Vicky Tu
CPSC 433 Computer Networks
2/23/2016

Assignment 3 Report

Part 2: Sequential and Multi-Threaded Servers

Your server also needs to implement a heartbeat monitoring URL service to integrate with a load balancer (e.g., Amazon Load Balancer we covered in class). In particular, a load balancer may query a virtual URL (i.e., no mapped file) named load (i.e., with request GET /load HTTP/1.0). If the server is willing to accept new connections, it should return status code 200; otherwise, it returns code 503 to indicate overloading. Your software design should allow "plugin", at run time, of different algorithms to compute overloading conditions. Please describe a particular design and implement it.

I made my implemented a heartbeat monitoring URL service by loading a class dynamically. I made a Balance class which implements the LoadBalancer interface. Balance will randomly determine if the server is willing to take a new connection or not. Balance is a plugin because I used Java Relection to make an instance of the class, which can be easily changed by changing the value of plugin on line 105 in Server.java.

Part 3.1: Asynchronous Server

We recommend that the software structure of your asynchronous server be based on v3 of the EchoServer that we discussed in class. You need to write a handler for the particular protocol. You can feel free to modify the structure if you see any way to improve it (fix error handling, etc). You need to document your changes.

The main changes I made were plugin in my parseConfig in AsyncServer (which was originally Server.java in v3 of the EchoServer folder), parseReq in EchoLineReadWriteHandler, and responseMessage in EchoLineReadWriteHandler. I also changed when processInBuffer determines when the request is complete and added a shutdownOutput when responseSent is set to true to that the client knows the response is completed.

The server should have a timeout thread. Upon accepting a new connection, the accept handler should register a timeout event with the timeout thread with a callback function. The timeout value is specified by IncompleteTimeout <timeout in seconds>. The default timeout value is 3 seconds. If the connection does not give a complete request to the server approximatelywithin timeout from the time of being accepted, the server should disconnect the connection. Note that the timeout monitoring thread should not directly close a channel that the dispatcher thread is still monitoring (why?). You need to think very carefully about the exact details of the interaction between these two threads, propose a software design, and implement it.

I implemented a timeout thread by making a global timer and MyTimerTask. When a clientKey is initiated in Acceptor, I create and schedule a MyTimerTask, which takes the clientKey as a

parameter. After the specified amount of seconds, MyTimerTask runs, during which is check if the clientKey's requestComplete has been set to true. If yes, it does nothing, but if not, it sets channelClosed to true and lets updateState of EchoLineReadWriteHandler actually close the channel.

Part 3.2: Comparison of Designs

A great way to learn about your design is to compare with other designs. You need to read the documents or code of two related frameworks: xsocket and netty.

Part 3.2(a): Comparison with xsocket

Although xsocket is no longer under active development, it provides a design alternative. Please read the source code and document of x-Socket, a high performance software library for reusable, asynchronous I/O servers. Please discuss in your report the following questions (please refer to the specific location when you refer to its document or source code:

How many dispatchers does x-Socket allow? If multiple, how do the dispatchers share workload?

According to http://xsocket.sourceforge.net/core/tutorial/V2/TutorialCore.htm x-socket allow number of CPUs + 1 dispatchers. Also a connection has one dispatcher throughout its lifetime. Dispatchers share the workload by specifying with org.xsocket.connection.dispatcher.maxHandles the maximum number of channels it can be connected to. If the number of channels needing to connect exceeds the number of dispatchers willing to accept, an additional dispatcher will be automatically created. Also as you get more connections you need make more dispatchers because each connection has one dispatcher for its lifetime.

What is the basic flow of a dispatcher thread?

A dispatcher thread is a thread initiated by a non-blocking connection that enables many clients to connect to the same server without the need for a dedicated server process for each client. In order to do that, it is created, channels are attached to it, and then it starts. During its lifetime it performs I/O read/write operations as needed and delegates call-back handling to the handler. In multi-threaded mode, the dispatcher calls the worker pool that calls the handler to handle call-back.

