CS205

C / C++

Stéphane Faroult
farault@sustc.edu.cn

As we have seen last time, although most layers can be (and usually are) written in C, we'll just consider here network exchanges at the application level.

Where does C network programming hit?

Application
Host-to-Host
Internet
Network Access

For us, here

Based on a SOCKET

To send a message you need a "socket", which is a stream , like a file. When you ask the system for a "socket", the system returns to you an `int` that identifies the socket. You need to pass some information to specify your target address.

Lots of weird structures

If you are really interested in the details, there is a very good guide on network programming on the web at http://beej.us/guide/bgnet/

WARNING

ALWAYS SCARY THE FIRST TIME

The principles

The domain says whether you want to use small (32bit), IP addresses, large ones, or if you don't care (a good choice). The type is about checking that messages arrive (you usually want that) and the protocol should be 0 for IP

1 Get a socket id (int) with the socket() call

```
#include <sys/socket.h>

int socket(int domain,
           int type,
           int protocol);
```

AF_INET
AF_INET6
AF_UNSPEC

SOCK_STREAM
SOCK_DGRAM

0 (IP)

The principles

Some of this information is usually specified in a special structure for network addresses.

1 Useful structure

Set everything else to 0

```
#include <netdb.h>
struct addrinfo {
    int ai_flags;       AI_PASSIVE
    int ai_family;      AF_UNSPEC
    int ai_socktype;    SOCK_STREAM
    int ai_protocol;    0
    socklen_t        ai_addrlen;
    char            *ai_canonname;
    struct  sockaddr *ai_addr;
    struct  addrinfo *ai_next;
};
```

The principles

You can also optionally specify if your calls will block until something comes on the network (the default), or return an error if there is no message.

1.5 Optional – specify socket behavior

It can be done in different ways.

```
#include <sys/socket.h>

int socket(int domain,
           int type,    SOCK_STREAM | SOCK_NONBLOCK
           int protocol);
```

The principles

2 Establish a connection using the connect() call that takes the socket id and a structure with host/port

```
#include <sys/socket.h>

int connect(int socket,
            const struct sockaddr *address,
            socklen_t address_len);
```

The principles

2

```
#include <netdb.h>
struct addrinfo {
    int ai_flags;
    int ai_family;
    int ai_socktype;
    int ai_protocol;
    socklen_t          ai_addrlen;
    char              *ai_canonname;
    struct  sockaddr  *ai_addr;
    struct  addrinfo  *ai_next;
};
```

```
#include <sys/socket.h>

int connect(int socket,
            const struct sockaddr *address,
            socklen_t address_len);
```

The call needs a complicated structure with a lot of information you don't know.

The principles

2

This information is retrieved by the system from other servers or local files, based on what you provide.

Dynamic Name Server

/etc/hosts
/etc/services

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *hostname,
                const char *servname,
                const struct addrinfo *hints,
                struct addrinfo **res);

void freeaddrinfo(struct addrinfo *ai);
```

name or IP address

port (as string) or name

what we know (not much)

list found by the system

The principles

Then you have very simple calls (it's like reading from/writing to a file) to send and receive messages.

3 Call **send()** to send a message

You can also call write()

4 If an answer is expected, call **recv()** to wait for it

You can also call read()

The principles

Rinse … Repeat …

5 Call **close()** to end the connection

What about writing a server?

What we have just seen is about writing a client program, a program that talks to a listening program, in other words a server. Writing a server program is hardly more complicated. The only difference is that if you want client programs to be able to talk to you, they must not only know on which computer your program is running, but on which port it's listening. We must decide on a port number.

Must use a fixed port

Read from configuration file!

~~< 1024~~

First rule: port numbers below 1024 are a definite and resounding NO, they are reserved for well defined, often highly technical services.

Beware for other ports!

Your port number must be a value between 1025 and 65532, but check with system engineers because some ports in this range may already be in use (Database Management Systems all use port numbers over 1024). Read the port number from a configuration file, it will be easy to change.

