Network Programming in ANSI Common Lisp with IOLib

Table of Contents

Revisions

Date	Version	Revision	Name
04/02/2010	0	0	Peter Keller (psilord@cs.wisc.edu)
02/11/2022	0	1	Angelo Rossi (angelo.rossi.homelab@gmail.com)

What is IOLib?

IOLib is a portable I/O library for ANSI Common Lisp. It includes socket interfaces for network programming with IPV4/IPV6 TCP and UDP, an I/O multiplexer that includes nonblocking I/O, a DNS resolver library, and a pathname library.

Where do I get IOLib?

The current version of IOLib is found here:

<http://common-lisp.net/project/iolib/download.shtml>

Please use the repository located in the Live Sources section for the most up to date version of IOLib.

1 Introduction

This tutorial loosely follows the exposition of network programming in "UNIX Network Programming, Networking APIs: Sockets and XTI 2nd Edition" by W. Richard Stevens ([STEVENS1997]). Many examples are derived from the source codes in that book. Major deviations from the C sources include converting the concurrent examples which use fork() into threaded examples which use the portable Bordeaux Threads package, more structured implementations of certain concepts such as data buffers and error handling, and general movement of coding style towards a Common Lisp viewpoint.

The scope of this version of the tutorial is:

1. Exposition suitable for programmers unfamiliar with ANSI Common Lisp
2. IPV4 TCP
3. Client/Server architecture
4. Iterative vs Concurrent (via threading) vs Multiplexed Server Design
5. Blocking and nonblocking I/O

It is intended, however, that this tutorial grows to contain the entirety of IOLib's API as detailed in the Future Directions section of this tutorial. As newer revisions of this tutorial are released, those gaps will be filled until the whole of the IOLib API has been discussed.

Finally, the example code in this tutorial is algorithmically cut from the actual example programs and inserted into the tutorial via a template generation method. The example codes have embedded in them a tiny markup language which facilitates this in the form (on a single line) of ';; ex-NNNb' to begin an example section, and ';; ex-NNNe' to end an example section--NNN stands for an enumeration integer for which each section's begin and end must match.

1.1 Acknowledgements

I would like to greatly thank Stelian Ionescu, the author of IOLib for his exposition of the various features of IOLib and his patience in our sometimes long conversations.

1.2 Supporting Code

The file package.lisp contains a small library of codes used widely in the examples. The supporting code implements:

1. The package containing the examples, called :iolib.examples.
2. The variables *host* and *port*, set to "localhost" and 9999 respectively. This is the default name and port to which client connect and servers listen. Servers usually bind to 0.0.0.0, however.
3. A small, but efficient, queue implementation, from "ANSI Common Lisp" by Paul Graham. The interface calls are:

   ```
   (make-queue)
   (enqueue obj q)
   (dequeue q)
   (empty-queue q)
   ```

4. :iolib.examples currently depends upon IOLib alone and uses packages :common-lisp, :iolib, and :bordeaux-threads.

1.3 Running the Examples

These examples were developed and tested on SBCL 1.0.33.30 running on an x86 Ubuntu 8.10 machine. They were ran with two sessions of SBCL running, one acting as a client, and the other as a server.

Supposing we'd like to start up the first example of the daytime server and connect to it with the first daytime client example. Initially, the server will bind to *host* and *port* and wait for the client to connect. We connect with the client to *host* and *port*, get the time, and exit.

First we'll start up a server:

```
Linux black > sbcl
This is SBCL 2.1.1.debian, an implementation of ANSI Common Lisp.
More information about SBCL is available at <http://www.sbcl.org/>.

SBCL is free software, provided as is, with absolutely no warranty.
It is mostly in the public domain; some portions are provided under
BSD-style licenses.  See the CREDITS and COPYING files in the
distribution for more information.
* (require :iolib.examples) ; much output!
* (in-package :iolib.examples)

#
* (run-ex1-server)
Created socket: #[fd=5]
Bound socket: #
Listening on socket bound to: 0.0.0.0:9999
Waiting to accept a connection...
[ server is waiting for the below client to connect! ]
Got a connection from 127.0.0.1:34794!
Sending the time...Sent!
T
*
```

Now we'll start up the client which connected to the above server:

```
Linux black > sbcl
This is SBCL 2.1.1.debian, an implementation of ANSI Common Lisp.
More information about SBCL is available at <http://www.sbcl.org/>.

SBCL is free software, provided as is, with absolutely no warranty.
It is mostly in the public domain; some portions are provided under
BSD-style licenses.  See the CREDITS and COPYING files in the
distribution for more information.
* (require :iolib.examples) ; much output!
* (in-package :iolib.examples)

#
* (run-ex1-client)
Connected to server 127.0.0.1:9999 via my local connection at 127.0.0.1:34794!
2/27/2010 13:51:48
T
*
```

In each client example, one can specify which host or port to which it should connect:

```
* (run-ex1-client :host "localhost" :port 9999)
Connected to server 127.0.0.1:9999 via my local connection at 127.0.0.1:34798!
2/27/2010 13:53:7
T
*
```

The servers can be told a port they should listen upon and in this tutorial, unless otherwise specified, will always bind to 0.0.0.0:9999 which means across all interfaces on the machine and on port 9999.

2 IPV4 TCP Client/Server Blocking and nonblocking I/O

2.1 Overview of Examples

The examples consist of a collection of clients and servers. They are split into two groups: a set of daytime clients and server, and echo clients and servers. In some of the examples, a certain network protocol, suppose end-of-file handling, must be matched between client and server causing further delineation.

Client protocols are matched to server protocols thusly:

Clients: ex1-client, ex2-client, ex3-client, can work with servers: ex1-server, ex2-server, ex3-server, ex4-server.

Clients: ex4-client, ex5a-client, can work with servers: ex5-server, ex6-server.

Clients: ex5b-client, can work with servers: ex7-server, ex8-server

Some clients and servers use the "daytime" series of protocols, those are ex1-client, ex2-client, ex3-client, and ex1-server, ex2-server, ex3-server, and ex4-server.

Some clients and servers use the "echo a line" series of protocols, those are ex4-client, ex5a-client, ex5b-client, and ex5-server, ex6-server, ex7-server, and ex8-server.

Even though much of the example source is included in the tutorial, it is recommended that the example sources be carefully read and understood in order to gain the most benefit from the tutorial.

2.2 Daytime Clients

In this section we show the evolution of a client which connects to a server and gets the time of day. Each example shows some kind of an incremental improvement to the previous one.

2.2.1 Daytime Client IVP4/TCP: ex1-client.lisp

This example is a very simple daytime client program which contacts a server, by default at *host* and *port*, returns a single line of text that is the current date and time, and then exits. It is written in more of a C style just to make it easy to compare with similar simple examples in other languages. It uses blocking, line oriented I/O.

The steps this program performs are:

1. The ex1-client.lisp entrance call:

   ```
   (defun run-ex1-client (&key (host *host*) (port *port*))
   ```

2. Create an active TCP socket:

   The socket creation function (MAKE-SOCKET ...) is the method by which one creates a socket in IOLib. It is very versatile and can be used to both create and initialize the socket in a single call.

   In this case, we use it simply and create an active IPV4 Internet stream socket which can read or write utf8 text and that understands a particular newline convention in the underlying data.

   One small, but important, deviation of IOLib sockets from Berkeley sockets is that when a socket is created, it is predestined to forever and unalterably be either an active or passive socket. Active sockets are used to connect to a server and passive sockets are used for a server's listening socket.

   ```
   ;; Create a internet TCP socket under IPV4
   (let ((socket (make-socket :connect :active
                              :address-family :internet
                              :type :stream
                              :external-format '(:utf-8 :eol-style :crlf)
                              :ipv6 nil)))
   ```

3. Specify the Server's IP address and port and establish a connection with the server:

   This bit of code contains many calls into IOLib and we shall examine each of them.

   The function LOOKUP-HOSTNAME takes as a string the DNS name for a machine and returns 4 values:

   A. an address

   B. a list of additional addresses(if existent)

   C. the canonical name of the host

   D. an alist of all the host's names with their respective addresses

   We use only the first return value, the address component, to pass to the function CONNECT.

   The function CONNECT will connect the socket to the address, but to a random port if the :port keyword argument is not specified. The average client codes usually use :wait t to block until the connect can resolve with a connected fd or an error. The exception to always using :wait t is if the client needs to connect to many servers at once, suppose a web client, or if a server is also a client in other contexts and wishes not to block.

   The functions REMOTE-HOST and REMOTE-PORT return the ip address and port of the remote connection associated with the connected socket. LOCAL-HOST and LOCAL-PORT return the information of the client's end of the connected socket. Analogous calls REMOTE-NAME and

LOCAL-NAME each return two values where the first value is the equivalent of *-host and the second value is the equivalent of *-port.

```
;; do a blocking connect to the daytime server on the port.
(connect socket (lookup-hostname host) :port port :wait t)
(format t "Connected to server ~A:~A via my local connection at ~A:~A!~%"
        (remote-host socket) (remote-port socket)
        (local-host socket) (local-port socket))
```

4. Read and display the server's reply:

   Now that the socket has been connected to the server, the server will send a line of text to the client. The client uses the standard Common Lisp function READ-LINE to read the information from the socket. The function READ-LINE blocks and will only return when an *entire line* is read. Once read, the line is emitted to *standard-output* via the function call FORMAT.

```
;; read the one line of information I need from the daytime
;; server.  I can use read-line here because this is a TCP socket.
(let ((line (read-line socket)))
  (format t "~A" line))
```

5. End program:

   We close the socket with the standard function CLOSE and return true so the return value of this example is t.

```
;; all done
(close socket)
t))
```

While this program works, it has some major flaws in it. First and foremost is that it doesn't handle any conditions that IOLib signals in common use cases. An example would be to run the ex1-client.lisp example without a daytime server running. In most, if not all, Common Lisp toplevels, you'll be dropped into the debugger on an unhandled SOCKET-CONNECTION-REFUSED-ERROR condition. Secondly, it isn't written in the Common Lisp style.

2.2.2 Daytime Client IVP4/TCP: ex2-client.lisp

In this example, we simply tackle the fact ex1-server.lisp can be shortened with an IOLib form to something where the application writer has less to do concerning cleaning up the socket object. It also uses line oriented blocking I/O.

The introduced macro WITH-OPEN-SOCKET calls MAKE-SOCKET with the arguments in question and binds the socket to the variable 'socket'. When this form returns, it will automatically close the socket.

This shortens the program so much, that the example can be included in its entirety:

```
(defun run-ex2-client (&key (host *host*) (port *port*))

  ;; We introduce with-open-socket here as a means to easily wrap
  ;; usually synchronous and blocking communication with a form that
  ;; ensures the socket is closed no matter how we exit it.
  (with-open-socket (socket :connect :active
                            :address-family :internet
                            :type :stream
```

```
                             :external-format '(:utf-8 :eol-style :crlf)
                             :ipv6 nil)

    ;; Do a blocking connect to the daytime server on the port.  We
    ;; also introduce lookup-hostname, which converts a hostname to an
    ;; 4 values, but in our case we only want the first, which is an
    ;; address.
    (connect socket (lookup-hostname host) :port port :wait t)
    (format t "Connected to server ~A:~A from my local connection at ~A:~A!~%"
            (remote-name socket) (remote-port socket)
            (local-name socket) (local-port socket))

    ;; read the one line of information I need from the daytime
    ;; server.  I can use read-line here because this is a TCP
    ;; socket. It will block until the whole line is read.
    (let ((line (read-line socket)))
      (format t "~A" line)
      t)))
```

This shorthand can go even further, if we add this to the WITH-OPEN-SOCKET flags

```
:remote-host (lookup-hostname host)
:remote-port port
```

then the underlying MAKE-SOCKET call will in fact connect the socket directly to the server before it is available for the body of the macro allowing us to remove the connect call entirely! In the early examples, however, we don't utilize IOLib's shorthand notations to this degree in order to make apparent how the library maps into traditional socket concepts. After one gains familiarity with the IOLib API, the situations where application of the shortcuts are useful become much easier to see.

