Simplified NS-3 application layer protocol development in Python

Sudarshan S, Aditya Kamath, Bhargav Reddy
Department of Computer Science and Engineering
Indian Institute of Technology Hyderabad, India
Email:{cs10b036, cs11b001, cs11b012}@iith.ac.in

***Abstract*—The socket APIs in most modern programming languages automatically handle multiple concurrent sessions, and only require the programmer to implement a request handler for a single request, even if the goal is to implement a multithreaded server that can handle an arbitrary number of clients in parallel. Additionally, they do not use a callback based API and they allow the programmer to send data to, or read data from, the socket whenever he/she wants, automatically waiting if the buffer is full or empty. In contrast, the NS-3 socket API is rather low level. It is callback-based and it does not transparently multiplex connections. This makes it a nontrivial job to implement a new application layer protocol in NS-3, irrespective of how simple the protocol is. In this work, we create a new socket API for NS-3 that aims to mirror the functionality of Python's SocketServer API (which is the de facto API for programming native socket applications in Python)**

I. Introduction

Unlike most modern programming platforms, NS-3's socket architecture is callback-based, due to the fact that the underlying simulator is event driven. This essentially prevents the application from sending data whenever it wants, and also prevents the application from querying the socket for data explicitly. Instead, NS-3 invokes a call back whenever new data is received at a socket, or when buffer space is freed. Additionally, NS-3's API is not transparent to multiple connections. This makes it relatively nontrivial to implement even simple applications in NS-3. For instance, even a simple packet sink application that does not perform any processing on the received data takes nearly 350 lines of code. Similarly, a simple data send application that sends zeroes takes nearly 400 lines of code.

In contrast, as shown in figure 1 the socket server API, which is the de facto standard for programming TCP servers in Python, provides a vastly simplified API. In order to create a fully multi-threaded multi-client application, all that the programmer has to do is to implement a handler function that handles a single request, reading incoming data from a file-like object that is automatically connected to the appropriate client and writing outgoing data to a file like object that is automatically connected to the client.

Perhaps it is due to this that there are Python implementations of almost all popular application layer protocols, while there are NS-3 implementations of only a select few.

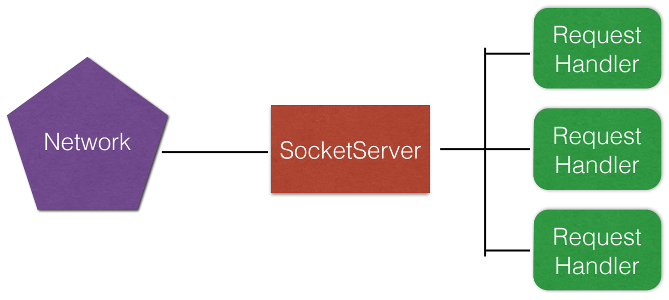

Fig. 1. The Python socket server API transparently multiplexes requests to the appropriate request handlers. It automatically handles creation and deletion of these request handlers.

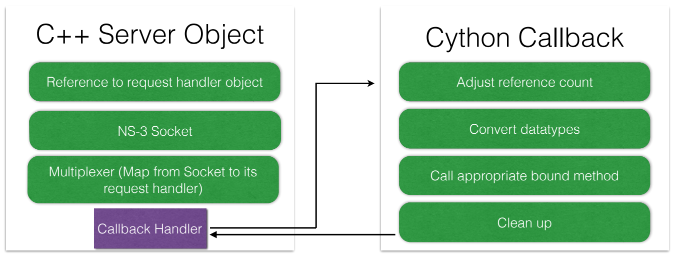
C++ Server Object
Reference to request handler object
NS-3 Socket
Multiplexer (Map from Socket to its request handler)
Callback Handler
Cython Callback
Adjust reference count
Convert datatypes
Call appropriate bound method
Clean up

Fig. 2. Our code consists of 2 parts: a C++ server that interfaces with NS-3, containing a multiplexer that associates sockets with their request handlers, and a set of callbacks written in Cython that handle the job of converting data types and calling Python functions.

II. Goal

The goal for this project was to implement a framework for NS-3 that automatically creates, multiplexes and destroys request handlers. This would allow a programmer to easily create applications that support an arbitrary number of clients: all that he/she would have to do is implement a single request handler that is callback-based, and the framework wold take care of creating a new request handler for each connection and deleting handlers when their connections are closed.

III. Implementation strategy

As depicted in figure 2, our implementation consists of two parts:

- A C++ server which is a valid NS-3 application
- A set of callbacks written in Cython

Our C++ server does not implement any application layer protocol logic. It simply invokes callbacks whenever something is

