ETEC3702 – Concurrency

Inter-Process Communication: Distributed Systems

https://youtu.be/F8dl07PtUlQ

IPC for Distributed Systems

Communicating Between Processes:

We have looked at a variety of ways of getting processes to communicate:

Queue Objects – We've seen these before with processes. Typically a FIFO.

Pipes – A two-ended communication mechanism that allows inter-process communication.

Value / Array Objects – Simple data objects that can be shared between processes.

Manager Objects – Special data objects that can be accessed by multiple processes.

Shared Memory – Shared memory blocks that can be accessed by name.

Each of these methods has advantages and disadvantages.

But all of them have a short-coming: The processes must be on the same OS / machine. This makes them not suitable for distributed / cluster-based parallel computing systems.

IPC for Distributed Systems

In other words, **none** of the previously discussed methods can easily allow a process on one system exchange data with a process on another system.

To achieve this, we need to use a different method.

There are several options available:

Standard Methods:

sockets – This is the standard sockets implementation for network programming.
multiprocessing.connection objects – An abstraction that is built upon sockets.

Libraries / Modules:

mpi4py – Python implementation of standard MPI (Message Passing Interface).

Pyro4 – A Python “remote object” library. Allows remote invocation of functions.
ray – A powerful Python module for remote and distributed application development.
others – There are many other libraries and modules for distributed systems.

IPC for Distributed Systems

Sockets

Sockets are one of the most widely used network communication methodologies.

Sockets is a library that can make use of numerous underlying network protocols such as: IPv4/IPv6(TCP, UDP), UNIX UDS, CAN, PACKET, etc.

Sockets also supports both stream-based and datagram-based communication.

Web browsers / servers are built using sockets programming.

Since the internet is largely built upon TCP/IP, we will briefly discuss how sockets can work with communicating data across a stream connection.

IPC for Distributed Systems

Sockets Programming

A **socket** can be thought of as a connection to the networking sub-system on a machine.

Before we can do anything else, we must create a socket:

```
import socket
# create an INET, STREAMing socket
s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

The two parameters are used to specify the “address family” and the “connection type” of the socket.

Here we’re using a socket with IP addressing and a TCP stream connection.

IPC for Distributed Systems

Sockets Programming

Once the socket has been opened, we can use it to connect to another process.

The process may be on the same machine we’re on, or on another machine somewhere else on the internet.

```
import socket

# create an INET streaming socket
s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Try connect to the process listening on remote port 80
s.connect(("www.yostlabs.com", 80))
```

Here we specify the name (or IP address) and port number of the machine we’d like to connect to.

IPC for Distributed Systems

Sockets Programming

For this to work, a process has to be listening on that machine / port.

The listening machine is called a “server” and the connecting machine is called a “client”.

The server code is a little more complex:

```
import socket

# create an INET streaming socket
serverSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# bind the socket to this machine and some port number.
serverSocket.bind((socket.gethostname(), 80))
# begin listening for incoming connections.
serverSocket.listen(5)
```

Now that we’re listening, we can accept incoming connections...

IPC for Distributed Systems

Sockets Programming

Now we can enter a loop that accepts connections.

```
import socket

# create an INET streaming socket
serverSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# bind the socket to this machine and some port number.
serverSocket.bind((socket.gethostname(), 80))
# begin listening for incoming connections.
serverSocket.listen(5)
while True:
    # accept a connection when one comes in.
    (clientSocket, clientAddress) = serverSocket.accept()
    # Now do something with the clientSocket
    # data can be exchanged with
    #     clientSocket.send() and clientSocket.recv()
    # Once the conversation is complete we can use
    #     clientSocket.close()
```

IPC for Distributed Systems

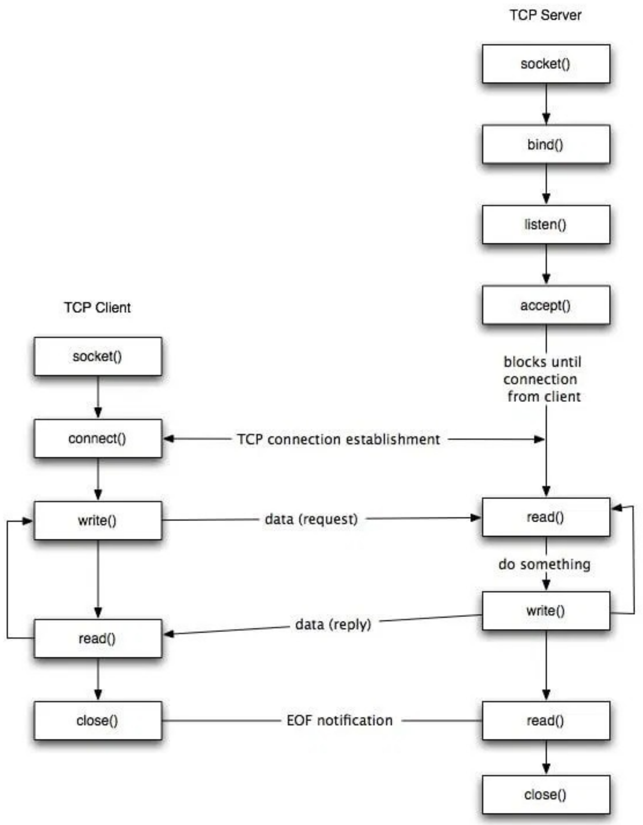
Sockets Programming

Note: for this to work, one machine must be listening (the server) and the other must be connecting (the client).

This means that we must take this into consideration when designing a distributed system that uses sockets like this.

Also note: There can be one server that handles multiple client connections.

On the following page is a complete example.



IPC for Distributed Systems

Complete Example:

```
#client side
import socket
mySocket=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
mySocket.connect(("127.0.0.1",0x1234))
data=mySocket.recv(2048)
print("data:",data.decode("latin"))
print("exiting.")
```

127.0.0.1 is local-host.
0x1234 is server’s port number

```
#server side
import socket
serverSocket=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
serverSocket.bind(("127.0.0.1", 0x1234))
serverSocket.listen(5)
while True:
     print("Waiting for a connection.")
     (clientSocket, clientAddress) = serverSocket.accept()
     print("connection from client:",clientAddress)
     clientSocket.send("Hello!".encode("latin"))
     clientSocket.close()
#never get here
```

127.0.0.1 is local-host.
0x1234 is a selected port number

“latin” encoding is Python3’s way to encode ASCII strings.

IPC for Distributed Systems

Sockets: Some Notes / Details

send() and recv() work on data streams.

When socket.send() is called it will return the number of bytes successfully sent. This number may not be all of the bytes of the message so send() may have to be called repeatedly for large messages.

Whe socket.recv() returns a block of data, it may not be all of the data from a given send() and it may be less than the number of bytes specified. Thus, recv() returns the number of bytes actually received. Those bytes may be an incomplete message or may contain additional bytes from a following send.

Since TCP does guarantee reliable transfer, when send() or receive() return a number of bytes sent / received, it is guaranteed that the transaction was successful.

IPC for Distributed Systems

Sockets: Final Details

Note that the server creates a new socket to talk to the connecting client.

This leaves the original listening socket open to accept other connections.

It is common to spawn a thread or process to handle each active socket connection so that the server is free to handle other connections concurrently.

This would be done in the server’s loop something like this:

```
while True:
    #accept a connection
    (clientSocket, clientAddress) = serverSocket.accept()
    #create a thread for this server
    cThread = threading.Thread(target=serverThread,args=(clientSocket,clientAddress))
    cThread.start()
```

IPC for Distributed Systems

Sockets: Dealing with Stream Data

Note that send() and recv() only deal with streams of byte data.

This means that we have to write code to ensure the correct length was sent and received.

This also means that if we want to send arbitrary data across a socket, we have to encode the data is some way.

To encode program data to be sent across a stream, we must serialize the data, we have a number of options:

binary – Binary formatted data. Use byte arrays, binary arrays, pack / unpack.
JSON – JavaScript Object Notation serialization. Used extensively on the web.
Pickle – Python's standard serialization module.
XML – Data organized using hierarchical nested tag structures.
MsgPack – A fast / compact cross-platform multi-language serialization library.
custom – We could write our own serialization encoding / decoding system.

IPC for Distributed Systems

Sockets: Dealing with Stream Data

The way to use these different methods varies by which you are using, but several share some commonalities.

JSON – JavaScript Object Notation serialization. Used extensively on the web.

Pickle – Python’s standard serialization module.

MsgPack – A fast / compact cross-platform multi-language serialization library.

All three of these can be used in a similar way:

dumps() - a method takes an object and converts it into a serialized version.

loads() - a method takes a serialized version of the data and converts it into an object.

This works for most basic data-types as well as container types like lists and dictionaries.

IPC for Distributed Systems

Sockets: Dealing with Stream Data

So, for example:

To serialize for transfer:

```
serializedData = json.dumps(someObject)
serializedData = pickle.dumps(someObject)
serializedData = msgpack.dumps(someObject)
```

To de-serialize upon receive:

```
someObject = json.loads(serializedData)
someObject = json.loads(serializedData)
someObject = json.loads(serializedData)
```

Note that there are some restrictions, but in general, this allows us to pass objects between processes.

IPC for Distributed Systems

Sockets: Dealing with Stream Data

Thus, the socket process ends up looking like this:

Sender → serialize → socket.send() → [Network] → socket.recv() → de-serialize → Receiver

Note that the sender and receiver can be anywhere: same machine, same cluster, remote machine on the internet, etc.

Also note that there are some details that are missing from above to ensure that complete messages have been sent/received.

It is also important to understand that this additional processing adds overhead.

Let’s take a quick look at some of those serialization modules...

IPC for Distributed Systems

Connection Objects: A simpler approach.

The multiprocessing module offers a slightly simpler approach to sending and receiving objects: **multiprocessing.connection**

Connection objects allow the sending and receiving of picklable objects or strings. They can be thought of as message oriented connected sockets.

Essentially, the connection object will automatically serialize and de-serialize objects and will automatically ensure that the data has been sent in its entirety.

So, it is doing some of the work for us.

We've actually seen connections like this in action when we used Pipe().

Connection objects will look familiar...

IPC for Distributed Systems

Connection Object Methods

send(obj) - Send an object to the other end of the connection which should be read using recv(). The object must be picklable. Very large pickles (approximately 32 MiB+, though it depends on the OS) may raise a ValueError exception.

recv() - Return an object sent from the other end of the connection using send(). Blocks until there is something to receive. Raises EOFError if there is nothing left to receive and the other end was closed.

close() - Close connection. Called automatically when the connection is garbage collected.

poll([timeout]) - Return whether there is any data available to be read. If timeout is not specified then it will return immediately. If timeout is a number then this specifies the maximum time in seconds to block. If timeout is None then an infinite timeout is used. Note that multiple connections may be polled at once by using multiprocessing.connection.wait().

IPC for Distributed Systems

Connection Object Methods (cont.)

fileno() - Return the file descriptor or handle used by the connection.

send_bytes(buffer[, offset[, size]]) - Send byte data from a bytes-like object as a complete message. If offset is given then data is read from that position in buffer. If size is given then size bytes will be read from buffer. Very large buffers may raise a ValueError.

recv_bytes([maxlength]) - Return a complete message of byte data sent from the other end of the connection as a string. Blocks until there is something to receive. Raises EOFError if there is nothing left to receive and the other end has closed. If maxlength is specified and the message is longer than maxlength then OSError is raised and the connection will no longer be readable.

recv_bytes_into(buffer[, offset]) - Read into buffer a complete message of byte data sent from the other end of the connection and return the number of bytes in the message. Blocks until there is something to receive. Raises EOFError if there is nothing left to receive and the other end was closed. buffer must be a writable bytes-like object. If offset is given then the message will be written into the buffer from that position. Offset must be a non-negative integer less than the length of buffer (in bytes). If the buffer is too short then a BufferTooShort exception is raised and the complete message is available as e.args[0] where e is the exception instance.

IPC for Distributed Systems

Connection Objects: Listeners and Clients

There are two classes that we can use to facilitate communication over Connection objects: Listener and Client.

Listener – This class waits to accept connections on some address / port. It has the ability to automatically use authentication to improve security. A Listener can be thought of as a connection server.

Client – This class connects to a corresponding Listener to establish a connection. Once the connection is established we can send / receive objects and data through the Connection object.

Let’s look at some example code...

IPC for Distributed Systems

```
#Listener Example
import multiprocessing.connection
address=('127.0.0.1',0x1234)
listener=multiprocessing.connection.Listener(address,authkey=b'secret')
while True:
    print("waiting for connection.")
    connection=listener.accept()
    print("Connection accepted from:",listener.last_accepted)
    playerData=connection.recv()
    print("Player Data Received:",playerData)
    connection.close()
```

```
#Client Example
import multiprocessing.connection
playerData={'name':'player1','x':100,'y':200,'health':100,'speed':0,'angle':90}
address=('127.0.0.1',0x1234)
connection=multiprocessing.connection.Client(address,authkey=b'secret')
print("Connected to:",address)
connection.send(playerData)
print("Player Data Sent. Closing connection.")
connection.close()
print("Done")
```

IPC for Distributed Systems

Listener Output:

```
waiting for connection.
Connection accepted from: ('127.0.0.1', 49162)
Player Data Received: {'name': 'player1', 'x': 100, 'y': 200, 'health': 100, 'speed':
0, 'angle': 90}
waiting for connection.
```

Client Output:

```
Connected to: ('127.0.0.1', 4660)
Player Data Sent. Closing connection.
Done
```

IPC for Distributed Systems

Connection Objects: Listeners and Clients

Note that using connection objects with listeners and clients can greatly simplify the exchange of data between two processes.

This can be used with two processes on the same machine or two process on different machines in different parts of the world.

Note that the object must be “pickle-able” to be sent using send / recv. Objects that aren’t pickle-able must be converted into buffers of be converted into some other data-representation before being transferred.

While Connections are doing the serialization / deserialization for us, there may still be advantages to doing it yourself using one of the other methods.

If you do use one of the other methods (like JSON or MsgPack) you can still use the Connection object’s sendbuffer() and recvbuffer() to simplify the socket interface.

IPC for Distributed Systems

Other Options:

There are other options for distributed computing.

Mostly they are in the form of other libraries and frameworks that use sockets underneath the hood.

Some of these are:

mpi4py – Python implementation of standard MPI (Message Passing Interface).
Pyro4 – A Python “remote object” library. Allows remote invocation of functions.
ray – A powerful Python module for remote and distributed application development.
others – There are many other libraries and modules for distributed systems.

The advantage of MPI is that it is a standard and allows communication with other distributed system. Pyro makes accessing remote function and method calls quite seamless. Ray offers a wide range of distributed data options.

IPC for Distributed Systems

No more things for today.

Stay safe!