2.2.3 Daytime Client IVP4/TCP: ex3-client.lisp

Now we come to condition handling, which can moderately affect the layout of your IOLib program. Any real program using IOLib must handle IOLib's signaled conditions which are common to the boundary cases of network programming. We've already seen one of these boundary cases when we tried to connect a daytime client to a server that wasn't running. The condition signaled in that case was: SOCKET-CONNECTION-REFUSED-ERROR. The stream interface has a set of conditions which IOLib will signal, and another lower level IOLib layer--which we'll come to in the nonblocking I/O examples have another set of conditions. There is some intersection between them and we will explore that later. For now, we'll just use the conditions associated with a stream.

Our rewrite of ex2-client.lisp into ex3-client.lisp (continuing to use line oriented blocking I/O) proceeds thusly:

1. We create a helper function which connects to the server and reads the daytime line:

   Notice the HANDLER-CASE macro around the portion of the function which reads the date from the server. In looking at the boundary conditions from the server given this protocol, we can receive an END-OF-FILE condition if the client connected, but before the server could respond it exited, closing the connection. Since in this case we're inside of a WITH-OPEN-SOCKET form, we simply note that we got an END-OF-FILE and let the cleanup forms of WITH-OPEN-SOCKET close the connection. If we don't catch this condition, then the program will break into the debugger and that isn't useful. It is usually debatable as to where one should handle conditions: either near to or far away from the generating calls. In these simple examples, no choice has any significant pros or cons. As your IOLib programs become more and more complex, however, it becomes more obvious at what abstraction level to handle signaled conditions.

```
(defun run-ex3-client-helper (host port)

  ;; Create a internet TCP socket under IPV4
  (with-open-socket
    (socket :connect :active
            :address-family :internet
            :type :stream
            :external-format '(:utf-8 :eol-style :crlf)
            :ipv6 nil)

    ;; do a blocking connect to the daytime server on the port.
    (connect socket (lookup-hostname host) :port port :wait t)
    (format t "Connected to server ~A:~A from my local connection at ~A:~A!~%"
            (remote-name socket) (remote-port socket)
            (local-name socket) (local-port socket))

    (handler-case
        ;; read the one line of information I need from the daytime
        ;; server.  I can use read-line here because this is a TCP
        ;; socket. It will block until the whole line is read.
        (let ((line (read-line socket)))
          (format t "~A" line)
          t)

      ;; However, let's notice the signaled condition if the server
      ;; went away prematurely...
      (end-of-file ()
        (format t "Got end-of-file. Server closed connection!")))))
```

2. Some conditions which are complete show-stoppers to the functioning of the code are caught at a higher level:

   Notice we catch the possible SOCKET-CONNECTION-REFUSED-ERROR from the connect inside of the function run-ex3-client-helper.

```
;; The main entry point into ex3-client
(defun run-ex3-client (&key (host *host*) (port *port*))
  (handler-case

      (run-ex3-client-helper host port)

    ;; handle a commonly signaled error...
    (socket-connection-refused-error ()
      (format t "Connection refused to ~A:~A. Maybe the server isn't running?~%"
              (lookup-hostname host) port))))
```

Here are some common conditions in IOLib (some from ANSI Common Lisp too) and under what situations they are signaled. In any IOLib program, *at least* these conditions should be handled where appropriate.

END-OF-FILE:

When a stream function such as READ, READ-LINE, etc...(but not RECEIVE-FROM), reads from a socket where the other end has been closed.

HANGUP:

When writing to a socket with a stream function such as WRITE, FORMAT, etc...(but not SEND-TO), if the socket is closed then this condition is signaled.

SOCKET-CONNECTION-RESET-ERROR:

When doing I/O on a socket and the other side of the socket sent a RST packet, this condition is signaled. It can also happen with the IOLIb function ACCEPT and similar.

SOCKET-CONNECTION-REFUSED-ERROR:

Signaled by connect if there is no server waiting to accept the incoming connection.

2.3 Daytime Servers

Now that we have completed the evolution of the daytime client, let's look at the daytime servers.

The exposition of the servers follows in style of the clients.

2.3.1 Daytime Server IVP4/TCP: ex1-server.lisp

This first example is an iterative server which handles a single client and then exits. The I/O is blocking and no error handling is performed. This is similar in scope to the ex1-client.lisp example.

1. Create the server socket:

   We see that the socket is :passive. Every socket in IOLib is predestined to be either an active or passive socket and since this is a server socket, it is passive. Also here we see that we can ask for the underlying fd of the socket with the function SOCKET-OS-FD.

```
(defun run-ex1-server (&key (port *port*))
  ;; Create a passive (server) TCP socket under IPV4 Sockets meant to
  ;; be listened upon *must* be created passively. This is a minor
  ;; deviation from the Berkeley socket interface.
  (let ((socket (make-socket :connect :passive
                             :address-family :internet
                             :type :stream
                             :external-format '(:utf-8 :eol-style :crlf)
                             :ipv6 nil)))
    (format t "Created socket: ~A[fd=~A]~%" socket (socket-os-fd socket))
```

2. Bind the socket

   Binding a socket is what gives it an endpoint to which clients can connect. The IOLib constant +IPV4-UNSPECIFIED+ represents 0.0.0.0 and means if a connection arrives on any interface, it will be accepted if it comes to the :port specified. The :reuse-addr keyword represents the socket option SO_REUSEADDR and states (among other things) that if the socket is in the TIME_WAIT state it can be reused immediately. It is recommended that all servers use :reuse-addr on their listening socket.

```
;; Bind the socket to all interfaces with specified port.
(bind-address socket
              +ipv4-unspecified+ ; which means INADDR_ANY or 0.0.0.0
              :port port
              :reuse-addr t)
(format t "Bound socket: ~A~%" socket)
```

3. Listen on the socket

   Listening on a socket allows clients to connect. In this example, we've specified that 5 pending connection can be queued up in the kernel before being accepted by the process.

```
;; Convert the sockxet to a listening socket
(listen-on socket :backlog 5)
(format t "Listening on socket bound to: ~A:~A~%"
        (local-host socket) (local-port socket))
```

4. Accept the client connection.

   Here we finally call the IOLib function ACCEPT-CONNECTION. We would like it to block, so we pass it :wait t. When ACCEPT-CONNECTION returns it will return a new socket which represents the connection to the client. ACCEPT-CONNECTION can return nil under some situations, such as on a slow server when the client sent a TCP RST packet in between the time the kernel sees the connection attempt and ACCEPT-CONNECTION is actually called. We also opt to use the function REMOTE-NAME, which returns two values, the ip address and port of the remote side of the socket.

```
;; Block on accepting a connection
(format t "Waiting to accept a connection...~%")
(let ((client (accept-connection socket :wait t)))
  (when client
    ;; When we get a new connection, show who it is from.
    (multiple-value-bind (who rport)
        (remote-name client)
      (format t "Got a connection from ~A:~A!~%" who rport))
```

5. Write the time to the client.

   Here we've figured out the time string and wrote it to the client. Notice we call the function FINISH-OUTPUT. This ensures that all output is written to the client socket. For streams using blocking I/O, it is recommended that every write to a blocking socket be followed up with a call to FINISH-OUTPUT.

```
;; Since we're using a internet TCP stream, we can use format
;; with it. However, we should be sure to call finish-output on
;; the socket in order that all the data is sent. Also, this is
;; a blocking write.
(multiple-value-bind (s m h d mon y)
    (get-decoded-time)
  (format t "Sending the time...")
  (format client "~A/~A/~A ~A:~A:~A~%" mon d y h m s)
  (finish-output client))
```

6. Close the connection to the client.

   We're done writing to the client, so close the connection so the client knows it got everything.

```
;; We're done talking to the client.
(close client)
(format t "Sent!~%"))
```

7. Close the server's socket.

   Since this is a one shot server, we close the listening socket and exit. In this and all other servers we call FINISH-OUTPUT to flush all pending message to *standard-output*, if any.

```
;; We're done with the server socket too.
(close socket)
(finish-output)
t)))
```

The above code is the basic idea for how a very simple TCP blocking I/O server functions. Like ex1-client, this server suffers from the inability to handle common signaled conditions such as a HANGUP from the client--which means the client went away before the server could write the time to it.

However, one major, and subtle, problem of this particular example is that the socket to the client is *not immediately closed* if the server happens to exit, say by going through the debugger back to toplevel--or a signaled condition, before writing the date to the client. If this happens, it can take a VERY long time for the socket to be garbage collected and closed. In this scenario, the client will hang around waiting for data which will never come until the Lisp implementation closes the socket when it gets around to collecting it. Garbage collection is an extremely nice feature of Common Lisp, but non-memory OS resources in general should be eagerly cleaned up. Clients can suffer from this problem too, leaving open, but unmanipulable, sockets to servers.

All clients or servers written against IOLib should either use some IOLib specific macros to handle closing of socket, Common Lisp's condition system like handler-case to catch the signaled conditions, or some other manual solution.

2.3.2 Daytime Server IVP4/TCP: ex2-server.lisp

Similarly to ex2-client, this server uses the macro WITH-OPEN-SOCKET to open the server socket. We introduce WITH-ACCEPT-CONNECTION to accept the client and convert this server from a single shot server to an iterative server which can handle, in a serial fashion only, multiple clients.

1. Serially accept and process clients:

   This portion of ex2-server shows the infinite loop around the accepting of the connection. The macro WITH-ACCEPT-CONNECTION takes the server socket and introduces a new binding: client, which is the accepted connection. We ensure to tell the accept we'd like to be blocking. If for whatever reason we exit the body, it'll clean up the client socket automatically.

```
;; Keep accepting connections forever.
(loop
    (format t "Waiting to accept a connection...~%")

    ;; Using with-accept-connection, when this form returns it will
    ;; automatically close the client connection.
    (with-accept-connection (client server :wait t)
      ;; When we get a new connection, show who it is from.
      (multiple-value-bind (who rport)
          (remote-name client)
        (format t "Got a connnection from ~A:~A!~%" who rport))

        ;; Since we're using a internet TCP stream, we can use format
        ;; with it. However, we should be sure to finish-output in
        ;; order that all the data is sent.
        (multiple-value-bind (s m h d mon y)
            (get-decoded-time)
          (format t "Sending the time...")
          (format client "~A/~A/~A ~A:~A:~A~%" mon d y h m s)
          (finish-output client)
          (format t "Sent!~%")
          (finish-output)
          t)))))
```

For very simple blocking I/O servers like this one, serially accepting and handling client connections isn't so much of a problem, but if the server does anything which takes a lot of time or has to send lots of data back and forth to many persistent clients, then this is a poor design. The means by which you exit this server is by breaking evaluation and returning to the toplevel. When this happens, the WITH-* forms automatically close the connection to the client.

2.3.3 Daytime Server IVP4/TCP: ex3-server.lisp

In this iterative and blocking I/O server example, we add the handling of the usual signaled conditions in network boundary cases often found with sockets. Like the earlier client where we introduced HANDLER-CASE, this involves a little bit of restructuring of the codes.

1. A helper function which opens a passive socket, binds it, and listens on it:

   There is nothing new in this portion of the code. We've seen this pattern before. In production code, we could probably shorten this further by having WITH-OPEN-SOCKET do the binding and connecting with appropriate keyword arguments.

```
(defun run-ex3-server-helper (port)
  (with-open-socket
    (server :connect :passive
            :address-family :internet
            :type :stream
            :ipv6 nil
            :external-format '(:utf-8 :eol-style :crlf))

    (format t "Created socket: ~A[fd=~A]~%" server (socket-os-fd server))

    ;; Bind the socket to all interfaces with specified port.
    (bind-address server +ipv4-unspecified+ :port port :reuse-addr t)
    (format t "Bound socket: ~A~%" server)

    ;; start listening on the server socket
    (listen-on server :backlog 5)
    (format t "Listening on socket bound to: ~A:~A~%"
            (local-host server)
            (local-port server))
```

2. Repeatedly handle clients in a serial fashion:

   The new material in this function is the HANDLER-CASE around sending the client the time information. The boundary conditions when writing to a client include the server getting a reset (RST) from the client or discovering the client had gone away and there is no-one to which to write. Since the write is contained within the WITH-ACCEPT-CONNECTION form, if any of these conditions happen, we simply notice that they happened and let the form clean up the socket when it exits. If we didn't catch the conditions, however, we'd break into the debugger.

