```
 1: // $Id: sockets.h,v 1.1 2015-05-12 18:48:40-07 - - $
 2:
 3: #ifndef __SOCKET_H__
 4: #define __SOCKET_H__
 5:
 6: #include <cstring>
 7: #include <stdexcept>
 8: #include <string>
 9: #include <vector>
10: using namespace std;
11:
12: #include <arpa/inet.h>
13: #include <netdb.h>
14: #include <netinet/in.h>
15: #include <string>
16: #include <sys/socket.h>
17: #include <sys/types.h>
18: #include <sys/wait.h>
19: #include <unistd.h>
20:
21: //
22: // class base_socket:
23: // mostly protected and not used by applications
24: //
25:
26: class base_socket {
27:    private:
28:       static constexpr size_t MAXRECV = 0xFFFF;
29:       static constexpr int CLOSED_FD = -1;
30:       int socket_fd {CLOSED_FD};
31:       sockaddr_in socket_addr;
32:       base_socket (const base_socket&) = delete; // prevent copying
33:       base_socket& operator= (const base_socket&) = delete;
34:    protected:
35:       base_socket(); // only derived classes may construct
36:       ~base_socket();
37:       // server_socket initialization
38:       void create();
39:       void bind (const in_port_t port);
40:       void listen() const;
41:       void accept (base_socket&) const;
42:       // client_socket initialization
43:       void connect (const string host, const in_port_t port);
44:       // accepted_socket initialization
45:       void set_socket_fd (int fd);
46:    public:
47:       void close();
48:       ssize_t send (const void* buffer, size_t bufsize);
49:       ssize_t recv (void* buffer, size_t bufsize);
50:       void set_non_blocking (const bool);
51:       friend string to_string (const base_socket& sock);
52: };
53:
```

```
54:
55: //
56: // class accepted_socket
57: // used by server when a client connects
58: //
59:
60: class accepted_socket: public base_socket {
61:    public:
62:       accepted_socket() {}
63:       accepted_socket(int fd) { set_socket_fd (fd); }
64: };
65:
66: //
67: // class client_socket
68: // used by client application to connect to server
69: //
70:
71: class client_socket: public base_socket {
72:    public:
73:       client_socket (string host, in_port_t port);
74: };
75:
76: //
77: // class server_socket
78: // single use class by server application
79: //
80:
81: class server_socket: public base_socket {
82:    public:
83:       server_socket (in_port_t port);
84:       void accept (accepted_socket& sock) {
85:          base_socket::accept (sock);
86:       }
87: };
88:
```

```
 89:
 90: //
 91: // class socket_error
 92: // base class for throwing socket errors
 93: //
 94:
 95: class socket_error: public runtime_error {
 96:    public:
 97:       explicit socket_error (const string& what): runtime_error(what){}
 98: };
 99:
100: //
101: // class socket_sys_error
102: // subclass to record status of extern int errno variable
103: //
104:
105: class socket_sys_error: public socket_error {
106:    public:
107:       int sys_errno;
108:       explicit socket_sys_error (const string& what):
109:                socket_error(what + ": " + strerror (errno)),
110:                sys_errno(errno) {}
111: };
112:
113: //
114: // class socket_h_error
115: // subclass to record status of extern int h_errno variable
116: //
117:
118: class socket_h_error: public socket_error {
119:    public:
120:       int host_errno;
121:       explicit socket_h_error (const string& what):
122:                socket_error(what + ": " + hstrerror (h_errno)),
123:                host_errno(h_errno) {}
124: };
125:
```

```
126:
127: //
128: // class hostinfo
129: // information about a host given hostname or IPv4 address
130: //
131:
132: class hostinfo {
133:    public:
134:       const string hostname;
135:       const vector<string> aliases;
136:       const vector<in_addr> addresses;
