Distributed Systems – Lecture Notes

Amnir Hadachi, Artjom Lind, Eero Vainikko

Contents

Introduction

These are the Lecture Notes for the course LTAT.06.007 Distributed Systems. The chapters appear in order to support learning the basic concepts of network programming and distributed systems. The aim is to give practical guidance and working examples for participants of the course to gain practical knowledge on how to build distributed applications.

Chapter 1

Introduction to Distributed Systems

Examples of Distributed Systems can be found everywhere: for example in the areas of finance and commerce, in applications to support the information society, creative industry and entertainment, healthcare, education, transport and logistics, science, environmental management *etc.* Here we will start with discussing the main properties of distributed systems as well as main challenges that one faces when starting dealing with them.

What is a distributed system?

Definition 1. A distributed system is one in which components located at networked computers communicate and coordinate their actions only by passing messages.

Definition 2. A distributed system consists of a collection of autonomous computers linked by a computer network and equipped with distributed system software. This software enables computers to coordinate their activities and to share the resources of the system hardware, software, and data.

When talking about a particular distributed systems architecture one might want to ask three key questions:

1. **What are the components?** Components in disributed systems are usually called nodes. First thing to ask is how many of what type and what purpose nodes are there? (For example computing nodes, special devices, sensors, network devices etc.)
   The nodes are connected with links - the components for information transportation.

2. **How do the nodes communicate?** Here come the communication protocols, communication technology. Also, lot of the functional properties of a distributed system depend on link speeds (latency and throughput). To make the communication to happen there is a need for communication software.

3. How **the coordination between the components** is achieved? Here the usual aim is to achieve as reliable system as possible. For this quite a few issues need to be solved, like **fault-tolerance**, **security**, well-defined **resource sharing policies** to avoid **dead-locks**, **guaranteed delivery** of services etc.

Sharing resources

All distributed systems involve certain resources to be shared between the components with the end users. What are these resources?

The resources can be **Hardware** (computers, servers, computer clusters, laptops, tablets, smartphones and smart-watches, Internet of Things (IoT) devices, sensors, data bandwidth on routes between nodes, CPU cycles etc.) Another group of resources can be described with the common term – **Data**. This involves data stored in databases, storage devices etc. But this includes also all kind of software together with its development (both proprietary as well as community-developed open source software.)

Note though that not every single resource is meant for sharing. But for those resources that are shareable we have to say that different resources are handled in different ways. However, some generic requirements can be outlined.

To be able to access some resource one needs to be able to name it. This means, we need a **(i) namespace** for identification of a group of resources. In order to get it working in a distributed environment there should be a mechanism for **(ii) name translation to network address**. And of course, in case of multiple access, possibly in a competing setup, well-defined **(iii) synchronization** is needed to avoid unconsistencies, dead-lock and starvation situations.

How to characterize a distributed system?

From above we can conclude, that distributed systems are characterized by **(a) concurrency of components**, meaning that all parts of a distributed system are independent and at the same time are competing for shared resources. The second feature that can be outlined is the **(b) lack of a global clock**. This means, that each processor on each node is running on its own clockrate and there is no synchroniztion of CPU clocks. Independence of the components yields also **(c) independent failures of components**, meaning that the design of a distributed system needs to take into account the fact that whichever node can crash at whichever time or become unavailable because of broken communication. This leads to a famous quote by Leslie Lamport: "*A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.*"

What makes distributed systems challenging?

One can easily name hundreds of challenges related to distributed systems' design and operation. We cover here only the following challenges: Scalability, Openness, Security, Heteroheneity of Components, Failure handling, Concurrency of components, Transparency and Providing quality of service.

Scalability

Consider an emerging new web-service, which is being developed while it still wasn't known and popular with users. Now say, suddenly word spreads around about this fantastic new possibility and the service will start to have 1000% more hits each month from all around the Globe. What can be a remedy? (You may know the answer that replication and caching might help here.) But the property of a distributed system to respond to growing number of requests is called Scalability.

Scalability is the ability to work when the system load or the number of users increases.

With designing and maintainig scalable distributed systems there exist a number of challenges:

(1) How to control the cost of physical resources? – one could build a perfect system with an unlimited amount of resources while, on the other hand, one could always try to minimise the cost as much as possible with taking certain risks. As one cannot exactly predict the future, the answer here would be flexibility. Like in elastic cloud systems nowadays.

(2) How to control performance loss? This is one of the main aims in designing reliable and lively distributed systems. General idea is to spread the load around in the system dynamically.

(3) Preventing software resources running out. For example, IPV4 32-bit Internet addresses were predicted to run out already around the Millennium time.

(4) Avoiding performance bottlenecks. This is actually one of the main concerns with the challenge (2). Careful planning and designing of the system is the way to go.

Openness

When talking about an open computer system the key question to ask is: ***(A) Can the system be extended and reimplemented in various ways?*** The main criteria to be able to answer "yes" to this question is that the key interfaces need to be published.

One can view a distributed system as a collection of independent computer systems that work together to fulfill a common goal. Therefore, with an open distributed system one should ask in addition to question (A) the following: ***(B) Can new resource-sharing services be added and made available for use by variety of client programs?*** To achieve this one would need (B1) Uniform communication mechanism and (B2) Published interfaces to shared resources.