```
                     bind()
                     listen()
connect()
                     accept()
```

When your server starts, it must therefore read its port number, then call the `bind()` function that states that it's using that port number (if another program is already using it, the call will fail). Then it must call in a loop the `listen()` function that does nothing but wait for an incoming connection. When a client program issues a `connect()` call, `listen()` returns out of its slumber, and the program must acknowledge the connection request by calling `accept()`. Then it can call `recv()` to read the incoming message, and reply to it with `send()`.
Needless to say, both sides must agree on a high level protocol: what should be the format of the answer to every request?

Turning it to C++

http://vichargrave.com/network-programming-design-patterns-in-c/

TCPStream
TCPConnector
TCPAcceptor (for writing a server)

This page explains in detail and provides interesting C++ classes to ease programming.

WARNING

The code is ALMOST complete
(small gaps to fill)

You should use `send()` and `recv()` rather than `write()` and `read()` as suggested in the post

There are some errors in the test application
(confuses NULL and \0)

```
class TCPStream {
    int         m_sd;
    std::string m_peerIP;
    int         m_peerPort;

  public:
    friend class TCPAcceptor;
    friend class TCPConnector;

    ~TCPStream();

    ssize_t send(const char* buffer, size_t len);
    ssize_t receive(char* buffer, size_t len, int timeout);

    std::string getPeerIP();
    int         getPeerPort();

  private:
    TCPStream(int sd, struct sockaddr_in* address);
    TCPStream();
    TCPStream(const TCPStream& stream);
};
```

Many interesting things in this class: you have friend classes. Like friend functions, all their methods can access what is private in this class. Constructors, in particular, are private, which is rather unusal. You'll never create a TCPStream directly, but used one created by a TCPAcceptor or TCPConnector object.

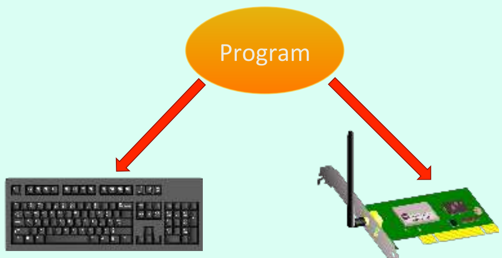
```
class TCPConnector {
  public:
    TCPStream* connect(const char* server, int port);

  private:
    int resolveHostName(const char* host, struct in_addr* addr);
};
```

It's the connect() method of a TCPConnector object (note: no coinstructor, it uses the default C++ constructor) that instantiates (creates) a TCPStream object and returns to you a pointer to this object.

TCPAcceptor not shown here

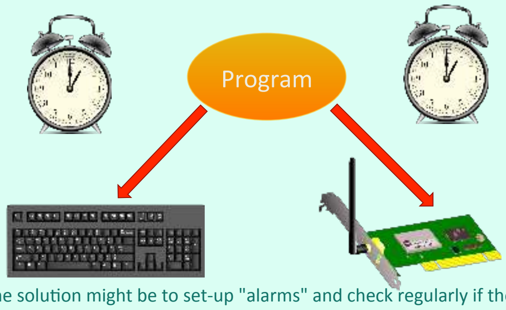
Practical issues

Reading from a network is like reading from the keyboard: it blocks. It can be a problem.



Think about a "chat" program. If it blocks on the keyboard, it cannot read what the other side is sending. If it blocks on the network, you cannot type.

Practical issues

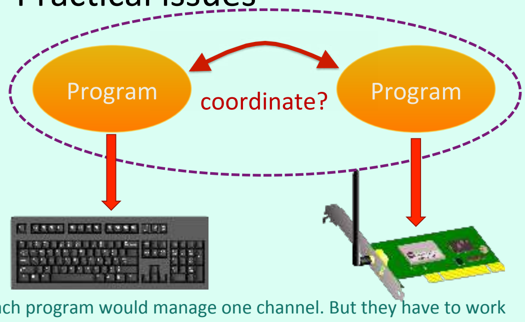


One solution might be to set-up "alarms" and check regularly if there is any input from the other channel, but it might be hard to code.

Practical issues

A better solution isn't to have a single program managing two channels, but coordinated programs working together.

Practical issues



Each program would manage one channel. But they have to work together, which is another issue!

Practical issues

Multithreading

This is an excellent case where having what is called "multiple threads" (parts of the program running at the same time and independently) would be an interesting solution. We'll soon see related topic, and perhaps we'll have time to talk a little about miltithreading proper.

A good network application example is the HTTP protocol that you are probably using everyday.

Sir Tim Berners-Lee

Photo: Enrique Dans

A HTTP server is a program that waits for requests formatted in a special way and listens on port 80 (all browsers know that).

Listens on port 80

HTTP is a "high-level" protocol, messages that must have a certain format and are sent over TCP/IP.

Protocol

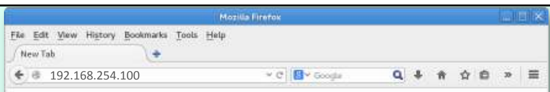
HTTP

TCP

It can be compared to correspondance rules when you are writing for business.

Protocol

(or you won't start a message to the White House with "Hi Donald")



If there is a HTTP server running on the machine the IP address is the one given above, and you type the address (or name, it will be translated) of the machine in your browser, it's interesting to see what the browser sends over the network.

"www.baidu.com"

