# Pipes

- Sharing files is a way for various processes to communicate among themselves (but this entails a number of problems).
- pipes are one means that Unix addresses one-way communication between two process (often parent and child).
- Simply stated: in a pipe, a process sends "down" the pipe data using a `write` and another (perhaps the same?) process receives data at the other end through with the help of a `read` call.

# Pipes

- ```
  #include <unistd.h>

  int pipe(int pipefd[2]);
  ```

- `pipe` creates a unidirectional data channel that can be used for interprocess communication in which `pipefd[0]` refers to the read end of the pipe and `pipefd[1]` to the write end.

- A pipe's real value appears when it is used in conjunction with a `fork()` and the fact that file descriptors remain open across a `fork()`.
  ```

# Somewhat of a useless example...

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#define MSGSIZE 16

char *msg1="Buenos Dias! #1";
char *msg2="Buenos Dias! #2";
char *msg3="Buenos Dias! #3";

main(){
  char inbuf[MSGSIZE];
  int p[2], i=0, rsize=0;
  pid_t pid;

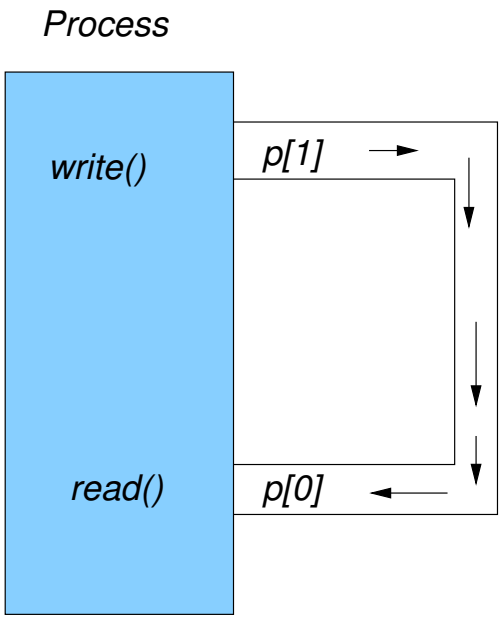
  if (pipe(p)==-1) { perror("pipe call"); exit(1);}

  write(p[1],msg1,MSGSIZE);
  write(p[1],msg2,MSGSIZE);
  write(p[1],msg3,MSGSIZE);

  for (i=0;i<3;i++){
        rsize=read(p[0],inbuf,MSGSIZE);
        printf("%.*s\n",rsize,inbuf);
        }
  exit(0);
}
```

# Here is what happens...

```
antoulas@sazerac:~/SysProMaterial/Set005/src$ ./a.out
Buenos Dias! #1
Buenos Dias! #2
Buenos Dias! #3
antoulas@sazerac:~/SysProMaterial/Set005/src$
```




The output and the input are part of the **same** process - not useful!

# Here is a *somewhat* more useful example...

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#define MSGSIZE 16

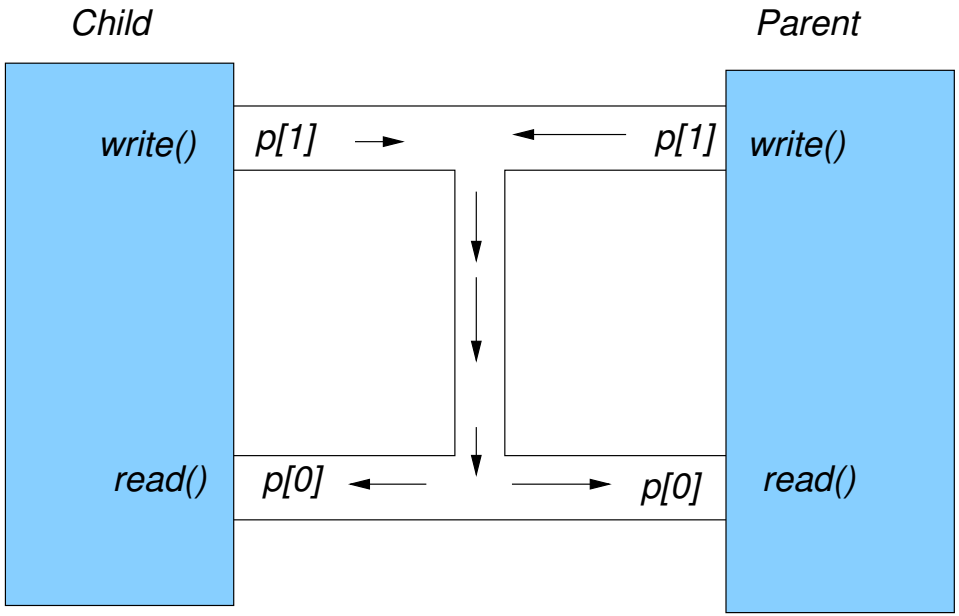
char *msg1="Buenos Dias! #1";
char *msg2="Buenos Dias! #2";
char *msg3="Buenos Dias! #3";

