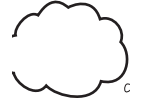
**11** sockets and networking

\***There’sno** **place** +

 **like** [**127.0.0.1**](127.0.0.1)+

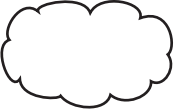




Servers-R-Us, how can I

help you?





A new client, darling? I always knew your BLABing would come in useful one day.

**Programs on different machines need to talk to each other.**

You’ve learned how to use I/O to communicate with files and how processes on the same machine can communicate with each other. Now you’re going to *reach out to the rest*

*of the world*, and learn how to write C programs that can talk to other programs **across the network** and **across the *world***. By the end of this chapter, you’ll be able to create **programs that behave as servers** and **programs that behave as clients**.

**this is a new chapter 467**

***knock-knock server***

Use <127.0.0.1> if you’re running the server on the same machine.

The Internet knock-knock server

C is used to write most of the low-level networking code on the

Internet. Most networked applications need two separate programs: a **server** and a **client**.

You’re going to build a server in C that tells jokes over the Internet. You’ll be able to start the server on one machine like this:

|  |
| --- |
| File Edit Window Help KnockKnock |
| **> ./ikkp\_server**  **Waiting for connection** |

Other than telling you it’s running, the server won’t display anything else on the screen. However, if you open a second console, you’ll be able to connect to the server using a client program called **telnet**. Telnet takes two parameters: the *address* of the server, and the *port*

the server is running on. If you are running telnet on the same machine as the server, you can use [**127.0.0.1**](127.0.0.1)for the address:

The server has responded.  You type in these responses. 尽

\*Do this!



|  |
| --- |
| **You’ll** **be** **using** **telnet** **quite** **a**  **lot** **in** **this**  **chapter** **to** **test** **our** **server** **code.**  *If you try to use the built-in*  *Windows telnet, you might have problems because of the way it communicates with the network. If you install the Cygwin version of telnet, you should be fine.* |

30000 is the number of the network port.

|  |
| --- |
| File Edit Window Help Who’sThere? |
| **> telnet** [**127.0.0.1**](127.0.0.1) **30000**  **Trying 127.0.0.1...**  **Connected to localhost.**  **Escape character is '^]'.**  **Internet Knock-Knock Protocol Server**  **Version 1.0**  **Knock! Knock! > Who's there? Oscar**  **> Oscar who?**  **Oscar silly question, you get a silly answer Connection closed by foreign host.**  **>** |

|  |  |
| --- | --- |
| You will need a **telnet** program in order to connect to the server. Most systems come with telnet already installed. You can  check that you have telnet by typing:  **telnet**  on the command line.  If you *don’t* have telnet, you can install it in one of these ways: | **Cygwin:**  Run the setup.exe program for Cygwin and search for *telnet*.  **Linux:**  Search for *telnet* in your package manager. On many systems, the package manager is called **Synaptic**.  **Mac:**  If you don’t have telnet, you can install it  from *www.macports.org* or *www.finkproject.org*. |

**468 *Chapter 11***

***sockets and networking***

Knock-knock server overview

The server will be able to talk to several clients at once. The client and the server will have a *structured conversation* called a **protocol**. There are different protocols used on the Internet. Some of

Aprotocol is

a structured conversation.

them are *low-level* protocols, like the *internet protocol* (IP), which

are used to control how binary 1s and 0s are sent around the

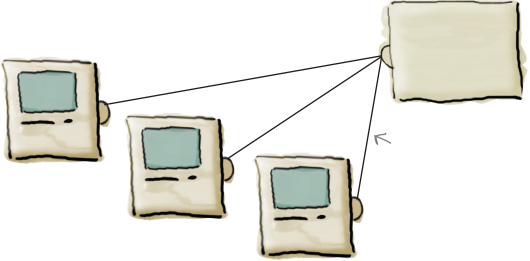
Internet. Other protocols are *high-level* protocols, like the *hypertext*

*transfer protocol* (HTTP), which controlshow web browsers talk to web servers. The joke server is going to use a custom high-level protocol called the *Internet knock-knock protocol* (IKKP).

**Server**

**Telnet client**

A client and server have a structured conversation called a protocol.

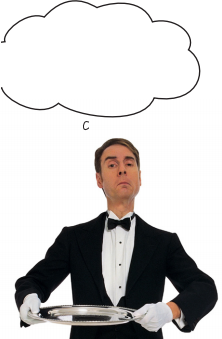
**Telnet client**

个

The server will talk to

several clients at once.

**Telnet client**

The client and the server will exchange messages like this:

Protocol demands that you reply with “Who’s there?” I shall therefore terminate this conversation forthwith.

|  |  |
| --- | --- |
| **Server: Client:** | |
| Knock knock! | |
| |  | | --- | | Oscar. | | Who’s there? |
| Oscar who? |
| Oscar silly question, you get a silly answer. | |

A **protocol** always has a strict set of rules. As long as the client and the server both follow those rules, everything is fine. But if one of

them breaks the rules, the conversation usually stops pretty abruptly.

**469**

***you here***

***blab***

BLAB: how servers talk to the Internet

When C programs need to talk to the outside world, they use **data streams** to read and write bytes. You’ve used data

streams that are connected to the files or Standard Input and Output. But if you’re going to write a program to talk to the network, you need a new kind of data stream called a ***socket***.

**#include <sys/socket.h>** You’ll need this header.

This is a

protocol number. You can leave it as 0.

...

**int listener\_d = socket(PF\_INET, SOCK\_STREAM, 0);**

ecie.descriptor

It’s an Internet socket. if (listener\_d == -1)

Bind to a port. Listen.

error("Can't open socket"); 

Before a server can use a socket to talk to a client program, This is the

it needs to go through four stages that you can remember error() function

with the acronym **BLAB**: **Bind, Listen, Accept, Begin**.

you created in the last chapter.

1. Bind to a port

Accept a connection. Begin talking.

Web: port 80.

Email: port 25.

Chat: port 5222.

Jokes: port 30000.

A computer might need to run several server programs at once. It might be sending out web pages, posting email, and running a chat server all at the same time. To prevent the different conversations from getting confused, each server uses a different **port**. A port is just like a channel on a TV. Different ports are used for different network services, just like different channels are used for different content.

When a server starts up, it needs to tell the operating

system which port it’s going to use. This is called ***binding the port***. The knock-knock server is going to use port

30000, and to bind it you’ll need two things: the **socket descriptor** and a **socket name**. A socket name is just a struct that means “Internet port 30000.”