```
int getaddrinfo(const char *hostname,
                const char *servname,
                const struct addrinfo *hints,
                struct addrinfo **res);
```

Note that your browser doesn't know the IP address of websites. It just knows that HTTP servers should listen on port 80. In the call to `getaddrinfo()` that you have seen earlier, the hostname will be the website name you have typed in, and your computer will get the actual IP address from other computers.

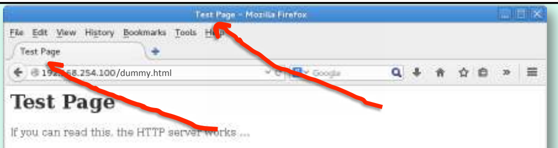
"80" or "http"

```
GET / HTTP/1.1
Host: 192.168.254.100
User-Agent: Mozilla/5.0 (Windows NT 5.1; rv:2.0.1) [...]
Accept: text/html,application/xhtml+xml,application/xml;[...]
Accept-Language: en-gb,en;q=0.5
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
Cache-Control: max-age=0
[empty line]
```

A message starts with a header (sometimes it's just a header) always terminated by an empty line. The first and second line are mandatory. The first one says which page should be returned (/ = homepage) and which is the HTTP version in use.
The second one repeats the name of the target (here, on a local network).

```
HTTP/1.1 200 OK
Date: [...]
Server: Apache/2/2/15 (Linux/SUSE)
Last-Modified: [date]
ETag: "2d7d-b7-4a3c52ef29046"
Accept-Ranges: bytes
Content-Length: 175
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html
[empty line]
<html>
  <head>
   <title>Test Page</title>
  </head>
  <body>
    <h1>Test Page</h1>
    <p>If you can read this, the HTTP server works ...</p>
  </body>
</html>
```

A reply from a webserver also follows a protocol. The first line specifies the protocol and provides a numerical code that gives the outcome of the previous request, followed by the meaning in clear, then the header contains various information, and the actual page displayed follows the empty line.

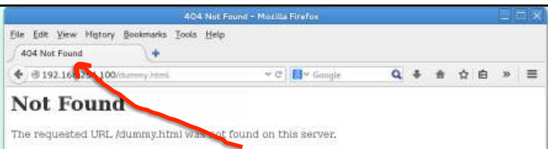


The title in the "head" section of the previous page is used by your browser to change the title of the window and the label on the tab.

GET /dummy.html HTTP/1.1

If you ask for a page that doesn't exist, the answer will obviously be different.

HTTP/1.1 404 Not Found



This page is entirely generated by the browser: the response simply was an error code.

Minimum HTTP message

"GET *<page>* HTTP/1.1\nHOST: *<host>*\n\n"

Many websites will drop the connection if there is no "User-Agent:" line, though (they also expect good manners). If you are interested in testing a program that sends HTTP requests and gets the answer, though, you can try this site, that was created for testing:

"GET /html HTTP/1.1\nHOST: httpbin.org\n\n"

Networking classes in action

I've written a small (obviously not complete) program that uses Vic Hargrave's classes to get HTML pages using the HTTP protocol, just to show how easy it becomes when the complicated code is wrapped in easy-to-use objects and methods.

I have created a HTTPCnx class that uses a TCPConnector and the TCPStream returned by the connect() method of the TCPConnector. The constructor only needs teh host name, as the prort is known.