137:       hostinfo (); // localhost
138:       hostinfo (hostent*);
139:       hostinfo (const string& hostname);
140:       hostinfo (const in_addr& ipv4_addr);
141:       friend string to_string (const hostinfo&);
142: };
143:
144: string localhost();
145: string to_string (const in_addr& ipv4_addr);
146:
147: #endif
148:
```

```
 1: // $Id: protocol.h,v 1.2 2015-05-12 18:59:40-07 - - $
 2:
 3: #ifndef __PROTOCOL__H__
 4: #define __PROTOCOL__H__
 5:
 6: #include <cstdint>
 7: #include <cstring>
 8: #include <iostream>
 9: using namespace std;
10:
11: #include "sockets.h"
12:
13: enum cix_command {CIX_ERROR = 0, CIX_EXIT,
14:                   CIX_GET, CIX_HELP, CIX_LS, CIX_PUT, CIX_RM,
15:                   CIX_FILE, CIX_LSOUT, CIX_ACK, CIX_NAK};
16:
17: size_t constexpr FILENAME_SIZE = 59;
18: struct cix_header {
19:    uint32_t nbytes {0};
20:    uint8_t command {0};
21:    char filename[FILENAME_SIZE] {};
22: };
23:
24: void send_packet (base_socket& socket,
25:                   const void* buffer, size_t bufsize);
26:
27: void recv_packet (base_socket& socket, void* buffer, size_t bufsize);
28:
29: ostream& operator<< (ostream& out, const cix_header& header);
30:
31: string get_cix_server_host (const vector<string>& args, size_t index);
32:
33: in_port_t get_cix_server_port (const vector<string>& args,
34:                                size_t index);
35:
36: #endif
37:
```

```
 1: // $Id: logstream.h,v 1.1 2015-05-12 18:48:40-07 - - $
 2:
 3: //
 4: // class logstream
 5: // replacement for initial cout so that each call to a logstream
 6: // will prefix the line of output with an identification string
 7: // and a process id.  Template functions must be in header files
 8: // and the others are trivial.
 9: //
10:
11: #ifndef __LOGSTREAM_H__
12: #define __LOGSTREAM_H__
13:
14: #include <cassert>
15: #include <iostream>
16: #include <string>
17: #include <vector>
18: using namespace std;
19:
20: #include <sys/types.h>
21: #include <unistd.h>
22:
23: class logstream {
24:    private:
25:       ostream& out;
26:       string execname_;
27:    public:
28:
29:       // Constructor may or may not have the execname available.
30:       logstream (ostream& out, const string& execname = ""):
31:                  out (out), execname_ (execname) {
32:       }
33:
34:       // First line of main should execname if logstream is global.
35:       void execname (const string& name) { execname_ = name; }
36:       string execname() { return execname_; }
37:
38:       // First call should be the logstream, not cout.
39:       // Then forward result to the standard ostream.
40:       template <typename T>
41:       ostream& operator<< (const T& obj) {
42:          assert (execname_.size() > 0);
43:          out << execname_ << "(" << getpid() << "): " << obj;
44:          return out;
45:       }
46:
47: };
48:
49: #endif
50:
```

```
// $Id: sockets.cpp,v 1.1 2015-05-12 18:48:40-07 - - $

#include <cerrno>
#include <cstring>
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

#include <fcntl.h>
#include <limits.h>

#include "sockets.h"

base_socket::base_socket() {
   memset (&socket_addr, 0, sizeof (socket_addr));
}

base_socket::~base_socket() {
   if (socket_fd != CLOSED_FD) close();
}

void base_socket::close() {
   int status = ::close (socket_fd);
   if (status < 0) throw socket_sys_error ("close("
                         + to_string(socket_fd) + ")");
   socket_fd = CLOSED_FD;
}

void base_socket::create() {
   socket_fd = ::socket (AF_INET, SOCK_STREAM, 0);
   if (socket_fd < 0) throw socket_sys_error ("socket");
   int on = 1;
   int status = ::setsockopt (socket_fd, SOL_SOCKET, SO_REUSEADDR,
                              &on, sizeof on);
   if (status < 0) throw socket_sys_error ("setsockopt");