main(){
  char inbuf[MSGSIZE];
  int p[2], i=0, rsize=0;
  pid_t pid;

  if (pipe(p)==-1) { perror("pipe call"); exit(1);}

  switch(pid=fork()){
  case -1: perror("fork call"); exit(2);
  case  0: write(p[1],msg1,MSGSIZE);   // if child then write!
           write(p[1],msg2,MSGSIZE);
           write(p[1],msg3,MSGSIZE);
           break;
  default: for (i=0;i<3;i++){          // if parent then read!
                rsize=read(p[0],inbuf,MSGSIZE);
                printf("%.*s\n",rsize,inbuf);
                }
           wait(NULL);
  }
  exit(0);
}
```

# Here is what happens now:




- Either process could write down the file descriptor *p[1]*.
- Either process could read from the file descriptor *p[0]*.
- Problem: pipes are intended to be unidirectional; if both processes start reading and writing indiscriminately, **chaos may ensue**.

# A much cleaner version

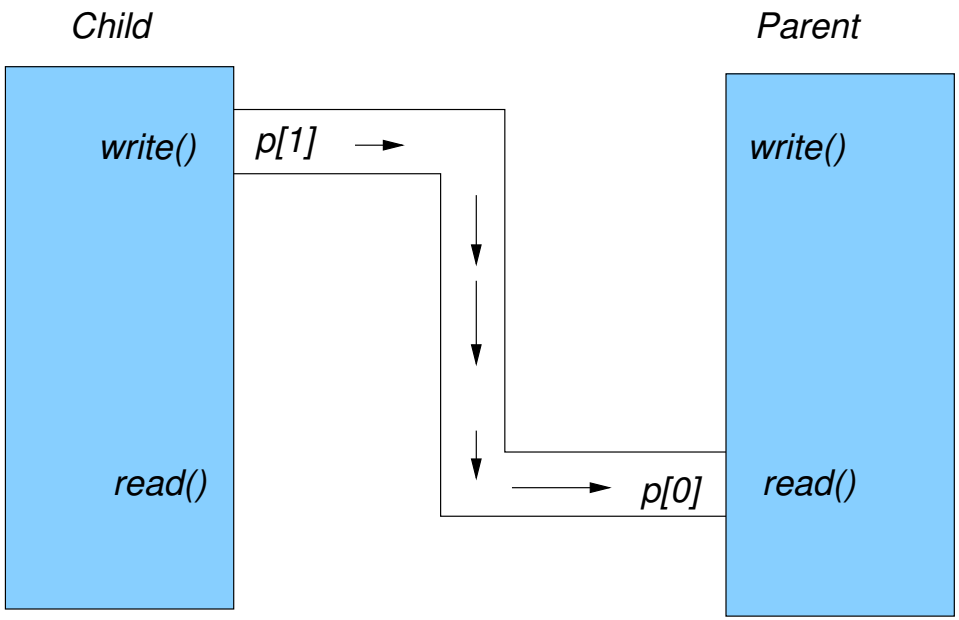
```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#define MSGSIZE 16

char *msg1="Buenos Dias! #1";
char *msg2="Buenos Dias! #2";
char *msg3="Buenos Dias! #3";

main(){
  char inbuf[MSGSIZE];
  int p[2], i=0, rsize=0;
  pid_t pid;
  if (pipe(p)==-1) { perror("pipe call"); exit(1);}

