Assignment 1

CPSC 433
David Z. Chen

February 24, 2012

1 Asynchronous Server

The architecture of the asynchronous server `SHTTPAsyncServer` closely follows that of the Asynchronous Server V3. A new `ReadWriteHander` (`SHTTPReadWriteHandler`) was implemented to handle HTTP requests. The request handling code was based heavily on the handler code used by the synchronous servers, `WebRequestHander`, but adapted for asynchronous socket I/O. Another change made was that the load balancing callback is now made to the `Dispatcher` object rather than the instance of the `SHTTPAsyncServer`.

In order to implement timeout, a separate thread that monitors timeout events of different opened sockets is instantiated along with the Dispatcher thread. When the Acceptor handler accepts a connection, after creating the `ReadWriteHandler`, it registers a timeout event for the `ReadWriteHandler`'s selection key. The selection key is added to the `IdleTimer`'s asynchronous queue, which is processed at the beginning of each iteration of the timeout thread's run loop. Later, when the `ReadWriteHandler` completes reading the request, it sends a cancellation request to the timeout thread, which is placed in another asynchronous queue in the timeout thread.

The timeout thread should not close the channel directly because the Dispatcher will still have the key registered and will not be able to know about the closing of the channel so that it may deregister the key. The timeout thread maintains two asynchronous queues and a `HashMap` that tracks the end-times of timeout events that are keyed by their selection key. The main loop of the timeout thread is as follows. First, it checks its timeout registration queue and adds each selection key to the HashMap with the end-time set to the current time plus the incomplete timeout set by the server's configuration file. Then, it checks its cancellation request queue and removes each of the requested timeout events from the HashMap.

Finally, it iterates through all of the timeout events in the HashMap. For each key-value pair found, it checks whether the current time has passed the end-time. If it has, then a new `IdleTimerTask` object is created, which is a simple Runnable whose `run()` method simply calls the method of the Dispatcher to close the channel associated with a particular selection key. The Dispatcher was modified to also contain an asynchronous queue based on `invokeLater()` from Santos's article. The timer thread adds the task to the Dispatcher's asynchronous queue using `invokeLater()`, and the Dispatcher closes the channel associated with that selection key and deregisters the key.

2 Comparison of Designs

Reflecting our discussion in class, asynchronous servers are extremely complex. The xsocket library is a hybrid architecture, both asynchronous and multi-threaded. With the significant gain in performance, also comes a significant gain in complexity and difficulty of tracing execution paths through the server.

2.1 Dispatcher threads

According to the xsocket documentation, by default, xsocket creates number of CPUs + 1 dispatcher threads. The dispatcher threads, represented by the `IoSocketDispatcher` class, are managed in a `IoSocketDispatcherPool`.

The dispatchers are created as follows. First, a `Server` object is created. The `Server` constructor calls `ConnectionUtils.getIoProvider().createAcceptor()` on line 489. The `createAcceptor()` method of `IoProvider` (IoProvider.java line 471) instantiates a `IoAcceptor` object (IoProvider.java line 478). The `IoAcceptor` constructor instantiates a `IoSocketDispatcherPool` (IoAcceptor.java line 119), passing it a size, which `IoSocketDispatcherPool` uses when it calls `setDispatcherSize()` (IoSocketDispatcherPool.java line 77) to set the size of the dispatcher thread pool. After it sets the size, it calls `updateDispatcher()` on line 290. The first time this method is called, it notices that the number of dispatcher threads running is less than the maximum and instantiates each dispatcher thread (line 178). Before instantiating each dispatcher, it creates a `IoUnsynchronizedMemoryManager` object for the dispatcher (line 172-176).

The way that the dispatcher pool shares the workload may be seen in the `accept()` method of `IoAcceptor`. The `nextDispatcher()` method of `IoSocketDispatcherPool` allows the dispatcher threads to be retrieved and assigned in a round-robin fashion. After retrieving the next dispatcher thread, the acceptor simply assigns it the new connection.