}

void base_socket::bind (const in_port_t port) {
   socket_addr.sin_family = AF_INET;
   socket_addr.sin_addr.s_addr = INADDR_ANY;
   socket_addr.sin_port = htons (port);
   int status = ::bind (socket_fd,
                        reinterpret_cast<sockaddr*> (&socket_addr),
                        sizeof socket_addr);
   if (status < 0) throw socket_sys_error ("bind(" + to_string (port)
                                           + ")");
}

void base_socket::listen() const {
   int status = ::listen (socket_fd, SOMAXCONN);
   if (status < 0) throw socket_sys_error ("listen");
}

```

```
 55:
 56: void base_socket::accept (base_socket& socket) const {
 57:    int addr_length = sizeof socket.socket_addr;
 58:    socket.socket_fd = ::accept (socket_fd,
 59:                  reinterpret_cast<sockaddr*> (&socket.socket_addr),
 60:                  reinterpret_cast<socklen_t*> (&addr_length));
 61:    if (socket.socket_fd < 0) throw socket_sys_error ("accept");
 62: }
 63:
 64: ssize_t base_socket::send (const void* buffer, size_t bufsize) {
 65:    int nbytes = ::send (socket_fd, buffer, bufsize, MSG_NOSIGNAL);
 66:    if (nbytes < 0) throw socket_sys_error ("send");
 67:    return nbytes;
 68: }
 69:
 70: ssize_t base_socket::recv (void* buffer, size_t bufsize) {
 71:    memset (buffer, 0, bufsize);
 72:    ssize_t nbytes = ::recv (socket_fd, buffer, bufsize, 0);
 73:    if (nbytes < 0) throw socket_sys_error ("recv");
 74:    return nbytes;
 75: }
 76:
 77: void base_socket::connect (const string host, const in_port_t port) {
 78:    struct hostent *hostp = ::gethostbyname (host.c_str());
 79:    if (hostp == NULL) throw socket_h_error ("gethostbyname("
 80:                               + host + ")");
 81:    socket_addr.sin_family = AF_INET;
 82:    socket_addr.sin_port = htons (port);
 83:    socket_addr.sin_addr = *reinterpret_cast<in_addr*> (hostp->h_addr);
 84:    int status = ::connect (socket_fd,
 85:                            reinterpret_cast<sockaddr*> (&socket_addr),
 86:                            sizeof (socket_addr));
 87:    if (status < 0) throw socket_sys_error ("connect(" + host + ":"
 88:                           + to_string (port) + ")");
 89: }
 90:
 91: void base_socket::set_socket_fd (int fd) {
 92:    socklen_t addrlen = sizeof socket_addr;
 93:    int rc = getpeername (fd, reinterpret_cast<sockaddr*> (&socket_addr),
 94:                          &addrlen);
 95:    if (rc < 0) throw socket_sys_error ("set_socket_fd("
 96:                        + to_string (fd) + "): getpeername");
 97:    socket_fd = fd;
 98:    if (socket_addr.sin_family != AF_INET)
 99:       throw socket_error ("address not AF_INET");
100: }
101:
102: void base_socket::set_non_blocking (const bool blocking) {
103:    int opts = ::fcntl (socket_fd, F_GETFL);
104:    if (opts < 0) throw socket_sys_error ("fcntl");
105:    if (blocking) opts |= O_NONBLOCK;
106:             else opts &= compl O_NONBLOCK;
107:    opts = ::fcntl (socket_fd, F_SETFL, opts);
108:    if (opts < 0) throw socket_sys_error ("fcntl");
109: }
110:
```

```
111:
112: client_socket::client_socket (string host, in_port_t port) {
113:    base_socket::create();
114:    base_socket::connect (host, port);
115: }
116:
117: server_socket::server_socket (in_port_t port) {
118:    base_socket::create();
119:    base_socket::bind (port);
120:    base_socket::listen();
121: }
122:
123: string to_string (const hostinfo& info) {
124:    return info.hostname + " (" + to_string (info.addresses[0]) + ")";
125: }
126:
127: string to_string (const in_addr& ipv4_addr) {
128:    char buffer[INET_ADDRSTRLEN];
129:    const char *result = ::inet_ntop (AF_INET, &ipv4_addr,
130:                                      buffer, sizeof buffer);
