For the latest version of this document, see http://www.wangafu.net/~nickm/libevent-book/TOC.html

To get the source for the latest version of this document, install git and run "git clone git://github.com/nmathewson/libevent-book.git"

A tiny introduction to asynchronous IO

Most beginning programmers start with blocking IO calls. An IO call is *synchronous* if, when you call it, it does not return until the operation is completed, or until enough time has passed that your network stack gives up. When you call "connect()" on a TCP connection, for example, your operating system queues a SYN packet to the host on the other side of the TCP connection. It does not return control back to your application until either it has received a SYN ACK packet from the opposite host, or until enough time has passed that it decides to give up.

Here's an example of a really simple client using blocking network calls. It opens a connection to www.google.com, sends it a simple HTTP request, and prints the response to stdout.

Example: A simple blocking HTTP client

```
/* For sockaddr_in */
#include <netinet/in.h>
/* For socket functions */
#include <sys/socket.h>
/* For gethostbyname */
#include <netdb.h>

#include <unistd.h>
#include <string.h>
#include <stdio.h>

int main(int c, char **v)
{
    const char query[] =
        "GET / HTTP/1.0\r\n"
        "Host: www.google.com\r\n"
        "\r\n";
    const char hostname[] = "www.google.com";
```

```
struct sockaddr_in sin;
struct hostent *h;
const char *cp;
int fd;
ssize_t n_written, remaining;
char buf[1024];

/* Look up the IP address for the hostname.   Watch out; this isn't
   threadsafe on most platforms. */
h = gethostbyname(hostname);
if (!h) {
    fprintf(stderr, "Couldn't lookup %s: %s", hostname, hstrerror(h_errno));
    return 1;
}
if (h->h_addrtype != AF_INET) {
    fprintf(stderr, "No ipv6 support, sorry.");
    return 1;
}

/* Allocate a new socket */
fd = socket(AF_INET, SOCK_STREAM, 0);
if (fd < 0) {
    perror("socket");
    return 1;
}

/* Connect to the remote host. */
sin.sin_family = AF_INET;
sin.sin_port = htons(80);
sin.sin_addr = *(struct in_addr*)h->h_addr;
if (connect(fd, (struct sockaddr*) &sin, sizeof(sin))) {
    perror("connect");
    close(fd);
    return 1;
}

/* Write the query. */
/* XXX Can send succeed partially? */
cp = query;
remaining = strlen(query);
while (remaining) {
  n_written = send(fd, cp, remaining, 0);
  if (n_written <= 0) {
    perror("send");
    return 1;
  }
  remaining -= n_written;
  cp += n_written;
}

/* Get an answer back. */
while (1) {
    ssize_t result = recv(fd, buf, sizeof(buf), 0);
    if (result == 0) {
        break;
    } else if (result < 0) {
```

```
            perror("recv");
            close(fd);
            return 1;
        }
        fwrite(buf, 1, result, stdout);
    }

    close(fd);
    return 0;
}
```

All of the network calls in the code above are *blocking*: the gethostbyname does not return until it has succeeded or failed in resolving www.google.com; the connect does not return until it has connected; the recv calls do not return until they have received data or a close; and the send call does not return until it has at least flushed its output to the kernel's write buffers.

Now, blocking IO is not necessarily evil. If there's nothing else you wanted your program to do in the meantime, blocking IO will work fine for you. But suppose that you need to write a program to handle multiple connections at once. To make our example concrete: suppose that you want to read input from two connections, and you don't know which connection will get input first. You can't say

Bad Example

```
/* This won't work. */
char buf[1024];
int i, n;
while (i_still_want_to_read()) {
    for (i=0; i<n_sockets; ++i) {
        n = recv(fd[i], buf, sizeof(buf), 0);
        if (n==0)
            handle_close(fd[i]);
        else if (n<0)
            handle_error(fd[i], errno);
        else
            handle_input(fd[i], buf, n);
    }
}
```

because if data arrives on fd[2] first, your program won't even try reading from fd[2] until the reads from fd[0] and fd[1] have gotten some data and finished.