```
class HTTPCnx {
    std::string   host;
    TCPConnector* connector;
    TCPStream*    stream;

    public:
        HTTPCnx(const char *host);
        ~HTTPCnx();
        std::string get(const char *page);
};
```

The constructor just calls the connect() method of the TCPConnector.

HTTPCnx.cpp

(most interesting bit)

```
string HTTPCnx::get(const char *page) {
   char    message[MSGLEN];
   string  full_message;
   ssize_t s;
   sprintf(message,"GET %s HTTP/1.1\nHOST: %s\n\n",
                   page, host.c_str());
   stream->send(message, strlen(message));
   full_message = "";
   do {
     s = stream->receive(message, MSGLEN, 1);
     if (s > 0) {
       full_message += string(message).substr(0, s-1);
     }
   } while (s > 0);
   return full_message;
}
```

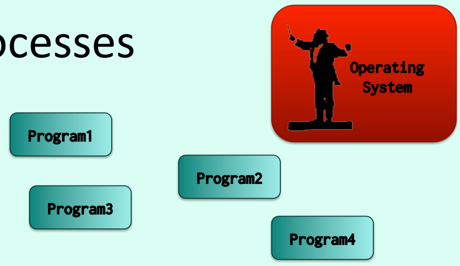
```
int main(int argc, char **argv) {
    if ((argc < 2) || (argc > 3)) {
        cerr << "Usage : " << argv[0] << " website [<page>]\n";
        return 1;
    }
    HTTPCnx *http = new HTTPCnx(argv[1]);
    if (http) {
       char   line[BUFFER_SZ];
       int    len;
       int    code;
       string response = "";
       response = http->get((argc == 3 ? argv[2] : "/"));
       if (response.length()) {
          cout << response << endl;
       } else {
          cerr << "No response" << endl;
       }
    } else {
       perror("Connection");
       cerr << "(httptest) Failed to connect\n";
    }
    return 0;
}
```

And here is a small test program.

Back to System Calls and processes

In today's complex software architectures, programs are permanently controlled, sometimes automatically restarted, or request flows diverted elsewhere to ensure service continuity. We are going to see a few useful functions for monitoring programs.

processes



Every process is identified by a process id (pid)

```
$ ps
  PID TTY           TIME CMD
 1800 ttys000    0:00.09 -bash
  826 ttys001    0:00.34 -bash
$
```

If you run the "ps" command on a Unix-like operating system, it will list you the running processes (TTY means "terminal"; I had two open console windows, "bash" is the program that accepts commands in them)

A process has easy access to two pids:

- Its own
- Its parent's

```
#include <unistd.h>

pid_t   getpid(void);
pid_t   getppid(void);
```

`getpid`: pid of the current process

`getppid`: pid of the parent process

A "pid_t" is a typedef'd integer

Important

Every process except process 1 is created by another process!

All processes have a parent process. We'll see this more in detail later. Nothing is born of nothing.

```
$ ps -o pid,ppid,time,comm
  PID  PPID      TIME  COMM
 1800  1799   0:00.09 -bash
  826   825   0:00.35 -bash
$
```

The "ps" command takes a lot of options. A particularly useful one is -o, followed by a comma separated list of things you want to see, here process id, parent process id, the time it has been running and the name of the command.

An easy way to create a subprocess:

You may already know `system()`. This function creates a subprocess that runs the command passed as a parameter.

```
#include <stdlib.h>

int system(const char *command);
```

Waits for command completion

Returns the return code of the command

run_ps.c

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    system("ps -o pid,ppid,time,comm");
    return 0;
}
```

A simple example that runs the previous command. Such a "wrapper" program can sometimes be useful; for instance, you can take advantage of conditional compiling for catering for subtle syntax variations between systems. Sometimes it's easier to do it that way than try to write a shell script that runs everywhere.

```
$ ./run_ps
  PID  PPID      TIME COMM
 1800  1799   0:00.09 -bash
  826   825   0:00.44 -bash
 2223   826   0:00.00 ./run_ps
$
```

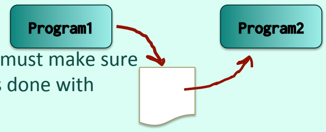
Here is the output of the previous command, which shows the run_ps program as a child process of the "bash" that runs in one of the consoles.

