Ping-pong and Echoes

Johan Montelius

HT2016

1 Introduction

In this tutorial you're going to explore different ways for two processes to communicate with each other. We will first look at *pipes*, something that you have probably used when working in the shell but now we will implement our own pipes. We will then look at *sockets*, an abstraction that can be used for process communication even if the processes reside on different machines in a network.

2 Pipes

To understand pipes you only have to understand how to read and write to streams such as standard input or standard output. A stream is of course nothing else but a sequence of bytes and we have already seen how `read()` and `write()` can be used to operate on a file descriptor.

When we create a pipe we will be given two file descriptors, one that will be used by the producer and one that can be used by a consumer. If we want to create two processes that can communicate of a pipe, we first create the pipe and then `fork()` two processes. Since any forked process will share the same file descriptor table, we can set up one process to use the `write descriptor` to produce data and the other to use the *read descriptor* to consume data.

2.1 flow control

The nice thing with the pipe is that it will take care of `flow control` i.e. the producer will be suspended if the consumer can not process the information quick enough. Let's run an experiment to see this in action, open a file called `flow.c` and add the following code.

We will let a producer send ten bytes in a burst and do this ten times. We will later change these numbers so let's define them as macros.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <sys/wait.h>
```

```
#define ITERATIONS 10
#define BURST 10
```

In the main procedure we first create a pipe providing an array where the procedure will store the two file descriptors. We then fork a new process determine if we're the mother or child process.

```
int main() {
  int descr[2];

  assert(0 == pipe(descr));

  if(fork() == 0 ) {
    /* consumer */
    :
  }

  /* producer  */
  :
  wait(NULL);
  printf("all done\n");
}
```

The consumer will read the burst of ten bytes each but will sleep for a second in between bursts. This is to simulate a process that is receiving some data and then spend some time processing this data.

```
for(int i = 0; i < ITERATIONS; i++) {
  for(int j = 0; j < BURST; j++) {
    int buffer[10];
    read(descr[0], &buffer, 10);
  }
  sleep(1);
}
printf("consumer done\n");
return 0;
```

The procedure will be more eager and will produce data as quickly as possible. To keep track of the pace we let it print a message by the end of each burst.

```
for(int i = 0; i < ITERATIONS; i++) {
  for(int j = 0; j < BURST; j++) {
    write(descr[1], "0123456789", 10);
  }
  printf("producer burst %d done\n", i);
}
printf("producer done\n");
```

That is it, compile and run the benchmark - explain what is happening. Now increase the size of the burst to a hundred, to a thousand - what is happening? What you see is an effect of *flow control*, something very important when it comes to any type of communication.

2.2 named pipes

Pipes are very easy to use since the only thing we need to understand is how to read a sequence of bytes from a file descriptor. They are also very easy to set up ... if you're forking a new process. The question is how to do it if you're not the one creating the new process.

It turns out that we can use the same naming scheme as for files to register and find pipes. We can thus create a pipe and register it under a file name. To do this we use the library call `mkfifo()` - check the man pages, note that it exists both as a command and as a library call (use `>man 3 mkfifo`).

We need half a dozen of include statements so create two files, `cave.c` and `baba.c`. They will look very much like the `flow.c` but now we split the producer (the cave) from the consumer (*Ali Baba*), and connect them through a pipe called `sesame`.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>


#define ITERATIONS 10
#define BURST 1000
```

In `cave.c` we have the producer and also the process that will create the pipe. This is were we call `mkfifo()` providing the name `sesame` and the *mode* of the pipe. Once the pipe is created we will *open* the pipe using a call `open()`. This is done in the same way as we would have opened any file. In this example we open it in write mode only since the producer will only write to the pipe. The rest of the file uses the producer part of `flow.c` i.e. looping over the `ITERATIONS` and `BURST`, the only difference is that we now write to `pipe`.

```
int main() {
  /* create the named pipe */
  int mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
```

```
    mkfifo("sesame", mode);

    int flag = O_WRONLY;
    int pipe = open("sesame", flag);
    :
    :
    printf(``producer done\n'');
    return 0;
}
```