Security

Security issues are gradually becoming more and more prevailing with computer systems in distributed environements making their way often even into the daily news headlines. This is due to real values are involved in the system, like bank transfers in digitalized transactions being moved over the networks connecting certain dedicated servers or credit card payments for goods or services by real users. This makes it an attractive target for hackers trying to interfare with the processes with the aim of earning or stealing some money, or some political reasons, or just out of the fun of hacking by itself. Common term for all security-related questions in distributed systems is **Cybersecurity**.

The main three terms that are the building blocks in distributed systems' security are:

Confidentiality – Protection against disclosure to unauthorized individuals – the techniques on how to conceal third parties from being able to access the actual content of the information even when being able to possess a copy of the data in an encrypted form. Key techniques here are (symmetric and unsymmetric) cryptographic algorithms providing the access to the original content of a message only by having corresponding credentials.

Integrity – Protection against alteration or corruption – ensuring that messages have not been changed or corrupted while travelling in an open network. The common technique is to use digital signatures based on public key encryption algorithms for proof of integrity.

Availability – Protection against interference with the means to access the resources – an effort to avoid denial of access and denial of service to shared resources in a distributed system.

Cybersecurity is a discipline by its own dealing with a big variety of attacks and protection techniques to avoid them. Some of the concepts are well-provable with relevant mathematical approaches while some are in a state of a conjecture with no known proof of existence of the opposite claim (like the conjecture, that the only way to find prime number multipliers of a big number is to use the method of trial-and-error – the basis of RSA algorithm for public-private key encryption systems.)

Heterogheneity of Components

With distributed systems we can tell two things for sure: 1. each of the nodes has some connection to at least one other node in the system and 2. each node has at least one CPU connected to the network interface running certain control program to be able to provide the ability of communication with the rest of the distributed system. All the rest might be (and usually is) difficult to define with common claims due to huge variety in hardware and software resources, operating systems, network connections, programming languages used for creating the needed functionality *etc.* All this contributes to the challenge of creating an operative environment with huge heterogeneity within the system components, connections and their control. An adopted way to handle heterogeneity is to use **middleware** - software layer providing programming abstraction while at the same time masking heterogeneity of underlying network technologies, pieces of hardware in use, operating systems, and bundles of software.

Failure handling

In an ideal World there would not be any failures at all. But with distributed systems we are talking about real thing developed by people, usually many people in collabration and often even spread around different locations around the Globe. Therefore one has to face the situations created by failures that have already occurred, but much better: to try to forecast all possible failure conditions to be able to prevent them before they have had their chance to show up. Developing a distributed systems is not just programming. It is about designing and playing through different scenarios with all kind of exceptions taken into account with the thought in mind that when nothing woks – how to still get the work done in the end?

There are certain techniques for dealing with failures. As the first measure one has to be able to **detect the failures**. Detected failures can then be **masked** to allow taking relevant action to handle them. With distributed systems one always has to take into account possibility of messages not going through due to network overloads and downtime. Therefore one has to design scenarios for message retransmission on the need. More serious cases are the ones with a crucial node itself crashing or becoming unavailable. One can consider **replication** of key services, databases, resources for the replica being able to take over the responsibilties as soon as such need occurs.

Therefore, one of the key distributed systems' properties as well as challenges is their ability of **tolerating failures**. When designing a system one might want to ask – what is the level of failure situations that the system is still able to **recover** from? The main goal is to achieve **high availability** – measure of the proportion of time that the resource is available for use.

Concurrency of components

As soon there are some resources to share between multiple parties there is built in a question of how to handle situations when there are more than one counterpart competing for the same resource simultaneously. It adds a new order magnitude to the complexity. In such cases there exist possibilities for dead-lock situations. But if one is not careful enough, there might also occur possibilities for unfair treatment towards certain

counterparts and possibilties for their starvation. A classical illustration to this is the example of Dining Philosophers.

In the case of distributed systems the challenge from the possibility of messages getting delayed or dropped adds an additional level of complexity. For a well-designed systems the proof of no deadlock situations as well as fair handling of shared resources' requests becomes therefore much more complex.

Transparency

While a user connects to the Internet with say, a request for certain information about different possibilities to travel from Tartu to say, Venice, [s]he does not want to know all the details about the underlying service details and sub-requests from different airlines, bus or train services etc. that happen seemlessly in the background. The user wants the answer to appear on her/his device screen and as quickly as possible. The service is **transparent** to the user in a way that there is no need to realize that there is a distributed system underneath serving the request and composing the answer.

Transparency - Concealment from the user and the application programmer of the separation of components in a Distributed Systems for the system to be perceived as a whole rather than a collection of independent components.

One can say that there exist **access transparency** - the user percieves (accesses) the distributed system as it were a single node instead of (possibly complex) set of queries between different nodes. Also, the user does not care where exactly the piece of information came from - **location transparency** - nor does the user care that accessed resource physical location got recently changed from a server in US to Ireland - **mobility transparency**. Also, usually the user does not want to get a message telling that the query was already n-th such a request at the very moment and it got served as the second in the queue – no, the user just needs the information – with the **concurreny transparency**. Likewise, the user need not to know that the piace of information came from a certain cache of got served by the server replica number X - **replication transparency**. Either, in case of any errors that might have occurred the user would like the system to get recovered instantly without any substantial delays or error messages - **failure transparency**.