What is the calling sequence until the onData method of EchoHandler (see EchoHandler, EchoServer, and EchoServerTest) is invoked? Please check this link for testing code: http://sourceforge.net/p/xsocket/code/HEAD/tree/xsocket/core/trunk/src/test/java/org/xsocket/connection/

EchoServer starts with complaining if you do not specify a listenport. After you call it correctly, it sets the worker pool size to 10. Then it makes an EchoServer with that listenport, which means it creates an EchoHandler and makes a server with that listenport and handler, with flushmode set to asynchronous. The server is started within that thread, which blocks until the server is

open. An mbean is created and registered for the server on the platform MBeanServer. When is connection is made to the server, the EchoHandler directs the flushmode to be set to asynchronous and autoflush to be turned off. Then when data comes in from the connection, onData is called.

How does x-Socket implement Idle timeout of a connection?

By default the idle timeout is around 24 days, but x-Socket highly suggests setting a more reasonable time, since idle time is the amount of the time elapsed during which connection received no data. The user can implementI IdleTimeoutHandler or IConnectionTimeoutHandler to handle timeouts. Basically the server has a Timer on it and using methods from these handlers allows that timer to be restarted.

Please give an example of how the library does testing (seehttp://sourceforge.net/p/xsocket/code/HEAD/tree/xsocket/core/trunk/src/test/java/org/xsocket/connection/EchoServerTest.java for an example). Please describe how you may test your server with idle timeout?

EchoServerTest tests the server side by creating a server, establishing a connection with the server, sending "test", and making sure it receives "test" back. It tests both side by creating a server and making three threads. On each of them, the thread establishes a blocking connection with the server, writes "hello" and checks that it got back "hello", writes "you" and checks that it got back "you", then closes the connection. It does this 10 times per thread. If for any thread/connection the right response was not received, the error is added to an array list and printed out at the end. I could test my server with timeout by making the one of the client threads sleep before sending "you", and compare how much error was received between the "you" and "hello" messages sent by that thread.

Part 3.2(b): Comparison with Netty

Netty is another Java async IO framework used by many; see for example use cases. Please read Netty user's guide and answer the following questions:

Netty provides multiple event loop implementations. In a typical server channel setting, two event loop groups are created, with one typically called the boss group and the second worker group. What are they? How does Netty achieve synchronization among them?

The boss group accepts an incoming connection. The worker group handles the traffic of the accepted connection once the boss group accepts the connection and registers the accepted connection to the worker. If only one EventLoopGroup is specified, it will act as the boss group and as the worker group. Netty uses ServerBootstrap to create a new ChannelFuture that is synced and handled by both the boss and worker group.

Method calls such as bind return ChannelFuture. Please describe how one may implement the sync method of a future.

The sync method of a Netty's future waits for the future until it is done, and rethrows the cause of the failure if the future failed. This was probably implemented by making a Java Future, then staying in a while loop until isDone() is true (sync is blocking), then returning get(). Or we can call notifyAll when the future is done to avoid busy wait.

Instead of using ByteBuffer, Netty introduces a data structure called ByteBuf. Please give one key difference between ByteBuffer and ByteBuf.

ByteBuf does not have a method like java.nio.ByteBuffer.flip() because it has two pointers; one for read operations and the other for write operations. The writer index increases when you write something to a ByteBuf while the reader index does not change. The reader index and the writer index represent where the message starts and ends respectively. This is great because we always need to flip before writing with a ByteBuffer because it has three pointers; position, limit, and capacity.

A major novel, interesting feature of Netty which we did not cover in class is ChannelPipeline. A pipeline may consist of a list of ChannelHander. Compare HTTP Hello World Server and HTTP Snoop Server, what are the handlers that each insert?

HTTPHelloWorldServer uses HTTPHelloWorldServerHandler while HTTPSnoopServer uses HTTPSnoopServerHandler. They are different in that the HTTPHelloWorldServerHandler send a HTTP response with "Hello World" while HTTPSnoopServerHandler sends a more detailed HTTP response. Instead of sending the same thing to every connection, the HTTPSnoopServerHandler sends header and decode the query and uses that in its response. Also the HTTPHelloWorldServerHandler uses a ChannelPipeline.