The consumer assumes that there is a pipe called `sesame` and will open this for reading. Complete the code by using the consumer part from `flow.c`.

```
int main() {
    /* open pipe for reading */
    int flag = O_RDONLY;
    int pipe = open("sesame", flag);
    :
    :
    printf("consumer done\n");
    return 0;
}
```

Compile and run the producer and consumer in the same shell, use the `&` notation to set a program to run in the background.

```
./cave&
[cave] 15321
> ./baba
producer burst 0 done
producer burst 1 done
   :
   :
```

Look at the man pages for `open()` (use `>man 3 open`). implicitly say that the file should not be created if it does not exist i.e. the call will suspend until we create the pipe. We also implicitly say that the reader should suspend waiting for the writer and also that the writer should wait for a reader (check `O_NONBLOCK`). This means that we equally well can start the producer before the consumer.

```
./baba&
[baba] 15351
> ./cave
producer burst 0 done
producer burst 1 done
```

:
:

2.3 marshaling

The easy way in which we can use pipes to communicate has a down-side. How do you send such a simple thing as a *floating-point* from one process to the other? The problem is of course that we're not sending data structures but bytes in the pipe. This means that we have to send the bytes across and then interpret them as a floating-point on the other side.

Let's try to send some floating-points from one process to the other. Make a copy of `flow.c` and call it `marshal.c`. Then do the following changes, the consumer part will look like this:

```
for(int i = 0; i < ITERATIONS; i++) {
  double buffer;
  read(descr[0], &buffer, sizeof(double));
  printf("received %f\n", buffer);
  sleep(1);
}
```

and the producer part will look like this:

```
for(int i = 0; i < ITERATIONS; i++) {
  double pi = 3.14*i;
  write(descr[1], &pi, sizeof(double));
}
```

Try this out and see that it works.

The reason why it works is of course because the producers and consumer represents floating-point data structures in the same way. Note that we have no clue of what this representation looks like but we know that if we send some bytes across a pipe and interpret them as the data structure that we know they represent then obviously it will work.

The above is not necessarily true if the processes were running on different machines, were written in different languages or even compiled with different versions of a compiler. We can use this simple way of marshaling data structures since we know that both the producer and the consumer agree on how things are represented.

Things do however become very complicated if we for example want to send a linked list across the pipe. Obviously you can not send a memory reference across a pipe so you would have to come up with your own representation of linked data structures.

The problem of turning data structures into sequences of bytes is called *marshaling*. We will not explore this further but you should realize that it is a problem.

3 Sockets

Pipes are a very efficient way of communicating between processes but it has one draw back - it is only one-way communication. In order for two processes to have a two-way communication you would have to use two pipes, one in each direction. This is of course easily done but it would be nice to have an abstraction that provided two-way communication directly - introducing *sockets*.

A *socket* is the abstraction of pipe communication were we do not assume a shared file system. We should be able to open up a communication channel with any process even if it is not running on the same machine. If the processes are actually running on the same machine, things will be as efficient as using pipes but the user level application do not need to be aware of how things are implemented.

You should already be familiar with network communication so we will not go through TCP and UDP but rather take a look at sockets from an applications point of view.

3.1 ping-pong

When we create a socket we choose the *domain* for the socket. Look up the different domains using `man socket`, as you can see the domains specify the networking protocol that the socket should use. There is however one domain that is a bit different, the `AF_UNIX` domain. This domain is used if we know that the processes are running on the same machine.

Let's implement a *ping-pong* experiment where two processes are sending ping messages to and from. Create two files `ping.c` and `pong.c`. The two processes will look very similar but `ping` will serve as the server and `pong` will act as a client i.e. it will find and connect to `ping`.