All the above mentioned challenges are together part of the final challenge we cover here:

Providing quality of service

Main nonfunctional properties of distributed systems that affect **Quality of Service (QoS)** are **Reliability**, **Security**, and **Performance**. With these properties having been met by the design and implementation, there is still possibility for the system loosing its original functionality due to changing system loads from increasing volume of requests as well as needs for changes in the system architecture itself to adapt to changing requirements. This is described as the system **adaptability**.

Chapter 2

Introduction to Computer Networking

Introduction

In this chapter we will introduce IP network and learn basics of IP communications. We start refreshing our knowledge of OSI model, make sure we remember what Physical Layer, Link Layer, Network and Transport layer standing for. Next we will discuss the term "protocol" and try to relate it to OSI model, making sure we understand the protocols can be related to different Layers of OSI model.

OSI Model and Protocols

1. ***OSI*** *model is a conceptual model, which characterizes and standardizes the communication functions of any computing system or telecommunication without regard to the technology involved or their internal structure.*

The model is divided into abstraction layers and they are as follows:

7. Application layer: Network Process to Application [Serves as the window for users and application processes to access the network services]

6. Presentation layer: Data Representation and Encryption [Formats the data to be presented to the Application layer. It can be viewed as the Translator for the network]

5. Session layer: Inter-host Communication [Allows session establishment between processes running on different stations]

4. Transport layer: End-to-End Connections and Reliability [Ensures that messages are delivered error-free, in-sequence, and with no losses or duplications]

3. Network layer: Path Determination and IP (Logical Addressing) [Controls the operations of the subnet, deciding which physical path the data takes]

2. Data link layer: MAC and LLC (Physical Addressing) [Provides error-free transfer of data frames from one node to another over the Physical layer]

1. Physical layer: Media, Signal, and Binary Transmission [Concerned with the transmission and reception of the unstructured raw bit stream over the physical medium]

OSI model was created by ISO and is considered a "reference" model to explain aspects of network communications where multiple protocols and technologies are evolved. OSI model is however not a standard that all the protocols are following.

2. *Communication protocol is a set of rules allowing two or more endpoints to exchange information. Rules defining the way the user data is broken down into data units*

3. *Protocol data unit unique peace of information delivered among the endpoints. Data unit is considered atomic (unbreakable) in the scope of protocol.*

Data unit may contain control information (header) or the user data (payload). In a layered network, the data unit of a higher layer becomes a payload of lower layer's data unit. Moreover, big data units of higher layer may be broken down and assigned as a payload to multiple data units of lower layer. Example of TCP/IP running on Ethernet: IP packet (Layer 3) on top of multiple Ethernet frames (Layer 2).

Data units:

- Data: higher abstraction layers (application, presentation, session)
- Segment: transport layer
- Packet: networking layer
- Frame: data link layer
- Bits: physical layer

Data movement between layers of the OSI model:

- Encapsulation:
- De-encapsulation

4. *TCP/IP model (simplified) derived from OSI model, however is only focusing on aspects of TCP/IP. For example session layer here is absent as the sessions in TCP/IP might be organized by transport layer (TCP) or application layer (UDP).*

1. Application layer
2. Transport layer
3. Internet layer
4. Network interface (Link) layer

Question: Let's remember a bit of Network Technology and try to organize the following protocols according to TCP/IP model: ARP, USB, RS-232, IEEE 802.11 , PPP , IPX , AppleTalk , IGMP , HTTP , FTP , IMAP, SMTP

What is a socket?

5. *A socket is one end-point of a two-way communication link between two programs or applications running on the network.*

In the scope of operating system the socket is a descriptor (identified by unique number).

6. *A descriptor is an abstract indicator (a unique number) assigned by the OS to any resource currently opened for I/O operations (read, write or both).*

The operations allowed on socket are therefore similar to file:

- socket() - Creating a bare socket object
- close() - Closing the socket

however additional methods are there on socket:

- bind() - Binding socket to local IP address and port
- connect() - Connecting a local socket to a remote address and port. Is required in case of TCP and is not there for UDP. Can you explain why ? (If not we will anyway discuss it later).
- ... more methods of socket you may find from socket API of Python

and in place of read/write/seek there are:

- `send()`: Send data through connected socket (TCP only)
- `sendto()`: Send data to particular recipient in UDP protocol
- `recvfrom()`: Receive data from particular recipient in UDP protocol

The role of the socket is to allow opening a communication channel that is used by programs or applications to transmit data back and forth locally or across Internet.
Sockets have two primary properties controlling how the transmission of data is done. These properties can be resumed as follows:

1. The address family controls the Open Systems Interconnection (OSI) network layer protocol used
2. The socket type controls the transport layer protocol.

Address family control

In python we have three supported address families:

1. **AF_INET**, which is used for IPv4 Internet addressing.
2. **AF_INET6**, is used for IPv6 Internet addressing.
3. **AF_UNIX** is the address family for Unix Domain Sockets (UDS).

