

Interprocess communication

- in Operating systems we find there are a number of mechanisms used for interprocess communication (IPC)
- the IPC mechanisms can be divided into two groups, those which work well using shared memory and those which work with non shared memory
- some common methods of IPC are: sockets, semaphores and mailboxes
- sockets and mailboxes are normally used by non shared memory programs
 - ie client and server on different machines

Interprocess communication in shared memory systems

- semaphores are more appropriate for multiple processes sharing some common memory
- we will be covering a semaphores and message passing after networking with sockets
- message passing
 - can be used in shared memory systems

Interprocess communication in non shared memory systems

- network sockets (Berkeley and System V Transport Layer Interface)
 - work well with programs (clients and servers) which do not share the same memory

- message passing
 - can be used in non shared memory systems

Berkeley Sockets

- the Berkeley interface to sockets ultimately gives the programmer a file descriptor on both client and server which can be both read from and written to
- this is elegant as the user application can map its functionality onto basic file primitives: read, write
- Berkeley sockets are available in many languages and available on most operating systems

Berkeley Sockets



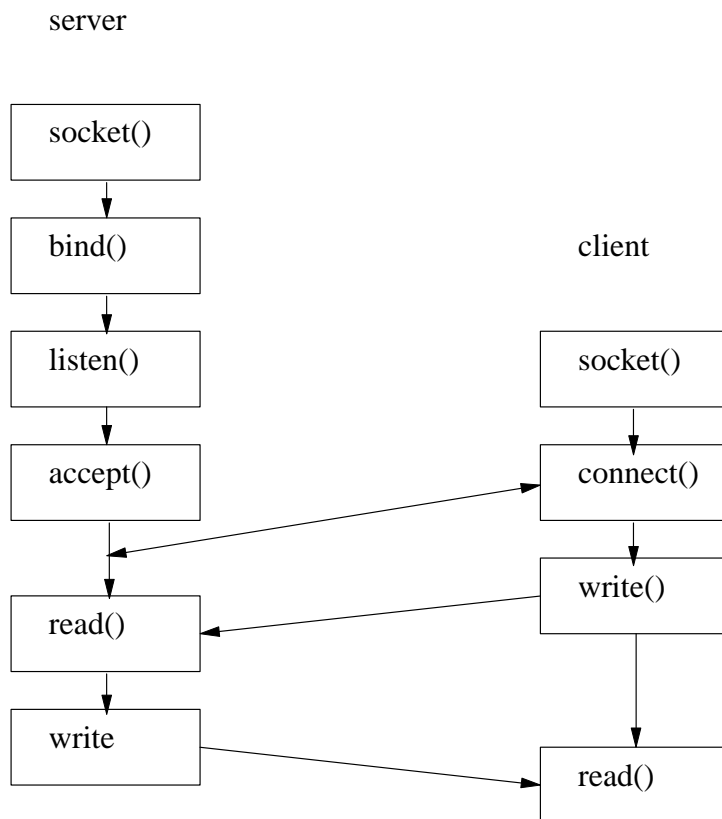
Program	Description	Function
server	create end point	socket ()
	bind address	bind ()
	specify queue	listen ()
	wait for connection	accept ()
client	create end point	socket ()
	bind address	bind ()
	connect to server	connect ()

Berkeley Sockets



Program	Description	Function
	transfer data	read() write() recv() send()
	datagrams	recvfrom() sendto()
	terminate	close() shutdown()

Connection oriented sockets (TCP sockets)



Consider Python Code for a TCP Server

tcpserver.py

```
#!/usr/bin/python

from socket import *
myHost = ""
myPort = 2000

# create a socket
s = socket(AF_INET, SOCK_STREAM)
# bind it to the server port number
s.bind((myHost, myPort))
# allow 5 pending connections
s.listen(5)

while True:
    # wait for next client to connect
    connection, address = s.accept()
    data = connection.recv(1024)
    while data:
        connection.send("echo -> " + data)
        data = connection.recv(1024)
    connection.close()
```


Consider Python Code for a TCP client



tcpclient.py

```
#!/usr/bin/python

import sys
from socket import *
serverHost = "localhost"
serverPort = 2000

# create a TCP socket
s = socket(AF_INET, SOCK_STREAM)

s.connect((serverHost, serverPort))
s.send("Hello world")
data = s.recv(1024)
print data
```

Testing the code

- open up an editor and type in the server Python code
- save it as `tcpserver.py`
- now open up a terminal and type
- ```
$ python tcpserver.py
```
- make a note of the FQDN of the server

## Testing the code

- open up another editor and type in the client Python code
- save it as `tcpclient.py`
- open up a terminal

## Testing the code

- `$ python tcpclient.py`
- notice that both client and server are working on the same machine

## Testing the code

- change the variable `serverHost` in `tcpclient.py` to the FDQN of your neighbours machine
  - and run your client again!

## Application protocol using TCP

- TCP is used by many application level protocols
  - a very common one is http
- let us build a tiny web server in Python!

# Tiny web server in Python



`mywebserver.py`

```
#!/usr/bin/python

from socket import *
myHost = ""
myPort = 2000

create a socket
s = socket(AF_INET, SOCK_STREAM)
bind it to the server port number
s.bind((myHost, myPort))
allow 5 pending connections
s.listen(5)
```

# Tiny web server in Python

mywebserver.py

```
while True:
 # wait for next client to connect
 connection, address = s.accept()
 data = connection.recv(1024)
 while data:
 reply = """HTTP-Version: HTTP/1.0 200 OK
Content-Length: 3012
Content-Type: text/html

<p>Hello world!</p>
<body>
"""
 connection.send(reply)
 data = connection.recv(1024)
 connection.close()
```



## Testing your web server

- open up a terminal and run

- `pythonmywebserver.py`

- now open up a browser and enter the url `<http://localhost:2000>`

- you should now have a start of a tiny web server

## Testing your web server

- we can see that a socket is created to give us access to manage the TCP port 2000
- in turn the program will read from the socket and form a http response
  - which is sent back to the client which renders the html after stripping it from the http packet

## UDP sockets

- we can also produce a UDP client and server
  - these are functionally different to TCP servers, despite the similarity between the Python code implementation

## UDP server

```
#!/usr/bin/python

from socket import *
myHost = ""
myPort = 2000

create a UDP socket
s = socket(AF_INET, SOCK_DGRAM)
bind it to the server port number
s.bind((myHost, myPort))

data, address = s.recvfrom(1024)
while data:
 print "UDP server:", data, "from", address
 s.sendto("echo -> " + data, address)
 data, address = s.recvfrom(1024)
```

# UDP client

udpclient.py

```
#!/usr/bin/python

import sys
from socket import *
serverHost = "localhost"
serverPort = 2000

create a UDP socket
s = socket(AF_INET, SOCK_DGRAM)

s.connect((serverHost, serverPort))
s.send("Hello world")
data = s.recv(1024)
print data
```