Please scan Netty implementation and give a high-level description of how ChannelPipeline is implemented.

ChannelPipeline is implemented by keeping a list of ChannelHandlers in a ChannelPipeline. The ChannelPipeline gives the developer access to the first and last/sink ChannelHandlers to pass ChannelBuffers to. One can add ChannelHandlers to the front and back of the list.

Part 4: Performance Benchmarking

I wrote generate.sh to make a request file for Apache based on requests.txt in the gen folder.

I wrote automate.sh to automate benchmarking.

You should submit a report on your server design.

SeqServer: This was my sequential server. It was based on Professor Yang's TCPServer.java. The idea is to open a Socket, read from the socket, parse the response, generate a response, send the response, then close the socket. All this was wrapped in a while loop.

MTServer: This was my per request thread server. It was based on Professor Yang's TCPServerMT.java. The idea is to open a Socket, read from the socket, parse the response, generate a response, send the response, then close the socket. All this happened in the run of a ServerRunnable thread. When a connection was accepted, a new thread would start.

WelServer: This was my thread pool with service threads completing on welcome socket server. It was based on Professor Yang's ShareWelcome.java. The idea is to open a ServerSocket welcomeSocket and create a pool of threads. While the welcomeSocket is synchronized, connections are accepted and a socket is created to serve the request. Once the request is served, the socket is closed and the thread can then serve another connection.

QServer: This was my thread pool with shared queue and busywait. It was based on Professor Yang's ShareQ.java. The idea is to open a QServer that has a ServerSocket welcomeSocket and a pool of QThreads. The busywait come from the fact the welcomeSocket is always checking to see if it can accept another connection. While the pool is synchronized, a socket is added and removed from the ServerSocket. Once the request is served, the socket is closed and the thread can then serve another connection.

NotifyServer: This was my thread pool with shared queue and suspension. It was based on Professor Yang's WaitAndNotify server. The idea is to open a NotifyServer that has a ServerSocket welcomeSocket and a pool of NotifyThreads. While the pool is synchronized, a socket is added and removed from the ServerSocket, except this time all threads are notified when a connection has been added. Once the request is served, the socket is closed and the thread can then serve another connection.

AsyncServer: This was my asynchronous server using select. It was based on Professor Yang's v3 of EchoServer. I opted to base it on Server and not EchoServer of that folder b/c Server is more modular and easier to add MyTimerTask to. The idea is the server has one dispatcher that iterates through the key, which are handled by Acceptor or EchoLineReadWriteHandler (yes, these are implementations of a more general interface, but these are where most of the action in the program happens). I only modified that things I mentioned in Part 3.1 of this report.

AsyncNIOServer: This was my asynchronous server in Java7. It was based on code from http://www.programmingopiethehokie.com/2014/03/asynchronous-non-blocking-io-java-echo.html. I opted to use CompletionHandlers instead of futures/listeners b/c future.get() blocks, which is what we don't want. The idea is the server has AsynchronousServerSocketChannel that accepts connections to create AsynchronousSocketChannels. On write, the AsynchronousSocketChannel reads into buffer, which we parse and generate a response to. It writes this response to the AsynchronousSocketChannel and if it's done writing, the channel closes.

Benchmarking Methodology:

I used automate.sh to run TestClient against AsyncNIOServer, my fastest server, with 1, 2, 3, 4, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, and 60 threads, each for 90 seconds each, on the dolphin Zoo node at port 6789. Each time, I restarted the server in order to clear out the cache. I always

requested the files in requests.txt. I also tested Apache the same way (same port, same node, same parallels, same time, same restarting of server, same time of day), but with apacheReq.txt. I graphically compared my performance below. I got very good throughput and delay time because Java has gotten very fast, but they are still strangely high so maybe my client was not calculating numbers very well. Apache was not very fast because everyone in the class was probably requesting from it.