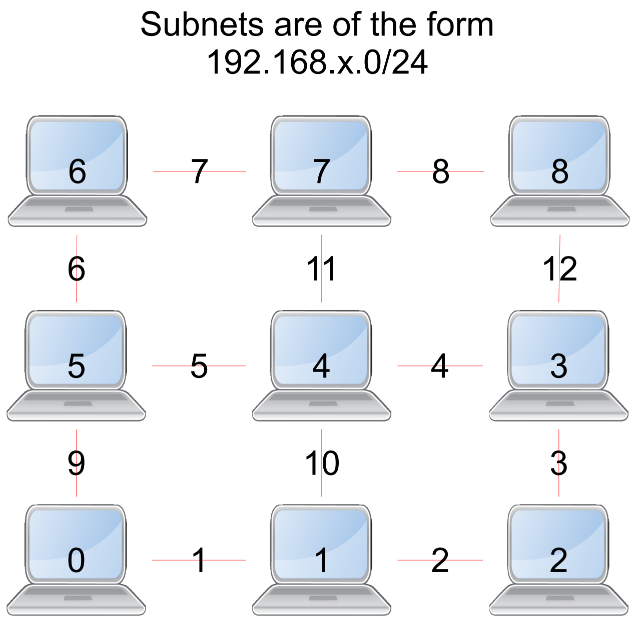

Fig. 3. The network that our test scenario creates. All links are point to point links

to be done. The Cython callbacks are responsible for creating and destroying request handlers, and for invoking appropriate member functions. We shall now analyse the two components of our program in detail:

A. *C++ server*

This is a regular NS-3 application that is broadly similar to the packet sink application that is distributed with NS-3's source. It maintains a set of callbacks, some of which are per connection and some of which are per server (although there can only be one callback registered for each type of event for each server, the callback is expected to take a user data parameter that is different for each connection). Whenever an event of interest occurs, this server calls the appropriate call back. Although such a generic framework could theoretically be used to implement NS-3 socket API bindings for any language, our implementation focuses on Python.

Apart from the various callbacks, the server maintains a "multiplexer"- which is a map from sockets to user data pointers (which essentially represent python request handler objects). This multiplexer is an extremely important part of our server since it is what allows request handlers to be transparently created, multiplexed, and destroyed, as appropriate

Whenever new data arrives along any connection, the server uses the multiplexer to figure out the appropriate user data pointer and then invokes the call back with the user data pointer as an argument. As is illustrated in figure 5 and explained below, since the user data pointer represents the Python object corresponding to the appropriate request handler, this can be used to generate a method call on the request handler.

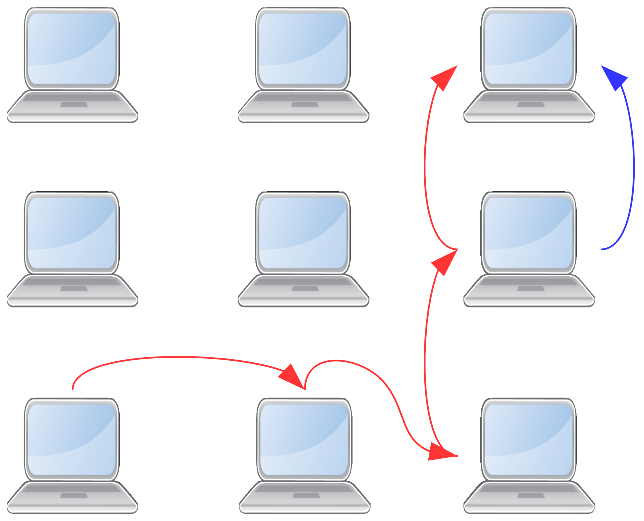

Fig. 4. The TCP flows that our test scenario creates. All flows are created using our server

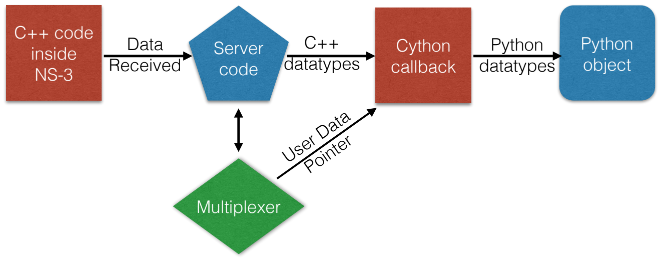

Fig. 5. The core logic behind our code is based on a C++ server that uses a multiplexer to decide what user data pointer to send to a Cython-based callback, which the callback interprets as a Python object that methods are to be invoked on

B. *Callbacks*

In order to prevent our core application from being dependent on Python (to leave open the possibility of expanding this framework to include other languages), all of the python related work is done by these callbacks. They are responsible for manipulating reference counts in Python to ensure that request handlers are automatically garbage collected when the attached connection closes. They are also responsible for converting between Python data types and NS-3 datatypes, which can be a nontrivial operation especially for strings. As part of this process, they are responsible for inter converting between Python objects and user data pointers (which are of type void*), and are also responsible for firing method calls on Python objects given their pointer representations.

In order to keep our code as simple as possible and open to expansion, we used Cython, an open-source language that is similar to Python, but which can access C and C++ datatypes transparently. Our Cython code is compiled into a Python module that can be dynamically loaded by NS-3's Python bindings

To allow for future expansion to include other languages, the Python callbacks have to be set on the server explicitly by calling functions in our python module. This is to allow extensibility in the future if there are other language bindings built atop our framework.

IV. Experimental evaluation

Our second assignment (HTTP server and client) was based on a pre release version of this code. In the assignment test cases, we considered multiple clients connected to the same server and verified that the multiplexer works correctly. Additionally, we have provided a test scenario that is similar to the scenario in the first assignment. The experimental setup is as illustrated in figure 3 and the flows established are as illustrated in figure 4.

V. Results and Conclusion

In conclusion, we successfully implemented a framework that makes it easy to create multi client TCP applications in Python. Although our framework remains callback based, the automatic multiplexing makes it extremely easy to write applications that can handle an arbitrary number of parallel clients