   One might ask what the value of catching these conditions here is at all since we don't actually do anything with them--other than printing a message and preventing the code from breaking into the debugger. For the purposes of the tutorial, it is intended that the reader induce the boundary cases manually and see the flow of the code and to understand exactly what conditions may be signaled under what conditions and how to structure code to deal with them. In production code where the author might not care about these conditions at all, one might simply ignore all the signaled conditions that writing to the client might cause.

   Of course, the appropriateness of ignoring network boundary conditions is best determined by context.

```
;; keep accepting connections forever.
(loop
    (format t "Waiting to accept a connection...~%")
    ;; Here we see with-accept-connection which simplifies closing
    ;; the client socket when are done with it.
    (with-accept-connection (client server :wait t)
      ;; When we get a new connection, show who it
      ;; is from.
      (multiple-value-bind (who rport)
          (remote-name client)
        (format t "Got a connnection from ~A:~A!~%" who rport))

        ;; Since we're using an internet TCP stream, we can use format
        ;; with it. However, we should be sure to finish-output in
        ;; order that all the data is sent.
        (multiple-value-bind (s m h d mon y)
            (get-decoded-time)
          (format t "Sending the time...")

          ;; Catch the condition of the client closing the connection.
          ;; Since we exist inside a with-accept-connection, the
          ;; socket will be automatically closed.
          (handler-case
              (progn
                (format client "~A/~A/~A ~A:~A:~A~%" mon d y h m s)
                (finish-output client))

            (socket-connection-reset-error ()
              (format t "Client reset connection!~%"))

            (hangup ()
              (format t "Client closed conection!~%")))

          (format t "Sent!~%"))))
```

3. End of the helper function, returns T to whomever called it:

```
t))
```

4. The entry point into this example:

   We handle the condition SOCKET-ADDRESS-IN-USE-ERROR which is most commonly signaled when we try to bind a socket to address which already has a server running on it or when the address is in the TIME_WAIT state. The latter situation is so common--usually caused by a server just having exited and another one starting up to replace it, that when binding addresses, one should supply the keyword argument :reuse-addr with a true value to BIND-ADDRESS to allow binding a socket to an address in TIME_WAIT state.

```
;; This is the main entry point into the example 3 server.
(defun run-ex3-server (&key (port *port*))
  (handler-case

      (run-ex3-server-helper port)

    (socket-address-in-use-error ()
      ;; Here we catch a condition which represents trying to bind to
      ;; the same port before the first one has been released by the
      ;; kernel.  Generally this means you forgot to put ':reuse-addr
      ;; t' as an argument to bind address.
      (format t "Bind: Address already in use, forget :reuse-addr t?")))

  (finish-output))
```

2.3.4 Daytime Server IVP4/TCP: ex4-server.lisp

This is the first of our concurrent servers and the last of our daytime protocol servers. Usually concurrency is introduced (in the UNIX environment) with the fork() library call which creates an entirely new process with copy-on-write semantics to handle the connection to the client. In this tutorial environment, we've chosen to render this idea with the portable threading library Bordeaux Threads. The I/O is still line oriented and blocking, however, when a thread blocks another can run giving the illusion of a server handling multiple clients in a non-blocking fashion.

We also introduce UNWIND-PROTECT ensures that various sockets are closed under various boundary conditions in the execution of the server. An UNWIND-PROTECT executes a single form, and after the evaluation, or interruption, of that form, evaluates a special cleanup form. The cleanup form is *always* evaluated and we use this to cleanup non-memory system resources like sockets.

Threads present their own special problems in the design of a server. Two important problems are: data races and thread termination. The tutorial tries very hard to avoid any data races in the examples and this problem is ultimately solvable using Bordeaux-Threads mutexes or condition variables. Our simple examples do not need mutexes as they do not share any data between themselves.

The harder problem is thread termination. Since the tutorial encourages experimentation with the clients and servers in a REPL, threads may leak when the server process' initial thread stops execution and goes back to the REPL. We use three API calls from the Bordeaux Threads: THREAD-ALIVE-P, ALL-THREADS, and DESTROY-THREAD--which are not to be used in normal thread programming. We do this here in order to try and clean up leaked threads so the clients know immediately when the server process stopped and we don't pollute the REPL with an ever increasing number of executing threads. The employed method of destroying the threads, on SBCL specifically, allows the invocation of the thread's UNWIND-PROTECT's cleanup form, which closes the socket to the client before destroying the thread. On other implementations of Common Lisp, we are not guaranteed that the thread's UNWIND-PROTECT cleanup form will be evaluated when we destroy it.

This method is also extremely heavy handed in that it uses the function IGNORE-ERRORS to ignore any condition that Bordeaux Thread's DESTROY-THREAD may have signaled, including important conditions like HEAP-EXHAUSTED-ERROR, an SBCL specific condition. In a real threaded server, the exiting of the initial thread (which means exiting of the runtime and termination of the entire Lisp process) will destroy all other threads as the process tears itself down and exits. This is the recommended way a threaded server should exit.

Since threading is implementation dependent for what guarantees are provided, any non-toy threaded network server will probably use the native implementation of threads for a specific Common Lisp implementation. An example difficult situation would be trying to terminate a thread which is blocked on I/O. Different implementations would handle this in different ways.

The two provided examples, ex4-server and ex5-server, provide a general idea for the structuring of the code to utilize threads.

Here is the dissection of ex4-server:

1. A special variable which will allow the initial thread to pass a client socket to a thread handling said client:

```
;; This variable is the means by which we transmit the client socket from
;; the initial thread to the particular thread which will handle that client.
(defvar *ex4-tls-client* nil)
```

2. A helper function which begins with the usual recipe for a server:

```
(defun run-ex4-server-helper (port)
  (with-open-socket
    (server :connect :passive
            :address-family :internet
            :type :stream
            :ipv6 nil
            :external-format '(:utf-8 :eol-style :crlf))

    (format t "Created socket: ~A[fd=~A]~%" server (socket-os-fd server))

    ;; Bind the socket to all interfaces with specified port.
    (bind-address server +ipv4-unspecified+ :port port :reuse-addr t)
    (format t "Bound socket: ~A~%" server)

    ;; start listening on the server socket
    (listen-on server :backlog 5)
    (format t "Listening on socket bound to: ~A:~A~%"
            (local-host server)
            (local-port server))
```

3. Forever more, accept a client connection on the listening socket and start a thread which handles it:

   There is a lot going on in this piece of code. The first thing to notice is the UNWIND-PROTECT and its cleanup form. The form which UNWIND-PROTECT is guarding is an infinite loop which does a blocking accept to get a client socket, rebinds *default-special-bindings* adding to its assoc list the binding for *ex4-tls-client*, and creates a thread which handles the client.

   The cleanup form walks all of the active client threads and destroys them, ignoring any conditions that may have arose while doing so. Destroying the threads prevents them from piling up and eventually causing havoc if many servers start and exit over time. In addition, it forces an eager close on the client sockets allowing any clients to know the server went away immediately.

```
;; Here we introduce unwind-protect to ensure we properly clean up
;; any leftover threads when the server exits for whatever reason.
;; keep accepting connections forever, but if this exits for
;; whatever reason ensure to destroy any remaining running
;; threads.
(unwind-protect
    (loop                         ; keep accepting connections...
        (format t "Waiting to accept a connection...~%")
        (finish-output)
        (let* ((client (accept-connection server :wait t))
               ;; set up the special variable according to the
               ;; needs of the Bordeaux Threads package to pass in
               ;; the client socket we accepted to the about to be
               ;; created thread.  *default-special-bindings* must
               ;; not be modified, so here we just push a new scope
               ;; onto it.
```

```
                (*default-special-bindings* (acons '*ex4-tls-client* client
                                                   *default-special-bindings*)))

           ;; ...and handle the connection!
           (when client
             (make-thread #'process-ex4-client-thread
                          :name 'process-ex4-client-thread))))

       ;; Clean up form for uw-p.
       ;; Clean up all of the client threads when done.
       ;; This code is here for the benefit of the REPL because it is
       ;; intended that this tutorial be worked interactively. In a real
       ;; threaded server, the server would just exit--destroying the
       ;; server process, and causing all threads to exit which then notifies
       ;; the clients.
       (format t "Destroying any active client threads....~%")
       (mapc #'(lambda (thr)
                 (when (and (thread-alive-p thr)
                   (string-equal "process-ex4-client-thread"
                                 (thread-name thr)))
                   (format t "Destroying: ~A~%" thr)
                   ;; Ignore any conditions which might arise if a
                   ;; thread happened to finish in the race between
                   ;; liveness testing and destroying.
                   (ignore-errors (destroy-thread thr))))
             (all-threads)))))
```

4. The beginning of the thread handling the client:

   When the thread is born, the aforementioned explicit binding of the client socket to *ex4-tls-client* takes effect via the *default-special-bindings* mechanism. By declaring *ex4-tls-client* ignorable, we inform the compiler that this variable is set "elsewhere" and no warning should be emitted about its possibly undefined value. In our case, this will always be defined at runtime in this server.

```
;;; The thread which handles the client connection.
(defun process-ex4-client-thread ()
  ;; This variable is set outside of the context of this thread.
  (declare (ignorable *ex4-tls-client*))
```

5. Send the time to the socket:

   The UNWIND-PROTECT in this form handles every possible case of leaving the evaluable function such as it completing normally, a condition being signaled, or by thread destruction--on SBCL! In all cases, the socket to the client is closed which cleans up OS resources and lets the client know right away the server has closed the connection. The HANDLER-CASE form here just informs us which of the common IOLib conditions may have been signaled while writing the time to the client.

```
;; We ensure the client socket is always closed!
(unwind-protect
    (multiple-value-bind (who port)
        (remote-name *ex4-tls-client*)
      (format t "A thread is handling the connection from ~A:~A!~%"
              who port)

      ;; Prepare the time and send it to the client.
      (multiple-value-bind (s m h d mon y)
          (get-decoded-time)
        (handler-case
            (progn
              (format t "Sending the time to ~A:~A..." who port)
              (format *ex4-tls-client*
                      "~A/~A/~A ~A:~A:~A~%"
                      mon d y h m s)
              (finish-output *ex4-tls-client*)
              (format t "Sent!~%"))

          (socket-connection-reset-error ()
            (format t "Client ~A:~A reset the connection!~%" who port))

          (hangup ()
            (format t "Client ~A:~A closed connection.~%" who port)))))

  ;; Cleanup form for uw-p.
  (format t "Closing connection to ~A:~A!~%"
          (remote-host *ex4-tls-client*) (remote-port *ex4-tls-client*))
  (close *ex4-tls-client*)))
```

It is a bit tricky to robustly handle closing of the client socket in the thread. For example, if we bound the special variable *ex4-tls-client* to a lexically scoped variable and then did the UNWIND-PROTECT form to close the lexically scoped variable, then if this thread wakes up and gets destroyed after the lexical binding, but before the UNWIND-PROTECT, we'd lose a socket to a client into the garbage collector.

Such incorrect code would look like:

```
;; This code is incorrect!
(defun process-ex4-client-thread ()
  (declare (ignorable *ex4-tls-client*))
  (let ((client *ex4-tls-thread*))
    ;; thread gets destroyed right here! client socket is left open!
    (unwind-protect
        ( [evaluable form] )
      (close client))))
```

6. The entry point into this example:

Like earlier servers, we call the helper function and catch what happens if :reuse-addr wasn't true in the BIND-ADDRESS function call.

```
;; The entry point into this example.
(defun run-ex4-server (&key (port *port*))
  (handler-case

      (run-ex4-server-helper port)

    ;; handle some common signals
    (socket-address-in-use-error ()
      (format t "Bind: Address already in use, forget :reuse-addr t?"))))
  (finish-output))
```

2.3.5 Daytime Client/Server Commentary

This concludes the examples using the daytime protocol. We've seen patterns emerge in how the simplest of clients and servers are built and began to reason about how to handle common signaled conditions. Threading, of course, increases the care one must have in order to ensure that data access and control flow is kept consistent.

