Creating a TCP Client

by Luke VanderHart

Problem

You want to open a TCP connection to a remote host, on a particular port.

Solution

Use Java interop to create an instance of java.net.Socket and connect to a remote host.

For example, the following code uses a Socket to create a TCP connection and send an HTTP GET request, returning the result as a string:

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

(defn send-request
  "Sends an HTTP GET request to the specified host, port, and path"
  [host port path]
  (with-open [sock (Socket. host port)
              writer (io/writer sock)
              reader (io/reader sock)
              response (StringWriter.)]
    (.append writer (str "GET " path "\n"))
    (.flush writer)
    (io/copy reader response)
    (str response)))
```

This function obtains instances of java.io.Writer and java.io.Reader to send and receive data to and from the remote server. By appending strings that conform to the HTTP specification to the writer, it forms a rudimentary HTTP client and executes a GET request to the specified endpoint. The results are then copied into an instance of java.io.StringWriter using the clojure.java.io/copy utility function, and returned as a string.

Invoking (send-request "google.com" 80 "/") at the REPL should return a very long string, consisting of the entire HTTP response that is the Google home page.

Discussion

This example uses the clojure.java.io namespace to obtain instances of java.io.Writer and java.io.Reader to read and write

textual data to/from the network socket. In point of fact, Socket instances are not actually limited to textual data, and it would be possible to obtain raw binary input and output streams just as easily using clojure.java.io/input-stream and clojure.java.io/output-stream, respectively. Since HTTP is a textual protocol, however, it makes more sense to use the higher-level features of Reader and Writer.

Caution	This example uses HTTP because it's a protocol that many readers are familiar with. In the real world, using a raw TCP socket for HTTP requests is almost certainly a terrible idea. There are a plethora of libraries that provide a much higher-level interface to HTTP requests and responses, and encapsulate a lot of pesky details such as escaping, encoding, and formatting.

Also note that the reader, the writer, and the socket itself are bound within the context of a with-open macro. This guarantees that the close method is called when they are finished, which releases the TCP connection. If the connection is not released, it will continue to consume resources on both the client and the server and may be subject to termination on the remote side.

When returning lazy sequences from a with-open context, it is important to fully realize those sequences using doall. This is because resources opened by with-open are *only* available inside the with-open block. The doall function fully realizes a collection, retaining its entire contents in memory:

```
(realized? (range 100))
;; -> false

(realized? (doall (range 100)))
;; -> true
```

Depending on your application, you may prefer to use the doseq macro. Instead of retaining the entire sequence, doseq executes its body for each element of the sequence. This is useful if you need to cause side effects for each element of a sequence, but need to hang on to the entire thing:

```
(doseq [n (range 3)]
  (println n))
;; *out*
;; 0
;; 1
;; 2
```

See Also

- [sec_network_io_tcp_server]
- Wikipedia on the TCP protocol