General Knowledge:

- **IPv4** addresses are composed of four octal values separated by dots (e.g. 10.10.10.6). This latter are commonly referred to as IP addresses.
- **IPv6** is the next generation of Internet protocol, which support 128bit addresses.
- **UDS** is an inter-process communication protocol available on POSIX-compliant systems

Socket types

When it comes to socket types usually we have either **SOCK_DGRAM** for user datagram protocol (UDP) or **SOCK_STREAM** for Transmission control protocol (TCP).

Note: Python contains other socket types but they are less commonly used, therefore they are not covered here.

UDP Protocol

7. *User Datagram Protocol (UDP) is a Transport Layer communication protocol relying on IP protocol for addressing the endpoints. It is used for establishing low-latency and loss tolerating (data delivery not guaranteed by UDP) connections between applications on the net.*

The UDP protocol is a message-oriented protocol and it does not need a long-lived connection (there is no dedicated pipes between the endpoints, and each message has to be addressed explicitly). The messages sent via UDP must fit within a single packet and the delivery is not guaranteed. This makes UDP communication very analogical to sending a mail by post without tracking it or delivery notification. In addition the size of the deliverable is limited by maximal datagram size, and it is impossible to send big amounts without fragmentation.

Note: Summarizing features of UDP

- Data send in datagrams, payload of the datagram can not be bigger then the maximal size of the datagram. Bigger blocks of data have to be split manually before sending and assembled back on delivery.[65,535 bytes (8 byte header + 65,527 bytes of data)]
- There is no delivery control, datagrams may be lost. If the delivery needs to be confirm it has to be programmed manually.
- The order of the packets is not preserved during delivery and they may be received in different order. If the order is important (for example for sending fragmented blocks) the sequencing has to programmed manually.
- It is possible to send one packet to many recipients (broadcast sending)
- It is possible to receive from any source by the same socket.
- It is possible to send and receive from the same socket interleaved.

In the next sections, we will step by step implement the simple messaging protocol between 2 endpoints relying on UDP.

UDP Socket

First of all lets make sure we understand the "creating" and "binding" operations of the sockets.

Creating socket

8. *Creating a socket is operation of registering new socket descriptor in OS (like opening a file for reading or writing). As a result of this operation, the OS has a new descriptor in the list of open descriptors.*

Please note that "creating" the socket is like building the mailbox of your home.

Here is an example code showing how to create the UDP socket and how to check what descriptor assigned to it by the OS:

```
# From socket module we import the required structures and constants.
from socket import AF_INET, SOCK_DGRAM, socket
# Sleep function will be used to keep application running for a while
from time import sleep
# And we also will need the process id given by OS when we execute this code
from os import getpid
# Main method
if __name__ == '__main__':
    print 'Application started'
    print 'OS assigned process id: %d' % getpid()
    # Creating a UDP/IP socket
    s = socket(AF_INET, SOCK_DGRAM)
    print 'UDP Socket created'
    print 'File descriptor assigned by OS: ', s.fileno()
    wait_secs = 60*5
    print 'Waiting %d seconds before termination ...' % wait_secs
    sleep(wait_secs)
    print 'Terminating ...'
```

Executing the code should print the following result, please note that letters X and Y will be denoting the process ID and the descriptor respectively and are specific to your environment (each time your run the code it will give different numbers):

```
Application started OS assigned process id: X
UDP Socket created
File descriptor assigned by OS:  Y
Waiting 300 seconds before termination ...
```

Now while the code is running for 5 minutes we have time to refer to the list of registered descriptors of the OS and check if we see our running application process with a corresponding descriptor:

1. in Linux systems the Kernel hold raw process information in folder-like structures (populated by Kernel). So the very primitive folder listing call is enough (the ls command). Issue the following command in shell, make sure in place of X you use the process ID reported by the executed Python code and in place of Y - the descriptor):

```
~$ ls -l /proc/X/fd/Y
```

... in the output you should now see the socket's descriptor that were assigned by the OS to a process number X:

```
lrwx------ 1 user user 64 Aug 18 13:13 Y -> socket:[129069931]
```

2. in MAC systems the unbound sockets are visible by lsof command, so it is enough to print all the UDP descriptors (lsof -i 4udp) and filter the ones assigned to specific process ID (grep X). Issue the following command in shell, make sure in place of X you use the process ID reported by the executed Python code:

```
$ lsof -i 4udp |grep X
```

... in the output you should now see the socket's descriptor that were assigned by the OS to a process number X:

```
Python    2062   user    3u  IPv4 0x90964583fac7dd81      0t0  UDP *:*
```

If you can see the descriptor of a socket you have created in your code, you may continue to the next step of binding a socket. The currently running application (if not terminated after 5 minutes) may be terminated manually (Ctrl+C if running in shell explicitly). You may notice that after the application was terminated all it's descriptors (including our UDP socket) are gone (closed automatically by the OS). Alternatively in your code you may issue close() method on socket before terminating the application (2 last lines of the reference code are modified):

```
    sleep(wait_secs)
    print 'Closing the UDP socket ...'
    s.close()
    print 'Terminating ...'
```

Binding socket

We know already:

1. Sockets represent transport endpoints exposed to an application willing to transfer data over network
2. UDP protocol is a transport protocol on top of IP addressing protocol

We will learn:

1. Transport address or socket address
2. UDP port
3. Binding to random free port
4. Binding to loop-back address only
5. Binding to all addresses

9. *Binding a socket is an operation of assigning a transport address to the socket (like declaring an address of the new mailbox to the post service - so the postman can deliver/send-out mails to/from the mailbox). As a result of this operation, the OS has the UDP socket in the list of bound sockets. Moreover at this point the socket can actually send/receive data and it is visible to remote systems (if there is network connectivity, if socket is not locked by firewall and if remote endpoint is not hidden behind NAT).*

10. *Transport address or socket address is the combination of an IP address and a port number.*

11. *UDP port it is used to help distinguish different user requests and optionaly a checksum to verify that the data arrived intact.*

In the following code we will create socket and bind it to loop-back address and UDP port 7777:

```
from socket import AF_INET, SOCK_DGRAM, socket
from os import getpid

if __name__ == '__main__':
    print 'Application started'
    print 'OS assigned process id: %d' % getpid()

    s = socket(AF_INET, SOCK_DGRAM)
    print 'UDP Socket created'
    print 'File descriptor assigned by OS: ', s.fileno()
    # Binding the UDP/IP socket to loopback address and port 7777
    s.bind(('127.0.0.1',7777))
    print 'Socket is bound to %s:%d' % s.getsockname()
     # Wait for user input before terminating application
    raw_input('Press Enter to teminate ...')

    s.close()
    print 'Terminating ...'
```

After starting the application it will stay running till the user hits "Enter" button. Keeping the application running, refer to active network connections table of the OS by issuing netstat command:

1. In Linux systems, from the shell

```
~$ netstat -anpu | grep X
```

   where X is the process ID printed by the executed Python code. As a result the following information will be shown (where X is PID reported by the executed Python code:

```
udp 0 0 127.0.0.1:7777 0.0.0.0:* X/python
```

2. In Mac from the terminal

```
~$ netstat  -n -p udp| grep X
```

   where X is the port number used in the Python code (e.g. 127.0.0.1.7777). The results will be as follows:

```
udp4       0      0  127.0.0.1.7777         *.*
```

As you can see in the output, the corresponding UDP socket issued by Python process (PID X) is registered with transport address 127.0.0.1:7777 which means the other applications/systems may refer to our running Python code by sending the UDP packets to UDP port 7777 on IP 127.0.0.1 (which limits the scope to the local OS only).

Questions:

1. What will happen to our socket if we modify the port number 0 in place of port 7777 in our code ? Please try it and take a look into netstat table.
2. What if do not want to limit the app. to loop-back address 127.0.0.1 ? What should be modified in the code ?
3. How do we bind to all interfaces without specifying all the IP-s in particular ?
4. Can we bind to port number lower than 1024 ? Try binding to port 1023 for example ...

Now we know how to create and bind UDP sockets for the application to be able to communicate to the other instances in the IP network using UDP protocol. Next we will discuss how to send / receive data over UDP sockets.

UDP packet (datagram)

12. *UDP packet or UDP datagram is transport unit of the UDP protocol. It contains 8 bytes of UDP header and the payload. The UDP header contains the following:*

- *Source port number (2 bytes)*
- *Destination port number (2 bytes)*
- *Datagram size (2 bytes)*
- *Checksum (2 bytes), computed over entire payload, other UDP header fields and IP header fields.*

Considering the fact that there is only 2 bytes reserved for storing the size of attached payload is limited to 64K.

13. *Transport unit (also referred as message unit or protocol data unit) is information delivered as an atomic unit among peer entities. Atomic here means no further fragmentation in the scope of the protocol. Data unit contains control information, address information and user data (payload).*

We may think of UDP datagram as an envelope for a written brief we would like to send using post service.

14. *The process of enveloping the user data into one or many datagrams is called encapsulation*

15. *De-encapsulation is then reverse process of assembling the user data from one or many received datagrams*

Sending a message inside UDP packet

In Python language the socket API takes care of creating UDP datagrams from user data. Here is an example of sending the string "Hello world!" to the peer by address 127.0.0.1:7778:

```
from socket import AF_INET, SOCK_DGRAM, socket
if __name__ == '__main__':
    print 'Application started'
    # Creating a UDP/IP socket
    s = socket(AF_INET, SOCK_DGRAM)
    # Binding the UDP/IP socket to address and port
    s.bind(('127.0.0.1',7777))
    print 'Socket is bound to %s:%d' % s.getsockname()
    # Sending the message
    message = 'Hello world!'
    destination = ('127.0.0.1',7778)
    s.sendto(message,destination)
    print 'Sent message to %s:%d' % destination
    print 'Payload lengh %d bytes: [%s]' % (len(message),message)
    raw_input('Press Enter to teminate ...')
    print 'Closing the UDP socket ...'
    s.close()
    print 'Terminating ...'
```

The output is then looking like:

```
Application started
Socket is bound to 127.0.0.1:7777
Sent message to 127.0.0.1:7778
Payload lengh 11 bytes: [Hello world!]
Press Enter to teminate ...
```

When we issue sendto(...) method on the bound socket object the new datagram is created using the data we provide over first argument (the message variable) then the destination is set to UDP header considering the second argument (the destination address IP and port). Afterwards, the source address is set using the address our socket is bound to (s.getsockname()). At the end, the length of the payload is set considering the size of the data (length of the message string). Finally the checksum is calculated using the payload and the filled header fields and the ready datagram is delegated to OS to be sent.