2.4 Echo Line Clients and Servers

These next examples focus on the echo protocol. This is simply a server that sends back to the client whatever the client wrote to it. A client can request to quit talking to a server (except ex8-server, where this feature isn't implemented) by sending the word "quit", on a line by itself. This tells the server to close the connection to the client once it has finished echoing the line. The closing of the client's read socket lets the client know the connection to the server went away and that it is time to exit. We also introduce the socket multiplexer interface which allows concurrent processing of socket connections. This is similar to how UNIX's select(), epoll(), or kqueue() works. Due to portability concerns on doing nonblocking operations on *standard-input* and *standard-output* (we can't easily do it) we are beholden to some form of blocking I/O in our clients because they interact with a human. We will explore true non-blocking I/O in the ex8-server example since that server only has to converse with connected clients.

2.5 Echo Clients

The echo clients are a group of programs which read a line from *standard-input*, write it to the server, read back the response from the server, and emit the result to *standard-output*. While there is a portable method to read "however much is available" from *standard-input*, there isn't the symmetrical method to write "whatever I'm able" to *standard-output*. For our client design, this means that all of these clients are line oriented and do blocking I/O when reading from *standard-input* and writing to *standard-output*.

2.5.1 Echo Client IPV4/TCP: ex4-client.lisp

This is a very basic echo client program that handles the usual conditions while talking to the server:

1. Connect to the server and start echoing lines:

   Here we use WITH-OPEN-SOCKET to create an active socket that we then use to connect to the server. We handle HANGUP, for when the server went away before the client could write to it, and END-OF-FILE, for when the server closes down the connection.

   Notice we call the function ex4-str-cli inside of a HANDLER-CASE macro. This allows us to not check for any signaled conditions in ex4-str-cli and greatly simplifies its implementation.

   In this specific example, we don't do anything other than notify that the condition happened since after that the socket gets closed via WITH-OPEN-SOCKET.

```
(defun run-ex4-client-helper (host port)

  ;; Create a internet TCP socket under IPV4
  (with-open-socket
    (socket :connect :active
            :address-family :internet
            :type :stream
            :external-format '(:utf-8 :eol-style :crlf)
            :ipv6 nil)

    ;; do a blocking connect to the daytime server on the port.
    (connect socket (lookup-hostname host) :port port :wait t)

    (format t "Connected to server ~A:~A from my local connection at ~A:~A!~%"
            (remote-host socket) (remote-port socket)
            (local-host socket) (local-port socket))

    (handler-case
        (ex4-str-cli socket)

      (socket-connection-reset-error ()
        (format t "Got connection reset. Server went away!"))

      (hangup ()
        (format t "Got hangup. Server closed connection on write!~%"))

      (end-of-file ()
        (format t "Got end-of-file. Server closed connection on read!~%")))))
```

2. Echo lines to the server:

   Until the user inputs "quit" on a line by itself, we read a line, send it to the server, read it back, and emit it to stdout. If any of the usual conditions are signaled here, the handler-case in the Step 0 code fires and we deal with it there.

When "quit" is entered, the line is sent on the round trip to the server like usual, but this time the server closes the connection to the client. Unfortunately, since the client is doing blocking I/O, we must read another line from *standard-input* before we get any signaled condition when IOLib discovers the socket has been closed by the server.

In practice, this means after the server closed the connection, the user must hit in order to drive the I/O loop enough to get the signaled condition.

```
;; read a line from stdin, write it to the server, read the response, write
;; it to stdout. If we read 'quit' then echo it to the server which will
;; echo it back to us and then close its connection to us.
(defun ex4-str-cli (socket)
  (loop
      (let ((line (read-line)))
        ;; send it to the server, get the response.
        (format socket "~A~%" line)
        (finish-output socket)
        (format t "~A~%" (read-line socket)))))
```

3. Entry point into the example:

   We handle the usual connection refused condition, but otherwise this step is unremarkable.

```
;; This is the entry point into this example
(defun run-ex4-client (&key (host *host*) (port *port*))
  (unwind-protect
      (handler-case

          (run-ex4-client-helper host port)

        ;; handle a commonly signaled error...
        (socket-connection-refused-error ()
          (format t "Connection refused to ~A:~A. Maybe the server isn't running?~%"
                  (lookup-hostname host) port)))

    ;; Cleanup form
    (format t "Client Exited.~%")))
```

2.5.2 *Echo Client IPV4/TCP: ex5a-client.lisp*

This is the first client to use the socket multiplexer to notice when the socket to the server is ready for reading or writing. While the multiplexer is often used in single threaded servers it can be used for clients--especially clients which may talk to multiple servers like web clients. Use of the multiplexer API will require a significant change in how the code is structured. It is not recommended that the multiplexer and threads be used simultaneously to handle network connections.

Keeping in mind the fact that we ALWAYS could block while reading from *standard-input* or writing to *standard-output*, we only attempt to read/write to the standard streams when the multiplexer thinks it can read/write to the server without blocking. This is a change from the traditional examples of how to do this in C because in C one can determine if STDIN or STDOUT are ready in the same manner as a network file descriptor.

The first big change from our previous examples is that we stop using WITH-OPEN-SOCKET since now we must manually control when the socket to the server must be closed. This is especially important for clients who use active sockets. The second change is how we do the creation and registering of the handlers for reading and writing to the server socket. The third change is how to unregister a handler and close the socket associated with it under the right conditions. Other changes will be explained as we meet them.

The main functions of the multiplexer API are:

```
(make-instance 'iomux:event-base ....)
```

Create an instance of the event-base, and associate some properties with it, such as event-dispatch should return if the multiplexer does not have any sockets it is managing. Passed an: :exit-when-empty - when no handlers are registered, event-dispatch will return.

```
(event-dispatch ...)
```

By default, sit in the multiplexer loop forever and handle I/O requests. It is passed the event-base binding and in addition: :once-only - run the ready handlers once then return. :timeout - when there is no I/O for a certain amount of time return.

```
(set-io-handler ...)
```

Associates a handler with a state to be called with a specific socket. Passed an:

- event-base binding
- :read or :write or :error keyword
- the handler closure

```
(remove-fd-handlers ...)
```

Removes a handler for a specific state with a specific socket. Passed an:

- event-base binding
- an fd
- one or more of :read t, :write t, :error t

Here is the example using this API.

1. The event base:

   The event-base is the object which holds the state of the multiplexer. It must be initialized and torn down as we'll see in the entry function to this example.

   ```
   ;; This will be an instance of the multiplexer.
   (defvar *ex5a-event-base*)
   ```

2. A helper function in which we create the active socket:

   Instead of using WITH-OPEN-SOCKET, we manually create the socket. We do this to better control how to close the socket. WITH-OPEN-SOCKET will try to FINISH-OUTPUT on the socket before closing it. This is bad if the socket had been previously closed or signaled a condition like HANGUP. Trying to write more data to an already hung up socket will simply signal another condition. To prevent layers of condition handling code, we explicitly handle closing of the socket ourselves.

   ```
   (defun run-ex5a-client-helper (host port)
     ;; Create a internet TCP socket under IPV4
     ;; We specifically do not use with-open-socket here since that form is
     ;; more suited for synchronous i/o on one socket. Since we do not use that
     ;; form, it is up to the handlers to decide to remove and close the socket
   ```

```
  ;; when the connection to the server should be closed.
  (let ((socket (make-socket :connect :active
                             :address-family :internet
                             :type :stream
                             :external-format '(:utf-8 :eol-style :crlf)
                             :ipv6 nil)))
```

3. Connect to the server, register the socket handlers:

   We protect the closing of the socket via UNWIND-PROTECT. We will talk about the ramifications of this decision in the next step which describes the UNWIND-PROTECT's cleanup form. In this section of code, we set up a read and write handler for the socket, and invoke the dispatch function, which will continue calling the handlers associated with the socket until the socket gets closed and the handlers unregistered. When this happens (see the entrance function step for why), EVENT-DISPATCH returns and we continue on to the cleanup form for the UNWIND-PROTECT.

   Setting up a handler in the multiplexer requires several arguments to the function set-io-handler. Here are what the arguments to that function are:

   a.
   ```
   *ex5a-event-base*
   ```
   This is the instance of the multiplexer for which we are setting up the handler.

   b.
   ```
   (socket-os-fd socket)
   ```
   This call returns the underlying operating system's file descriptor associated with the socket.

   c.
   ```
   :read
   ```
   This keyword states that we'd like to call the handler when the socket is ready to read. There is also :write and :error.

   d.
   ```
   (make-ex5a-str-cli-read socket (make-ex5a-client-disconnector socket))
   ```
   The make-ex5a-str-cli-read function returns a closure over the socket and another closure returned by the make-ex5a-client-disconnector function. This function is what will be called when the socket is ready for reading. We will shortly explain the signature of this function and what gets passed to it by the multiplexer. The disconnector function will be called by the returned reader function if the reader function thinks that it needs to close the socket to the server.

```
(unwind-protect
    (progn
      ;; do a blocking connect to the echo server on the port.
      (connect socket (lookup-hostname host) :port port :wait t)

      (format t "Connected to server ~A:~A from my local connection at ~A:~A!~%"
              (remote-host socket) (remote-port socket)
              (local-host socket) (local-port socket))

      ;; set up the handlers for read and write
      (set-io-handler *ex5a-event-base*
                      (socket-os-fd socket)
```

```
                       :read (make-ex5a-str-cli-read socket
                                                     (make-ex5a-client-disconnector socket)))

       (set-io-handler *ex5a-event-base*
                       (socket-os-fd socket)
                       :write (make-ex5a-str-cli-write socket
                                                       (make-ex5a-client-disconnector socket)))

       (handler-case
           ;; keep processing input and output on the fd by
           ;; calling the relevant handlers as the socket becomes
           ;; ready. The relevant handlers will take care of
           ;; closing the socket at appropriate times.
           (event-dispatch *ex5a-event-base*)

         ;; We'll notify the user of the client if a handler missed
         ;; catching common conditions.
         (hangup ()
           (format t "Uncaught hangup. Server closed connection on write!%"))
         (end-of-file ()
           (format t "Uncaught end-of-file. Server closed connection on read!%"))))
```

4. Cleanup form for UNWIND-PROTECT:

   In the cleanup form, we always close the socket and we pass the function close :abort t to try and close the socket in any way possible. If we just tried closing the socket, then we might cause another condition to be signaled if a previous condition, like HANGUP, had already affected the socket. :abort t avoids that case. If the socket is already closed by a handler by the time we get here, closing it again hurts nothing.

```
;; Cleanup expression for uw-p.
;; Try to clean up if the client aborted badly and left the socket open.
;; It is safe to call close mutiple times on a socket.
;; However, we don't want to finish-output on the socket since that
;; might signal another condition since the io handler already closed
;; the socket.
(format t "Client safely closing open socket to server.~%")
(close socket :abort t))))
```

5. Make the writer function for when the socket is ready to write:

   This function returns a closure which is called by the multiplexer when it is ready to read something from the server. The arguments to the closure are fd, the underlying file descriptor for the ready socket, event, which could be :read, :write, or :error if the handler was registered multiple times, and exception, which is nil under normal conditions, :error under an error with the socket, or :timeout, if we were using timeout operations when dealing with the socket.

   The closure will read a line with the function READ-LINE and write it to the server. The read will be blocking, but hopefully the write won't be since the multiplexer told us we could perform the write and not block. Obviously, is we write an enormous line, then we might block again, and in this case the FINISH-OUTPUT on the socket will push the data in a blocking I/O fashion until it is done and we return from the handler. So while this closure for the most part writes when ready, there are cases under which it'll still block.

   In this handler, if there is a signaled condition either reading from *standard-input* (the END-OF-FILE condition) or writing to the server socket (the HANGUP condition), we invoke the disconnector closure and pass it :close. When we get to the description of the disconnector function, you'll see what that means.