Sometimes people solve this problem with multithreading, or with multi-process servers. One of the simplest ways to do multithreading is with a separate process (or thread) to deal with each connection. Since each connection has its own process, a blocking IO call that waits for one connection won't make any of the other connections' processes block.

Here's another example program. It is a trivial server that listens for TCP connections on port 40713, reads data from its input one line at a time, and writes out the ROT13 obfuscation of line each as it arrives. It uses the Unix fork() call to create a new process for each incoming connection.

Example: Forking ROT13 server

```
/* For sockaddr_in */
#include <netinet/in.h>
/* For socket functions */
#include <sys/socket.h>

#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#define MAX_LINE 16384

char
rot13_char(char c)
{
    /* We don't want to use isalpha here; setting the locale would change
     * which characters are considered alphabetical. */
    if ((c ⩾ 'a' && c ⩽ 'm') || (c ⩾ 'A' && c ⩽ 'M'))
        return c + 13;
    else if ((c ⩾ 'n' && c ⩽ 'z') || (c ⩾ 'N' && c ⩽ 'Z'))
        return c - 13;
    else
        return c;
}

void
child(int fd)
{
    char outbuf[MAX_LINE+1];
    size_t outbuf_used = 0;
    ssize_t result;

    while (1) {
        char ch;
        result = recv(fd, &ch, 1, 0);
        if (result == 0) {
            break;
        } else if (result == -1) {
            perror("read");
            break;
        }

        /* We do this test to keep the user from overflowing the buffer. */
        if (outbuf_used < sizeof(outbuf)) {
            outbuf[outbuf_used++] = rot13_char(ch);
        }

        if (ch == '\n') {
```

```
            send(fd, outbuf, outbuf_used, 0);
            outbuf_used = 0;
            continue;
        }
    }
}

void
run(void)
{
    int listener;
    struct sockaddr_in sin;

    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = 0;
    sin.sin_port = htons(40713);

    listener = socket(AF_INET, SOCK_STREAM, 0);

#ifndef WIN32
    {
        int one = 1;
        setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &one, sizeof(one));
    }
#endif

    if (bind(listener, (struct sockaddr*)&sin, sizeof(sin)) < 0) {
        perror("bind");
        return;
    }

    if (listen(listener, 16)<0) {
        perror("listen");
        return;
    }



    while (1) {
        struct sockaddr_storage ss;
        socklen_t slen = sizeof(ss);
        int fd = accept(listener, (struct sockaddr*)&ss, &slen);
        if (fd < 0) {
            perror("accept");
        } else {
            if (fork() == 0) {
                child(fd);
                exit(0);
            }
        }
    }
}

int
main(int c, char **v)
{
```

```
    run();
    return 0;
}
```

So, do we have the perfect solution for handling multiple connections at once? Can I stop writing this book and go work on something else now? Not quite. First off, process creation (and even thread creation) can be pretty expensive on some platforms. In real life, you'd want to use a thread pool instead of creating new processes. But more fundamentally, threads won't scale as much as you'd like. If your program needs to handle thousands or tens of thousands of connections at a time, dealing with tens of thousands of threads will not be as efficient as trying to have only a few threads per CPU.

But if threading isn't the answer to having multiple connections, what is? In the Unix paradigm, you make your sockets *nonblocking*. The Unix call to do this is:

```
fcntl(fd, F_SETFL, O_NONBLOCK);
```

where fd is the file descriptor for the socket.
[A file descriptor is the number the kernel assigns to the socket when you open it. You use this number to make Unix calls referring to the socket.] Once you've made fd (the socket) nonblocking, from then on, whenever you make a network call to fd the call will either complete the operation immediately or return with a special error code to indicate "I couldn't make any progress now, try again." So our two-socket example might be naively written as:

Bad Example: busy-polling all sockets

```
/* This will work, but the performance will be unforgivably bad. */
int i, n;
char buf[1024];
for (i=0; i < n_sockets; ++i)
    fcntl(fd[i], F_SETFL, O_NONBLOCK);

while (i_still_want_to_read()) {
    for (i=0; i < n_sockets; ++i) {
        n = recv(fd[i], buf, sizeof(buf), 0);
        if (n == 0) {
            handle_close(fd[i]);
        } else if (n < 0) {
            if (errno == EAGAIN)
                 ; /* The kernel didn't have any data for us to read. */
            else
                 handle_error(fd[i], errno);
         } else {
            handle_input(fd[i], buf, n);
         }
    }
}
```

Now that we're using nonblocking sockets, the code above would *work*… but only barely. The performance will be awful, for two reasons. First, when there is no data to read on either connection the loop will spin indefinitely, using up all your CPU cycles. Second, if you try to handle more than one or two connections with this approach you'll do a kernel call for each one, whether it has any data for you or not. So what we need is a way to tell the kernel "wait until one of these sockets is ready to give me some data, and tell me which ones are ready."

The oldest solution that people still use for this problem is select(). The select() call takes three sets of fds (implemented as bit arrays): one for reading, one for writing, and one for "exceptions". It waits until a socket from one of the sets is ready and alters the sets to contain only the sockets ready for use.

Here is our example again, using select:

Example: Using select

```
/* If you only have a couple dozen fds, this version won't be awful */
fd_set readset;
int i, n;
char buf[1024];

while (i_still_want_to_read()) {
    int maxfd = -1;
    FD_ZERO(&readset);

    /* Add all of the interesting fds to readset */
    for (i=0; i < n_sockets; ++i) {
         if (fd[i]>maxfd) maxfd = fd[i];
         FD_SET(fd[i], &readset);
    }

    /* Wait until one or more fds are ready to read */
    select(maxfd+1, &readset, NULL, NULL, NULL);

    /* Process all of the fds that are still set in readset */
    for (i=0; i < n_sockets; ++i) {
        if (FD_ISSET(fd[i], &readset)) {
            n = recv(fd[i], buf, sizeof(buf), 0);
            if (n == 0) {
                handle_close(fd[i]);
            } else if (n < 0) {
                if (errno == EAGAIN)
                     ; /* The kernel didn't have any data for us to read. */
                else
                     handle_error(fd[i], errno);
             } else {
                handle_input(fd[i], buf, n);
             }
        }
    }
}
```

And here's a reimplementation of our ROT13 server, using select() this time.

