

Lab 07

Basic Threading for Concurrent Applications

Goals

Whether developing algorithms to run in parallel on a single processor, or to run concurrently across multiple machines over a network, issues of programming concurrent applications are important to to familiarize yourself with. Threads offer a model of concurrent execution that are useful when building servers that need to be able to handle multiple incoming client connections concurrently (at the same time). In this lab we will learn a bit about concurrent programming using threads in python, and building concurrent server applications.

Instructions

For this and all future labs, all of your work will be done and submitted through your own student repository on the nisl class server. If you have not done so already, you need to check out a working copy of your personal repository before you can begin this lab.

Exercise 7.0: Preliminaries

Check out a working copy of your students repository if you haven't already done so. Your student repository will have a url designation of the form:

<http://nisl.tamu-commerce.edu/repo/csci553/username>

Where you need to substitute your own nisl account username for the last portion above.

In your working copy, create and add to your repository a directory called *lab07* (make sure you name it exactly as shown, make sure you not only create the directory, but you run the appropriate command to add the directory to be under version control). All work for this lab is to be added and checked into the *lab07* directory of your repository.

Exercise 7.1: Python Threads and Concurrent Programming

If you want your application to perform several tasks at once, you can use threads. Python can handle threads, but many developers find thread programming to be very tricky.

Introduction

Threads allow applications to perform multiple tasks at once. Multi-threading is important in many applications, from primitive servers to today's complex and hardware-demanding games, so, naturally, many programming languages sport the ability to deal with threads. This includes Python.

However, Python's support for multi-threading is not without limitations and consequences, as Guido van Rossum writes:

"Unfortunately, for most mortals, thread programming is just Too Hard to get right.... Even in Python -- every time someone gets into serious thread programming, they send me tons of bug

reports, and half of them are subtle bugs in the Python interpreter, half of them are subtle problems in their own understanding of the consequences of multiple threads...."

Before we begin to look at the code that makes threading work, the most important feature we must look at is Python's global interpreter lock. If two or more threads were to attempt to manipulate the same object at the same time, problems would inevitably pop up. The global interpreter lock fixes this. Only one thread can perform an action at any given time. Python automatically switches between threads when it is needed.

The threading module provides an easy way to work with threads. Its Thread class may be subclassed to create a thread or threads. The run method should contain the code you wish to be executed when the thread is executed. This sound simple, right? Well, it is (Put this in a file called MyThread1.py):

```
#!/usr/bin/env python
import threading
# extend Thread class and override the run method
class MyThread(threading.Thread):

    def run(self):
        print "Insert some thread stuff here."
        print "It'll be executed...yeah..."
        print "There's not much to it."

# instantiate the MyThread class and run it
myThread = MyThread()
myThread.start()
```

Of course, it's no fun having just one thread. Just like us humans, threads get lonely after a while. Let's create a group of threads (put this in file MyThread2.py):

```
#!/usr/bin/env python
import threading

theVar = 1

# extend Thread class and override the run method
class MyThread(threading.Thread):

    # override Threads run class
    def run(self):
        global theVar
        print "This is thread", theVar, "speaking."
        print "Hello and goodbye."
        theVar = theVar + 1

# instantiate a number of MyThread classes and run
for i in range(20):
    myThread = MyThread()
    myThread.start()
```

Now let's actually do something semi-useful with the threading module. Servers often use threads to handle multiple clients at once. Let's build a simple but extendable server. When a client opens a connection with the server, the server will create a new thread to handle the client. To send the client's data to the thread, we will need to override the Thread class's `__init__` method to accept parameters. The server will now send the thread on its way and then wait for new clients. Each thread will send a pickled object to the appropriate client and then print no more than ten strings received from the client. (A pickled object is basically an object that has been reduced to a few characters. This is useful for storing objects for later use and for sending objects over a network). Call the file `PickleServer.py`:

```
#!/usr/bin/env python
import threading
import socket
import pickle

# We'll pickle a list of numbers:
someList = [1, 2, 7, 9, 0]
pickledList = pickle.dumps(someList)

# our thread class:
class ClientThread(threading.Thread):

    # Override Thread's __init__ method to accept parameters
    def __init__(self, channel, details):
        self.channel = channel
        self.details = details
        # chain with superclasses constructor
        threading.Thread.__init__(self)

    # Override Thread's run method to implement thread logic
    def run(self):
        print "Received connection:", self.details[0]
        # first send the pickled list over the socket channel to the
        # client
        self.channel.send(pickledList)
        # now read 10 messages that the client sends to us
        for x in range(10):
            print self.channel.recv(1024)
        self.channel.close()
        print "Closed connection:", self.details[0]

# Set up the server
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(('', 11123)) # listen any interface port 11123
server.listen(5)
```

```
# Have the server serve forever
while True:
    channel, details = server.accept()
    clientThread = ClientThread(channel, details)
    clientThread.start()
```