We need some header files and among the usual suspects we find `socket.h` that will provide the support for sockets and `un.h` that is the special `AF_UNIX` domain.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <assert.h>

#define NAME "pingpong"
#define TURNS 10
```

The name `pingpong` is the address that we will use to register the socket. As with pipes we will use the regular file system to provide name resolution.

In both files we create a socket and a socket address `{AF_UNIX, NAME}`.

```
int main(void) {
  int sock;

  assert((sock = socket(AF_UNIX, SOCK_STREAM, 0)) != -1);

  struct sockaddr_un name = {AF_UNIX, NAME};
```

In `ping.c` we then *bind* the socket to the address i.e. we create the `pingpong` name in the file system in order for the `pong` process to find it. We also start to *listen* to the socket and thereby tell the operating system that we are prepared to accept connecting clients. The call to `assert()` will suspend until a client connects and then return a new socket descriptor `cfd`. This is our two-way connection that we can use for communication.

```
  assert(bind(sock, (struct sockaddr *)&name, sizeof(name)) != -1);

  assert(listen(sock, 5) != -1);

  int cfd;

  assert((cfd = accept(sock, NULL, NULL)) != -1);
```

The file `pong.c` looks very similar but now we do not have to bind the socket nor wait for incoming connection. We simply us the socket and the socket address to connect to the server.

```
  assert(connect(sock, (struct sockaddr *)&name, sizeof(name)) != -1);
```

The call to `connect()` will only return a error if something goes wrong but it will also connect the socket so we can use it to communicate with the `ping` process.

Both the `ping.c` and `pong.c` files then contain a section were we use `send()` and `recv()` to pass messages to a fro. Below is the code for `ping.c` and you will have to adapt it for `pong.c`.

```
  for(int i = 0; i < TURNS; i++) {
    char buffer[5];
    assert(send(cfd, "ping", 4, 0) != -1);
    assert(recv(cfd, buffer, 4, 0) != -1);
    buffer[4] = 0;
    printf("ping received %s\n", buffer);
  }
```

The file `ping.c` will of course look very similar, only difference is that we will use `sock` when we communicate and that we will first receive a message before sending a message.

If everything compiles you should be able to run the experiment as follows:

```
> ./ping &
[ping] 4427
> ./pong
ping received ping
pong received ping
ping received ping
   :
```

If you run it a second time you will receive the strange message:

```
ping: ping.c:21: main: Assertion ‘bind(sock,  .....
aborted (core dumped)
```

If you look in your directory using `ls -l` you will see the file name `pingpong` with the strange description `srwxrwxr-x`. The `s` means that this is not a regular file but rather a name of a socket. When we try to bind the socket to a name we get an error since the name is already take. To prevent this we should `unlink()` the name when we're done. Remove the file and add `unlink(NAME)` to the end of `ping.c`.

You also see that people add a call to `unlink()` before doing a call to `bind()` but then one should of course be sure, that no one is actually using the name.

3.2 datagrams

One problem with the solution that we have so far is that we want to send messages at the application layer but the communication layer only provides an abstraction of a stream of bytes. In our program we have solved this problem by knowing that a message is four bytes. It's a bit more complicated if messages can be of arbitrary size.

We could, and many communication protocols work this way (http for example), encode the length of a message in the first byte (or two) and then read as many bytes as required. However, since the problem is so often encountered there is an abstraction that provides what we're looking for - *datagrams*.

Assume that we want to send text strings to a server and we want the server to convert the string to lower case and then send it back. We can implement this quite easily using the datagram socket type. Create two files `tolower.c` and `conv.c`. They will look very similar so we will go through them and point out the differences.

We will of course use some header files but by now you should be able to figure out which ones to use using the `man` command. We start by showing the main procedure.