**#include <arpa/inet.h>**

 You’ll need this header for creating Internet addresses.

**struct sockaddr\_in name;**

...

These lines create a name for the

port meaning “Internet port 30000.”

**name.sin\_family = PF\_INET;**

**name.sin\_port = (in\_port\_t)htons(30000); name.sin\_addr.s\_addr = htonl(INADDR\_ANY);**

**int c = bind (listener\_d, (struct sockaddr \*) &name, sizeof(name));**

if (c == -1)

error("Can't bind to socket");

**470 *Chapter 11***

***sockets and networking***

2. Listen

If your server becomes popular, you’ll probably get lots of clients connecting to it at once. Would you like the clients to wait in a

queue for a connection? The listen() system call tells the

operating system how long you want the queue to be: You’ll use a queue with a length of 10.

**if (listen(listener\_d, 10) == -1) error("Can't listen");**

Calling listen() with a queue length of 10 means that up to 10 clients can try to connect to the server at once. They won’t all be immediately answered, but they’ll be able to wait. The 11th client will be told the server is too busy.

3. Accept a connection

The first 10 clients will be able to wait.

The 11th and 12th

will be told the

server is too busy.

Once you’ve bound a port and set up a listen queue, you then

just have to…wait. Servers spend most of their lives waiting for clients to contact them. The accept() system call waits until a client contacts the server, and then it returns a **second socket**

**descriptor** that you can use to hold a conversation on.

client\_addr will store details about

**struct sockaddr\_storage client\_addr;** the client who’s just connected. **unsigned int address\_size = sizeof(client\_addr);**

**int connect\_d = accept(listener\_d, (struct sockaddr \*)&client\_addr, &address\_size); if (connect\_d == -1)**

**error("Can't open secondary socket");**



This new **connection descriptor** (connect\_d) is the one that the server will use to …

**Begin talking.**

|  |
| --- |
| Why do you think the accept() system  call creates the descriptor for a new  socket? Why don’t servers just use the socket they created to listen to the port? |

***you are here*** **471**

***send()***

A socket’s not your typical data stream

So far, data streams have all been the same. Whether you’re

connected to files or Standard Input/Output, you’ve been able to use functions like fprintf() and fscanf() to talk to

them. But sockets area little different. A socket is *two way*: it

can be used for input *and* output. That means it needs different functions to talk to it.

If you want to output data on a socket, you can’t use

fprintf(). Instead, you use a function called **send()**:

This is the message you’re going to send over the network.



**char \*msg = "Internet Knock-Knock Protocol Server\r\nVersion 1.0\r\nKnock! Knock!\r\n> "; if (send(connect\_d, msg, strlen(msg), 0) == -1)**

**error("send");** ithe is gssage' iial fs r advanced

descriptor.

**Remember**: it’s important to always check the return

value of system calls like send(). Network errors are really common, and your servers will have to cope with them.



Geek

|  |
| --- |
| What port should I use?  You need to be careful when you choose a port number for a server application. There are lots of different servers available, and you need to make sure you don’t use a port number that’s normally used for some other program. On Cygwin and most Unix-style machines, you’ll find a file called ***/etc/services*** that lists the ports used by most of the common servers. When you choose a port, make sure there isn’t another application that already uses the same one.  Port numbers can be between 0 and 65535, and you need to decide whether you want to  use a low number (< 1024) or a high one. Port numbers that are lower than 1024 are usually only available to the superuser or administrator on most systems. This is because the low port numbers are reserved for well-known services, like web servers and email servers. Operating systems restrict these ports to administrators only, to prevent ordinary users from starting  unwanted services.  **Most of the time, you’ll probably want to use a port number greater than 1024.** |

Bits

**472 *Chapter 11***

***sockets and networking***

|  |  |  |
| --- | --- | --- |
| The includes are removed  to save space.  int main(int argc, char \*argv[]) {  char \*advice[] = {  "Take smaller bites\r\n",  "Go for the tight jeans. No | | This server generates random advice for any client that connects to it, but it’s not quite complete. You need to fill in the missing system calls. Also, this version of the code will send back a single piece of advice  and then end. Part of the code needs to be inside a loop. Which part?  they do NOT make you look fat.\r\n", |
| "One word: inappropriate\r\n",  "Just for today, be honest. Tell your boss what you \*really\* think\r\n", "You might want to rethink that haircut\r\n"  };  int listener\_d = (PF\_INET, SOCK\_STREAM, 0);  struct sockaddr\_in name;  name.sin\_family = PF\_INET;  name.sin\_port = (in\_port\_t)htons(30000); name.sin\_addr.s\_addr = htonl(INADDR\_ANY); | | |
|  | (listener\_d, (struct sockaddr \*) &name, sizeof(name)); (listener\_d, 10); | |
| puts("Waiting for connection");  struct sockaddr\_storage client\_addr;  unsigned int address\_size = sizeof(client\_addr);  int connect\_d = (listener\_d, (struct sockaddr \*)&client\_addr, &address\_size); char \*msg = advice[rand() % 5];  (connect\_d, msg, strlen(msg), 0); close(connect\_d);  return 0; }  And for a bonus point, if you add in the missing #include statements, the program will work. But what has the programmer missed out? **Hint: look at the system calls.**  The programmer has forgotten to | | |

***you are here*** **473**

***code written***

|  |
| --- |
| This server generates random advice for any client that connects to it,  but it’s not quite complete. You needed to fill in the missing system calls. Also, this version of the code will send back a single piece of advice and then end. Part of the code needs to be inside a loop. Which part?  int main(int argc, char \*argv[]) {  char \*advice[] = {  "Take smaller bites\r\n",  "Go for the tight jeans. No they do NOT make you look fat.\r\n", "One word: inappropriate\r\n",  "Just for today, be honest. Tell your boss what you \*really\* think\r\n", "You might want to rethink that haircut\r\n"  };  int listener\_d = · socket · (PF\_INET, SOCK\_STREAM, 0);  Create a socket.  struct sockaddr\_in name;  name.sin\_family = PF\_INET;  name.sin\_port = (in\_port\_t)htons(30000); Bind the socket to port 30000.  name.sin\_addr.s\_addr = htonl(INADDR\_ANY);  · bind · (listener\_d, (struct sockaddr \*) &name, sizeof(name));  · listen · (listener\_d, 10);  Set to the listen queue depth to 10. puts("Waiting for connection");  while (1) {  You need to loop the accept/begin talking section.  struct sockaddr\_storage client\_addr;  unsigned int address\_size = sizeof(client\_addr);  int connect\_d = · accept · (listener\_d, (struct sockaddr \*)&client\_addr, &address\_size); char \*msg = advice[rand() % 5];  Accept a connection from a client.  · send · (connect\_d, msg, strlen(msg), 0); } close(connect\_d);  Begin talking to the client. return 0;  }  And for a bonus point, if you add in the missing #include statements, the program will work. But what has  sr**t: l**   **calls.** eo,u if1.socket, bind, listen, |

**474 *Chapter 11***

***sockets and networking***

 ~~Test Drive~~

Let’s compile the advice server and see what happens.