  switch(pid=fork()){
  case -1: perror("fork call"); exit(2);
  case  0: close(p[0]);                  // child is writing
           write(p[1],msg1,MSGSIZE);
           write(p[1],msg2,MSGSIZE);
           write(p[1],msg3,MSGSIZE);
       break;
  default: close(p[1]);                  // parent is reading
           for (i=0;i<3;i++){
        rsize=read(p[0],inbuf,MSGSIZE);
        printf("%.*s\n",rsize,inbuf);
        }
        wait(NULL);
  }
  exit(0);
}
```

# .. and pictorially:



# Another Example

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#define  READ 0
#define  WRITE 1
#define  BUFSIZE 100
char *mystring = "This is a test only; this is a test";

int main(void){
        pid_t  pid;
        int fd[2], bytes;
        char message[BUFSIZE];

  if (pipe(fd) == -1){ perror("pipe"); exit(1); }

  if ( (pid = fork()) == -1 ){ perror("fork"); exit(1); }

  if ( pid == 0 ){    //child
        close(fd[READ]);
        write(fd[WRITE], mystring, strlen(mystring)+1);
        close(fd[WRITE]);
        }
  else  {                 // parent
        close(fd[WRITE]);
        bytes=read(fd[READ], message, sizeof(message));
        printf("Read %d bytes: %s \n",bytes, message);
        close(fd[READ]);
        }
}
```

# Outcome:

```
antoulas@sazerac:~/src-set005$ ./pipes-example-p50
Read 36 bytes: This is a test only; this is a test
antoulas@sazerac:~/src-set005$
```

- Anytime, *read/write-ends* are not needed, make sure they are closed off.

# `read()` call and pipes

- If a process has opened up a pipe for `write` but has not written anything yet, a potential `read()` blocks.
- if a pipe is empty and no process has the pipe open for `write()`, a `read()` returns 0.

# Example with pipe and dup2

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#define  READ 0
#define  WRITE 1

int main(int argc, char *argv[]){
pid_t  pid;
int fd[2], bytes;

if ( argc!= 3) {printf("./a.out prog1 prg2"); exit(23);}

if (pipe(fd) == -1){ perror("pipe"); exit(1); }
if ( (pid = fork()) == -1 ){ perror("fork"); exit(1); }
if ( pid != 0 ){    // parent and writer
        close(fd[READ]);
        dup2(fd[WRITE],1);
        close(fd[WRITE]);
        execlp(argv[1], argv[1], NULL);
        perror("execlp");
        }
else    {             // child and reader
        close(fd[WRITE]);
        dup2(fd[READ],0);
        close(fd[READ]);
        execlp(argv[2],argv[2],NULL);
        }
}
```

# Some outcomes:

```
antoulas@sazerac:~/src$ ./a.out ls wc
antoulas@sazerac:~/src$       22       22      244
antoulas@sazerac:~/src$
antoulas@sazerac:~/src$ ./a.out ps sort
 3420 pts/4    00:00:00 bash
 6849 pts/4    00:00:00 ps
 6850 pts/4    00:00:00 sort
  PID TTY          TIME CMD
antoulas@sazerac:~/src$ ./a.out ls head
a.out
exec-demo.c
execvp-1.c
execvp-2.c
fork1.c
fork2.c
mychain.c
mychild2.c
myexit.c
mygetlimits.c
antoulas@sazerac:~/src$
antoulas@sazerac:~/src$
```

# The size of a pipe

- The size of data that can be written down a pipe is **finite**.
- If there is no more space left in the pipe, an impending write will *block* (until space becomes again available).
- POSIX designates this limit to be 512 Bytes.
- Unix systems often display *much higher* capacity for this buffer area.
- The following is a small program that helps discover “real” upper-bounds for this limit.

# The *size* program

```c
#include <signal.h>
#include <unistd.h>
#include <limits.h>
#include <stdlib.h>
#include <stdio.h>

int count=0;
void alrm_action(int);

main(){
int p[2];
int pipe_size=0;
char c='x';
static struct sigaction act;

// set up the signal handler
act.sa_handler=alrm_action;
sigfillset(&(act.sa_mask));

if ( pipe(p) == -1) { perror("pipe call"); exit(1);}
pipe_size=fpathconf(p[0], _PC_PIPE_BUF);
printf("Maximum size of (atomic) write to pipe: %d bytes\n", pipe_size);
printf("The respective POSIX value %d\n",_POSIX_PIPE_BUF);
```

# The *size* program