Once the disconnector closure is invoked, the handler will have been removed and the socket closed. This will make EVENT-DISPATCH return since the only socket it was multiplexing for was closed--because we've told the multiplexer to do so when it was made!

```
(defun make-ex5a-str-cli-write (socket disconnector)
  ;; When this next function gets called it is because the event dispatcher
  ;; knows the socket to the server is writable.
  (lambda (fd event exception)
    ;; Get a line from stdin, and send it to the server
    (handler-case
        (let ((line (read-line)))
          (format socket "~A~%" line)
          (finish-output socket))

      (end-of-file ()
        (format t "make-ex5a-str-cli-write: User performed end-of-file!~%")
        (funcall disconnector :close))

      (hangup ()
        (format t
                "make-ex5a-str-cli-write: server closed connection on write!~%")
        (funcall disconnector :close)))))
```

6. Make the reader function for when the socket is ready to read:

   This piece of code is very similar to the previous step's code, we just handle the appropriate conditions and after reading the line from the server emit it to *standard-output*. Again, even though we are told we can read from the server without blocking, if the read is large enough we will continue to block until read-line reads the all the data and the newline.

```
(defun make-ex5a-str-cli-read (socket disconnector)
  ;; When this next function gets called it is because the event dispatcher
  ;; knows the socket from the server is readable.
  (lambda (fd event exception)
    ;; get a line from the server, and send it to *standard-output*
    (handler-case
        ;; If we send "quit" to the server, it will close its connection to
        ;; us and we'll notice that with an end-of-file.
        (let ((line (read-line socket)))
          (format t "~A~%" line)
          (finish-output))

      (end-of-file ()
        (format t "make-ex5a-str-cli-read: server closed connection on read!~%")
        (funcall disconnector :close)))))
```

7. The disconnector function:

   This function returns a closure which takes an arbitrary number of arguments. If the arguments to the invoked closure contain :read, :write, or :error, the respective handler on the associated socket is removed. If none of those three are supplied, then all handlers for that socket are removed. Additionally if :close is specified, the socket is closed. While not all features of this function is used in this example, this function (or a similar one using the correct event-base special variable) is used whenever we use the multiplexer in an example.

   The closure is called whenever a handler believes it should unregister itself or another handler, or close the socket. Because we will often close the socket in the disconnector closure, we can't use WITH-OPEN-SOCKET to automatically close the socket because WITH-OPEN-SOCKET may try to flush data on the socket, signaling another condition.

```
(defun make-ex5a-client-disconnector (socket)
  ;; When this function is called, it can be told which callback to remove, if
  ;; no callbacks are specified, all of them are removed! The socket can be
  ;; additionally told to be closed.
  (lambda (&rest events)
    (format t "Disconnecting socket: ~A~%" socket)
    (let ((fd (socket-os-fd socket)))
      (if (not (intersection '(:read :write :error) events))
          (remove-fd-handlers *ex5a-event-base* fd :read t :write t :error t)
          (progn
            (when (member :read events)
              (remove-fd-handlers *ex5a-event-base* fd :read t))
            (when (member :write events)
              (remove-fd-handlers *ex5a-event-base* fd :write t))
            (when (member :error events)
              (remove-fd-handlers *ex5a-event-base* fd :error t)))))
    ;; and finally if were asked to close the socket, we do so here
    (when (member :close events)
      (close socket :abort t))))
```

8. The entry point for this example and setting up the event-base:

   This function is much more complex than in examples that do not use the multiplexer. Protected by an UNWIND-PROTECT, we first initialize the event base my calling make-instance 'iomux:event-base. Here is where we pass the keyword argument :exit-when-empty t which states that the event-dispatch function should return when there are no more registered handlers. Once that is done, we call the helper, catching a common condition and waiting until we return.

```
;; This is the entry point for this example.
(defun run-ex5a-client (&key (host *host*) (port *port*))
  (let ((*ex5a-event-base* nil))
    (unwind-protect
        (progn
          ;; When the connection gets closed, either intentionally in the client
          ;; or because the server went away, we want to leave the multiplexer
          ;; event loop. So, when making the event-base, we explicitly state
          ;; that we'd like that behavior.
          (setf *ex5a-event-base*
                (make-instance 'iomux:event-base :exit-when-empty t))
          (handler-case
              (run-ex5a-client-helper host port)

            ;; handle a commonly signaled error...
            (socket-connection-refused-error ()
              (format t "Connection refused to ~A:~A. Maybe the server isn't running?~%"
                      (lookup-hostname host) port))))
```

9. The cleanup form for UNWIND-PROTECT:

   This cleanup form closes the *ex5a-event-base* instance. IOLib defines a method for the generic function CLOSE which accepts an event-base and performs the necessary work to shut it down.

```
;; Cleanup form for uw-p
;; ensure we clean up the event base regardless of how we left the client
;; algorithm
(when *ex5a-event-base*
  (close *ex5a-event-base*))
(format t "Client Exited.~%")
(finish-output))))
```

While this program works just fine with human input, it has a failure when reading batch input. The failure is that when we get the END-OF-FILE condition when *standard-input* closes, we _immediately_ unregister the read/write handlers to the server, close the socket and exit the program. This destroys any in-flight data to/from the server and lines being echoed may be lost.

2.5.3 Echo Client IPV4/TCP: ex5b-client.lisp

In order to fix the batch input problem of ex5a-client, we will use the shutdown function which allows us to inform the server we are done writing data, but leave the socket open so we can read the rest of the responses from the server. This effectively closes only one-half of the TCP connection. The server has to be made aware of this kind of protocol so it doesn't assume the client completely exited when it gets an END-OF-FILE from the client and shuts down the whole connection throwing away any queued data for the client.

This client is nearly identical to ex5a-client except we shut down the write end of the connection to the server when we get END-OF-FILE from *standard-input* and wait until we get all of the data back from the server. The server signifies to us that it has sent all of the pending data by closing the write end of its connection. The client sees the closing of the server's write end as an END-OF-FILE on the socket connected to the server.

We show this example as a difference to ex5aq-client.

1. Shutdown the write end of the socket to the server:

   Here we use the expanded functionality of the disconnector closure. After we shut down the write end of our TCP connection, we call (funcall disconnector :write) which states only to remove the write (to the server) handler, but leave the connection open. After this happens, there is no way we can read from *standard-input* again. Once the server sends the final data and the closes its connection to this client, we remove the read handler, which removes the last handler, and causes the EVENT-DISPATCH function to return, which ends the client computation.

```
(defun make-ex5b-str-cli-write (socket disconnector)
  ;; When this next function gets called it is because the event dispatcher
  ;; knows the socket to the server is writable.
  (lambda (fd event exception)
    ;; Get a line from stdin, and send it to the server
    (handler-case
        (let ((line (read-line)))
          (format socket "~A~%" line)
          (finish-output socket))

      (end-of-file ()
        (format t
                "make-ex5b-str-cli-write: User performed end-of-file!~%")
        ;; Shutdown the write end of my pipe to give the inflight data the
        ;; ability to reach the server!
        (format t
                "make-ex5b-str-cli-write: Shutting down write end of socket!~%")
        (shutdown socket :write t)
        ;; since we've shut down the write end of the pipe, remove this handler
        ;; so we can't read more data from *standard-input* and try to write it
        ;; it to the server.
        (funcall disconnector :write))

      (hangup ()
        (format t
                "make-ex5b-str-cli-write: server closed connection on write!~%")
        (funcall disconnector :close)))))
```

Be aware that even if both directions on one end of a connection are shutdown, close still must be called upon the socket in order to release resources held by the operating system.

2.6 Echo Servers

The echo servers, paired to clients as per the beginning of this tutorial, further evolve to using the multiplexer and becoming more fine grained with respect to when I/O is done until we reach the ability to perform nonblocking I/O of arbitrary read/write sizes.

2.6.1 Echo Server IPV4/TCP: ex5-server.lisp

This threaded server is very similar to ex4-server, but instead of sending only the time, each thread handles an echo protocol to a client. While this is still a blocking I/O server, only a single thread talking to a client gets blocked, not the whole server. Other than the server not honoring batch input from the client correctly, this is a common model for a class of servers due to its nonblocking behavior.

1. The special variable used to communicate the client socket to the thread:

```
;; The special variable used to hold the client socket for the thread
;; managing it.
(defvar *ex5-tls-client* nil)
```

2. The usual prologue to a server:

```
(defun run-ex5-server-helper (port)
  (with-open-socket
    (server :connect :passive
            :address-family :internet
            :type :stream
            :ipv6 nil
            :external-format '(:utf-8 :eol-style :crlf))

    (format t "Created socket: ~A[fd=~A]~%" server (socket-os-fd server))

    ;; Bind the socket to all interfaces with specified port.
    (bind-address server +ipv4-unspecified+ :port port :reuse-addr t)
    (format t "Bound socket: ~A~%" server)

    ;; start listening on the server socket
    (listen-on server :backlog 5)
    (format t "Listening on socket bound to: ~A:~A~%"
            (local-host server)
            (local-port server))
```

3. First half of creating the client threads:

```
;; keep accepting connections forever, but if this exits for whatever
;; reason ensure to destroy any remaining running threads.
(unwind-protect
    (loop
        (format t "Waiting to accept a connection...~%")
        (finish-output)
        (let* ((client (accept-connection server :wait t))
          ;; set up the special variable to store the client
          ;; we accepted...
               (*default-special-bindings*
                 (acons '*ex5-tls-client* client
                         *default-special-bindings*)))

          ;; ...and handle the connection!
          (when client
            (make-thread #'process-ex5-client-thread
                         :name 'process-ex5-client-thread))))
```

4. Second half, the cleanup form for the UNWIND-PROTECT:

   We make sure to clean up only the client threads!

```
;; Clean up form for uw-p.
;; Clean up all of the client threads when done.
;; This code is here for the benefit of the REPL because it is
;; intended that this tutorial be worked interactively. In a real
;; threaded server, the server would just exit--destroying the
;; server process, and causing all threads to exit which then notifies
;; the clients.
(format t "Destroying any active client threads....~%")
(mapc #'(lambda (thr)
          (when (and (thread-alive-p thr)
                     (string-equal "process-ex5-client-thread"
                                   (thread-name thr)))
            (format t "Destroying: ~A~%" thr)
            ;; Ignore any conditions which might arise if a
            ;; thread happened to finish in the race between
            ;; liveness testing and destroying.
            (ignore-errors (destroy-thread thr))))
      (all-threads)))))
```

5. Handle the client and deal with signaled conditions:

   In this function, we ensure that under all conditions of the execution of this function, if something goes wrong, we eagerly close the socket to the client so it is not leaked into the garbage collector. We also handle numerous conditions the the client could generate while talking to it in the function str-ex5-echo.

```
;; The thread which handles the client connection.
(defun process-ex5-client-thread ()
  ;; declared ignorable because this dynamic variable is bound outside
  ;; of the context of this function.
  (declare (ignorable *ex5-tls-client*))
  ;; no matter how we get out of the client processing loop, we always
  ;; close the connection.
  (unwind-protect
      (multiple-value-bind (who port)
          (remote-name *ex5-tls-client*)
        (format t "A thread is handling the connection from ~A:~A!~%"
                who port)

        (handler-case
            ;;  perform the actual echoing algorithm
            (str-ex5-echo *ex5-tls-client* who port)
          (socket-connection-reset-error ()
            (format t "Client ~A:~A: connection reset by peer.~%"
                    who port))

          (end-of-file ()
            (format t "Client ~A:~A closed connection for a read.~%"
                    who port)
            t)

          (hangup ()
            (format t "Client ~A:~A closed connection for a write.~%"
                    who port)
            t)))

    ;; cleanup form of the unwind-protect
    ;; We always close the connection to the client, even if this
    ;; thread gets destroyed (at least in SBCL this cleanup form gets
    ;; run when this thread is destroyed).
    (format t "Closing connection to ~A:~A!~%"
            (remote-host *ex5-tls-client*) (remote-port *ex5-tls-client*))
    (close *ex5-tls-client*)
    t))
```

6. Actually perform the echo protocol to the client:

   Read lines from the client and echo them back. All of this I/O is blocking. If we see "quit" from the client, then exit the loop, which causes the UNWIND-PROTECT cleanup form in step 4 to fire and close the connection to the client.

```
;; The actual function which speaks to the client.
(defun str-ex5-echo (client who port)
  ;; here we let signaled conditions on the boundary conditions of the
  ;; client (meaning it closes its connection to us on either a read or
  ;; a write) bail us out of the infinite loop
  (let ((done nil))
    (loop until done
        do
          (let ((line (read-line client)))
            (format t "Read line from ~A:~A: ~A~%" who port line)
            (format client "~A~%" line)
            (finish-output client)
            (format t "Wrote line to ~A:~A: ~A~%" who port line)

            ;; Exit the thread when the user requests it with 'quit'.
            ;; This forces a close to the client socket.
            (when (string= line "quit")
              (setf done t))
            t))))
```

7. The entrance function into this example:

```
;; This just checks for some error conditions so we can print out a nice
;; message about it.
(defun run-ex5-server (&key (port *port*))
  (handler-case

      (run-ex5-server-helper port)

    ;; handle some common conditions
    (socket-address-in-use-error ()
      (format t "Bind: Address already in use, forget :reuse-addr t?")))

  (finish-output))
```

2.6.2 *Echo Server IPV4/TCP: ex6-server.lisp*

This is the first of the echo servers which use the multiplexer to handle multiple clients concurrently. It is a single threaded program. As mentioned before, one shouldn't mix the multiplexer and threads together to handle network connections.