2.2 Workflow of dispatcher

As described above, a dispatcher thread is represented by the `IoSocketDispatcher` class. Its constructor is defined in IoSocketDispatcher.java line 106-127. The constructor is passed a memory manager object and a string name. It then opens a `Selector`. `IoSocketDispatcher` is `Runnable`. The `run()` method of the dispatcher is as follows:

Listing 1: Main loop of dispatcher

```
    public void run() {

        // set thread name and attach dispatcher id to thread
        Thread.currentThread().setName(name);
        THREADBOUND_ID.set(id);

        DIRECT_CALL_COUNTER.set(0);

        if (LOG.isLoggable(Level.FINE)) {
            LOG.fine("selector " + name + " listening...");
        }
```

```
219
220         int handledTasks = 0;
221
222         while(isOpen.get()) {
223             try {
224                 int eventCount = selector.select(5000);
225
226                 handledTasks = performRegisterHandlerTasks();
227                 handledTasks += performKeyUpdateTasks();
228
229                 if (eventCount > 0) {
230                     handleReadWriteKeys();
231                 }
232
233                 handledTasks += performDeregisterHandlerTasks();
234
235                 checkForLoooping(eventCount + handledTasks,
                        lastTimeWokeUp);
236
237             } catch (Throwable e) {
238                 // eat and log exception
239                 if (LOG.isLoggable(Level.FINE)) {
240                     LOG.fine("[" + Thread.currentThread().getName() +
                            "] exception occurred while processing. Reason "
                             + DataConverter.toString(e));
241                 }
242             }
243         }
244
245
246         for (IoSocketHandler socketHandler : getRegistered()) {
247             socketHandler.onDeregisterEvent();
248         }
249
250         try {
251             selector.close();
252         } catch (Exception e) {
253             // eat and log exception
254             if (LOG.isLoggable(Level.FINE)) {
255                 LOG.fine("error occured by close selector within
                        tearDown " + DataConverter.toString(e));
256             }
257         }
258     }
```

The main loop iterates as long as the Dispatcher is open. Each time through the loop, it calls `select()` (line 224). Then, it calls `performRegisterHandlerTasks()` (line 226) which runs all of the tasks on the asynchronous queue `registerQueue` to handle each of the register handler tasks. Then, it calls texttttperformKeyUpdateTasks() (line 227) which similarly runs each task for updating selection keys on the asynchronous queue `keyUpdateQueue`. Then, if handlers have been registered and if selection keys have been updated, it means that there could be read and write events that need to be handled. Thus, it calls `handleReadWriteKeys()` which, similar to our dispatcher, loops through the set of selection keys and calls the read and write handlers for the read and write events. Then, it calls `performDeregisterHandlerTasks()` to run the tasks on the asynchronous queue `deregisterQueue` for deregistering handlers. Finally, it checks whether the thread is idle-looping by calling the parent class's `checkForLooping()` method, which has the thread either sleep or reinitialize the selector if idle looping is detected. Finally, if the dispatcher thread has closed, the loop terminates. Then, all of the `IoSocketHandler` objects are deregistered and the selector is closed.

2.3 `EchoHandler` calling sequence

2.3.1 System Initialization

The trace will begin with the initialization of the server. In EchoServerTest.java on line 50, a new `EchoServer` object is created with 0 as the listen port number. The constructor of `EchoServer` is defined on EchoServer.java line 54. In the constructor, a `EchoHandler` is first instatiated on line 56. Then, a new `Server` object is instantiated on line 68 and is passed the port number and the `EchoHandler`. The server's constructor (Server.java line 483) calls `setHandler()` (line 498) which creates a `HandlerAdapter` object that wraps the `EchoHandler` passed to it.

The server is then set to asynchronous flush mode for the connection (line 69) and is then started (line 71). After the server is started, the server's `run()` method is invoked (Server.java line 586). Here, the server records the startup time (line 593), creates a handler to be invoked when the server shuts down (line 595), registers it (line 599) and calls the acceptor's `listen()` method (line 603).