131:    if (result == NULL) throw socket_sys_error ("inet_ntop");
132:    return result;
133: }
134:
135: string to_string (const base_socket& sock) {
136:    hostinfo info (sock.socket_addr.sin_addr);
137:    return info.hostname + " (" + to_string (info.addresses[0])
138:           + ") port " + to_string (ntohs (sock.socket_addr.sin_port));
139: }
140:
```

```
141:
142: string init_hostname (hostent* host) {
143:    if (host == nullptr) throw socket_h_error ("gethostbyname");
144:    return host->h_name;
145: }
146:
147: vector<string> init_aliases (hostent* host) {
148:    if (host == nullptr) throw socket_h_error ("gethostbyname");
149:    vector<string> init_aliases;
150:    for (char** alias = host->h_aliases; *alias != nullptr; ++alias) {
151:       init_aliases.push_back (*alias);
152:    }
153:    return init_aliases;
154: }
155:
156: vector<in_addr> init_addresses (hostent* host) {
157:    vector<in_addr> init_addresses;
158:    if (host == nullptr) throw socket_h_error ("gethostbyname");
159:    for (in_addr** addr =
160:                  reinterpret_cast<in_addr**> (host->h_addr_list);
161:         *addr != nullptr; ++addr) {
162:       init_addresses.push_back (**addr);
163:    }
164:    return init_addresses;
165: }
166:
167: hostinfo::hostinfo (hostent* host):
168:    hostname (init_hostname (host)),
169:    aliases (init_aliases (host)),
170:    addresses (init_addresses (host)) {
171: }
172:
173: hostinfo::hostinfo(): hostinfo (localhost()) {
174: }
175:
176: hostinfo::hostinfo (const string& hostname):
177:           hostinfo (::gethostbyname (hostname.c_str())) {
178: }
179:
180: hostinfo::hostinfo (const in_addr& ipv4_addr):
181:           hostinfo (::gethostbyaddr (&ipv4_addr, sizeof ipv4_addr,
182:                                      AF_INET)) {
183: }
184:
185: string localhost() {
186:    char hostname[HOST_NAME_MAX] {};
187:    int rc = gethostname (hostname, sizeof hostname);
188:    if (rc < 0) throw socket_sys_error ("gethostname");
189:    return hostname;
190: }
191:
```

```
 1: // $Id: protocol.cpp,v 1.2 2015-05-12 18:59:40-07 - - $
 2:
 3: #include <unordered_map>
 4: #include <string>
 5: using namespace std;
 6:
 7: #include "protocol.h"
 8:
 9: const unordered_map<int,string> cix_command_map {
10:    {int (CIX_ERROR), "CIX_ERROR"},
11:    {int (CIX_EXIT ), "CIX_EXIT" },
12:    {int (CIX_GET  ), "CIX_GET"  },
13:    {int (CIX_HELP ), "CIX_HELP" },
14:    {int (CIX_LS   ), "CIX_LS"   },
15:    {int (CIX_PUT  ), "CIX_PUT"  },
16:    {int (CIX_RM   ), "CIX_RM"   },
17:    {int (CIX_FILE ), "CIX_FILE" },
18:    {int (CIX_LSOUT), "CIX_LSOUT"},
19:    {int (CIX_ACK  ), "CIS_ACK"  },
20:    {int (CIX_NAK  ), "CIS_NAK"  },
21: };
22:
23:
24: void send_packet (base_socket& socket,
25:                   const void* buffer, size_t bufsize) {
26:    const char* bufptr = static_cast<const char*> (buffer);
27:    ssize_t ntosend = bufsize;
28:    do {
29:       ssize_t nbytes = socket.send (bufptr, ntosend);
30:       if (nbytes < 0) throw socket_sys_error (to_string (socket));
31:       bufptr += nbytes;
32:       ntosend -= nbytes;
33:    }while (ntosend > 0);
34: }
35:
36: void recv_packet (base_socket& socket, void* buffer, size_t bufsize) {
37:    char* bufptr = static_cast<char*> (buffer);
38:    ssize_t ntorecv = bufsize;
39:    do {
40:       ssize_t nbytes = socket.recv (bufptr, ntorecv);
41:       if (nbytes < 0) throw socket_sys_error (to_string (socket));
42:       if (nbytes == 0) throw socket_error (to_string (socket)
43:                                            + " is closed");
44:       bufptr += nbytes;
45:       ntorecv -= nbytes;
46:    }while (ntorecv > 0);
47: }
48:
49: ostream& operator<< (ostream& out, const cix_header& header) {