Example: select()-based ROT13 server

```
/* For sockaddr_in */
#include <netinet/in.h>
/* For socket functions */
#include <sys/socket.h>
/* For fcntl */
#include <fcntl.h>
/* for select */
#include <sys/select.h>

#include <assert.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

#define MAX_LINE 16384

char
rot13_char(char c)
{
    /* We don't want to use isalpha here; setting the locale would change
     * which characters are considered alphabetical. */
    if ((c ⩾ 'a' && c ⩽ 'm') || (c ⩾ 'A' && c ⩽ 'M'))
        return c + 13;
    else if ((c ⩾ 'n' && c ⩽ 'z') || (c ⩾ 'N' && c ⩽ 'Z'))
        return c - 13;
    else
        return c;
}

struct fd_state {
    char buffer[MAX_LINE];
    size_t buffer_used;

    int writing;
    size_t n_written;
    size_t write_upto;
};

struct fd_state *
alloc_fd_state(void)
{
    struct fd_state *state = malloc(sizeof(struct fd_state));
    if (!state)
        return NULL;
    state→buffer_used = state→n_written = state→writing =
        state→write_upto = 0;
    return state;
}

void
```

```
free_fd_state(struct fd_state *state)
{
    free(state);
}

void
make_nonblocking(int fd)
{
    fcntl(fd, F_SETFL, O_NONBLOCK);
}

int
do_read(int fd, struct fd_state *state)
{
    char buf[1024];
    int i;
    ssize_t result;
    while (1) {
        result = recv(fd, buf, sizeof(buf), 0);
        if (result ⩽ 0)
            break;

        for (i=0; i < result; ++i)  {
            if (state→buffer_used < sizeof(state→buffer))
                state→buffer[state→buffer_used++] = rot13_char(buf[i]);
            if (buf[i] == '\n') {
                state→writing = 1;
                state→write_upto = state→buffer_used;
            }
        }
    }

    if (result == 0) {
        return 1;
    } else if (result < 0) {
        if (errno == EAGAIN)
            return 0;
        return -1;
    }

    return 0;
}

int
do_write(int fd, struct fd_state *state)
{
    while (state→n_written < state→write_upto) {
        ssize_t result = send(fd, state→buffer + state→n_written,
                              state→write_upto - state→n_written, 0);
        if (result < 0) {
            if (errno == EAGAIN)
                return 0;
            return -1;
        }
        assert(result ≠ 0);
```

```
        state->n_written += result;
    }

    if (state->n_written == state->buffer_used)
        state->n_written = state->write_upto = state->buffer_used = 0;

    state->writing = 0;

    return 0;
}

void
run(void)
{
    int listener;
    struct fd_state *state[FD_SETSIZE];
    struct sockaddr_in sin;
    int i, maxfd;
    fd_set readset, writeset, exset;

    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = 0;
    sin.sin_port = htons(40713);

    for (i = 0; i < FD_SETSIZE; ++i)
        state[i] = NULL;

    listener = socket(AF_INET, SOCK_STREAM, 0);
    make_nonblocking(listener);

#ifndef WIN32
    {
        int one = 1;
        setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &one, sizeof(one));
    }
#endif

    if (bind(listener, (struct sockaddr*)&sin, sizeof(sin)) < 0) {
        perror("bind");
        return;
    }

    if (listen(listener, 16)<0) {
        perror("listen");
        return;
    }

    FD_ZERO(&readset);
    FD_ZERO(&writeset);
    FD_ZERO(&exset);

    while (1) {
        maxfd = listener;

        FD_ZERO(&readset);
        FD_ZERO(&writeset);
```

```
        FD_ZERO(&exset);

        FD_SET(listener, &readset);

        for (i=0; i < FD_SETSIZE; ++i) {
            if (state[i]) {
                if (i > maxfd)
                    maxfd = i;
                FD_SET(i, &readset);
                if (state[i]->writing) {
                    FD_SET(i, &writeset);
                }
            }
        }

        if (select(maxfd+1, &readset, &writeset, &exset, NULL) < 0) {
            perror("select");
            return;
        }

        if (FD_ISSET(listener, &readset)) {
            struct sockaddr_storage ss;
            socklen_t slen = sizeof(ss);
            int fd = accept(listener, (struct sockaddr*)&ss, &slen);
            if (fd < 0) {
                perror("accept");
            } else if (fd > FD_SETSIZE) {
                close(fd);
            } else {
                make_nonblocking(fd);
                state[fd] = alloc_fd_state();
                assert(state[fd]);/*XXX*/
            }
        }

        for (i=0; i < maxfd+1; ++i) {
            int r = 0;
            if (i == listener)
                continue;

            if (FD_ISSET(i, &readset)) {
                r = do_read(i, state[i]);
            }
            if (r == 0 && FD_ISSET(i, &writeset)) {
                r = do_write(i, state[i]);
            }
            if (r) {
                free_fd_state(state[i]);
                state[i] = NULL;
                close(i);
            }
        }
    }
}

int
```

```
main(int c, char **v)
{
    setvbuf(stdout, NULL, _IONBF, 0);

    run();
    return 0;
}
```

But we're still not done. Because generating and reading the select() bit arrays takes time proportional to the largest fd that you provided for select(), the select() call scales terribly when the number of sockets is high.
[On the userspace side, generating and reading the bit arrays can be made to take time proportional to the number of fds that you provided for select(). But on the kernel side, reading the bit arrays takes time proportional to the largest fd in the bit array, which tends to be around *the total number of fds in use in the whole program*, regardless of how many fds are added to the sets in select().]

