lives. So we begin with a basic overview of networking and sockets, discussing how a connection is made to a particular port on a machine. We also discuss the need for a clear communication protocol of which both client and server are aware.

As a first example we aim to query the current time by connecting to an internet time server. While most time servers use more advanced protocols, some NIST servers still support the basic daytime protocol running on port 13. When a connection is made, the server sends a string with several space-separated fields. The second field is the date in "YYYY-MM-DD" format and the third is the time in "HH:MM:SS" format.

The only class we need to introduce is the socket class. For this assignment there are only two relevant methods of that class. The connect method establishes a new connections, requiring a two-tuple as a parameter, to specify the server address and port. The other method is recv which returns a string sent from the server. Though this first example does not require sending data, we will need the send method for other tasks and so we introduce it naturally in contrast to recv. Since students are familiar with means for inputing strings from and outputting strings to a user, as well as reading strings from and writing strings to a file, the concept of recv and send is straightforward. The main difference is that the recv function takes a single parameter of a maximum message length. While they are not used to having to deal with buffers, they do understand that huge amounts of data might be transmitted over the internet and you may wish to limit the amount of data coming at any one time. We chose to use a buffer size of 1024. The complete program for accessing the time and day is in Figure 1.

With the simple data format, no data is ever sent along the connection to the server. The students only need to process the data returned using the split function of the str class that will cut the response string along whitespace and put the result in a list of strings. While slightly different from what students were used to, this program was not hard to comprehend and made an excellent starting point.

After completing this example in class, students are prepared to write slightly more complicated clients. In some cases, students implemented a basic telnet application and others wrote programs to download and save a webpage. Another good activity is adding error checking and robustness to the time client.

4. DAY 2: A BASIC SERVER

After seeing a very simple client, we wanted to continue by writing a server with a simple protocol. We chose to implement an echo server since the protocol is trivial. We needed to introduce students to two library classes: TCPServer and BaseRequestHandler. In a more advanced networking course we might provide more emphasis regarding the different types of connections, explaining the differences between UDP and TCP. Yet for CS1, we wanted the most basic explanation, namely that there are multiple kinds of servers and the one that we were going to use is based upon TCP.

To construct a server instance, the students need a network address and port to listen on and a way to handle incoming connections. We decided to run the server on port 9000 and went on to explain how a request handler worked. The pertinent features of the server class are as follows. Every time a computer connects to the server, a request is generated and an instance of BaseRequestHandler is constructed. The handle function of that class is called to process the request, and a member variable called request is the socket used to communicate back and forth with the client. Remember that students had already dealt with sockets in the previous example. The new concept for this example was that of a handler. Even though we had not previously seen event-driven programming, the concept was easily grasped.

The source code for the server is in Figure 2. To implement the server, students write a class called EchoHandler derived from the BaseRequestHandler and must override exactly one function. For an echo server, the handle function reads a message from the socket and immediately writes it back to the same socket, and then terminates (implicitly closing that connection). To prepare a server they create an instance of a TCPServer that listens on the chosen port and uses the EchoHandler class to handle incoming messages. The server is then started in a mode that handles requests continuously until the program is stopped.

While there were several concepts and techniques for the students to acquaint themselves with, no single part of the program was overwhelming. Furthermore, from a network programming perspective there was nothing present in the code that did not absolutely have to be there. This simplicity helps avoid confusion among the students. While this server may not be as robust or advanced as one you might write in a network programming course, it is not bad for one day in a CS1 class. Other examples that students have worked on include a web server that only handled "GET" requests. This was slightly more complicated, but the basic techniques are the same.

Message Type	Format
Joining the room with given identity	`'ADD %s\n'% screenName`
Sending a message to everyone	`'MESSAGE %s\n'% content`
Send a private message	`'PRIVATE %s\n%s\n'% (recipient,content)`
Quit the chat room	`'QUIT\n'`

Figure 3: The network protocol for the chat server and client.

5. DAY 3: THE CHAT SERVER

After writing a simple server like the echo server, students are ready for a more complex server; in this case a chat server. The two additional complexities are the fact that the connections are persistent and that information needs to be shared between connections.