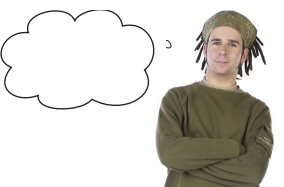
|  |
| --- |
| File Edit Window Help I’mTheServer |
| **> gcc advice\_server.c -o advice\_server > ./advice\_server**  **Waiting for connection** |

Then, while the server is still running, open a second console and connect to the server using telnet a couple of times.

|  |
| --- |
| File Edit Window Help I’mTelnet |
| **> telnet** [**127.0.0.1**](127.0.0.1) **30000**  **Trying 127.0.0.1...**  **Connected to localhost.**  **Escape character is '^]'. One word: inappropriate**  **Connection closed by foreign host. > telnet** [**127.0.0.1**](127.0.0.1) **30000**  **Trying 127.0.0.1...**  **Connected to localhost.**  **Escape character is '^]'.**  **You might want to rethink that haircut Connection closed by foreign host.**  **>** |

That’s great, the server works. Here, you’reusing <127.0.0.1> as the IP address, because the client is running on the same machine as the

server. But you could have connected to the server from anywhere on the network and we’d have gotten the same response.

Working, you

say? Hmm…I think there might be a problem …

***you are here*** **475**

***starting problems***

Sometimes the server doesn’tstart properly

Server console

Client console

|  |
| --- |
| File Edit Window Help I’mTheClient |
| **> telnet** [**127.0.0.1**](127.0.0.1) **30000**  **Trying 127.0.0.1...**  **Connected to localhost.**  **Escape character is '^]'. One word: inappropriate**  **Connection closed by foreign host. >** |

If I start the server, then run the client

one time, it works …

|  |
| --- |
| File Edit Window Help I’mTheServer |
| **> ./advice\_server**  **Waiting for connection** |

The server’s started.

server sends back a response.

The

Server console



…but then, if I stop the server and restart it real quick, the client can’t get a response anymore!

Hitting

Ctrl-C kills the server.

>

Client console

restarted.

The server’s

|  |  |  |
| --- | --- | --- |
| File Edit Window Help I’mTheClient   |  | | --- | | File Edit Window Help I’mTheServer | | **> ./advice\_server**  **Waiting for connection ^C**  **> ./advice\_server**  **Waiting for connection** | |
| **> telnet** [**127.0.0.1**](127.0.0.1) **30000**  **Trying 127.0.0.1...**  **telnet: connect to address 127.0.0.1: Connection refused telnet: Unable to connect to remote host**  **>** |

The server *looks* like it’s starting correctly the second time, but the client can’t get any response from it. Why is that?

Remember that the code was written **without any error checking**. Let’sadd a little error check into the code and see if we can figure out what’shappening.

WTF??!?!??

Where’s The Feedback????

**476 *Chapter 11***

***sockets and networking***

Why your mom always told you to check for errors

|  |  |
| --- | --- |
| If you add an error check on the line that binds the socket to a port:  bind (listener\_d, (struct sockaddr \*) | From this…  &name, sizeof (name)); |

…to this



if (bind (listener\_d, (struct sockaddr \*) &name, sizeof(name)) == -1)

error("Can't bind the port");  This is calling the error function you wrote a while

back. It will display the cause of the error and exit.

Then you’ll get a little more information from the server if it is stopped and restarted quickly:

|  |
| --- |
| File Edit Window Help I’mTheServer |
| **> ./advice\_server**  **Waiting for connection ^C**  **> ./advice\_server**  **Can't bind the port: Address already in use >** |

The bind fails!

If the server has responded to a client and then gets stopped and restarted, the call to the bind system call fails. But because the

original version of the program never checked for errors, the rest of the server code ran eventhough it couldn’t use the server port.

Bound ports are sticky

When you binda socket to a port, the operating system will

prevent anything else from rebinding to it for the next 30

seconds or so, and that includes the program that bound the port in the first place. To get around the problem, you just need to set an option on the socket before you bind it:

You need an int variable to store the option.

 Setting it to 1 means “Yes, reuse the port.”

int reuse = 1;

ALWAYS check for

errors on system calls.

if (**setsockopt(listener\_d, SOL\_SOCKET, SO\_REUSEADDR,** error("Can't set the reuse option on the socket");

This code makes the socket **reuse the port** when it’sbound.

That means you can stop and restart the server and there will be no errors when you bind the port a second time.

**(char \*)&reuse, sizeof(int))** == -1)

个

This makes the socket reuse the port.

***you are here*** **477**

***recv()***

Reading from the client

You’ve learned how to send data to the client, but what about *reading* from the client? In the same way that sockets have a special send() function to write data, they also have a **recv()** function to read data.

**<bytes read> = recv(<descriptor>, <buffer>, <bytes to read>, 0);**

If someone types in a line of text into a client and hits return, therecv() function stores the text into a character array like this:

recv() will return the value 14, because there are 14 characters sent from the client.

|  |
| --- |
| r |

|  |
| --- |
| h |

|  |
| --- |
| t |

|  |
| --- |
| e |

|  |
| --- |
| e |

|  |
| --- |
| ? |

|  |
| --- |
| \r |

|  |
| --- |
| \n |

|  |
| --- |
| s |

|  |
| --- |
| h |

|  |
| --- |
| o |

|  |
| --- |
| ' |

|  |
| --- |
| W |

There area few things to remember:

**The characters are not terminated with a \0 character.**

¥

**When someone types text in telnet, the string always ends \r\n.**

¥

**The recv() will return the number of characters, or –1 if there’s an error, or**

¥

**0 if the client has closed the connection.**

**You’re not guaranteed to receive all the characters in a single call to recv().**

¥

This last point is important. It means you might have to call recv() more than once:

You might need to call

recv() a few times to

get all the characters.