As you can see here we do not have a structure for an UDP datagram to manipulate, all manipulation is done by socket API. We have therefore to take into account the facts: that data blocks bigger than 2^16 have to be split manually into smaller blocks before issuing `send()` method of the socket API. The second fact is that if skip the binding socket and we send the data explicitly after creating the socket, the socket API will do binding automatically and socket will be bound to ('0.0.0.0',0). Which means the first unused port and all addresses on all interfaces. You may test it using alternative code below:

```
from socket import AF_INET, SOCK_DGRAM, socket
if __name__ == '__main__':
    print 'Application started'
    # Creating a UDP/IP socket
    s = socket(AF_INET, SOCK_DGRAM)
    # Sending the message (socket is still unbound)
    message = 'Hello world!'
    destination = ('127.0.0.1',7778)
```

```
    s.sendto(message,destination)
    print 'Sent message to %s:%d' % destination
    print 'Payload lengh %d bytes: [%s]' % (len(message),message)
    # Let's check how the socket API did bind the socket
    print 'Socket was automatically bound to %s:%d' % s.getsockname()
    raw_input('Press Enter to teminate ...')
    print 'Closing the UDP socket ...'
    s.close()
    print 'Terminating ...'
```

The output is then looking like:

```
Application started
Sent message to 127.0.0.1:7778
Payload lengh 11 bytes: [Hello world!]
Socket was automatically bound to 0.0.0.0:40154
Press Enter to teminate ...
```

As you can see the socket API did automatically assign the pattern 0.0.0.0 as destination IP and randomly pick up port 40154 for our socket.

Receiving message

Communication is typically happening between at least 2 entities, therefore in our example we need the second entity. The first one was the sender described in previous section, and it's transport endpoint (socket address) was 127.0.0.1:7777. The listener is the second and it's address may be easily deducted from the sender's code above (it is 127.0.0.1 7778). So both endpoints happen to run on the same host OS. Let's introduce the code of the listener/receiver:

```
from socket import AF_INET, SOCK_DGRAM, socket
if __name__ == '__main__':
    print 'Application started'
    # Creating a UDP/IP socket
    s = socket(AF_INET, SOCK_DGRAM)
    # Binding the UDP/IP socket to address and port
    s.bind(('127.0.0.1',7778))
    print 'Socket is bound to %s:%d' % s.getsockname()
    # Receiving the message, maximal payload to receive in one peace
    recv_buffer_length = 1024
    print 'Waiting for message ...'
    message,source = s.recvfrom(recv_buffer_length)
    print 'Received message from %s:%d' % source
    print 'Payload lengh %d bytes: [%s]' % (len(message),message)
    raw_input('Press Enter to teminate ...')
    print 'Closing the UDP socket ...'
    s.close()
    print 'Terminating ...'
```

As you can see the code is not that much different compared to the sender's code, however we can notice one new function recvfrom called on the socket with 1024 as an argument and returning the payload (message) and the sender's address (source) at once. The value 1024 is critical as we do not know what will be the size

of the message, we just agree we receive in 1024 bytes in one attempt (and if there is more we have the call recvfrom again).

The output is then looking like:

```
Application started
Socket is bound to 127.0.0.1:7778
Waiting for message ...
Received message from 127.0.0.1:50516
Payload lengh 11 bytes: [Hello world!]
Press Enter to teminate ...
```

Our message is successfully received.

Please note, here recvfrom() method will block the code execution till something is received on corresponding socket by the OS. Calling blocking receive with no timeout we assume the remote endpoint will eventually send the data. Otherwise the receive method will block forever as the default timeout is set to None.

Dealing with Big Amount of Data

Sending big message in multiple UDP packets with no additional header but using defined message terminator

```
from socket import AF_INET, SOCK_DGRAM,  socket

if __name__ == '__main__':
    print 'Application started'
    # Creating a UDP/IP socket
    s = socket(AF_INET, SOCK_DGRAM)
    # Sending the message (socket is still unbound)
    # We will use the break line to indicate the end of message
    # then the receiver may receive without knowing the full length
    # of the message he is receiving
    term = '\n'
    message = 'Hello world!'*7000+term
    max_len = 1024 # Maximal pay-load size to send in one UDP packet
    destination = ('127.0.0.1',7778)

    # In case there is more data, send it in peaces
    m = message # this will be reduced in progress of sending
    sent = 0    # how much was sent
    parts = 0   # count total sent packets
    # Cut the message into blocks and send out till the message is over
    while sent < len(message):
        m_send = m[:max_len] if len(m) > max_len else m
        m = m[max_len:]
        sent += s.sendto(m_send,destination)
        parts += 1
        print 'Sent %d bytes of %d ...' % (sent,len(message))

    print 'Sent message to %s:%d in %d parts' % (destination + (parts,))
    print 'Pay-load length %d bytes: [%s]' % (len(message),message)