Now we need to build a client that connects to the server, retrieves the pickled object, reconstructs the pickled object and finally sends ten messages, closing the connection, call the file `PickleClient`:

```
#!/usr/bin/env python
import socket
import pickle

# connect to the server
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.connect( ('localhost', 11123) )

# Retrieve and unpickle the list object
print pickle.loads(server.recv(1024))

# send 10 messages to the server
for x in range(1,11):
    server.send("Hey server, client sends msg " + str(x))

# close the connection cleanly
server.close()
```

Of course, the above client doesn't take advantage of our server's multi-threading capabilities. Only one thread is spawned, which really defeats the purpose of multi-threading. Let's thread the client to make things a bit more interesting. Each thread will connect to the server and execute the code above (call the file `PickleThreadingClient.py`):

```
#!/usr/bin/env python
import threading
import socket
import pickle

# here's our thread for sending messages to the server
class ConnectionThread(threading.Thread):

    # override the Thread classes run method
    def run(self):
        # connect to the server
        server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        server.connect( ('localhost', 11123) )

        # Retrieve an unpickle the list object:
```

```

print pickle.loads(server.recv(1024))

# send 10 messages to the server
for x in range(1,11):
    server.send("Hey srv, client sends msg " + str(x) + "\n")

# close the connection cleanly
server.close()

# Lets spawn a few threads
for x in range(10):
    connectionThread = ConnectionThread()
    connectionThread.start()

```

It's important to remember that threads don't start up instantly. Creating too many of them can slow down your application. It takes time to spawn and later kill threads. Threads can also eat up valuable system resources in large applications. This problem is easily solved by creating a set number of threads (a thread pool) and assigning them new tasks, basically recycling them. Connections would be accepted and then pushed to a thread as soon as it finished with the previous client. A thread pool is a commonly seen paradigm when creating threaded applications.

If you still don't understand, compare it to a doctor's office. Let's say that there are five doctors. These are our threads. Patients (clients) walk into the office, and if the doctors are busy, they are seated in the waiting room.

Obviously, we'll need something that can transfer client data to our threads without running into problems (it will need to be "thread safe"). Python's Queue module does this for us. Client information is stored in a Queue object, where threads can pull them out when needed.

Let's recreate our server to take advantage of a pool of threads (call the following `PickleThreadPoolServer.py`):

```

#!/usr/bin/env python
import threading
import socket
import pickle
import Queue

# We'll pickle a list of numbers:
someList = [1, 2, 7, 9, 0]
pickledList = pickle.dumps(someList)
# Create ur Queue to hold incoming connection requests
clientPool = Queue.Queue(0)

# A revised version of our previous server thread class:
class ClientThread(threading.Thread):

```

```
# Note that we do not override Thread's __init__ method
# The Queue module makes this unnecessary
def run(self):
    # Have our thread serve "forever"
    while True:
        # Get a client out of the queue
        client = clientPool.get()

        # Check if we actually have a client variable
        if client != None:
            print "Received connection:", client[1][0]
            client[0].send(pickledList)
            for x in range(10):
                print client[0].recv(1024)
            client[0].close()
            print "Closed connection:", client[1][0]

# Start two threads, this is the thread pool.
# If we wanted to have 5 or N threads in the thread pool we
# would start up 5 or N instead.
for x in range(2):
    clientThread = ClientThread()
    clientThread.start()

# Set up the server
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind( ('', 11123) )
server.listen(5)

# have the server serve forever
while True:
    clientPool.put(server.accept())
```

As you can see, it's a little more complex than our previous server, but it's not amazingly complicated. The client we created in the previous section can be used to test this server, just like the previous server.