Different operating systems have provided different replacement functions for select. These include poll(), epoll(), kqueue(), evports, and /dev/poll. All of these give better performance than select(), and all but poll() give O(1) performance for adding a socket, removing a socket, and for noticing that a socket is ready for IO.

Unfortunately, none of the efficient interfaces is a ubiquitous standard. Linux has epoll(), the BSDs (including Darwin) have kqueue(), Solaris has evports and /dev/poll… *and none of these operating systems has any of the others*. So if you want to write a portable high-performance asynchronous application, you'll need an abstraction that wraps all of these interfaces, and provides whichever one of them is the most efficient.

And that's what the lowest level of the Libevent API does for you. It provides a consistent interface to various select() replacements, using the most efficient version available on the computer where it's running.

Here's yet another version of our asynchronous ROT13 server. This time, it uses Libevent 2 instead of select(). Note that the fd_sets are gone now: instead, we associate and disassociate events with a struct event_base, which might be implemented in terms of select(), poll(), epoll(), kqueue(), etc.

Example: A low-level ROT13 server with Libevent

```
/* For sockaddr_in */
#include <netinet/in.h>
/* For socket functions */
#include <sys/socket.h>
/* For fcntl */
#include <fcntl.h>
```

```
#include <event2/event.h>

#include <assert.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

#define MAX_LINE 16384

void do_read(evutil_socket_t fd, short events, void *arg);
void do_write(evutil_socket_t fd, short events, void *arg);

char
rot13_char(char c)
{
    /* We don't want to use isalpha here; setting the locale would change
     * which characters are considered alphabetical. */
    if ((c ⩾ 'a' && c ⩽ 'm') || (c ⩾ 'A' && c ⩽ 'M'))
        return c + 13;
    else if ((c ⩾ 'n' && c ⩽ 'z') || (c ⩾ 'N' && c ⩽ 'Z'))
        return c - 13;
    else
        return c;
}

struct fd_state {
    char buffer[MAX_LINE];
    size_t buffer_used;

    size_t n_written;
    size_t write_upto;

    struct event *read_event;
    struct event *write_event;
};

struct fd_state *
alloc_fd_state(struct event_base *base, evutil_socket_t fd)
{
    struct fd_state *state = malloc(sizeof(struct fd_state));
    if (!state)
        return NULL;
    state→read_event = event_new(base, fd, EV_READ|EV_PERSIST, do_read, state);
    if (!state→read_event) {
        free(state);
        return NULL;
    }
    state→write_event =
        event_new(base, fd, EV_WRITE|EV_PERSIST, do_write, state);

    if (!state→write_event) {
        event_free(state→read_event);
        free(state);
        return NULL;
```

```
    }

    state->buffer_used = state->n_written = state->write_upto = 0;

    assert(state->write_event);
    return state;
}

void
free_fd_state(struct fd_state *state)
{
    event_free(state->read_event);
    event_free(state->write_event);
    free(state);
}

void
do_read(evutil_socket_t fd, short events, void *arg)
{
    struct fd_state *state = arg;
    char buf[1024];
    int i;
    ssize_t result;
    while (1) {
        assert(state->write_event);
        result = recv(fd, buf, sizeof(buf), 0);
        if (result <= 0)
            break;

        for (i=0; i < result; ++i)  {
            if (state->buffer_used < sizeof(state->buffer))
                state->buffer[state->buffer_used++] = rot13_char(buf[i]);
            if (buf[i] == '\n') {
                assert(state->write_event);
                event_add(state->write_event, NULL);
                state->write_upto = state->buffer_used;
            }
        }
    }

    if (result == 0) {
        free_fd_state(state);
    } else if (result < 0) {
        if (errno == EAGAIN) // XXXX use evutil macro
            return;
        perror("recv");
        free_fd_state(state);
    }
}

void
do_write(evutil_socket_t fd, short events, void *arg)
{
    struct fd_state *state = arg;

    while (state->n_written < state->write_upto) {
```

```
        ssize_t result = send(fd, state->buffer + state->n_written,
                              state->write_upto - state->n_written, 0);
        if (result < 0) {
            if (errno == EAGAIN) // XXX use evutil macro
                return;
            free_fd_state(state);
            return;
        }
        assert(result != 0);

        state->n_written += result;
    }

    if (state->n_written == state->buffer_used)
        state->n_written = state->write_upto = state->buffer_used = 1;

    event_del(state->write_event);
}

void
do_accept(evutil_socket_t listener, short event, void *arg)
{
    struct event_base *base = arg;
    struct sockaddr_storage ss;
    socklen_t slen = sizeof(ss);
    int fd = accept(listener, (struct sockaddr*)&ss, &slen);
    if (fd < 0) { // XXXX eagain??
        perror("accept");
    } else if (fd > FD_SETSIZE) {
        close(fd); // XXX replace all closes with EVUTIL_CLOSESOCKET */
    } else {
        struct fd_state *state;
        evutil_make_socket_nonblocking(fd);
        state = alloc_fd_state(base, fd);
        assert(state); /*XXX err*/
        assert(state->write_event);
        event_add(state->read_event, NULL);
    }
}

void
run(void)
{
    evutil_socket_t listener;
    struct sockaddr_in sin;
    struct event_base *base;
    struct event *listener_event;

    base = event_base_new();
    if (!base)
        return; /*XXXerr*/

    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = 0;
    sin.sin_port = htons(40713);
```

```
    listener = socket(AF_INET, SOCK_STREAM, 0);
    evutil_make_socket_nonblocking(listener);

#ifndef WIN32
    {
        int one = 1;
        setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &one, sizeof(one));
    }
#endif

    if (bind(listener, (struct sockaddr*)&sin, sizeof(sin)) < 0) {
        perror("bind");
        return;
    }

    if (listen(listener, 16)<0) {
        perror("listen");
        return;
    }

    listener_event = event_new(base, listener, EV_READ|EV_PERSIST, do_accept, (void*)base);
    /*XXX check it */
    event_add(listener_event, NULL);

    event_base_dispatch(base);
}

int
main(int c, char **v)
{
    setvbuf(stdout, NULL, _IONBF, 0);

    run();
    return 0;
}
```