```

```
31      raw_input('Press Enter to terminate ...')
32      print 'Closing the UDP socket ...'
33      s.close()
34      print 'Terminating ...'
```

Receiving big message in multiple UDP packets with no additional header but using defined message terminator

```
1  from socket import AF_INET, SOCK_DGRAM, socket
2
3  if __name__ == '__main__':
4      print 'Application started'
5      # Creating a UDP/IP socket
6      s = socket(AF_INET, SOCK_DGRAM)
7      # Binding the UDP/IP socket to address and port
8      s.bind(('127.0.0.1',7778))
9      print 'Socket is bound to %s:%d' % s.getsockname()
10
11     # Receiving the message, maximal pay-load to receive in one peace
12     recv_buffer_length = 1024
13     term = '\n'      # Terminator indicating the message is over
14     message = ''     # This will grow in progress of receiving
15
16     print 'Waiting for message ...'
17     # Append message block by block till terminator is found
18     while not message.endswith('\n'):
19         m,source = s.recvfrom(recv_buffer_length)
20         print 'Received block of %d from %s:%d' % ( (len(m),) +source)
21         message +=m
22
23     print 'Total lengh %d bytes: [%s]' % (len(message),message)
24     raw_input('Press Enter to teminate ...')
25
26     print 'Closing the UDP socket ...'
27     s.close()
28     print 'Terminating ...'
```

Reflection: as you can see the receiver does not distinguish the senders (assuming there is only one sender sending at once). What will happen if multiple senders will send simultaneously messages to the receiver ?

Ensuring Packet Arrival Order
UDP protocol does not guarantee that the packets arrive in order; however, if the design requires it then the developer must ensure the order manually, typically using the sequence markers inside blocks. Regarding the previous send/receive example developer would want to add a sequence number at the beginning of each block created. Receiver must then be aware of the existence of a sequence marker in the message, and handle the order of received blocks before appending to the message being received.

Note: In our example we chose to use the terminator to mark the end of message, however we could avoid using special terminator character and just add the total message length into first block of the message.

UDP Multicasting

Why multicasting is important? Imagine you are in a situation where you have to send the **same** message or data to many hosts (without the need of iteratively sending the message to each peer individually). Multicasting provides a solution to our case. Since in our case multicasting is based UDP, the transmission is by default not reliable. As we explain before using the UDP does not give us any feedback about the status of the process of delivering the message or the data. Therefore, the user who sent a multicast has no idea how many peers really received the packets (except if the peers were designed to send a feedback manually).

Note: Multicast allows you to send message only to the interested parties and the message is transmitted only once (saves a lot of bandwidth). In order to receive multicast the receiver must be subscribed to particular multicast group in the network. Multicast group is just a specific IP address ranges (according to IANA):

- 224.0.0.0 - 224.0.0.255 [Reserved for special "well-known" multicast addresses].
- 224.0.1.0 - 238.255.255.255 [Global scope multi-cast, for cross-network multi-casting].
  - This is however typically blocked by ISPs (preventing flood from private to public subnets).
- 239.0.0.0 - 239.255.255.255 [Administratively-scoped (local) multicast addresses].
  - This section of addresses have actually more segregation into sub-cathegories, but we are not going into details here as we will be using only the two first address ranges.

Sending Multicast

Now we will create a multicast sender that will send a message to a multicast group. The following example illustrate how to do it:

```
#!/usr/bin/python
#
# Implements Python UDP multicast sender
#
# Based on the code from:
# http://stackoverflow.com/questions/603852/multicast-in-python

'''Simple UDP Multicast sender
    @author: devel
'''
# Tired of print ------------------ ----------------------------------------------
# ... setup Python logging as defined in:
# https://docs.python.org/2/library/logging.html
import logging
```

```
15 FORMAT = '%(asctime)-15s %(levelname)s %(message)s'
16 logging.basicConfig(level=logging.DEBUG,format=FORMAT)
17 LOG = logging.getLogger()
18 # Needed imports ----------------------------------------------------------
19 # Socket from socket import socket
20 # Socket attributes
21 from socket import AF_INET, SOCK_DGRAM, SOL_SOCKET, SO_REUSEADDR
22 from socket import IPPROTO_IP, IP_MULTICAST_LOOP, IP_DEFAULT_MULTICAST_TTL
23 # from time import sleep
24 # Constants  --------------------------------------------------------------
25 # The multi-cast group address we are going to use
26 __mcast_addr = '239.1.1.1'
27 # The multi-cast group port
28 __mcast_port = 53124
29 # The multi-cast time-to-live (router hops)
30 __mcast_ttl = 4
31 # Broadcast interval seconds
32 __mcast_delay = 5
33 # Particular message the server sends to identify himself
34 __mcast_message = u'Hello world!'
35 # Initialization  ---------------------------------------------------------
36 # Declare UDP socket
37 __s = socket(AF_INET, SOCK_DGRAM)
38 LOG.debug('UDP socket declared ...')
39 # Reusable UDP socket? I am not sure we really need it ...
40 __s.setsockopt(SOL_SOCKET,SO_REUSEADDR,1)
41 LOG.debug('UDP socket reuse set ...')
42 # Enable loop-back multi-cast - the local machine will also receive multicasts
43 __s.setsockopt(IPPROTO_IP,IP_MULTICAST_LOOP,1)
44 LOG.debug('Enabled loop-back multi-casts ...')
45 # Set multi-cast time-to-live, let say we go up 4 sub-nets
46 __s.setsockopt(IPPROTO_IP, IP_DEFAULT_MULTICAST_TTL, __mcast_ttl)
47 LOG.debug('Set multicast time-to-live [%d] hops ...', __mcast_ttl)
48 # For multi-casting we do not need any specific bindings
49 # ... socket will be bound automatically to interface with sub-net providing
50 # default gateway, the port will be also defined randomly
51 # Serving -----------------------------------------------------------------
52 # Server forever (Ctrl+C to kill)
53 __n = 0
54 while 1:
55     try:
56         # Send out multi-cast message
57         __s.sendto(__mcast_message, (__mcast_addr, __mcast_port))
58         __n += 1
59         LOG.debug('[%d] Multi-cast sent ...', __n)
60         sleep(__mcast_delay)
61     except (KeyboardInterrupt, SystemExit) as e:
62         LOG.info('Terminating server ...\n')
63         break
64 # Termination -------------------------------------------------------------
65 # Clean-up the socket
66 __s.close()
67 LOG.debug('Socket closed ...')
```

```
LOG.info("Server terminated ...")
```