The topic of persistence can lead to interesting classroom discussions. In one course students came to the realization that a multi-threaded server was needed to handle the multiple connections; this recognition came without even knowing what a thread was or the proper terminology. In Python, creating a multi-threaded server is straightforward; a ThreadingTCPServer instance is constructed instead of a TCPServer. This takes care of all of the threading issues and ensures that students do not have to worry about anything other then the program logic.

This was also the first chance students had to develop their own network protocol. In class they decided that each message would be preceded by one of `ADD`, `REMOVE`, `MESSAGE` or `PRIVATE`. These commands are followed by the messages to transmit or other necessary information.

Since messages need to be relayed from one connection to another, a dictionary is kept to store all the connections. This dictionary also allows private messages to be sent to a specific socket located based on username. The complete source for the server is in Figure 4.

To test the server (in the absence of a matching client), students used a telnet client to connect to the server and simulated the network protocol. This gave them a somewhat usable chat client. Testing their program also demonstrated the unpredictable consequences which result if a protocol is not properly used by both client and server. This experience could lead to writing a more robust server.

Prior to our development of this CS1 module, students in our curriculum did not write a server of this complexity until either the operating systems course or well into our network programming course. In fact, the CS1 students were amused to discover that the students in the operating systems course were also writing a chat server and client that very same week.

```
from SocketServer import ThreadingTCPServer,
    BaseRequestHandler

socketLookup = dict( )

def broadcast(announcement):
  for connection in socketLookup.values( ):
    connection.send(announcement)

class ChatHandler(BaseRequestHandler):
  def handle(self):
    username = 'Unknown'
    active = True
    while active:
      message = self.request.recv(1024)
      if message:
        command = message.split( )[0]
        data = message[1+len(command): ] # the rest

        if command == 'ADD':
          username = data.strip( )
          socketLookup[username] = self.request
          broadcast('NEW %s\n' % username)
        elif command == 'MESSAGE':
          broadcast('MESSAGE %s\n%s\n' % (username,data) )
        elif command == 'PRIVATE':
          rcpt = data.split('\n')[0]
          if rcpt in socketLookup:
            content = data.split('\n')[1]
            message = 'PRIVATE %s\n%s\n' %
                (username,content)
            socketLookup[rcpt].send(message)
        elif command == 'QUIT':
          active = False
          broadcast('LEFT %s\n' % username) # inform others
          self.request.send('GOODBYE\n') # acknowledge

    socketLookup.pop(username)
    self.request.close( )

myServer = ThreadingTCPServer( ('my.net.addr', 9000),
    ChatHandler)
myServer.serve_forever( )
```

Figure 4: A simple chat server.

6. DAY 4: THE CHAT CLIENT

To use the chat server effectively, the next step is writing a dedicated chat client. This turned out to be the most difficult part of the week. The main reason for the extra complexity is that the client has two separate threads: one for sending messages and one for receiving. This client could easily be skipped; however, we also wanted to expose our students to some minimal threading examples.

To have a separate thread in a Python program you need to write a class derived from Thread and override the run function to implement the second thread. While the details of the thread implementation were new to the students the concepts were not hard for them to deal with. As beginners, they have no a priori reason to believe that writing multi-threaded programs is difficult.