W h o ' s  t h e r e ? \r \n 

That means recv() can be tricky to use. It’s best to wrap recv() in a function that stores a simple \0-terminated string in the array it’sgiven.

Something like this:

 This reads all the characters

int read\_in(int socket, char \*buf, int len) {

until it reaches ‘\n’.

char \*s = buf; int slen = len;

int c = recv(socket, s, slen, 0); z一 while ((c > 0) && (s[c-1] != '\n')) {

s += c; slen -= c;

c = recv(socket, s, slen, 0); }

if (c < 0) In case there’s an error

return c;

else if (c == 0) Nothing read; send

buf[0] = '\0';  back an empty string. else

Replace the ‘\r’

s[c-1]='\0';  return len - slen;

character with a ‘\0’ .

}

**478 *Chapter 11***

Keep reading until there are no more characters or you reach ‘\n’.

Piste

Go Off

|  |
| --- |
| This is one way of  simplifying recv(), but could *you* do  better? Why not write your own version of  read\_in() and let us know at *headfirstlabs.com*. |

***sockets and networking***



Here are some other functions that are useful when you are

Ready-Bake Code

writing a server. Do you understand how each of them works?

You’ve used this error function

a LOT in this book.

Don’t call this function if you

want the program to keep running.

strerror(errno));

int open\_listener\_socket() {

int s = socket(PF\_INET, SOCK\_STREAM, 0); if (s == -1)

error("Can't open socket");

|  |
| --- |
| Display the error…  void error(char \*msg) {  fprintf(stderr, "%s: %s\n", msg, exit(1);  }  …then stop the program. |

Create an Internet streaming socket. 

Yes, reuse the socket (so you can

restart the server without problems).

return s; }

|  |
| --- |
| void bind\_to\_port(int socket, int port) {  struct sockaddr\_in name; name is Internet port 30000.  name.sin\_family = PF\_INET;  name.sin\_port = (in\_port\_t)htons(30000); name.sin\_addr.s\_addr = htonl(INADDR\_ANY); int reuse = 1;  if (setsockopt(socket, SOL\_SOCKET, SO\_REUSEADDR, (char \*)&reuse, sizeof(int)) == -1)  error("Can't set the reuse option on the socket");  int c = bind (socket, (struct sockaddr \*) &name, sizeof(name)); if (c == -1)  error("Can't bind to socket"); } |

Grab port 30000.

|  |
| --- |
| { int result = send(socket, s, strlen(s), 0); ’oa’ rt( iee’ssrlm.  int say(int socket, char \*s) Send a string to a client.  if (result == -1)  there’s just a problem with one client.  fprintf(stderr, "%s: %s\n", "Error talking to the client", strerror(errno)); return result;  } |

**Now** **that** **you** **have** **a** **set** **of** **server** **functions,** **let’s** **try** **them** **out…**

***you are here*** **479**

int listener\_d;

void handle\_shutdown(int sig) {

if (listener\_d)

close(listener\_d);

fprintf(stderr, "Bye!\n"); exit(0);

}

|  |  |
| --- | --- |
| This will store the  main listener  If someone hits Ctrl-C when the server  is running, this function will close the  socket before the program ends.  socket for  the server. | Now it’s time to write the code for the **Internet knock-knock server**. You’re going to write a little more code than usual, but you’ll be able to use the ready-bake code from the previous page. Here’s the start of the program.  The ready-bake functions from the previous page go here.  #include <stdio.h>  #include <string.h>  #include <errno.h>  #include <stdlib.h>  #include <sys/socket.h> #include <arpa/inet.h> #include <unistd.h>  #include <signal.h> |

**480 *Chapter 11***

***sockets and networking***

|  |
| --- |
| Now it’s over to you to write the main function. You’ll need to create a new server socket and store it in listener\_d. The socket will be bound to port 30000, and the queue depth should be set to 10. Once that’s done, you need to write code that works like this: |
| ↓ ↓  Get connection from client  ↓  Say, “Knock! Knock!”    Check that they say, “Who’sthere?”    Say, “Oscar”  Check that they say, “Oscar who?”  Say, “oscar silly question, you get a silly answer” |
| Try to check error codes and if the user says the wrong thing, just send an error message, close the connection, and then wait for another client.  **Goodluck!** |

***you are here*** **481**

int listener\_d;

void handle\_shutdown(int sig) {

if (listener\_d)

close(listener\_d);

fprintf(stderr, "Bye!\n"); exit(0);

}

|  |  |
| --- | --- |
| This will store the  main listener  If someone hits Ctrl-C when the server  is running, this function will close the  socket before the program ends.  socket for  the server. | Now it’s time to write the code for the **Internet knock-knock server**. You were to write a little more code than usual, but you’ll be able to use the ready-bake code from the previous page. Here’s the start of the program.  The ready-bake functions from the previous page go here.  #include <stdio.h>  #include <string.h>  #include <errno.h>  #include <stdlib.h>  #include <sys/socket.h> #include <arpa/inet.h> #include <unistd.h>  #include <signal.h> |

***sockets and networking***

|  |
| --- |
| Listen for a connection.  〓〓 -I)  This is the kind of code you should have written. Is yours similar? It doesn’t matter if the code is *exactly* the same. The important thing is that your code can tell the joke in the right way, and cope with errors.  int main(int argc) char 米argvcJ)  {  if (catch\_signaI(SI6INT) handIe\_shutdown) error(“Can)t set the interrupt handIern);  This wiII caII handIe\_shutdown() if CtrI-C is hit.  Iistener\_d 〓 open\_Iistener\_socket(); bind to port(Iistener\_d) 30000);  Create a socket on port 30000. Set the Iisten-queue Iength to I0.  if (Iisten(Iistener\_d) I0) 〓〓 -I)  error(“Can)t Iistenn);  struct sockaddr\_storage cIient\_addr;  unsigned int address\_si乙e 〓 si乙eof(cIient\_addr);  puts(“Waiting for connectionn);  char bufc255J;  whiIe (I) {  int connect\_d 〓 accept(Iistener\_d) (struct sockaddr 米)钅cIient\_addr) 钅address\_si乙e); if (connect\_d 〓〓 -I)  error(“Can)t open secondar丫 socketn); Send data to the cIient.  if (sa丫(connect\_d)  “Internet Knock-Knock ProtocoI Server\r\nVersion I.0\r\nKnock! Knock!\r\n>“) !〓 -I) {  read\_in(connect\_d) buf) si乙eof(buf)); <— Read data from the cIient.  if (strncasecmp(“Who)s there?n) buf) I2))  sa丫(connect\_d)“You shouId sa丫 (Who)s there?)!n);  Checking the user)s answers. eIse {  if (sa丫(connect\_d)“Oscar\r\n>“) !〓 -I) { read\_in(connect\_d) buf) si乙eof(buf));  if (strncasecmp(“Oscar who?n) buf) I0))  sa丫(connect\_d)“You shouId sa丫 (Oscar who?)!\r\nn);  eIse  sa丫(connect\_d)“Oscar siII丫 question) 丫ou get a siII丫 answer\r\nn);  } }  c}Iose(connect\_d);  CIose the secondar丫 socket we used for the conversation.  }  return 0; } |