We explore a new concept with the multiplexer in that the listening server socket is itself registered with the multiplexer. The read handler (called the listener handler in this context) associated with this socket becomes ready when a client has connected to the server address. Thus, once the listening socket is ready the listener handler accepts the client and associates the line echo protocol callback with the client's socket in the multiplexer.

The I/O design of this server is such that if the client connection is ready to read, we read a line, then immediately write the line back to the client in the same function without waiting to see if it is ready for writing. Since we are still using blocking I/O, this is ok. The reason for this example's design was to minimize the complexity of using the multiplexer in order to introduce the listener handler. Later examples become much more complex as we push the multiplexer API farther.

1. The variable which holds the multiplexer instance:

   ```
   ;; This variable represents the multiplexer state.
   (defvar *ex6-server-event-base*)
   ```

2. A hash table of client connections:

   We record each client that connects to the server into a hash table socket keyed by the list (ip address port) and associate with it a value of the client's socket. This is so that under any conditions of the server exiting we can eagerly close any open connections to clients in a cleanup form.

   ```
   ;; This holds any open connections to clients as keys in the table. The values
   ;; is a list containing the host and port of the connection. We use this to
   ;; close all connections to the clients, if any, when the server exits.  This
   ;; allows all clients to notice the server had gone away.
   (defvar *ex6-server-open-connections*)
   ```

3. Create and bind the server socket:

   We protect how we manipulate the server socket with an UNWIND-PROTECT so we ensure to close the socket at the end of the server's computation or if something went wrong.

   ```
   ;; Set up the server and server clients with the multiplexer
   (defun run-ex6-server-helper (port)

     ;; We don't use with-open-socket here since we may need to have a
     ;; finer control over when we close the server socket.
   ```

```
  (let ((server (make-socket :connect :passive
                             :address-family :internet
                             :type :stream
                             :ipv6 nil
                             :external-format '(:utf-8 :eol-style :crlf))))
    (unwind-protect
        (progn
          (format t "Created socket: ~A[fd=~A]~%" server (socket-os-fd server))
          ;; Bind the socket to all interfaces with specified port.
          (bind-address server +ipv4-unspecified+ :port port :reuse-addr t)
          (format t "Bound socket: ~A~%" server)

          ;; start listening on the server socket
          (listen-on server :backlog 5)
          (format t "Listening on socket bound to: ~A:~A~%"
                  (local-host server)
                  (local-port server))
```

4. Register a listener handler on the server socket and start dispatching events with the multiplexer:

```
;; Set up the initial listener handler for any incoming clients
(set-io-handler *ex6-server-event-base*
                (socket-os-fd server)
                :read
                  (make-ex6-server-listener-handler server))

;; keep accepting connections forever.
(handler-case
    (event-dispatch *ex6-server-event-base*)

  ;; Just in case any handler misses these conditions, we
  ;; catch them here.
  (socket-connection-reset-error ()
    (format t "~A~A~%"
            "Caught unexpected reset by peer! "
            "Client connection reset by peer!"))
  (hangup ()
    (format t "~A~A~%"
            "Caught unexpected hangup! "
            "Client closed connection on write!"))
  (end-of-file ()
    (format t "~A~A~%"
            "Caught unexpected end-of-file! "
            "Client closed connection on read!"))))
```

5. When the server stops handling clients, we close the server socket:

```
;; Cleanup expression for uw-p.
;; Ensure the server socket is closed, regardless of how we left
;; the server.
(close server))))
```

6. The listener handler:

Once the returned closure from this function is called by the multiplexer on the ready server socket, we accept the client with a blocking accept. We then save the client connection in our table and register the line echo closure with the socket. The line echo closure will also contain a disconnector function as in previous usages of the multiplexer.

```
;; When the multiplexer states the server socket is ready for reading
;; it means that we have a client ready to accept. So we accept it and
;; then register the accepted client socket back into the multiplexer
;; with the appropritate echo protocol function.
(defun make-ex6-server-listener-handler (socket)
  (lambda (fd event exception)

    ;; do a blocking accept, returning nil if no socket
    (let* ((client (accept-connection socket :wait t)))
      (when client
        (multiple-value-bind (who port)
            (remote-name client)
          (format t "Accepted a client from ~A:~A~%" who port)

          ;; save the client connection in case we need to close it later
          ;; when the server exits.
          (setf (gethash `(,who ,port) *ex6-server-open-connections*) client)

          ;; set up an line echo function for the client socket.
          (set-io-handler *ex6-server-event-base*
                          (socket-os-fd client)
                          :read (make-ex6-server-line-echoer client
                                                             who
                                                             port
                                                             (make-ex6-server-disconnector client))))))))
```

7. The line echo closure generator:

   This function returns a closure which is then bound to a client socket in the multiplexer. When the socket is ready, we read a line form the client and write it back to the client immediately. Since this is blocking I/O the whole server will wait until this transaction is complete. This means that a client which sends one byte of ASCII that is not a newline can cause the whole server to block for all clients. This serious defect is remedied with non-blocking I/O, which we show in a later example.

```
;; This function returns a function that reads a line, then
;; echoes it right back onto the socket it came from. This is blocking
;; i/o.  This code can suffer denial of service attacks like on page
;; 167 of "Unix Network Programming 2nd Edition: Sockets and XTI", by
;; Richard Stevens.
(defun make-ex6-server-line-echoer (socket who port disconnector)
  (format t "Creating line-echoer for ~A:~A~%" who port)
  (lambda (fd event exception)
    (handler-case
        (let ((line (read-line socket))) ;; read a line from the client
          (format t "Read ~A:~A: ~A~%" who port line)
          (format socket "~A~%" line) ;; write it the client
          (finish-output socket)
          (format t "Wrote ~A:~A: ~A~%" who port line)

          ;; close the connection to the client if it asked to quit
          (when (string= line "quit")
            (format t "Client requested quit!~%")
            (funcall disconnector who port)))

      (socket-connection-reset-error ()
        ;; Handle the usual and common conditions we'll see while
        ;; talking to a client
```

```
         (format t "Client's connection was reset by peer.~%")
         (funcall disconnector who port))

       (hangup ()
         (format t "Client went away on a write.~%")
         (funcall disconnector who port))

       (end-of-file ()
         (format t "Client went away on a read.~%")
         (funcall disconnector who port)))))
```

8. The disconnector closure generator:

   This function returns a closure that removes all the handlers from the socket in question and then closes it. Notice that this means this server is not capable of handling batch input from a client, since when it receives the END-OF-FILE on the read from a client, will immediately tear down the connection destroying any in flight data. After closing the socket, we also remove it from our table of open connections.

```
;; If we decide we need to disconnect ourselves from the client, this will
;; remove all the handlers and remove the record of our connection from
;; *ex6-server-open-connections*.
(defun make-ex6-server-disconnector (socket)
  (lambda (who port)
    (format t "Closing connection to ~A:~A~%" who port)
    (remove-fd-handlers *ex6-server-event-base* (socket-os-fd socket))
    (close socket)
    (remhash `(,who ,port) *ex6-server-open-connections*)))
```

9. Initialize the event-base, the connection table, and start the server:

   This code is the beginning of the UNWIND-PROTECT form which protects the server's socket resources.

```
;; This is the entrance function into this example.
(defun run-ex6-server (&key (port *port*))
  (let ((*ex6-server-open-connections* nil)
        (*ex6-server-event-base* nil))
    (unwind-protect
        (handler-case
            (progn
              ;; Clear the open connection table and init the event base
              (setf *ex6-server-open-connections*
                    (make-hash-table :test #'equalp)

                    *ex6-server-event-base*
                    (make-instance 'event-base))

              (run-ex6-server-helper port))

          ;; handle a common signal
          (socket-address-in-use-error ()
            (format t "Bind: Address already in use, forget :reuse-addr t?")))
```

10. Cleanup the client connections and close the event-base:

When the server exits we walk the *ex6-server-open-connections* hash and eagerly close every client we find there. After we are done, we close the event-base. This ensures every thing is cleaned up properly.

```
;; Cleanup form for uw-p
;; Close all open connections to the clients, if any. We do this
;; because when the server goes away we want the clients to know
;; immediately. Sockets are not memory, and can't just be garbage
;; collected whenever. They have to be eagerly closed.
(maphash #'(lambda (k v)
             (format t "Closing a client connection to ~A~%" k)
             ;; We don't want to signal any conditions on the close...
             (close v :abort t))
         *ex6-server-open-connections*)

;; and clean up the multiplexer too!
(when *ex6-server-event-base*
  (close *ex6-server-event-base*))
(format t "Server Exited~%")
(finish-output))))
```

This server uses the multiplexer in a simple fashion because only one handler is registered for a client. That handler reads, then writes the data back to the client. The scope of the data read from the client never has to leave the handler function.

2.6.3 Echo Server IPV4/TCP: ex7-server.lisp

This example is different than ex6-server because it fully separates the reading and writing of data to a client into different handler functions. This requires an architectural change to the server in order to be able to keep the data from the client "somewhere" before being able to write it back to the client when the multiplexer determines it can written to the client. We introduce an io-buffer object, implemented in terms of a closure and one per client, which stores the in-flight data until the client is ready to accept the writes from the server.

Storage of client data introduces a problem in that if the client writes lots of data to the server but happens to never be ready to accept it back from the server, the server will consume all memory and run out of resources. We attempt to prevent this from happening, though not perfectly.

When the io-buffer is created for a client, we state we only would like a certain number of bytes to be read from the client. Of course, since we're using read-line with blocking I/O and the client could write a tremendous amount of data before a newline, we can't completely enforce our storage policy in this server. If the client, though, is well-behaved in that it sends reasonable sized lines of text--a rarity in the real world, our implemented policy is sufficient. When we reach the nonblocking I/O server example, we'll find that we can perfectly enforce the per client data storage policy.

This server honors batch input from the client. When it sees the END-OF-FILE from the client, and it still has data to write, the server will attempt to write the rest of the data out as the multiplexer says the client is ready to receive it.

Since this example is quite long the server portion will just be shown as a difference to ex6-server.

1. The listener handler:

   The important code in this function is the call to make-ex7-io-buffer. This function returns a closure, here called io-buffer, which takes one argument, either :read-a-line or :write-a-line. When the funcall of io-buffer with the appropriate argument happens, *another* closure is returned and this is the closure registered with the appropriate ready state in the multiplexer.

This returned closure has bound in its lexical scope the storage needed for the client.

Both closures returned by :read-a-line and :write-a-line have access to the same storage space unique to this object io-buffer. This is the means by which the client's write handler can get access to the data read by the client's read handler.

```
;; Create the listener closure which accepts the client and registers the
;; buffer functions with it.
(defun make-ex7-server-listener-handler (socket)
  (lambda (fd event exception)
    ;; do a blocking accept, returning nil if no socket
    (let* ((client (accept-connection socket :wait t)))
      (when client
        (multiple-value-bind (who port)
            (remote-name client)
          (format t "Accepted a client from ~A:~A~%" who port)

          ;; save the client connection in case we need to close it later.
          (setf (gethash `(,who ,port) *ex7-open-connections*) client)

          ;; We make an io-buffer, which takes care of reading from the
          ;; socket and echoing the information it read back onto the
          ;; socket.  The buffer takes care of this with two internal
          ;; handlers, a read handler and a write handler.
          (let ((io-buffer (make-ex7-io-buffer client who port
                                              (make-ex7-server-disconnector client))))

            ;; set up an line echo function for the client socket.  The
            ;; internals of the buffer will perform the appropriate
            ;; registration/unregistration of the required handlers at
            ;; the right time depending upon data availability.

            (set-io-handler *ex7-event-base*
                            (socket-os-fd client)
                            :read (funcall io-buffer :read-a-line))

            (set-io-handler *ex7-event-base*
                            (socket-os-fd client)
                            :write (funcall io-buffer :write-a-line)))))))
```

2. The disconnector function:

This function is almost identical to a previous example used in ex5a-client. The only difference is the special variable it references.