The complete source for the client is in Figure 5. The second thread is implemented in the Incoming class and, as the name implies, deals with incoming messages. It continues reading messages from the server and printing the messages to the screen. Since the students spend a fair amount of time on string processing earlier in the term, the text processing for extracting data from the messages is among the more straightforward portions of the program.

```
from socket import socket
from threading import Thread

server = socket( )
server.connect( ('my.net.addr', 9000) ) # or a remote host
username = raw_input('What is your name: ').strip( )
server.send('ADD %s\n' % username )

class Incoming(Thread):
  def run(self):
    stillChatting = True
    while stillChatting:
      incoming = server.recv(1024) # wait for more news
      lines = incoming.split('\n')[:-1]
      i = 0
      while i < len(lines):
        command = lines[i].split( )[0]
        param = lines[i][len(command)+1: ]
        if command == 'GOODBYE':
          stillChatting = False
        elif command == 'NEW':
          print '==>', param, 'has joined the chat room'
        elif command == 'LEFT':
          print '==>', param, 'has left the chat room'
        elif command == 'MESSAGE':
          i += 1 # need next line for content
          print '==>', param + ': ' + lines[i]
        elif command == 'PRIVATE':
          i += 1 # need next line for content
          print '==> [private]', param + ': ' + lines[i]
        i += 1

incomingThread = Incoming( )
incomingThread.start( )

active = True # main thread for user input
while active:
  message = raw_input( ) # wait for more user input
  if message.strip( ):
    if message.rstrip( ).lower( ) == 'quit':
      server.send('QUIT\n')
      active = False
    elif message.split( )[0].lower( ) == 'private':
      colon = message.index(':')
      friend = message[7:colon].strip( )
      server.send('PRIVATE %s\n%s\n' % (friend,
                                        message[1+colon:]) )
    else:
      server.send('MESSAGE ' + message)
```

Figure 5: The chat client.

The main thread deals with the transmission of messages to the server. It continues to read user input and sends the messages to the server. In the case of private messages or leaving the chatroom, there are slightly different procedures. The complexity of the two threads were roughly equivalent and several of the students implemented their code with the roles of the two threads reversed.

A sample session, as viewed by the user of this software, appears in Figure 6. Lines 5, 10, 12, 14 and 16 were actually commands typed by this user. The rest is a trace of the relevant chat room activity. Notice that the command issued at line 5 is reflected back at line 6 as it is broadcasted to the entire room. The commands issued at lines 12 and 14 were private messages to `hot dog` and thus not broadcast to the room. The incoming message at line 13 was one which was sent privately to this user from `hot dog`.

```
1  What is your name: cool cat
2  ==> cool cat has joined the chat room
3  ==> cold turkey has joined the chat room
4  ==> cold turkey: Hi everyone!
5  Hi turkey
6  ==> cool cat: Hi turkey
7  ==> hot dog has joined the chat room
8  ==> hot dog: How's everyone doing?
9  ==> cold turkey: pretty good
10 I'm okay
11 ==> cool cat: I'm okay
12 private hot dog: Are you hungry?
13 ==> [private] hot dog:  Sure. Wanna eat?
14 private hot dog: Yep
15 ==> hot dog has left the chat room
16 quit
17 ==> cool cat has left the chat room
```

Figure 6: A sample user session with our chat client.

7. CONCLUSION

With a few well chosen examples we were able to introduce our students to basic network programming in a few class days. Our examples were written in Python and took advantage of its simple syntax and built-in libraries. However, the same can be done in other languages.

Once students have learned how to pass information in both directions and have the server maintain the state, students are ready to write more complicated applications. In another semester, this included writing a networked poker game. The poker game has a server that ran the game and simple clients based on the chat clients. This provided the students an opportunity to design their own communication protocols and incorporate more complex logic into the game.

From the students' perspective, the examples that we did in class covered the techniques needed for them to write more advanced applications. It also gave them the opportunity to write programs closer to what they expect out of real world applications.

8. REFERENCES

[1] Joint Task Force on Computing Curricula. *Computing Curricula 2001: Computer Science Final Report.* IEEE Computer Society and the Association for Computing Machinery, Dec. 2001. http://www.computer.org/education/cc2001/final.

[2] R. Morelli and R. Walde. *Java, Java, Java, Object-Oriented Problem Solving*. Prentice Hall, third edition, 2006.

[3] N. Parlante, S. A. Wolfram, L. I. McCann, E. Roberts, C. Nevison, J. Motil, J. Cain, and S. Reges. Nifty assignments. In *Proc. 37th SIGCSE Technical Symp. on Computer Science Education (SIGCSE)*, pages 562–563, Houston, Texas, Mar. 2006.

[4] D. E. Stevenson and P. J. Wagner. Developing real-world programming assignments for CS1. In *Proc. 11th Annual Conf. on Innovation and Technology in Computer Science (ITiCSE)*, pages 158–162, Bologna, Italy, June 2006.