The acceptor contains a reference to a `LifeCycleHandler` object. The acceptor's `listen()` method first calls the `onConnected()` method of the `LifeCycleHandler` (IoAcceptor.java line 205) which initializes the server's listener objects. Then, the acceptor calls `accept()` to enter its main loop (IoAcceptor.java line 206).

2.3.2 Making the connection

To trace the call sequence for a request, we start in the main loop of the acceptor in IoAcceptor.java lines 212-233. The acceptor's main loop iterates as long as the acceptor is open. It first calls `accept()` on the acceptor's `ServerSocketChannel`. Note that in the constructor of `IoAcceptor` on line 111, `serverChannel` is set to be blocking. As a result, this `accept()` call will block. Once a connection is established, the acceptor retrieves the next dispatcher thread from the dispatcher thread pool in a round-robin manner (line 219), creates a `IoChainableHandler` (line 220), which is passed the dispatcer and the socket channel. Following the `createIoHandler()`, we see that the type of `IoChainableHandler` is `IoSocketHandler` (IoProvider.java line 512). Finally, the acceptor calls the `onConnectionAccepted()` method of the `LifeCycleHandler`, passing it the `IoChainableHandler` (line 223) and increments the number of connections accepted.

The `LifeCycleHandler` checks to see if the maximum number of connections have been exceeded (Server.java line 1134). If not, it creates a `NonBLockingConnection` object to represent the connection and initializes the connection (lines 1170-1171) and sets the timeout values (more in next section). The `NonBlockingConnection` is also the `EchoHandler`.

The initialization of `NonBlockingConnection` is fairly complex. It creates a new `HandlerAdapter` to contain the `EchoHandler` (NonBlockingConnection.java line 722). On line 1191 of Server.java, the `init()` method of the `NonBlockingConnection` is called and is passed the `IoChainableHandler`. Tracing to NonBlockingConnection.java line 1042, we see that this overloaded method simply calls the other `init()` method, passing the `IoSocketHandler` and a `IoHandlerCallback` called `ioHandlerCallback`. Note that the `IoHandlerCallback` class, defined on line 2465, is actually a callback object for the `NonBlockingConnection` itself, and this will be important later.

This `init()` method of the `NonBlockingConnection` then calls the `init()` method of the handler (line 1049), pasing the `IoHandlerCallback` callback object for itself. Finally, we see that the handler registers itself with the dispatcher in read mode (IoSocketHandler.java line 142). When the dispatcher registers the handler, it registers the handler's channel with its selector. As described in the xsocket documentation, a connection is associated with the same dispatcher for its entire lifecycle. We now continue our trace of the execution in the dispatcher's main loop.

2.3.3 Dispatcher

The dispatcher calls `select()` on line 224 of IoSocketDispatcher.java. If it detects that new handlers have been registers or selection keys have been updated, it calls `handleReadWriteKeys()` on line 230 (IoSocketDispatcher.java). This method is defined on line 261. It loops through each selection key. For each selection key, it obtains the attached `IoSocketHandler` on line 272. We recall that in the `EchoServerTest`, a short string `"test\r\n"` is written to the server.

Thus, at this point, the key is readable, and the dispatcher calls `onReadableEvent()` on line 278, which in turn calls the `onReadableEvent()` method of the handler on line 301. Then, on line 296 of IoSocketHandler.java, the handler in turn calls `onData()` of its reference to the `IoHandlerCallback` of the `NonBlockingConnection`. This method is defined on line 1208 of `NonBlockingConnection` and simply calls `appendDataToReadBuffer()` if the `ByteBuffer` pased to it is not null. The `appendDataToReadBuffer()` method is inherited from its parent class, `AbstractNonBlockingStream` and is defined on line 1484 of AbstractNonBlockingStream.java. Here, the `ByteBuffer` and size read are appended to a `readQueue` and `onPostAppend()` is called (lines 1485 and 1486), which checks if the read buffer is full and suspends the current I/O thread (not the dispatcher thread) if the read buffer is full. In short, the sequence of calls starting with the `onData()` call of the `IoHandlerCallback` handles the sequence of asynchronous reads until the entire request has been read.