***you are here*** **483**

***test drive***

 ~~Test Drive~~

Now that you’ve written the time to compile it and fire it

The server’s waiting for a connection, so open a separate console and connect to it with telnet:

Client console

knock-knock server, it’s

up.

Server console

|  |
| --- |
| File Edit Window Help I’mTheServer |
| **> gcc ikkp\_server.c -o ikkp\_server > ./ikkp\_server**  **Waiting for connection** |

The server can tell

you ajoke, but what happens if you break the protocol and

send back an invalid response?

Client console 

The server is able to validate the data you send it and close the connection immediately. Once you’redone running the server, you can switchback to the server window and hit Ctrl-C to close it down neatly. It even sends you a farewell message:

Server console 

|  |
| --- |
| File Edit Window Help I’mTheClient |
| **> telnet** [**127.0.0.1**](127.0.0.1) **30000**  **Trying 127.0.0.1...**  **Connected to localhost.**  **Escape character is '^]'.**  **Internet Knock-Knock Protocol Server**  **Version 1.0**  **Knock! Knock! > Who's there? Oscar**  **> Oscar who?**  **Oscar silly question, you get a silly answer Connection closed by foreign host.** |

|  |
| --- |
| File Edit Window Help I’mTheClient |
| **> telnet** [**127.0.0.1**](127.0.0.1) **30000**  **Trying 127.0.0.1...**  **Connected to localhost.**  **Escape character is '^]'.**  **Internet Knock-Knock Protocol Server**  **Version 1.0**  **Knock! Knock! > Come in**  **You should say 'Who's there?'!Connection closed by foreign host. >** |

|  |
| --- |
| File Edit Window Help I’mTheServer |
| **> gcc ikkp\_server.c -o ikkp\_server > ./ikkp\_server**  **Waiting for connection ^CBye!**  **>** |

That’s great! The server does everything you need it to do.

**Or** **does** **it?**

**484 *Chapter 11***

***sockets and networking***

The server can only talk to one person at a time

There’sa problem with the current server code. Imagine

someone connects to it and he is a little slow with his responses:



The server is running

on a machine out on

the Internet.

|  |
| --- |
| File Edit Window Help I’mTheClient |
| **> telnet knockknockster.com 30000 Trying knockknockster.com...**  **Connected to localhost.**  **Escape character is '^]'.**  **Internet Knock-Knock Protocol Server**  **Version 1.0**  **Knock! Knock! > Who's there?**  **Oscar >** |



Oh, wait! Oscar! Oh, I know this one… Oh, it’s so funny … Oscar…Oscar who? Hey,that’s no, wait…don’t tell me …

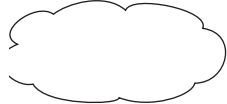
It’s … like …

Then, if someone else tries to get

through to the server, she can’t; it’s busy with the first guy:



|  |
| --- |
| File Edit Window Help I’mAnotherClient |
| **> telnet knockknockster.com 30000 Trying knockknockster.com...**  **Connected to localhost.**  **Escape character is '^]'.** |



Oh, great! I can’t get

through to the server and I can’t even Ctrl-C my way out of telnet. What gives?

The problem is that the server is still busy talking to the first

guy. The main server socket will keep the client waiting until the server calls the accept() system call again. But because of the guy already connected, it will be sometime before that happens.

|  |
| --- |
| The server can’t respond to the second user, because it is busy dealing with the first. What have you learned that might help you deal with *both* clients *at once*? |

***you are here*** **485**

***different clients, different sockets***

You can fork() a process for each client

When the clients connect to the server, they start to have a

conversation on a separate, newly created socket. That means the main server socket is free to go and find another client. So let’s do that.

When a client connects, you can fork() a separate child process to deal with the conversation between the server and the client.



Child 

process



Parent process



Client

Hey, great to see you! I’ll

just hand you over to someone who can deal with you.

While the client is talking to the child process, the server’s parent process can go connect to the next client.



Knock! Knock!

The parent and child use different sockets

One thing to bear in mind is that the parent server process will

only need to use the main listener socket. That’sbecause the

main listener socket is the one that’s used to accept() new

connections. On the other hand, the child process will only ever need to deal with the secondary socket that gets created by the

accept() call. That means once the parent has forked the

child, the parent can close the secondary socket and the child can close the main listener socket.

|  |  |
| --- | --- |
| child, the parent can **close(connect\_d);** close this socket. **close(listener\_d);**  After forking the | Once the child gets  created, it can  k close this socket. |

Who’s there?

there are no

Dumb Questions

**If I create a new process for**

**each client, what happens if hundreds of clients connect? Will my machine create hundreds of processes?**

Yes. If you think your server will get a lot of clients, you need to control how many processes you create. The child

can signal you when it’s finished with a client, and you can use that to maintain a count of current child processes.