50:    const auto& itor = cix_command_map.find (header.command);
51:    string code = itor == cix_command_map.end() ? "?" : itor->second;
52:    cout << "{" << header.nbytes << "," << code << "="
53:         << int (header.command) << ",\"" << header.filename
54:         << "\"}";
55:    return out;
56: }
57:
```

```
58:
59: string get_cix_server_host (const vector<string>& args, size_t index) {
60:    if (index < args.size()) return args[index];
61:    char* host = getenv ("CIX_SERVER_HOST");
62:    if (host != nullptr) return host;
63:    return "localhost";
64: }
65:
66: in_port_t get_cix_server_port (const vector<string>& args,
67:                                size_t index) {
68:    string port = "-1";
69:    if (index < args.size()) port = args[index];
70:    else {
71:       char* envport = getenv ("CIX_SERVER_PORT");
72:       if (envport != nullptr) port = envport;
73:    }
74:    return stoi (port);
75: }
76:
```

```
 1: // $Id: cix.cpp,v 1.2 2015-05-12 18:59:40-07 - - $
 2:
 3: #include <iostream>
 4: #include <string>
 5: #include <vector>
 6: #include <unordered_map>
 7: using namespace std;
 8:
 9: #include <libgen.h>
10: #include <sys/types.h>
11: #include <unistd.h>
12:
13: #include "protocol.h"
14: #include "logstream.h"
15: #include "sockets.h"
16:
17: logstream log (cout);
18: struct cix_exit: public exception {};
19:
20: unordered_map<string,cix_command> command_map {
21:    {"exit", CIX_EXIT},
22:    {"help", CIX_HELP},
23:    {"ls"  , CIX_LS  },
24: };
25:
26: void cix_help() {
27:    static vector<string> help = {
28:       "exit         - Exit the program.  Equivalent to EOF.",
29:       "get filename - Copy remote file to local host.",
30:       "help         - Print help summary.",
31:       "ls           - List names of files on remote server.",
32:       "put filename - Copy local file to remote host.",
33:       "rm filename  - Remove file from remote server.",
34:    };
35:    for (const auto& line: help) cout << line << endl;
36: }
37:
38: void cix_ls (client_socket& server) {
39:    cix_header header;
40:    header.command = CIX_LS;
41:    log << "sending header " << header << endl;
42:    send_packet (server, &header, sizeof header);
43:    recv_packet (server, &header, sizeof header);
44:    log << "received header " << header << endl;
45:    if (header.command != CIX_LSOUT) {
46:       log << "sent CIX_LS, server did not return CIX_LSOUT" << endl;
47:       log << "server returned " << header << endl;
48:    }else {
49:       char buffer[header.nbytes + 1];
50:       recv_packet (server, buffer, header.nbytes);
51:       log << "received " << header.nbytes << " bytes" << endl;
52:       buffer[header.nbytes] = '\0';
53:       cout << buffer;
54:    }
55: }
56:
```

```
 57:
 58: void usage() {
 59:    cerr << "Usage: " << log.execname() << " [host] [port]" << endl;
 60:    throw cix_exit();
 61: }
 62:
 63: int main (int argc, char** argv) {
 64:    log.execname (basename (argv[0]));
 65:    log << "starting" << endl;
 66:    vector<string> args (&argv[1], &argv[argc]);
 67:    if (args.size() > 2) usage();
 68:    string host = get_cix_server_host (args, 0);
 69:    in_port_t port = get_cix_server_port (args, 1);
 70:    log << to_string (hostinfo()) << endl;
 71:    try {
 72:       log << "connecting to " << host << " port " << port << endl;
 73:       client_socket server (host, port);
 74:       log << "connected to " << to_string (server) << endl;
 75:       for (;;) {
 76:          string line;
 77:          getline (cin, line);
 78:          if (cin.eof()) throw cix_exit();
 79:          log << "command " << line << endl;
 80:          const auto& itor = command_map.find (line);
 81:          cix_command cmd = itor == command_map.end()