```
sigaction(SIGALRM, &act, NULL);
while (1) {
       alarm(10);                /* set alarm                          */
       write(p[1], &c, 1);       /* do the writing of the character */
       alarm(0);                 /* reset alarm                        */
       if (++count % 4096 == 0 )/* report every 4kbytes written out*/
          printf("%d characters in pipe\n",count);
      }
}

void alrm_action(int signo){
        printf("Alrm-Handler: write blocked after %d characters \n",count);
        exit(0);
}
```

# Outcome of execution:

```
antoulas@sazerac:~/src-set005$ ./pipe-size
Maximum size of (atomic) write to pipe: 4096 bytes
The respective POSIX value 512
4096 characters in pipe
8192 characters in pipe
12288 characters in pipe
16384 characters in pipe
20480 characters in pipe
24576 characters in pipe
28672 characters in pipe
32768 characters in pipe
36864 characters in pipe
40960 characters in pipe
45056 characters in pipe
49152 characters in pipe
53248 characters in pipe
57344 characters in pipe
61440 characters in pipe
65536 characters in pipe
Alrm-Handler: write blocked after 65536 characters
antoulas@sazerac:~/src-set005$
```

- A `write` on a pipe will execute *atomically* – all data transfered in a single kernel operation.
- Otherwise, writing occurs in stages.
- If multiple processes write to the same pipe, data will inevitably become mingled.

# What happens to file descriptors/pipes after an `exec`

- A copy of file descriptors and pipes are inherited by the child (as well as the signal state and the scheduling parameters).
- Although the file descriptors are "available" and accessible by the child, their **symbolic names are not!!**
- How do we "pass" descriptors to the program called by `exec`?
  - pass such descriptors as inline parameters
  - use standard file descriptors: 0, 1 and 2 ("as is").

# demo-file-pipes-exec creates a child & calls execlp

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>

#define READ 0
#define WRITE 1

main(int argc, char *argv[]){
    int fd1[2], fd2[2], filedesc1= -1;
    char myinputparam[20];
    pid_t pid;

    // create a number of file(s)/pipe(s)/etc
    if ( (filedesc1=open("MytestFile", O_WRONLY|O_CREAT, 0666)) == -1){
        perror("file creation"); exit(1);
        }
    if ( pipe(fd1) == -1 ) {
        perror("pipe"); exit(1);
        }
    if ( pipe(fd2)== -1 ) {
                perror("pipe"); exit(1);
                }
```

```
    if ( (pid=fork()) == -1){
        perror("fork"); exit(1);
        }

    if ( pid!=0 ){          // parent process - closes off everything
        close(filedesc1);
        close(fd1[READ]); close(fd1[WRITE]);
        close(fd2[READ]); close(fd2[WRITE]);
        close(0); close(1); close(2);
        if (wait(NULL)!=pid){
            perror("Waiting for child\n"); exit(1);
            }
        }
    else {
        printf("filedesc1=%d\n", filedesc1);
        printf("fd1[READ]=%d, fd1[WRITE]=%d,\n",fd1[READ], fd1[WRITE]);
        printf("fd2[READ]=%d, fd2[WRITE]=%d\n", fd2[READ], fd2[WRITE]);
        dup2(fd2[WRITE], 11);
        execlp(argv[1], argv[1], "11", NULL);
        perror("execlp");
    }
}
```

# write-portion replaces the image of the child

```
#include <stdio.h> .....
#define READ 0
#define WRITE 1

main(int argc, char *argv[] ) {
    char message[]="Hello there!";
    // although the program is NOT aware of the logical names
    // can access/manipulate the file descriptors!!!
    printf("Operating after the execlp invocation! \n");
    if ( write(3,message, strlen(message)+1)== -1)
        perror("Write to 3-file \n");
    else    printf("Write to file with file descriptor 3 succeeded\n");
    if ( write(5, message, strlen(message)+1) == -1)
        perror("Write to 5-pipe");
    else    printf("Write to pipe with file descriptor 5 succeeded\n");
    if ( write(7, message, strlen(message)+1) == -1)
        perror("Write to 7-pipe");
    else    printf("Write to pipe with file descriptor 7 succeeded\n");
    if ( write(11, message, strlen(message)+1) == -1)
                perror("Write to 11-dup2");
        else    printf("Write to dup2ed file descriptor 11 succeeded\n");
    if ( write(13, message, strlen(message)+1) == -1)
                perror("Write to 13-invalid");
        else    printf("Write to invalid file descriptor 13 not feasible\n");
    return 1;
}
```