But how can two programs talk to each other? We can contemplate various options.

How can processes **communicate** with each other?

FILE *OK if sequential operations*

Files are easy, synchronization isn't: the reader must make sure that the writer is done with writing.



Socket *for different hosts* *"listening"*

Sockets are a bit of an overkill on a single computer, and assume that someone is listening.

If you want to grab the attention of a program on the same computer that isn't listening, the best is to send it a signal.

Signals

```
#include <signal.h>

int kill(pid_t pid, int sig);
```

Predefined value (symbol)

The "kill" name only refers to ONE usage of signals.

> 0 specific process

0 processes in the same group

-1 other processes of the same user

There may be some slight differences between systems

```
$ kill -l
 1) SIGHUP    2) SIGINT      3) SIGQUIT  4) SIGILL
 5) SIGTRAP   6) SIGABRT     7) SIGEMT   8) SIGFPE
 9) SIGKILL  10) SIGBUS     11) SIGSEGV 12) SIGSYS
13) SIGPIPE  14) SIGALRM    15) SIGTERM 16) SIGURG
17) SIGSTOP  18) SIGTSTP    19) SIGCONT 20) SIGCHLD
21) SIGTTIN  22) SIGTTOU    23) SIGIO   24) SIGXCPU
25) SIGXFSZ  26) SIGVTALRM  27) SIGPROF 28) SIGWINCH
29) SIGINFO  30) SIGUSR1    31) SIGUSR2
$
```

The "kill -l" command lists all available signals.
The default behavior when a program receives a signal depends on the signal.

```
$ kill -l
 1) SIGHUP    2) SIGINT      3) SIGQUIT  4) SIGILL
 5) SIGTRAP   6) SIGABRT     7) SIGEMT   8) SIGFPE
 9) SIGKILL  10) SIGBUS     11) SIGSEGV 12) SIGSYS
13) SIGPIPE  14) SIGALRM    15) SIGTERM 16) SIGURG
17) SIGSTOP  18) SIGTSTP    19) SIGCONT 20) SIGCHLD
21) SIGTTIN  22) SIGTTOU    23) SIGIO   24) SIGXCPU
25) SIGXFSZ  26) SIGVTALRM  27) SIGPROF 28) SIGWINCH
29) SIGINFO  30) SIGUSR1    31) SIGUSR2
$
```

Ignored by default

Of those, SIGCHLD, which means that the status of a child process has changed, can be useful for monitoring.

```
$ kill -l
 1) SIGHUP    2) SIGINT      3) SIGQUIT  4) SIGILL
 5) SIGTRAP   6) SIGABRT     7) SIGEMT   8) SIGFPE
 9) SIGKILL  10) SIGBUS     11) SIGSEGV 12) SIGSYS
13) SIGPIPE  14) SIGALRM    15) SIGTERM 16) SIGURG
17) SIGSTOP  18) SIGTSTP    19) SIGCONT 20) SIGCHLD
21) SIGTTIN  22) SIGTTOU    23) SIGIO   24) SIGXCPU
25) SIGXFSZ  26) SIGVTALRM  27) SIGPROF 28) SIGWINCH
29) SIGINFO  30) SIGUSR1    31) SIGUSR2
$
```

Stop (suspend) process

You can restart the process by sending SIGCONT to it.

```
$ kill -l
 1) SIGHUP    2) SIGINT      3) SIGQUIT  4) SIGILL
 5) SIGTRAP   6) SIGABRT     7) SIGEMT   8) SIGFPE
 9) SIGKILL  10) SIGBUS     11) SIGSEGV 12) SIGSYS
13) SIGPIPE  14) SIGALRM    15) SIGTERM 16) SIGURG
17) SIGSTOP  18) SIGTSTP    19) SIGCONT 20) SIGCHLD
21) SIGTTIN  22) SIGTTOU    23) SIGIO   24) SIGXCPU
25) SIGXFSZ  26) SIGVTALRM  27) SIGPROF 28) SIGWINCH
29) SIGINFO  30) SIGUSR1    31) SIGUSR2
$
```

