Creating a TCP Server

by Luke VanderHart

Problem

You want to open up a socket on a port to use as a low-level TCP server.

Solution

Use Java interop on the java.net.ServerSocket class to create a TCP listener. Use the functions in clojure.java.io to obtain input and output streams (or readers and writers) to read and write data to the socket:

```
(require '[clojure.java.io :as io])
(import '[java.net ServerSocket])

(defn receive
  "Read a line of textual data from the given socket"
  [socket]
  (.readLine (io/reader socket)))

(defn send
  "Send the given string message out over the given socket"
  [socket msg]
  (let [writer (io/writer socket)]
      (.write writer msg)
      (.flush writer)))

(defn serve [port handler]
  (with-open [server-sock (ServerSocket. port)
              sock (.accept server-sock)]
    (let [msg-in (receive sock)
          msg-out (handler msg-in)]
      (send sock msg-out))))
```

This code defines three functions. receive and send deal with reading and writing string data from and to a socket, using the clojure.java.io/reader and clojure.java.io/writer functions. Both of these accept a java.net.Socket as an argument and will return a java.io.Reader or java.io.Writer built from the socket's input and output streams.

The serve function handles actually creating an instance of ServerSocket on a particular port. It also takes a handler function, which will be used to process the incoming request and determine a response message.

After creating an instance of ServerSocket, serve immediately calls its accept method, which blocks until a TCP connection is established. When a client connects, it returns the session as an instance of java.net.Socket.

It then passes the socket to the receive function, which opens up a reader on it and blocks until it receives a full line of input, terminated by a newline character (\n). When it receives one, it calls the handler function with the resulting value, and calls send to send the response using a writer opened on the same socket. send also calls the flush method on the writer to ensure that all the data is actually sent back to the client, instead of being buffered in the Writer instance.

After sending the response, the serve function returns. Because it used the with-open macro when creating the server socket and the TCP session socket, it will invoke the close function on each before returning, which disconnects the client and ends the session.

To try it out, invoke the serve function in the REPL. For a simple example, use (serve 8888 #(.toUpperCase %)). Note that it won't return right away; it blocks, waiting for a client to connect.

To connect to the server you can use a *telnet* client, which is installed by default on nearly every operating system. To use it, open up a command-line window:

```
$ telnet localhost 8888
Trying ::1...
Connected to localhost.
Escape character is '^]'.
```

At this point you can type anything you like (in the following example, the input is "Hello, World!"). When you finish, make sure you type Enter or Return to send a newline character:

```
$ telnet localhost 8888
Trying ::1...
Connected to localhost.
Escape character is '^]'.
Hello, World!
HELLO, WORLD!Connection closed by foreign host
```

As you can see, as soon as you type a newline, the server responds with the uppercase version of your input (as per the handler function) and then immediately terminates the connection. In the REPL, you will find that the serve function has finally returned.

Discussion

This example uses readers and writers, which deal solely in textual data, to make the concepts of working with sockets easier to demonstrate. Of course, an actual socket is not limited to strings and can send and receive any kind of binary data.

To do this, simply use the clojure.java.io/input-stream and clojure.java.io/output-stream functions instead of the clojure.java.io/reader and clojure.java.io/writer functions, respectively, which return java.io.InputStream and java.io.OutputStream objects. These provide APIs for reading and writing raw bytes, rather than just strings and characters.

One thing you may have noticed about the example is that, unlike a traditional server, it doesn't actually continue to accept incoming connections after the serve function returns. For ongoing use, typically you'd like to be able to serve multiple incoming connections.

Fortunately, this is relatively straightforward to do given the concurrency tools that Clojure provides. Modifying the serve function to work as a persistent server requires three changes:

- Run the server on a separate thread so it doesn't block the REPL.
- Don't close the server socket after handling the first request.
- After handling a request, loop back to immediately handle another.

Also, because the server will be running on a non-REPL thread, it would be good to provide a mechanism for terminating the server other than killing the whole JVM.

The modified code looks like this:

```
(defn serve-persistent [port handler]
  (let [running (atom true)]
    (future
      (with-open [server-sock (ServerSocket. port)]
        (while @running
          (with-open [sock (.accept server-sock)]
            (let [msg-in (receive sock)
                  msg-out (handler msg-in)]
              (send sock msg-out))))))
    running))
```

The key feature of this code is that it launches the server socket asynchronously inside a future and calls the accept method inside of a loop. It also creates an atom called running and returns it, checking it each time it loops. To stop the server, reset the atom to false, and the loop will break:

```
(def a (serve-persistent 8888 #(.toUpperCase %)))
;; -> #'my-server/a

;; Server is running, will respond to multiple requests

(reset! a false)
;; -> false
;; Server is stopped, will stop serving requests after the next one
```

When to Use Sockets

As you can see from these examples, raw server sockets are a fairly low-level networking construct. Using them effectively means either creating your own data protocol or re-implementing an existing one, and handling all the fiddly bits of connecting, flushing, and disconnecting input and outputs streams yourself.

If your communication needs can be met by some existing protocol or communication technique (such as HTTP, SSH, or a message queue), you should almost certainly use that instead. There are widely available servers and libraries for these protocols that allow programming at a much higher level of abstraction, with much better performance and resiliency.

Still, understanding how all these different techniques work on a low level is valuable. At least as far as the JVM is concerned, most networking code ultimately bottoms out in calls to the raw socket mechanisms described in this recipe. Understanding how they work is key to understanding how higher-level networking tools (such as HTTP requests or JMS queues) actually work.

See Also

- The API documentation for ServerSocket and Socket objects in Java
- The API documentation for the clojure.java.io namespace
- [sec_network_io_tcp_client]
- Wikipedia on the TCP protocol