Since the io-buffer knows under what conditions it should register or unregister specific handlers for the client socket, we need to be able to selectively remove them without disturbing the others.

```
(defun make-ex7-server-disconnector (socket)
  ;; When this function is called, it can be told which callback to remove, if
  ;; no callbacks are specified, all of them are removed! The socket can be
  ;; additionally told to be closed.
  (lambda (who port &rest events)
    (let ((fd (socket-os-fd socket)))
      (if (not (intersection '(:read :write :error) events))
          (remove-fd-handlers *ex7-event-base* fd :read t :write t :error t)
          (progn
            (when (member :read events)
              (remove-fd-handlers *ex7-event-base* fd :read t))
            (when (member :write events)
              (remove-fd-handlers *ex7-event-base* fd :write t))
            (when (member :error events)
              (remove-fd-handlers *ex7-event-base* fd :error t)))))
     ;; and finally if were asked to close the socket, we do so here
     (when (member :close events)
       (format t "Closing connection to ~A:~A~%" who port)
       (finish-output)
       (close socket)
       (remhash `(,who ,port) *ex7-open-connections*))))
```

Now we come to the description of the ex7-io-buffer code base. This code base interacts directly with the event-base multiplexer instance in order to register and unregister handlers to the client. Handlers are only registered when there is data to write, or room to read more data up to the buffer size.

1. The io-buffer closure generator and associated lexical storage:

   These are the variables closed over which represent the internal state of the closure and hold the data from the client. In particular note is the fact we keep track of when a handler is registered (since this object can register or unregister the handlers in and of itself) and whether or not we've seen the END-OF-FILE from a client. The line-queue will hold the actual data from the client.

```
(defun make-ex7-io-buffer (socket who port disconnector &key (max-bytes 4096))
  (let ((line-queue (make-queue))
        (bytes-left-to-write 0)
        (read-handler-registered nil)
        (write-handler-registered nil)
        (eof-seen nil))
```

2. The read-a-line closure:

   This is the function which will ultimately be registered with the multiplexer hence the arguments it expects. Its job is to read a line from the client when the multiplexer said the client was readable and then store the line into the line-queue. If we have read a line, we immediately register the write-a-line handler with the multiplexer since we need to know when the client will be ready to accept it. If it turns out there is more data stored than the high-water mark we set, we unregister the read handler so we don't continue to keep reading data. If we get END-OF-FILE, but there is nothing left to write, then this handler performs a small optimization and closes the socket to the client and unregisters everything. This prevents a needless loop through the multiplexer in this case.

   The handling of END-OF-FILE is interesting in that we unregister the read handler, since we won't need it anymore, and mark that we've seen the END-OF-FILE. At this point, the only thing the multiplexer has to do with respect to this client is to write all of the lines stored in the line-queue out to the client and close the connection to the client.

   Of the various conditions that can be signaled, the SOCKET-CONNECTION-RESET-ERROR condition is the one which will shut down the whole connection by removing all handlers in the multiplexer for this client and ultimately throw away any in-flight data.

```
(labels
  ;; If this function notices that there is data to write, it will
  ;; set the io-handler on the socket for the write handler.
  ;; If the function notices it has read >= than the max-bytes
  ;; it will remove itself from the handler *after* ensuring the
  ;; write handler is set up properly.
  ((read-a-line (fd event exception)
     (handler-case
         (let ((line (format nil "~A~%" (read-line socket)))) ; add a \n
           (format t "Read from ~A:~A: ~A" who port line)
           (enqueue line line-queue)
           (incf bytes-left-to-write (length line))

           (when (> bytes-left-to-write 0)
             ;; If the write handler isn't registered, then do
             ;; it now since I have data to write.
             (unless write-handler-registered
               (set-io-handler *ex7-event-base*
                               (socket-os-fd socket)
                               :write
                               #'write-a-line)
               (setf write-handler-registered t)))

           ;; Now, if there is more data than I should be
           ;; reading, remove myself from the io handler. When
           ;; the write handler notices that, after writing some
           ;; data, more of it can be read, it will reregister
           ;; the io handler for the read socket.
           (when (>= bytes-left-to-write max-bytes)
             (funcall disconnector who port :read)
             (setf read-handler-registered nil)))

       (socket-connection-reset-error ()
         ;; If the client resets its connection, we close
         ;; everything down.
         (format t "Client ~A:~A: Connection reset by peer~%" who port)
         (funcall disconnector who port :close))

       (end-of-file ()
         ;; When we get an end of file, that doesn't necessarily
         ;; mean the client went away, it could just mean that
         ;; the client performed a shutdown on the write end of
         ;; its socket and it is expecting the data stored in
         ;; the server to be written to it.  However, if there
         ;; is nothing left to write and our read end is close,
         ;; we shall consider it that the client went away and
         ;; close the connection.
         (format t "Client ~A:~A produced end-of-file on a read.~%"
                 who port)
         (if (zerop bytes-left-to-write)
             (funcall disconnector who port :close)
             (progn
               (funcall disconnector who port :read)
               (setf read-handler-registered nil)
               (setf eof-seen t))))))
```

3. The write-a-line closure:

   This function is somewhat symmetrical to read-a-line. It will register and unregister itself or the read handler based upon how much data is available to read/write. If the END-OF-FILE is seen and there is nothing left to write, it will close the connection to the client and unregister everything.

```
;; This function will notice that if it has written enough bytes to
;; bring the bytes-left-to-write under max-bytes, it will re-register
;; the reader io handler. If there is no data to write, it will,
;; after ensuring the read handler is registered, unregister itself
;; as to not be called constantly on a write ready socket with no
;; data to write.
(write-a-line (fd event exception)
  (handler-case
      (progn
        ;; If we have something to write to the client, do so.
        (when (> bytes-left-to-write 0)
          (let ((line (dequeue line-queue)))
                (format socket "~A" line) ;; newline is in the string.
                (finish-output socket)
                (format t "Wrote to ~A:~A: ~A" who port line)
                (decf bytes-left-to-write (length line))))

            ;; If we see we've fallen below the max-bytes mark,
            ;; re-register the read handler to get more data for
            ;; us. However, don't reregister the read handler if
            ;; we've seen that the client closed our read end of
            ;; our socket.
            (when (< bytes-left-to-write max-bytes)
              (unless (or eof-seen read-handler-registered)
                (set-io-handler *ex7-event-base*
                                (socket-os-fd socket)
                                :read
                                #'read-a-line)
                (setf read-handler-registered t)))

            ;; If we notice that we don't have any data to write
            ;; AND have seen the end of file from the client,
            ;; then we close the connection to the client since
            ;; it will never speak to us again and we're done
            ;; speaking to it.
            ;;
            ;; If notice we've written all of our data and there
            ;; might be more to do later, then unregister the
            ;; write handler so we don't get called
            ;; unnecesarily. This might mean that sometimes we'll
            ;; have to make an extra trip through the
            ;; event-dispatcher to perform the write if we read
            ;; more from the client and it reregisters us.
            (when (zerop bytes-left-to-write)
              (if eof-seen
                  (funcall disconnector who port :close)
                  (progn
                    (funcall disconnector who port :write)
                    (setf write-handler-registered nil)))))

        (socket-connection-reset-error ()
          ;; If I happen to get a reset, make sure the connection
          ;; is closed.  I shouldn't get this here, but if you
```

```
            ;; tinker with the flow of this example, it is a good
            ;; guard to have.
            (format t "Client ~A:~A: connection reset by peer.~%" who port)
            (funcall disconnector who port :close))

          (hangup ()
            ;; In this server, if the client doesn't accept data,
            ;; it also means it will never send us data again. So
            ;; close the connection for good.
            (format t "Client ~A:~A got hangup on write.~%" who port)
            (funcall disconnector who port :close)))))
```

4. The returned closure, which represents the io-buffer:

   This is the actual closure returned by make-ex7-io-buffer and which is used to gain access into the read-a-line and write-a-line functions. It takes a single argument, either the keywords :read-a-line or :write-a-line, and returns a reference to either internal function.

```
;; This is the actual function returned from make-ex7-io-buffer
;; which allows us access to the read/writer in the scope of the
;; closure.  We will ask for the correct functions when setting
;; up the io handlers.  NOTE: By simply asking for the handler,
;; I've assumed it is to be immediately put into an iolib event
;; handler. This is why they are considered registered at this point.
(lambda (msg)
  (cond
    ((equalp msg :read-a-line)
     (setf read-handler-registered t)
     #'read-a-line)
    ((equalp msg :write-a-line)
     (setf write-handler-registered t)
     #'write-a-line)
    (t
     (error "make-ex7-buffer: Please supply :read-a-line or :write-a-line~%")))))))
```

While this server still uses blocking I/O, we've laid the foundations for nonblocking I/O and memory storage enforcement. The foundations specifically are separating the read/write handlers into different pieces and having shared lexical bindings between them.

2.6.4 Echo Server IPV4/TCP: ex8-server.lisp

This server uses nonblocking I/O and the multiplexer to concurrently talk to the clients.

Architecturally, it is very similar to ex7-server, but the io-buffer for this server is implemented with much different internals. Whereas in ex7-server reading from a client used the stream function READ-LINE, writing used the stream function FORMAT, and the strings from the client were kept in a queue, now we use RECEIVE-FROM and SEND-TO along with an array of unsigned-bytes as a buffer to read/write actual bytes from the socket.

Accessing the socket through the stream API is different than doing it through the almost raw socket API which we are about to use. RECEIVE-FROM and SEND-TO are not part of the stream interface. They are a lower level API in IOLib being closer to the underlying OS abstraction and as a consequence have a somewhat different set of conditions that they can signal. These different conditions have the form isys: like: isys:epipe, isys:ewouldblock, etc. There is some intersection with the condition names signaled by the stream API, such as: SOCKET-CONNECTION-RESET-ERROR, and SOCKET-CONNECTION-REFUSED.

[TODO figure out complete list!]

An example of the ramifications of this API is RECEIVE-FROM. Comparing against the stream interface whose READ-LINE will signal an END-OF-FILE when the reading socket has been closed by the client, the function RECEIVE-FROM will return 0, signifying the end of file. The stream function FORMAT will signal HANGUP if it tries to write to a socket where the client has gone away. SEND-TO might not signal, or otherwise produce, any error at all when writing to a socket where the client has gone away--usually it is on the next RECEIVE-FROM that it is discovered the client went away. The bytes that SEND-TO wrote simply vanish!

With IOLib, it may surprise you to be told that all underlying fds in the previous examples have been nonblocking! This is why we specified :wait t for ACCEPT-CONNECTION and CONNECT.

The IOLib library internally ensures that the stream interface blocks according to the requirements of ANSI Common Lisp. However, when we use SEND-TO and RECEIVE-FROM we automatically gain the benefit of the non-blocking status on the underlying fd. This is why in this example we don't explicitly set the underlying fd to non-blocking status--it already is!

The server code itself is described as a difference from ex7-server, but the io-buffer for this nonblocking server (in file ex8-buffer.lisp) will be described in its entirety. Also, this server honors the batch input requirement from example client ex-5b-client, which you should use against this server.