Q:

A:

**486 *Chapter 11***

***sockets and networking***

|  |
| --- |
| This is a version of the server code that has been changed to  fork a separate child process to talk to each client…except it’s not quite finished. See if you can figure out the missing pieces of code.  while (1) {  int connect\_d = accept(listener\_d, (struct sockaddr \*)&client\_addr, &address\_size);  if (connect\_d == -1)  error("Can't open secondary socket");  if ( ) {  close( ); if (say(connect\_d,  "Internet Knock-Knock Protocol Server\r\nVersion 1.0\r\nKnock! Knock!\r\n> ") != -1) {  read\_in(connect\_d, buf, sizeof(buf));  if (strncasecmp("Who's there?", buf, 12))  say(connect\_d, "You should say 'Who's there?'!");  else {  if (say(connect\_d, "Oscar\r\n> ") != -1) { read\_in(connect\_d, buf, sizeof(buf));  if (strncasecmp("Oscar who?", buf, 10))  say(connect\_d, "You should say 'Oscar who?'!\r\n");  else  say(connect\_d, "Oscar silly question, you get a silly answer\r\n");  } }  }  close( );  <— What should the child do when the conversation is done?  }  close( ); } |

***you are here*** **487**

***code written***

|  |  |  |  |
| --- | --- | --- | --- |
| This is a version of the server code that has been changed to fork a separate child process to talk to each client—except it’s not quite finished. You were to figure out the missing  pieces of code.  while (1) {  int connect\_d = accept(listener\_d, (struct sockaddr \*)&client\_addr, &address\_size);  if (connect\_d == -1)  error("Can't open secoh;ild process, and you know that if if ( !fork() ) { the fork() call returns 0, you must be in the child. | | | |
| close( · listener\_d ); if (say(connect\_d, | In the child, you need to close the main listener socket. | The child will use only socket to talk to the | the connect\_d client. |
| "Internet Knock-Knock Protocol Server\r\nVersion 1.0\r\nKnock! Knock!\r\n> ") != -1) {  read\_in(connect\_d, buf, sizeof(buf));  if (strncasecmp("Who's there?", buf, 12))  say(connect\_d, "You should say 'Who's there?'!");  else {  if (say(connect\_d, "Oscar\r\n> ") != -1) { read\_in(connect\_d, buf, sizeof(buf));  if (strncasecmp("Oscar who?", buf, 10))  say(connect\_d, "You should say 'Oscar who?'!\r\n");  else  say(connect\_d, "Oscar silly question, you get a silly answer\r\n");  } Once the conversation’s over, the child } can close the socket to the client.  } ↓  close( · connect\_d );  exit(0);  Once the child process has finished talking, it should exit.  } That will prevent it from falling into the main server loop.  close( · connect\_d · );  } | | | |

**488 *Chapter 11***

***sockets and networking***

 ~~Test Drive~~

Let’s try the modified version of the server. You can compile and run it in the same way:

Server console

|  |
| --- |
| File Edit Window Help I’mTheServer |
| **> gcc ikkp\_server.c -o ikkp\_server > ./ikkp\_server**  **Waiting for connection** |

If you open a separate console and start telnet, you can connect, just like you did before:

Client console

Everything seems the same, but if you leave the client running with the joke half-told, you should be able to see what’s changed:

|  |
| --- |
| File Edit Window Help I’mTheClient |
| **> telnet** [**127.0.0.1**](127.0.0.1) **30000**  **Trying 127.0.0.1...**  **Connected to localhost.**  **Escape character is '^]'.**  **Internet Knock-Knock Protocol Server**  **Version 1.0**  **Knock! Knock! > Who's there?**  **Oscar >** |

If you open a third console, you will see that there are now two processes for the server: one for the parent and one for the child:

The ps command shows running processes in Unix and Cygwin.

|  |
| --- |
| File Edit Window Help I’mJustCurious |
| **> ps**  **PID TTY TIME CMD**  **14324 ttys002 0:00.00 ./ikkp\_server**  **14412 ttys002 0:00.00 ./ikkp\_server**  **>** |

The parent process



That means you can connect, even while the first client is still talking to the server:

Another client console

**Now** **that** **you’ve** **built** **an** **Internet** **server,** **let’s** **go** **look** **at** **what** **it** **takes** **to** **build** **a** **client,** **by** **writing** **something** **that** **can**

**read** **from** **the** **Web.**

The child process



|  |
| --- |
| File Edit Window Help I’mAnotherClient |
| **> telnet** [**127.0.0.1**](127.0.0.1) **30000**  **Trying 127.0.0.1...**  **Connected to localhost.**  **Escape character is '^]'.**  **Internet Knock-Knock Protocol Server**  **Version 1.0**  **Knock! Knock! >** |

***you are here*** **489**

***the client***

Writing a web client

What if you want to write your own client program? Is it really *that* different from a server? To see the similarities

and differences, you’re going to write a **web client** for the hypertext transfer protocol (HTTP).

HTTP is a lot like the Internet knock-knock protocol you coded earlier. All protocols are *structured conversations*. Every time a web client and server talk, they say the same kind of things. Open telnet and see how to download

[*http://en.wikipedia.org/wiki/O*](http://en.wikipedia.org/wiki/O)*’Reilly\_Media.*



Do this!



Most web servers run on port 80.

|  |
| --- |
| File Edit Window Help I’mJustCurious |
| **> telnet en.wikipedia.org 80**  **Trying 91.198.174.225...**  **Connected to wikipedia-lb.esams.wikimedia.org.**  **Escape character is '^]'.**  This is the path that follows the hostname in the URL.  **GET /wiki/O'Reilly\_Media http/1.1**  **Host: en.wikipedia.org**  **HTTP/1.0 200 OK Server: Apache**  In HTTP/1.1, you need to say what hostname you are using.  **...**  **<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "**[**http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd**](http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd)**">**  **<html lang=en" dir="ltr" class"client-nojs"**  **xmlns="**[**http://www.w3.org/1999/xhtml**](http://www.w3.org/1999/xhtml)**"> <head>**  **<title>O'Reilly Media - Wikipedia, the free encyclopedia</title> ...** And this is the HTML for the web page. |

When your program connects to the web server, it will need to

send at least three things: 

¥ **A GET command**

Most web clients actually send a lot more information,

but you’ll just send the minimum amount.

GET /wiki/O'Reilly\_Media http/1.1

¥

¥

**The hostname**

Host: en.wikipedia.org

**A blank line**

**But** **before** **you** **can** **send** **any** **data** **at** **all** **to** **the** **server,** **you** **need** **to** **make** **a** **connection** **from**

This is the numeric

address of Wikipedia.

You might get a slightly 

different address when

you try it.

You need to type in these two lines.

刁

And then you need to hit return and leave a blank line.

The server first responds with some extra details about the web page.

**the** **client.** **How** **do** **you** **do** **that?**

***sockets and networking***

Clients are in charge

Clients and servers communicate using sockets, but the way that each gets hold of a socket is a little different. You’ve

already seen that ***servers*** use the BLAB sequence:

**1 Bind a port.**

**2 Listen.**

**3 Accept a conversation.**

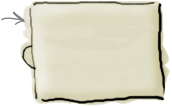
I was taught

**4 Begin talking.**

never to speak

until I’m spoken to.