 82:                          ? CIX_ERROR : itor->second;
 83:          switch (cmd) {
 84:             case CIX_EXIT:
 85:                throw cix_exit();
 86:                break;
 87:             case CIX_HELP:
 88:                cix_help();
 89:                break;
 90:             case CIX_LS:
 91:                cix_ls (server);
 92:                break;
 93:             default:
 94:                log << line << ": invalid command" << endl;
 95:                break;
 96:          }
 97:       }
 98:    }catch (socket_error& error) {
 99:       log << error.what() << endl;
100:    }catch (cix_exit& error) {
101:       log << "caught cix_exit" << endl;
102:    }
103:    log << "finishing" << endl;
104:    return 0;
105: }
106:
```

```
 1: // $Id: cixd.cpp,v 1.4 2015-06-05 14:25:38-07 - - $
 2:
 3: #include <iostream>
 4: #include <string>
 5: #include <vector>
 6: using namespace std;
 7:
 8: #include <libgen.h>
 9: #include <sys/types.h>
10: #include <unistd.h>
11:
12: #include "protocol.h"
13: #include "logstream.h"
14: #include "sockets.h"
15:
16: logstream log (cout);
17: struct cix_exit: public exception {};
18:
19: void reply_ls (accepted_socket& client_sock, cix_header& header) {
20:    FILE* ls_pipe = popen ("ls -l", "r");
21:    if (ls_pipe == NULL) {
22:       log << "ls -l: popen failed: " << strerror (errno) << endl;
23:       header.command = CIX_NAK;
24:       header.nbytes = errno;
25:       send_packet (client_sock, &header, sizeof header);
26:    }
27:    string ls_output;
28:    char buffer[0x1000];
29:    for (;;) {
30:       char* rc = fgets (buffer, sizeof buffer, ls_pipe);
31:       if (rc == nullptr) break;
32:       ls_output.append (buffer);
33:    }
34:    pclose (ls_pipe);
35:    header.command = CIX_LSOUT;
36:    header.nbytes = ls_output.size();
37:    memset (header.filename, 0, FILENAME_SIZE);
38:    log << "sending header " << header << endl;
39:    send_packet (client_sock, &header, sizeof header);
40:    send_packet (client_sock, ls_output.c_str(), ls_output.size());
41:    log << "sent " << ls_output.size() << " bytes" << endl;
42: }
43:
```

```
44:
45: void run_server (accepted_socket& client_sock) {
46:    log.execname (log.execname() + "-server");
47:    log << "connected to " << to_string (client_sock) << endl;
48:    try {
49:       for (;;) {
50:          cix_header header;
51:          recv_packet (client_sock, &header, sizeof header);
52:          log << "received header " << header << endl;
53:          switch (header.command) {
54:             case CIX_LS:
55:                reply_ls (client_sock, header);
56:                break;
57:             default:
58:                log << "invalid header from client" << endl;
59:                log << "cix_nbytes = " << header.nbytes << endl;
60:                log << "cix_command = " << header.command << endl;
61:                log << "cix_filename = " << header.filename << endl;
62:                break;
63:          }
64:       }
65:    }catch (socket_error& error) {
66:       log << error.what() << endl;
67:    }catch (cix_exit& error) {
68:       log << "caught cix_exit" << endl;
69:    }
70:    log << "finishing" << endl;
71:    throw cix_exit();
72: }
73:
74: void fork_cixserver (server_socket& server, accepted_socket& accept) {
75:    pid_t pid = fork();
76:    if (pid == 0) { // child
77:       server.close();
78:       run_server (accept);
79:       throw cix_exit();
80:    }else {
81:       accept.close();
82:       if (pid < 0) {
83:          log << "fork failed: " << strerror (errno) << endl;
84:       }else {
85:          log << "forked cixserver pid " << pid << endl;
86:       }
87:    }
88: }
89:
```

```
 90:
 91: void reap_zombies() {
 92:    for (;;) {