The ex8-server codes:

1. The listener handler (first half):

   Accept and store the client connection.

```
(defun make-ex8-server-listener-handler (socket)
  (lambda (fd event exception)
    ;; do a blocking accept, returning nil if no socket
    (let* ((client (accept-connection socket :wait t)))
      (when client
        (multiple-value-bind (who port)
            (remote-name client)
          (format t "Accepted a client from ~A:~A~%" who port)

          ;; save the client connection in case we need to close it later.
          (setf (gethash `(,who ,port) *ex8-open-connections*) client)
```

2. The listener handler (second half):

   Like ex7-server, we register the read and write handlers. Notice though that we changed the keywords to the io-buffer closure to be :read-some-bytes and :write-some-bytes. This better represents what the io-buffer is actually doing.

```
;; We make an io-buffer, which takes care of reading from the
;; socket and echoing the information it read back onto the
;; socket.  The buffer takes care of this with two internal
;; handlers, a read handler and a write handler.
(let ((io-buffer
       (make-ex8-io-buffer client who port
                           (make-ex8-server-disconnector client))))

  ;; set up an unsigned byte echo function for the
  ;; client socket.  The internals of the buffer will
  ;; perform the appropriate registration/unregistration of
  ;; the required handlers at the right time depending upon
  ;; data availability.
```

```
   (set-io-handler *ex8-event-base*
                   (socket-os-fd client)
                   :read
                   (funcall io-buffer :read-some-bytes))

   (set-io-handler *ex8-event-base*

                   (socket-os-fd client)
                   :write
                   (funcall io-buffer :write-some-bytes)))))))
```

The rest of the server is extremely similar to ex7-server.

Now, we'll show the io-buffer specific to ex8-server.

1. The internal state of the io-buffer closure:

   The binding echo-buf is an unsigned-byte array of size max-bytes. This is where data from the client is stored before it is written back to the client.

   The binding read-index keeps track of the beginning of the empty space in the echo-buf buffer where more data could be stored during a read.

   The binding write-index keeps track of how much data has been written to the client. It moves towards read-index, and when it has the same value as read-index it means that there is no data left to write to the client.

   The bindings read-handler-registered and write-handler-registered allow the io-buffer to know when it has registered a handler for reading and writing data.

   The binding eof-seen marks when the client has closed its write connection to the server. The server will push out all data to the client, then close socket to the client.

```
(defun make-ex8-io-buffer (socket who port disconnector &key (max-bytes 16384))
  (let ((echo-buf (make-array max-bytes :element-type 'unsigned-byte))
        (read-index 0)
        (write-index 0)
        (read-handler-registered nil)
        (write-handler-registered nil)
        (eof-seen nil))
```

2. Reading bytes form the client:

   In this function, we will convert the return value 0 of RECEIVE-FROM on the read of a closed socket into a signaled END-OF-FILE condition to keep the structure of our code similar to what has transpired before. Once we read some bytes, we increment the read-index pointer and ensure to register a write handler to write the data back out. We optimize the writing process a little bit and try to write the data out immediately without checking to see if the socket is ready. Then if there is no more room in the echo-buf array, we unregister ourselves so we don't try and read more data from the client until we are ready to accept it (by having written all of the data back to the client). We mark the END-OF-FILE flag and unregister the read handler if we see the client has closed its connection. We optimize the knowledge that if we have no more data to write we just close the connection to the client.

```
(labels
  ;; This is the function responsible for reading bytes from the client.
  ((read-some-bytes (fd event exception)
     (handler-case
         (progn
           ;; Read however much we are able.
           (multiple-value-bind (buf bytes-read)
               (receive-from socket
                             :buffer echo-buf
                             :start read-index
                             :end max-bytes)

             ;; Unlike read-ing from a stream, receive-from
             ;; returns zero on an end-of-file read, so we turn
             ;; around and signal that condition so our
             ;; handler-case can deal with it properly like our
             ;; other examples.
             (when (zerop bytes-read)
               (error 'end-of-file))

             (format t "Read ~A bytes from ~A:~A~%" bytes-read who port)
             (incf read-index bytes-read))

           ;; Register the write handler if there is data to
           ;; write.
           ;;
           ;; Then, try to write some data to the socket right
           ;; away even though it might not be ready simply to
           ;; avoid another go around. The write-some-bytes
           ;; function must be able to catch econnreset because
           ;; this connection may be closed at the time of this
           ;; call. Normally, if the multiplexer has told me I
           ;; could write then it'd be ok, but since this write
           ;; is outside of the multiplexer and an optimization,
           ;; it needs to check.
           (when (/= write-index read-index)
             (unless write-handler-registered
               (set-io-handler *ex8-event-base*
                               (socket-os-fd socket)
                               :write
                               #'write-some-bytes)
               (setf write-handler-registered t))

             ;; See if I can write it right away!
             (write-some-bytes fd :write nil))

           ;; If I'm out of room to store more data then remove
           ;; myself from the io handler. When the write handler
           ;; notices that it has finished writing everything,
           ;; all indicies get set back to zero and the write
           ;; handler removes itself.  If write-some-bytes in
           ;; the call above worked, then read-index might not
           ;; equal max-bytes when this line of code gets
           ;; executed.
           (when (= read-index max-bytes)
             (funcall disconnector who port :read)
             (setf read-handler-registered nil)))

       (socket-connection-reset-error ()
         ;; Handle the client sending a reset.
         (format t "Client ~A:~A: connection reset by peer.~%" who port)
         (funcall disconnector who port :close))

       (end-of-file ()
         ;; When we get an end of file, that doesn't necessarily
         ;; mean the client went away, it could just mean that
         ;; the client performed a shutdown on the write end of
         ;; its socket and it is expecting the data stored in
         ;; the server to be written to it.  However, if there
         ;; is nothing left to write and our read end is closed,
         ;; we shall consider it that the client went away and
         ;; close the connection.
         (format t "Client ~A:~A produced end-of-file on a read.~%"
                 who port)
         (if (= read-index write-index)
             (funcall disconnector who port :close)
             (progn
               (funcall disconnector who port :read)
               (setf read-handler-registered nil)
               (setf eof-seen t))))))
```

3. Writing bytes to the client:

   While there are more bytes to write, we write them, keeping track of how much we wrote. Once we are out of data to write, we unregister the write handler, since we don't want to be called unnecessarily--usually the client socket is always ready to write. If we've seen the eof marker and are out of data, we close the client connection and are done. If we haven't seen it, then we determine if we are at the end of the buffer, if so, we reset the indices to the beginning. Either way, we re-register the read handler to acquire more data.

   We handle some new conditions here: isys:ewouldblock is needed because sometimes the underlying OS will mark an fd as ready to write when in fact it isn't when we get around to writing it. We might also see this condition when we tried to optimize the write of the data in the read handler since we did it outside of the multiplexer--this is idiomatic and saves a trip through the multiplexer more often than not. Seeing isys:ewouldblock simply aborts the write and we'll try again later. Under some conditions, send-to will signal an isys:epipe error, which means the client closed its connection. It is similar to a HANGUP condition in a format call with the stream API. We treat it similarly to a HANGUP.

```
;; This is the function responsible for writing bytes to the client.
(write-some-bytes (fd event exception)
  (handler-case
      (progn
        ;; If there is data to be written, write it.  NOTE:
        ;; There is often no indication of failure to write
        ;; with send-to. If I'm writing to a closed (by the
        ;; client) socket, it could be that send-to tells me
        ;; nothing is wrong and returns the number of bytes
        ;; written. In this case, nothing was written but we
        ;; have no way of knowing. Usually in this case, the
        ;; read handler will get a 0 bytes read on the socket
        ;; and we can know the connection is broken.
        (when (> read-index write-index)
          (let ((wrote-bytes (send-to socket echo-buf
                                      :start write-index
                                      :end read-index)))
            (format t "Wrote ~A bytes to ~A:~A~%" wrote-bytes who port)
            (incf write-index wrote-bytes)))

        ;; If we see we're out of data to write and we saw an eof,
        ;; then close the connection, we're done. If we didn't see an
        ;; eof, then unregister the write handler and reregister the
        ;; read handler to get more data. If the buffer indices
        ;; are at the very end, reset them to the beginning.
        (when (= read-index write-index)
          (if eof-seen
              (funcall disconnector who port :close)
              (progn

                ;; nothing more to write, so unregister writer
                (funcall disconnector who port :write)
                (setf write-handler-registered nil)

                ;; If we're at the end of the buffer, move to the
                ;; beginning so there is more room for data.
                (when (= read-index write-index max-bytes)
                  (setf read-index 0
                        write-index 0))

                ;; Reregister the read handler to get more data
                (unless read-handler-registered
                  (set-io-handler *ex8-event-base*
                                  (socket-os-fd socket)
                                  :read
                                  #'read-some-bytes)
                  (setf read-handler-registered t))))))

    (socket-connection-reset-error ()
      ;; If for somer eaon the client reset the network connection,
      ;; we'll get this signal.
      (format t "Client ~A:~A: connection reset by peer.~%" who port)
      (funcall disconnector who port :close))

    (isys:ewouldblock ()
      ;; Sometimes this happens on a write even though it
      ;; might have been marked as ready. Also we might have
      ;; asked to write on an unknown status socket. Ignore
      ;; it and we will try again later.
      (format t "write-some-bytes: ewouldblock~%")
      nil)

    (isys:epipe ()
      ;; In this server, if the client doesn't accept data,
      ;; it also means it will never send us data again. So
      ;; close the connection for good.
      (format t "Client ~A:~A got hangup on write.~%" who port)
      (funcall disconnector who port :close)))))
```

4. The returned closure of the io-buffer:

   Much like make-ex7-io-buffer, we return one of the internal closures which are appropriate for reading or writing by the multiplexer.

```
;; This is the function returned from make-ex8-io-buffer which
;; allows us access to the read/writer in the scope of the
;; closure.  We will ask for the correct functions when setting
;; up the io handlers.  NOTE: By simply asking for the handler,
;; I've assumed it is to be immediately put into an iolib event
;; handler. This is why they are considered registered at this
;; point.
(lambda (msg)
  (cond
    ((equalp msg :read-some-bytes)
     (setf read-handler-registered t)
     #'read-some-bytes)
    ((equalp msg :write-some-bytes)
     (setf write-handler-registered t)
     #'write-some-bytes)
    (t
     (error "make-ex8-buffer: Please supply :read-some-bytes or :write-some-bytes~%"))))))
```

2.7 Future Directions

Of course, more information should go into this tutorial, such as non-blocking connects/accepts, urgent TCP data, UDP examples, and IPV6. As time permits or contributions come in, these will be added.

2.8 Appendix A

This holds a rough approximation between the sources in this tutorial and the original sources in the network programming book by Stevens mentioned in the beginning of the tutorial. Aspects about the implementation of each client or server are summarized here.

2.8.1 The Clients

Figure 1.5, page 6

- ex1-client: Blocking I/O, daytime client, C Style

Figure 1.5, page 6

- ex2-client: Blocking I/O, daytime client, Lisp Style

Figure 1.5, page 6

- ex3-client: ex2-client, but with much more error handling

Figure 5.4, 5.5, page 114, 115

- ex4-client: Blocking I/O, line oriented

Figure 6.9, page 157

- ex5a-client: I/O multiplexing with iolib, line oriented, blocking I/O
- note: since this is still blocking I/O, I'm using *standard-input* and friends. Also note, with batch input, it will close the socket with in-flight data still present which is incorrect.

Figure 6.13, page 162

- ex5b-client: Same as ex5a-client EXCEPT shutdown is called when the input reaches end-of-file as to prevent in flight data from being destroyed on the way to the server.

2.8.2 The servers

Figure 4.11, page 101

- ex1-server: Iterative, blocking I/O daytime server, C Style, no error handling, one shot, line oriented

Figure 4.11, page 101

- ex2-server: Iterative, blocking I/O daytime server, Lisp Style, no error handling, loop forever, line oriented

Figure 4.11, page 101

- ex3-server: daytime server, ex2-server, but with error handling, line oriented

Figure 4.13, page 105

- ex4-server: daytime server, concurrent, blocking I/O, line oriented

Figure 5.2, 5.3, page 113, 114

- ex5-server: Concurrent, blocking I/O, echo server, line oriented

Figure 6.21,6.22 page 165,166

- ex6-server: I/O multiplexing of clients with iolib, line oriented, blocking I/O

Figure 6.21, 6.22 page 165,166

- ex7-server, ex7-buffer: individual I/O handlers for read/write, I/O multiplexing of clients with iolib, line oriented, blocking I/O, has problem with denial of service, page 167.

2.8 Appendix A

Figure 15.3, 15.4, 15.5 page 400-403 - ex8-server, ex8-buffer: nonblocking I/O, event-dispatch, send-to, receive-from

3 Bibliography

STEVENS1997	Stevens, W. Richard (1997) UNIX Network Programming, Networking APIs: Sockets and XTI 2nd Ed. Prentice Hall. ISBN■ 978-0134900124.