(Other things to note in the code: instead of typing the sockets as "int", we're using the type evutil_socket_t. Instead of calling fcntl(O_NONBLOCK) to make the sockets nonblocking, we're calling evutil_make_socket_nonblocking. These changes make our code compatible with the divergent parts of the Win32 networking API.)

What about convenience? (and what about Windows?)

You've probably noticed that as our code has gotten more efficient, it has also gotten more complex. Back when we were forking, we didn't have to manage a buffer for each connection: we just had a separate stack-allocated buffer for each process. We didn't need to explicitly track whether each socket was reading or writing: that was implicit in our location in the code. And we didn't need a structure to track how much of each operation had completed: we just used loops and stack variables.

Moreover, if you're deeply experienced with networking on Windows, you'll realize that Libevent probably isn't getting optimal performance when it's used as in the example above. On Windows, the way you do fast asynchronous IO is not with a select()-like interface: it's by using the IOCP (IO Completion Ports) API. Unlike all the fast networking APIs, IOCP does not alert your program when a socket is *ready* for an operation that your program then has to perform. Instead, the program tells the Windows networking stack to *start* a network operation, and IOCP tells the program when the operation has finished.

Fortunately, the Libevent 2 "bufferevents" interface solves both of these issues: it makes programs much simpler to write, and provides an interface that Libevent can implement efficiently on Windows *and* on Unix.

Here's our ROT13 server one last time, using the bufferevents API.