Terminate process

SIGINT is what Ctrl-C delivers, SIGSEGV is a segmentation violation (dangling pointer), a zero-divide causes SIGFPE (floating-point exception), and so forth.

```
$ kill -l
 1) SIGHUP    2) SIGINT      3) SIGQUIT  4) SIGILL
 5) SIGTRAP   6) SIGABRT     7) SIGEMT   8) SIGFPE
 9) SIGKILL  10) SIGBUS     11) SIGSEGV 12) SIGSYS
13) SIGPIPE  14) SIGALRM    15) SIGTERM 16) SIGURG
17) SIGSTOP  18) SIGTSTP    19) SIGCONT 20) SIGCHLD
21) SIGTTIN  22) SIGTTOU    23) SIGIO   24) SIGXCPU
25) SIGXFSZ  26) SIGVTALRM  27) SIGPROF 28) SIGWINCH
29) SIGINFO  30) SIGUSR1    31) SIGUSR2
$
```

... with a core dump

A core dump is a file that contains an image of memory when the program crashes, to help with debugging. This files are very big and their generation is often disabled by system administrators.

Special signal:

0 = test if process is alive

(kill() returns 0 if no such process)

Finally, signal 0 is a dummy signal which isn't delivered to the "target", but allows the sender to know whether the process corresponding to the pid is up and running. This is very much used by monitoring system that check the good health of a system, and generate alerts when a critical process is down.

Almost all signals can be caught and handled!

"signal handler"

`void handler(int sig)`

typedef'd as sig_t

Beware:

In some systems, handlers are automatically deactivated after being called.

Must reset themselves.

SIGKILL and **SIGSTOP** cannot be caught or ignored

Signals

The easy way to trap a signal is by associating a handler to a signal with function `signal()`, which isn't considered a "system call"

```
#include <signal.h>
```

3

```
sig_t signal(int sig, sig_t func);
```

returns previous settings

SIG_IGN Ignore

SIG_DFL Default behavior

`void handler(int sig)`

The hard way to trap a signal is calling the `sigaction()` function which, also declared in the same `signal.h` as `signal()`, is documented in section 2 (system calls) of the manual.
`sigaction()` allows for finer handling of signals than `signal()` can. For instance, nothing prevents with `signal()` a handler from being itself interrupted while processing a signal. Function `sigaction()` allows to mask interrupts and work uninterrupted when needed, and so forth.

Signals

Also in signal.h

2

```
struct  sigaction {
   union __sigaction_u __sigaction_u;  /* signal handler */
          sigset_t sa_mask;          /* signal mask to apply */
          int      sa_flags;         /* signal options*/
  };

  union __sigaction_u {
          void    (*__sa_handler)(int);
          void    (*__sa_sigaction)(int, struct __siginfo *,
                         void *);
  };

  #define sa_handler      __sigaction_u.__sa_handler
  #define sa_sigaction    __sigaction_u.__sa_sigaction

  int sigaction(int sig, const struct sigaction *restrict act,
              struct sigaction *restrict oact);
```

beeper.c

```
#include <stdio.h>
#include <unistd.h>
#include <time.h>

#define SLEEP_TIME   10

int main() {
    int         nap = SLEEP_TIME;
    time_t      now;
    struct tm *t;

    while (1) {
        now = time(NULL);
        t = localtime(&now);
        printf("%02d:%02d:%02d beep!\n",
                t->tm_hour, t->tm_min, t->tm_sec);
        sleep(nap);
    }
    return 0;
}
```

We can easily illustrate signals with a simple program that runs an infinite loop, and displays the time and "beep!" every 10 seconds.
We can stop this program with Ctrl-C.

If we add this function to the program:

```
void handler(int sig) {
    printf("Tada!\n");
}
```

then define it as the `SIGINT` handler before the start of the loop:

```
(void)signal(SIGINT, handler);
```

Ctrl-C will no longer stop the program, but display *Tada!*
If we want to stop the program, we must open another window, run "ps" to find out the process id of the beeper, and issue something such as

`$ kill -SIGKILL <pid>` ("kill pid" sends SIGTERM)