A server spends most of its life waiting for afresh connection from a client. Until a client connects, a server really can’t

do anything. Clients don’t have that problem. A client can

connect and start talking to a server whenever it likes. This is the sequence for a ***client***:

Server

**1 Connect to a remote port.**

**2 Begin talking.**

Remote ports and IP addresses

When a server connects to the network, it just has to decide

which port it’s going to use. But clients need to know a little

more: they need to know the port of the remote server, but

they also need to know its **internet protocol (IP) address**:

<208.201.239.100> Addresses with four digits are in IP version 4 format. Most will eventually be replaced with longer version 6 addresses.

Internet addresses are kind of hard to remember, which is

why most of the time human beings use **domain names**. A domain name is just an easier-to-remember piece of text like:

*www.oreilly.com*

Eventhough human beings prefer domain names, the actual packets of information that flow across the network only use the numeric IP address.

***you are here*** **491**

***client sockets***

Create a socket for an IP address

Once your client knows the address and port number of the server, it can create a **client socket**. Client sockets and server sockets are created the same way:

**int s = socket(PF\_INET, SOCK\_STREAM, 0);**

The difference between client and server code is what they do

with sockets once they’recreated. A server will **bind** the socket to a *local* port, but a client will **connect** the socket to a *remote port*:

To save space, the error check here. for errors.

examples won’tinclude the

But in your code, always check

**struct sockaddr\_in si;**

**memset(&si, 0, sizeof(si)); si.sin\_family = PF\_INET;**

These lines

create a socket

address for

<208.201.239.100>

on port 80.

**si.sin\_addr.s\_addr = inet\_addr("**[**208.201.239.100**](208.201.239.100)**");**

**si.sin\_port = htons(80);**

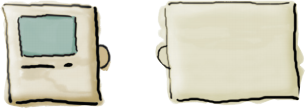
**connect(s, (struct sockaddr \*) &si, sizeof(si));**

This line

connects the socket to the remote port.

Server <208.201.239.100>

Client

7

Port 80



Hello? I don’t want

to know how to connect a socket to an IP address. I’m actually **human**…I

want to connect to a real domain name.

**The** **above** **code** **works** **only** **for** **numeric** **IP** **addresses.**

To connect a socket to a remote domain name, you’ll need afunction called getaddrinfo().

**492 *Chapter 11***

***sockets and networking***

getaddrinfo() gets addresses for domains

The DNS is a gigantic address book.

The *domain name system* is a huge address book. It’saway of converting a domain name like *www.oreilly.com* into the kinds of numeric IP addresses that computers need to address the packets of information they send across the network.

Create a socket for a domain name





|  |  |
| --- | --- |
| Domain name | Address |
| en.wikipedia.org | <91.198.174.225> |
| www.oreilly.com | <208.201.239.100> |
| www.oreilly.com | <208.201.239.101> |

Most of the time, you’ll want your client code to use the DNS

trts. ,syn,oa to Some large sites have Computers need IP

construct your client sockets in a slightly different way: several IP addresses. addresses to create

network packets.

**#include <netdb.h>** ’lodil(ner

...

**struct addrinfo \*res; struct addrinfo hints;**

This creates a **memset(&hints, 0, sizeof(hints));** getaddrinfo() expects

**..\_****p** **\_PR;EAM;** the port to be a string.

www.oreilly.com.  **getaddrinfo("www.oreilly.com", "80", &hints, &res);**

The getaddrinfo() constructs a new data structure on the **heap** called a *naming resource*. The naming resource represents a port on a server with a given domain name. Hidden away

inside the naming resource is the IP address that the computer will need. Sometimes very large domains can have several IP addresses, but the code here will simply pick one of them. You can then use the naming resource to create a socket.

Now you can create the socket using the naming resource.

**int s = socket(res->ai\_family, res->ai\_socktype, res->ai\_protocol);**

Finally, you can connect to the remote socket. Because the

res->ai\_addrlen is the size of the address in memory.

naming resource was created on the heap, you’ll need to tidy it away with a function called **freeaddrinfo()**:

’i**(****,****(**(**e****a**a)**, re****)**i**;**,fyrlneo

functions you used for the server. That means you should have enough information now to write a web client …

***you are here*** **493**

***magnets muddled***



Code Magnets

Here is the code for a web client that will download the contents of a page from Wikipedia and display it on the screen. The web page will be passed as an argument to the program. Think carefully about the data you need to send to a web server running HTTP.

#include <stdio.h>

#include <string.h>

#include <errno.h>

#include <stdlib.h>

#include <sys/socket.h> #include <arpa/inet.h> #include <unistd.h>

#include <netdb.h>

void error(char \*msg) {

fprintf(stderr, "%s: %s\n", msg, strerror(errno)); exit(1);

}

int open\_socket(char \*host, char \*port) {

struct addrinfo \*res; struct addrinfo hints;

memset(&hints, 0, sizeof(hints)); hints.ai\_family = PF\_UNSPEC;

hints.ai\_socktype = SOCK\_STREAM;

if (getaddrinfo(host, port, &hints, &res) == -1) error("Can't resolve the address");

int d\_sock = socket(res->ai\_family, res->ai\_socktype, res->ai\_protocol);

if (d\_sock == -1)

error("Can't open socket");

int c = connect(d\_sock, res->ai\_addr, res->ai\_addrlen);

freeaddrinfo(res); if (c == -1)

error("Can't connect to socket"); return d\_sock;

}

**494 *Chapter 11***

***sockets and networking***

int say(int socket, char \*s) {

int result = send(socket, s, strlen(s), 0); if (result == -1)

fprintf(stderr, "%s: %s\n", "Error talking to the server", strerror(errno));

return result; }

int main(int argc, char \*argv[])

{

int d\_sock;

d\_sock = ; char buf[255];

sprintf(buf, , argv[1]); say(d\_sock, buf);

say(d\_sock, ); char rec[256];

int bytesRcvd = recv(d\_sock, rec, 255, 0);

while (bytesRcvd) {

if (bytesRcvd == -1)

error("Can't read from server");

rec[bytesRcvd] = ; printf("%s", rec);

bytesRcvd = recv(d\_sock, rec, 255, 0); }

; return 0;

}

**'\0'**

**"Host: en.wikipedia.org\r\n\r\n"**

**"\r\n"**

|  |
| --- |
| **"GET /wiki/%s http/1.1\r\n"** |

**close(d\_sock)**

**open\_socket("en.wikipedia.org", "80")**

|  |
| --- |
| **"Host: en.wikipedia.org\r\n"** |

***you are here*** **495**

***magnets unmuddled***



Code Magnets Solution

Here is the code for a web client that will download the contents of a page from Wikipedia and display it on the screen. The web page will be passed as an argument to the program. You were to think carefully about the data you need to send to a web server running HTTP.