```
#define SERVER "lower"
#define MAX 512

int main(void)  {
  int sock;
  char buffer[MAX];

  /* A socket is created */
  assert((sock = socket(AF_UNIX, SOCK_DGRAM, 0)) != -1);

  struct sockaddr_un name = {AF_UNIX, SERVER};

  assert(bind(sock, (struct sockaddr *) &name, sizeof(struct sockaddr_un)) != -1);
      :
```

The server, `tolower.c`, will as before create a socket but now we state that it is of `SOCK_DGRAM` type. As before we register it under a name in order for the client to find us.

The client, `conv.c`, will look the same but we register the socket under a name of its own. Note that we will later use the `SERVER` name so keep this in the file. We also provide a text to test the service.

```
#define TEST "This is a TeSt to SEE if iT wORks"
#define CLIENT "help"

int main(void) {
     :
    struct sockaddr_un name = {AF_UNIX, CLIENT};
     :
```

It is now time for the server to wait for incoming messages and we do this using the socket directly i.e. there is no call to `listen()`. We create a data structure to hold the address of the client that connects and then call `recvfrom()`. When this call returns we will have a message in the buffer that we provided - since it is a datagram service we will have the whole message.

```
    struct sockaddr_un client;
    int size = sizeof(struct sockaddr_un);

    while(1) {
      int n;
      n = recvfrom(sock, buffer, MAX-1, 0, (struct sockaddr *) &client, &size);
      if (n == -1)
        perror("server");

      buffer[n] = 0;
```

```
    printf("Server received: %s\n",buffer);

    for (int i= 0; i < n; i++)
      buffer[i] = tolower((unsigned char) buffer[i]);

    assert(sendto(sock, buffer, n, 0, (struct sockaddr *) &client, size) == n);
  }
```

The server will convert the message to lower case letters and then send it back to the client. Notice that we know who we should send it to since we received the address of the client in the call to `recvfrom()`. There is no established connection between the two processes, the two messages are independent from each other.

On the client side we have a similar scenario but here we know who we want to send the message to. This is where we need the name `SERVER`, in our call to `sendto()` we must provide the address of the server.

```
  struct sockaddr_un server = {AF_UNIX, SERVER};
  int size = sizeof(struct sockaddr_un);

  int n = sizeof(TEST);

  assert(sendto(sock, TEST, n, 0, (struct sockaddr *) &server, size) != -1);
  n = recvfrom(sock, buffer, MAX-1, 0, (struct sockaddr *) &server, &size);
  if (n == -1)
    perror("server");

  buffer[n] = 0;
  printf("Client received: %s\n", buffer);
  unlink(CLIENT);
  exit(0);
}
```

Hard coding the address of the client might not be a very good idea if we want to run multiple instances of the client. The names must of course not collide and we would be better of with selecting the addresses dynamically.

3.3 sequence of datagrams

The beauty of datagrams is that they are independent from each other and this is also the problem. We are not guaranteed that messages are delivered in order nor will we be informed if a datagram is lost.

This will of course not happen if we have the two processes running in the same operating system but it is a scenario if we run across the Internet. If this is the case we might want to use the `SOCK_SEQPACKET` that will give us a reliable ordered delivery of messages.

3.4 address families

We have seen how sockets can use the file system as the address but there are many different *address families* that we can use. The most used is the *internet protocol* that will allow us to communicate between two processes on different machines in a network. The socket abstraction allows us to change the underlying communication protocol and allows the application level to remain the same.

To demonstrate how easy it is to set up two processes that communicate over the network we will implement an *echo server*, a process that simply accepts incoming messages and then return them to their destination. The implementation will look very much like the `tolower.c` and `conv.c` so you can use these files as templates for two files called `echo.c` and `hello.c`.

Both files need the same set of header files, notice the `netinet/ip.h` and `arpa/inet.h` that will give us support for the IP sockets and some address translations that will come in handy.

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <arpa/inet.h>
#include <assert.h>
```

The echo process will create a socket and bind it to the *port* `8080`. The procedure `htons()` and `htonl()` will convert short (16-bit) and long (32-bit) integers to *network byte order* i.e. the order the two or four bytes should appear when we send them to be interpreted correctly by the network infrastructure.