Then, we return to the `IoSocketHandler` which then calls its `IoHandlerCallback`'s `onPostData()` method on line 297 of IoSocketHandler.java. This method simply calls the `onPostData()` method of `NonBlockingConnection`, which obtains the `HandlerAdapter` containing `EchoHandler` atomically and calls the `HandlerAdapter`'s `onData()` method (NonBlockingConnection.java line 1316). This method is defined on line 164 of HandlerAdapter.java. This method eventually calls `performOnData()`, which is defined on line 221. Finally, `performOnData()` calls `EchoHandler`'s `onData()` method. This completes our trace.

2.4 Idle timeout

The xsocket framework tests for two kinds of timeout: connection and idle timeout. Idle timeout occurs when no data is received for a certain amount of time defined by the idle timeout time whereas the connection timeout is the timeout time for the entire lifetime of the connection.

A client-server connection is represented by a `NonBlockingConnection` object. In the `init()` method of the inner class `Server.LifeCycleHandler` (Server.java line 1184), the `setIdleTimeoutMillis()` method (line 1194) and the `setConnectionTimeoutMillis()` methods (line 1195) of `NonBlockingConnection` are called to set the idle and connection timeout times. Both methods are methods of the `IConnection` interface, which is used to represent an end-to-end connection or session that contains the backing socket channel.

The `NonBlockingConnection` class also provides the method `checkIdleTimeout()` (line 1401) and `checkConnectionTimeout()` (line 1427) that check whether the an idle timeout or a connection timeout has occurred.

The actual timeout checking is done by a `WachdogTask`, a private inner class of `ConnectionManager` (ConnectionManager.java line 237). This class extends `TimerTask` and is scheduled to be run at regular time intervals by a timer. Every time the `WachdogTask` is run, it calls `check()` (defined ConnectionManger.java line 257) which iterates through each `NonBlockingConnection` and calls the two aforementioned timeout checking methods (lines 294 and 299) via `checkTimeout()` (line 291).

2.5 Testing

Testing for xsocket is usually by done by instantiating a server object and executing client side code that connects with the server through a local socket and having the client code assert certain expected output from the server. In the `EchoServerTest`, the test code creates a `EchoServer` object, opens a local socket to the server, writes to it, and asserts if the same message is echoed back. This is possible because the server runs in separate threads. The second test `testXSocketBothSide` similarly sends the server requests and tests whether the response received is as expected, but it executes that code in a separate thread that simulates the client.

A way to test our server with idle timeout would be similar to the way that the xsocket `TimeoutTest` performs testing. The `TimeoutTest` tests both idle and connection timeouts by setting a timeout on the server, opening a connection from the client, and then stalling for a certain amount of time that would cause the server to timeout. To test for a specific timeout event, such as an idle timeout and not a connection timeout, it sets the connection timeout to a t_c and the idle timeout to a t_i such that $t_c > t_i$, open a connection, and have the client send an imcomplete request and sleep for a time t such that $t_i < t < t_c$ and then check that the idle timeout event was triggered but the connection timeout has not.

Since our server does not implement connection timeout, we can simply assume that as long as data is being received, the connection will remain open. Let our timeout time be t seconds. With proper adjustments to our server code, our test program can instantiate a server, then open a socket to the server and simply sleep for $t + 1$ seconds then wake up and check that the server has timeout and closed the connection.