 93:       int status;
 94:       pid_t child = waitpid (-1, &status, WNOHANG);
 95:       if (child <= 0) break;
 96:       log << "child " << child
 97:            << " exit " << (status >> 8)
 98:            << " signal " << (status & 0x7F)
 99:            << " core " << (status >> 7 & 1) << endl;
100:    }
101: }
102:
103: void signal_handler (int signal) {
104:    log << "signal_handler: caught " << strsignal (signal) << endl;
105:    reap_zombies();
106: }
107:
108: void signal_action (int signal, void (*handler) (int)) {
109:    struct sigaction action;
110:    action.sa_handler = handler;
111:    sigfillset (&action.sa_mask);
112:    action.sa_flags = 0;
113:    int rc = sigaction (signal, &action, nullptr);
114:    if (rc < 0) log << "sigaction " << strsignal (signal) << " failed: "
115:                    << strerror (errno) << endl;
116: }
117:
```

```
118:
119: int main (int argc, char** argv) {
120:    log.execname (basename (argv[0]));
121:    log << "starting" << endl;
122:    vector<string> args (&argv[1], &argv[argc]);
123:    signal_action (SIGCHLD, signal_handler);
124:    in_port_t port = get_cix_server_port (args, 0);
125:    try {
126:       server_socket listener (port);
127:       for (;;) {
128:          log << to_string (hostinfo()) << " accepting port "
129:              << to_string (port) << endl;
130:          accepted_socket client_sock;
131:          for (;;) {
132:             try {
133:                listener.accept (client_sock);
134:                break;
135:             }catch (socket_sys_error& error) {
136:                switch (error.sys_errno) {
137:                   case EINTR:
138:                      log << "listener.accept caught "
139:                          << strerror (EINTR) << endl;
140:                      break;
141:                   default:
142:                      throw;
143:                }
144:             }
145:          }
146:          log << "accepted " << to_string (client_sock) << endl;
147:          try {
148:             fork_cixserver (listener, client_sock);
149:             reap_zombies();
150:          }catch (socket_error& error) {
151:             log << error.what() << endl;
152:          }
153:       }
154:    }catch (socket_error& error) {
155:       log << error.what() << endl;
156:    }catch (cix_exit& error) {
157:       log << "caught cix_exit" << endl;
158:    }
159:    log << "finishing" << endl;
160:    return 0;
161: }
162:
```

```
 1: # $Id: Makefile,v 1.1 2015-05-12 18:48:40-07 - - $
 2:
 3: GPP      = g++ -g -O0 -Wall -Wextra -std=gnu++11
 4:
 5: DEPFILE  = Makefile.dep
 6: HEADERS  = sockets.h protocol.h logstream.h
 7: CPPLIBS  = sockets.cpp protocol.cpp
 8: CPPSRCS  = ${CPPLIBS} cix.cpp cixd.cpp
 9: LIBOBJS  = ${CPPLIBS:.cpp=.o}
10: CIXOBJS  = cix.o ${LIBOBJS}
11: CIXDOBJS = cixd.o ${LIBOBJS}
12: OBJECTS  = ${CIXOBJS} ${CIXDOBJS}
13: EXECBINS = cix cixd
14: LISTING  = Listing.ps
15: SOURCES  = ${HEADERS} ${CPPSRCS} Makefile
16:
17: all: ${DEPFILE} ${EXECBINS}
18:
19: cix: ${CIXOBJS}
20:         ${GPP} -o $@ ${CIXOBJS}
21:
22: cixd: ${CIXDOBJS}
23:         ${GPP} -o $@ ${CIXDOBJS}
24:
25: %.o: %.cpp
26:         ${GPP} -c $<
27:
28: ci:
29:         - checksource ${SOURCES}
30:         - cid + ${SOURCES}
31:
32: lis: all ${SOURCES} ${DEPFILE}
33:         mkpspdf ${LISTING} ${SOURCES} ${DEPFILE}
34:
35: clean:
36:         - rm ${LISTING} ${LISTING:.ps=.pdf} ${OBJECTS} Makefile.dep
37:
38: spotless: clean
39:         - rm ${EXECBINS}
40:
41: dep:
42:         - rm ${DEPFILE}
43:         make --no-print-directory ${DEPFILE}
44:
45: ${DEPFILE}:
46:         ${GPP} -MM ${CPPSRCS} >${DEPFILE}
47:
48: again: ${SOURCES}
49:         make --no-print-directory spotless ci all lis
50:
51: include ${DEPFILE}
52:
```