Example: A simpler ROT13 server with Libevent

```
/* For sockaddr_in */
#include <netinet/in.h>
/* For socket functions */
#include <sys/socket.h>
/* For fcntl */
#include <fcntl.h>

#include <event2/event.h>
#include <event2/buffer.h>
#include <event2/bufferevent.h>

#include <assert.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

#define MAX_LINE 16384

void do_read(evutil_socket_t fd, short events, void *arg);
void do_write(evutil_socket_t fd, short events, void *arg);

char
rot13_char(char c)
{
    /* We don't want to use isalpha here; setting the locale would change
     * which characters are considered alphabetical. */
    if ((c ⩾ 'a' && c ⩽ 'm') || (c ⩾ 'A' && c ⩽ 'M'))
        return c + 13;
    else if ((c ⩾ 'n' && c ⩽ 'z') || (c ⩾ 'N' && c ⩽ 'Z'))
        return c - 13;
    else
        return c;
}
```

```
void
readcb(struct bufferevent *bev, void *ctx)
{
    struct evbuffer *input, *output;
    char *line;
    size_t n;
    int i;
    input = bufferevent_get_input(bev);
    output = bufferevent_get_output(bev);

    while ((line = evbuffer_readln(input, &n, EVBUFFER_EOL_LF))) {
        for (i = 0; i < n; ++i)
            line[i] = rot13_char(line[i]);
        evbuffer_add(output, line, n);
        evbuffer_add(output, "\n", 1);
        free(line);
    }

    if (evbuffer_get_length(input) ⩾ MAX_LINE) {
        /* Too long; just process what there is and go on so that the buffer
         * doesn't grow infinitely long. */
        char buf[1024];
        while (evbuffer_get_length(input)) {
            int n = evbuffer_remove(input, buf, sizeof(buf));
            for (i = 0; i < n; ++i)
                buf[i] = rot13_char(buf[i]);
            evbuffer_add(output, buf, n);
        }
        evbuffer_add(output, "\n", 1);
    }
}

void
errorcb(struct bufferevent *bev, short error, void *ctx)
{
    if (error & BEV_EVENT_EOF) {
        /* connection has been closed, do any clean up here */
        /* ... */
    } else if (error & BEV_EVENT_ERROR) {
        /* check errno to see what error occurred */
        /* ... */
    } else if (error & BEV_EVENT_TIMEOUT) {
        /* must be a timeout event handle, handle it */
        /* ... */
    }
    bufferevent_free(bev);
}

void
do_accept(evutil_socket_t listener, short event, void *arg)
{
    struct event_base *base = arg;
    struct sockaddr_storage ss;
    socklen_t slen = sizeof(ss);
    int fd = accept(listener, (struct sockaddr*)&ss, &slen);
    if (fd < 0) {
```

```
        perror("accept");
    } else if (fd > FD_SETSIZE) {
        close(fd);
    } else {
        struct bufferevent *bev;
        evutil_make_socket_nonblocking(fd);
        bev = bufferevent_socket_new(base, fd, BEV_OPT_CLOSE_ON_FREE);
        bufferevent_setcb(bev, readcb, NULL, errorcb, NULL);
        bufferevent_setwatermark(bev, EV_READ, 0, MAX_LINE);
        bufferevent_enable(bev, EV_READ|EV_WRITE);
    }
}

void
run(void)
{
    evutil_socket_t listener;
    struct sockaddr_in sin;
    struct event_base *base;
    struct event *listener_event;

    base = event_base_new();
    if (!base)
        return; /*XXXerr*/

    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = 0;
    sin.sin_port = htons(40713);

    listener = socket(AF_INET, SOCK_STREAM, 0);
    evutil_make_socket_nonblocking(listener);

#ifndef WIN32
    {
        int one = 1;
        setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &one, sizeof(one));
    }
#endif

    if (bind(listener, (struct sockaddr*)&sin, sizeof(sin)) < 0) {
        perror("bind");
        return;
    }

    if (listen(listener, 16)<0) {
        perror("listen");
        return;
    }

    listener_event = event_new(base, listener, EV_READ|EV_PERSIST, do_accept, (void*)base);
    /*XXX check it */
    event_add(listener_event, NULL);

    event_base_dispatch(base);
}
```

```
int
main(int c, char **v)
{
    setvbuf(stdout, NULL, _IONBF, 0);

    run();
    return 0;
}
```

How efficient is all of this, really?

XXXX write an efficiency section here. The benchmarks on the libevent page are really out of date.

Last updated 2014-03-31 12:23:10 EDT