#include <stdio.h>

#include <string.h>

#include <errno.h>

#include <stdlib.h>

#include <sys/socket.h> #include <arpa/inet.h> #include <unistd.h>

#include <netdb.h>

void error(char \*msg) {

fprintf(stderr, "%s: %s\n", msg, strerror(errno)); exit(1);

}

int open\_socket(char \*host, char \*port) {

struct addrinfo \*res; struct addrinfo hints;

memset(&hints, 0, sizeof(hints)); hints.ai\_family = PF\_UNSPEC;

hints.ai\_socktype = SOCK\_STREAM;

if (getaddrinfo(host, port, &hints, &res) == -1) error("Can't resolve the address");

int d\_sock = socket(res->ai\_family, res->ai\_socktype, res->ai\_protocol);

if (d\_sock == -1)

error("Can't open socket");

int c = connect(d\_sock, res->ai\_addr, res->ai\_addrlen); freeaddrinfo(res);

if (c == -1)

error("Can't connect to socket"); return d\_sock;

}

***sockets and networking***

int {

say(int socket, char \*s)

int result = send(socket, if (result == -1)

s, strlen(s), 0);

fprintf(stderr, "%s: %s\n", strerror(errno));

talking to

server",

the

"Error

return result; }

main(int

char

\*argv[])

int

argc,

{

int d\_sock;

d\_sock =

Create a string for the path

**open\_socket("en.wikipedia.org", "80")**

;

char buf[255];

to the page you want.



sprintf(buf, ·

|  |
| --- |
| **"GET /wiki/%s http/1.1\r\n"** |

, argv[1]);

say(d\_sock, buf);

This sends the

**"Host:**

**en.wikipedia.org\r\n\r\n"**

say(d\_sock, );

host data as well as a blank line.

char rec[256];

int bytesRcvd = recv(d\_sock, rec, 255, 0); while (bytesRcvd) {

if (bytesRcvd == -1)

to the end of the array of

to make it a proper string.

error("Can't read from server");

Add a ‘\0’

characters

rec[bytesRcvd] = ;

**'\0'**

printf("%s", rec);

bytesRcvd = recv(d\_sock, rec, 255, 0); }

**close(d\_sock)**

;

return 0; }

|  |
| --- |
| **"\r\n"** |

|  |
| --- |
| **"Host: en.wikipedia.org\r\n"** |

***you are here*** **497**

***test drive***

 ~~Test Drive~~

If you compile and run the web client, you make it

|  |  |
| --- | --- |
| download a page from Wikipedia like this: |  |
| You’ll have to replace any spaces with underscore | (\_) characters. |

|  |
| --- |
| **> gcc wiki\_client.c -o wiki\_client**  File Edit Window Help I’mTheWebClient  **> ./wiki\_client "O'Reilly\_Media"**  **HTTP/1.0 200 OK**  At the beginning, you’ll get the response HEADERS. These tell you things about the server and the web page.  **Date: Fri, 06 Jan 2012 20:30:15 GMT**  **Server: Apache**  **...**  **Connection: close**  **<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "**[**http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd**](http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd)**">**  **<html lang="en" dir="ltr" class="client-nojs" xmlns="http://www.w3.org/1999/xhtml"> <head>**  **<title>O'Reilly Media - Wikipedia, the free encyclopedia</title>**  **<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />**  **...**  Then you get the contents of the web page from Wikipedia. |

**It** **works!**

The client took the name of the page from the

command line and then connected to Wikipedia to

download the page. Because it’s constructing the *path* to the file, you need to make sure that the you replace any spaces in the page name with underscore (\_)

characters.



Go Off

|  |
| --- |
| Why not update the code to automatically replace characters like spaces for you? For more details on how to replace characters for web addresses, see:  [*http://www.w3schools.com/tags/re*](http://www.w3schools.com/tags/re)*f\_urlencode.asp* |

Piste

***sockets and networking***

~~b"iesiom~~

Q: **Should I create sockets with IP addresses or domain names?**

Most of the time, you’ll want to use domain names. They’re easier to remember, and

A:

A:

occasionally some servers will change their numeric addresses but keep the same domain names.

Q: **So, do I even need to know how to connect to a numeric address?**

Yes. If the server you are connecting to is not registered in the domain name system,

such as machines on your home network, then you will need to know how to connect by IP. Q: **Can I use getaddrinfo()with a numeric address?**

A:Yes, you can. But if you *know* that the address you are using is a numeric IP, the first version of the client socket code is simpler.

|  |  |
| --- | --- |
| ■ A protocol is a structured conversation.  ■ Servers connect to local ports.  ■ Clients connect to remote ports.  ■ Clients and servers both use sockets to communicate. | ■ You write data to a socket with send().  ■ You read data from a socket with recv().  ■ HTTP is the protocol used on the Web. |

***you are here*** **499**

***c toolbox***

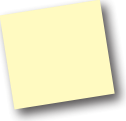
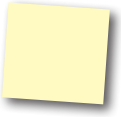
Your C Toolbox

**You’ve** **got** **Chapter** **11** **under** **your** **belt,** **and** **now** **you’ve**

**added** **sockets** **and** **networking**

**to** **your** **toolbox.** **For** **a** **complete** **list**

**of** **tooltips** **in** **the** **book,** **see** **Appendix** **ii.**



Telnet is a

simple network

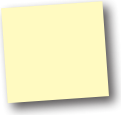
client.

Create

sockets with

the socket()

function.

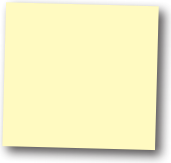


Use fork()

to cope with

several clients

at once.



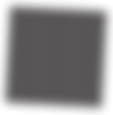
Servers BLAB:

B = bind()

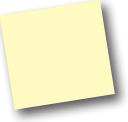
L = listen()

A = accept()

B = Begin talking



DNS =



Domain name

system

getaddrinfo()

finds

addresses by

domain.