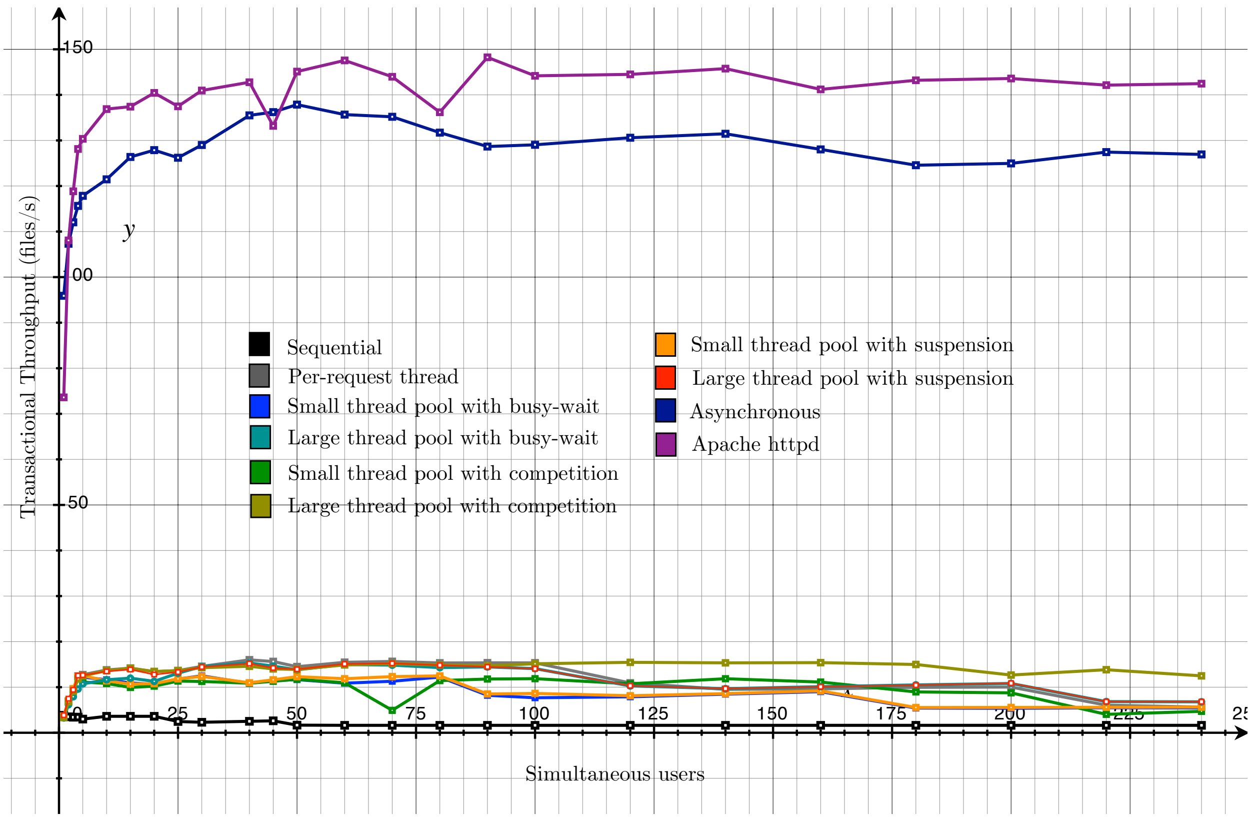

Figure 1: Performance benchmarks with transactional throughput on the y axis and number of simultaneous users (threads) on the x axis.

3 Performance benchmarking

A set of test files and request pattern files were generated with the scripts provided in gen.tar. The same request pattern was used for all of the servers. Each server was tested on a separate zoo machine, and the zoo Apache server was benchmarked per the instructions on the assignment page. The thread pool servers were each tested with a small thread pool of 4 threads and a large thread pool of 24 threads. Transactional throughput in files per second was recorded for each benchmark. The results of the benchmark were graphed and are shown in Figure 1.

As expected, the sequential server performs the worst. With a large number of simultaneous users, it becomes overwhelmed. The per-request server performs the best out of the threaded servers initially. However, at a certain point, the thread overhead begins to overtake the performance increase.

Clearly, the asynchronous server greatly outperforms the sequential and all of the threaded servers by serveral orders of magnitude, though it is slightly outperformed by the more highly-optimized Apache. Although the event driven architecure is much more complex than the synchronous multithreaded ones, the improvement in performance is dramatic.