# Running the last two programs...

## Execution with no parameter:

```
antoulas@sazerac:~/src-set005$
antoulas@sazerac:~/src-set005$ ./demo-file-pipes-exec
filedesc1=3
fd1[READ]=4, fd1[WRITE]=5,
fd2[READ]=6, fd2[WRITE]=7
antoulas@sazerac:~/src-set005$
```

## Execution with parameter:

```
antoulas@sazerac:~/src-set005$ ./demo-file-pipes-exec ./write-portion
filedesc1=3
fd1[READ]=4, fd1[WRITE]=5,
fd2[READ]=6, fd2[WRITE]=7
Operating after the execlp invocation!
Write to file with file descriptor 3 succeeded
Write to pipe with file descriptor 5 succeeded
Write to pipe with file descriptor 7 succeeded
Write to dup2ed file descriptor 11 succeeded
Write to 13-invalid: Bad file descriptor
antoulas@sazerac:~/src-set005$
```

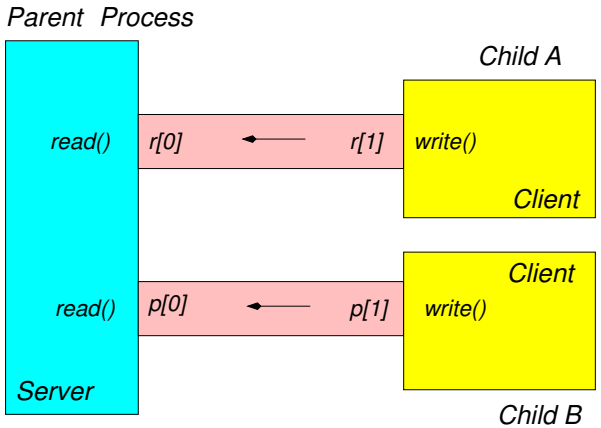
# redirecting output in an unusual way..

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define SENTINEL -1

main(){
    pid_t pid;
    int fd=SENTINEL;

    printf("About to run who into a file (in a strange way!)\n");
    if ( (pid=fork())== SENTINEL){
        perror("fork"); exit(1);
        }

    if ( pid == 0 ){  // child
        close(1);
        fd=creat("userlist", 0644);
        execlp("who","who",NULL);
        perror("execlp");
        exit(1);
        }
    if ( pid != 0){  // parent
        wait(NULL);
        printf("Done running who - results in file \"userlist\"\n");
        }
}
```

# Running the program

```
antoulas@sazerac:~/src-set005$
antoulas@sazerac:~/src-set005$ ./demo-trick
About to run who into a file (in a strange way!)
Done running who - results in file "userlist"
antoulas@sazerac:~/src-set005$
antoulas@sazerac:~/src-set005$
antoulas@sazerac:~/src-set005$ more userlist
ad         :0           Apr  2 15:43 (:0)
ad         pts/8        Apr  3 15:34 (:0.0)
ad         pts/0        Apr  3 01:54 (:0.0)
ad         pts/6        Apr  3 10:08 (:0.0)
ad         pts/9        Apr  3 15:34 (:0.0)
ad         pts/10       Apr  3 15:34 (:0.0)
ad         pts/11       Apr  3 15:34 (:0.0)
antoulas@sazerac:~/src-set005$
```

# Reading from *Multiple Pipes*




- Server should be able to deal with the situation where there is data present on more than one pipes.
- If nothing is pending, the server should block.
- If data arrives in one of more pipe, server has to become aware of where the information is to receive it.