We (I assume that you only have one network connection) do not specify the ip-address but allow the operating system (`INADDR_ANY`) to choose this for us.

```
#define PORT 8080
#define MAX 512

int main(void)  {
    int sock;
    char buffer[MAX];

    /* A socket is created */
    assert((sock = socket(AF_INET, SOCK_DGRAM, 0)) != -1);

    struct sockaddr_in name;
```

```
    name.sin_family = AF_INET;
    name.sin_port = htons(PORT);
    name.sin_addr.s_addr = htonl(INADDR_ANY);

    assert(bind(sock, (struct sockaddr *) &name, sizeof(name)) != -1);
        :
```

We then do exactly as what we did in `tolower.c` but we do not convert the string to lower-case before sending it back. We can add another print out statement to see who we got the message from. The procedure `inet_ntoa()` will take a network address and turn it into a string and `ntohs()` will take a short network address and turn it into an integer.

```
  printf("Server: received: %s\n", buffer);
  printf("Server: from destination %s %d\n", inet_ntoa(client.sin_addr), ntohs(clie
```

That's it for the echo server, the client `hello.c` will be almost as simple. We include the same header files but now of course need the address of the server in order to know how to connect. You can specify the server ip-address as a string as shown in the example below but there are other possibilities. In this example we use the *loop-back address 127.0.0.1* i.e. the machine it self.

```
#define MAX 512
#define TEST "Hello, hello"

#define SERVER_IP "127.0.0.1"
#define SERVER_PORT 8080
```

The hello process of course needs a socket by its own and bind this socket to an address in order for the server to send back a reply. In this case we allow that operating system to select a port number for us since we will only use this for reply messages, we do not need to use a known port number.

```
    struct sockaddr_in name;

    name.sin_family = AF_INET;
    name.sin_port = 0;
    name.sin_addr.s_addr = htonl(INADDR_ANY);

    assert(bind(sock, (struct sockaddr *) &name, sizeof(name)) != -1);
```

We can now create the address of the server, this is were we need the server port number and server ip-address.

```
    struct sockaddr_in server;

    server.sin_family = AF_INET;
```

```
server.sin_port = htons(SERVER_PORT);
server.sin_addr.s_addr = inet_addr(SERVER_IP);
```

This is it, we now do exactly what we did in the file `conv.c` i.e. send a message and wait for the reply. If everything works you should be able to start the echo server and then run `hello` from your own machine. If you check you ip-address and change the `hello.c` you should be able to do the same thing across the network.

3.5 Alternatives

One alternative mechanism for process communication is the concept of *signals*. Signals are however not intended as a tool for process communication, rather a way for the operating system to alert the process of events and possibly for a process to control the faith of another process i.e. kill it or suspend its execution. Using signals for communication is not advisable since it is likely to introduce more problems that it will actually solve.

Another concept is the use of *shared memory*. We have already seen how two threads in a process can collaborate when they share the same memory and it is possible for two processes to do almost the same. One process can map a file into memory to provide direct access to its content and if two processes map the same file they will share this content. It is however very different from the memory shared by two threads since the file is mapped into different virtual memory regions. To choose this approach you really have to know what you're doing.

The socket abstraction layer is what everything on the Internet is built on top of. It might however not be the best things to use when you're building an application that will run on a single operating system. You would like to have more support from a communication layer to handle for example `authentication`, `multicasting` and *publish subscribe* functionality. There are many messaging abstraction layers that provide this but to explore these require another tutorial. If you want to explore something by yourself that is highly related to operating systems, take a look at *D-Bus*.

4 Summary

Pipes, a simple way to use communication abstraction that used the notion of reading and writing to file descriptors in the same way as you would access a file.

Sockets provide an abstraction on a higher level that provides two-way communication using either byte streams, messages, sequences of messages etc. We have several networking protocols to choose from and we can change the choice of *address family* without changing the main part of our application.