For clarity, the graphs of the performance of threaded and sequential servers is shown in detail in Figure 2. For each of the thread pool servers, running with a large thread pool indeed increases throughput to a certain extent. However, this performance increase is nominal

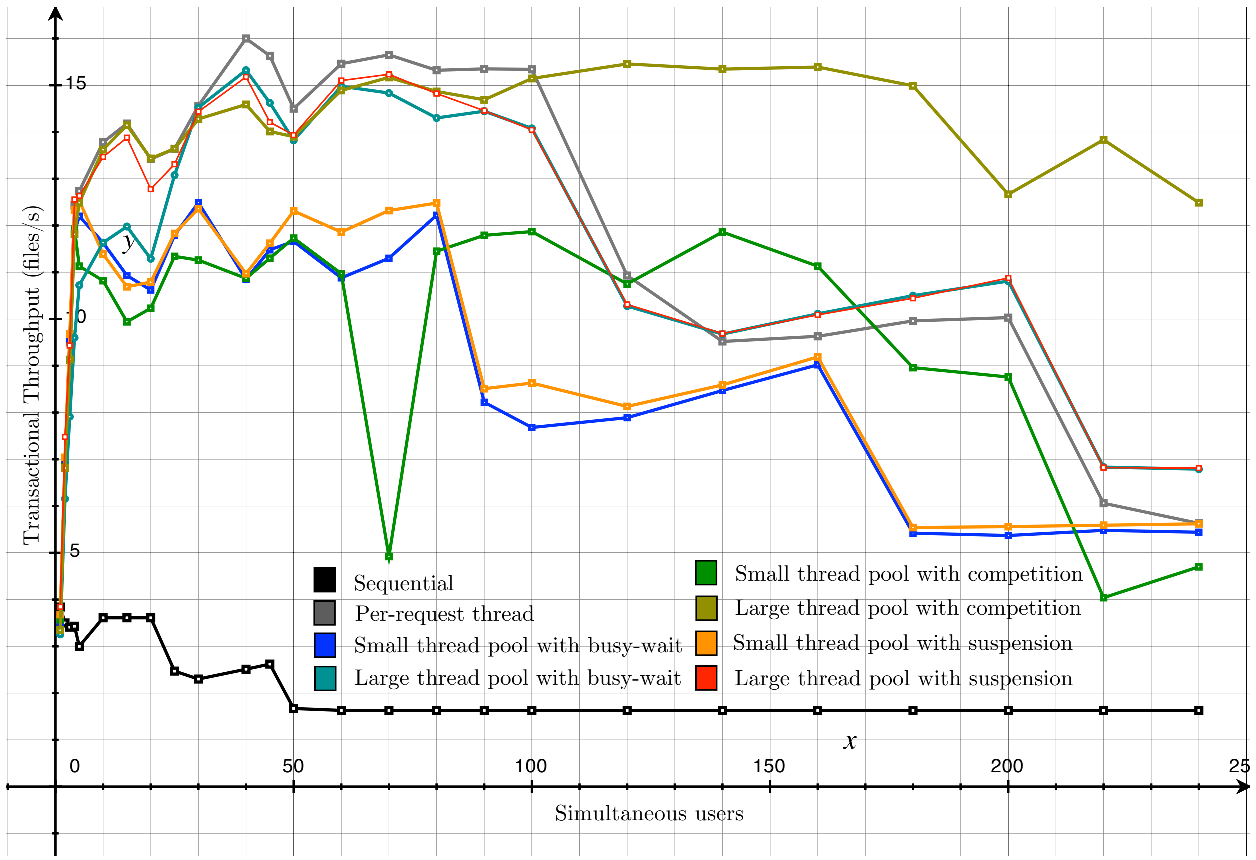

Figure 2: Performance benchmarks in detail showing only the thread pool, per-request, and sequential servers.

compared to the performance increase given by using asynchronous I/O. Furthermore, it is likely that at a certain point, the thread overhead will begin to overtake any performance advantage of a larger thread pool.