```
#include <sys/select.h>
int select(int nfds, fd_set *readfds, fd_set *writefds,
                     fd_set *exceptfds, struct timeval *timeout);
```

# “Limitations” of Pipes

Classic pipes have at least two drawbacks:

- Processes using pipes must share **common ancenstry**.
- Pipes are **NOT** permanent (persistent).

# Named Pipes (FIFOs)

- The `FIFOs` ("**named pipes**") address above defficiencies.
  - FIFOs are permanent on the file system
  - Enable the first-in/first-out one communication channel.
  - `read` and `write` operations function similarly to pipes.
  - A FIFO has an owner and access permissions (as usual).
  - A FIFO can be opened, read, written and finally closed.
  - A FIFO **cannot be** seeked.
  - Blocking and non-blocking versions use `<fcntl.h>`
    $\rightarrow$ why non-blocking FIFOs??

# Creation of FIFOs (System Program & API)

- System program: `/bin/mknod nameofpipe p`
  - `nameofpipe` name of FIFO.
  - Note the `p` parameter above and the `p` in `prw-r--r--` below:

```
antoulas@sazerac:~/Set005/src$ mknod kitsos p
antoulas@sazerac:~/Set005/src$ ls -l kitsos
prw-r--r-- 1 antoulas antoulas 2010-04-22 16:58 kitsos
antoulas@sazerac:~/Set005/src$ man mkfifo
```

- `int mkfifo(const char *pathname, mode_t mode)`
  - `pathname`: whete the FIFO is created on the filesuystem
  - included files:

```
#include <sys/types.h>
#include <sys/stat.h>
```

  - `mode` represents the designated access permissions for: owner, group, others.

# A simple client-server application with a FIFO

→ *server program:* `receivemessages.c`

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

#define MSGSIZE 65

char *fifo = "myfifo";

main(int argc, char *argv[]){
        int fd, i, nwrite;
        char msgbuf[MSGSIZE+1];

        if (argc>2) {
                printf("Usage: receivemessage & \n");
                exit(1);
                }

        if ( mkfifo(fifo, 0666) == -1 ){
                if ( errno!=EEXIST ) {
                        perror("receiver: mkfifo");
                        exit(6);
                        }
                }
```

# The server program: receivemessages.c

```
        if ( (fd=open(fifo, O_RDWR)) < 0){
                perror("fifo open problem");
                exit(3);
                }
        for (;;){
                if ( read(fd, msgbuf, MSGSIZE+1) < 0) {
                        perror("problem in reading");
                        exit(5);
                        }
                printf("\nMessage Received: %s\n", msgbuf);
                fflush(stdout);
                }
}
```

# The client program: sendmessages.c

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <error.h>
#define MSGSIZE 65
char *fifo = "myfifo";

main(int argc, char *argv[]){
    int fd, i, nwrite;
    char msgbuf[MSGSIZE+1];

    if (argc<2) { printf("Usage: sendamessage ... \n"); exit(1); }
    if ( (fd=open(fifo, O_WRONLY| O_NONBLOCK)) < 0)
        { perror("fife open error"); exit(1); }

    for (i=1; i<argc; i++){
        if (strlen(argv[i]) > MSGSIZE){
            printf("Message with Prefix %.*s Too long - Ignored\n",10,argv[i]);
            fflush(stdout);
            continue;
            }
        strcpy(msgbuf, argv[i]);
        if ((nwrite=write(fd, msgbuf, MSGSIZE+1)) == -1)
            { perror("Error in Writing"); exit(2); }
        }
    exit(0);
}
```

# Running the client-server application

→ Setting up the *server*:

```
antoulas@sazerac:~/Set005/src$ ./receivemessages &
[1] 2662
antoulas@sazerac:~/Set005/src$
```

→ Running the *client*:

```
antoulas@sazerac:~/Set005/src$ ./sendmessages Nikos
antoulas@sazerac:~/Set005/src$ ./sendmessages Nikos "Alexis Delis"
antoulas@sazerac:~/Set005/src$ ./sendmessages "Alex Delis" "Apostolos
     Despotopoulos"
antoulas@sazerac:~/Set005/src$ ./sendmessages Thessaloniki+Komotini+Xanthi+
     Kavala kkkkkkkkkkkkkkkkkkkkkkkkk....kkkkkkkkkk
antoulas@sazerac:~/Set005/src$
```

→ Observing the behavior of the *server*:

```
antoulas@sazerac:~/Set005/src$
Message Received: Nikos
Message Received: Nikos
Message Received: Alexis Delis
Message Received: Alex Delis
Message Received: Apostolos Despotopoulos
Message Received: Thessaloniki+Komotini+Xanthi+Kavala
Message with Prefix kkkkkkkkkk Too long - Ignored
```