Output should look like:

```
2016-08-19 16:28:05,828 DEBUG UDP socket declared ...
2016-08-19 16:28:05,828 DEBUG UDP socket reuse set ...
2016-08-19 16:28:05,828 DEBUG Enabled loop-back multi-casts ...
2016-08-19 16:28:05,828 DEBUG Set multicast time-to-live [4] hops ...
2016-08-19 16:28:05,828 DEBUG [1] Multi-cast sent ...
2016-08-19 16:28:10,833 DEBUG [2] Multi-cast sent ...
2016-08-19 16:28:15,838 DEBUG [3] Multi-cast sent ...
2016-08-19 16:28:20,841 DEBUG [4] Multi-cast sent ...
^C
2016-08-19 16:28:24,321 INFO Terminating server ...
2016-08-19 16:28:24,322 DEBUG Socket closed ...
2016-08-19 16:28:24,322 INFO Server terminated
```

Receiving Multicast

At this level, we will create the receiver server.

The example below illustrate source code of the multicast receiver:

```
#!/usr/bin/python
#
# Implements Python UDP multicast receiver
#
# Based on the code from:
# http://stackoverflow.com/questions/603852/multicast-in-python

'''Simple UDP Multicast receiver
    @author: devel
'''
# Tired of print ------------------ ----------------------------------------
# ... setup Python logging as defined in:
# https://docs.python.org/2/library/logging.html
import logging
FORMAT = '%(asctime)-15s %(levelname)s %(message)s'
logging.basicConfig(level=logging.DEBUG,format=FORMAT)
LOG = logging.getLogger()
# Needed imports ------------------------------------------------------------
# Socket from socket
import socket
# Socket attributes
from socket import AF_INET, SOCK_DGRAM, SOL_SOCKET, SO_REUSEADDR, SOL_IP
from socket import IPPROTO_IP, IP_MULTICAST_LOOP, IP_DEFAULT_MULTICAST_TTL
from socket import inet_aton, IP_ADD_MEMBERSHIP
# Constants  ----------------------------------------------------------------
# The multi-cast group address we are going to use
__mcast_addr = '239.1.1.1'
# The multi-cast group port
__mcast_port = 53124
# The multi-cast time-to-live (router hops)
```

```
__mcast_ttl = 4
# Receiving buffer
__mcast_buffer = 1024
# Initialization  ---------------------------------------------------------
# Declare UDP socket
__s = socket(AF_INET, SOCK_DGRAM)
LOG.debug('UDP socket declared ...')
# Reusable UDP socket? I am not sure we really need it ...
__s.setsockopt(SOL_SOCKET,SO_REUSEADDR,1)
LOG.debug('UDP socket reuse set ...')
# Enable loop-back multi-cast - the local machine will also receive multicasts
__s.setsockopt(IPPROTO_IP,IP_MULTICAST_LOOP,1)
LOG.debug('Enabled loop-back multi-casts ...')
# Set multi-cast time-to-live, let say we go up 4 sub-nets
__s.setsockopt(IPPROTO_IP, IP_DEFAULT_MULTICAST_TTL, __mcast_ttl)
LOG.debug('Set multi-cast time-to-live [%d] hops ...', __mcast_ttl)
# Bind UDP socket to listen to muli-casts
__s.bind((__mcast_addr,__mcast_port))
LOG.debug('Socket bound to %s:%s' % __s.getsockname())
__s.setsockopt(SOL_IP, IP_ADD_MEMBERSHIP, \
               inet_aton(__mcast_addr) + inet_aton('0.0.0.0'))
# Receiving ---------------------------------------------------------------
# Listen forever (Ctrl+C) to kill
while 1:
    try:
        # Receive multi-cast message
        message,addr = __s.recvfrom(__mcast_buffer)
        LOG.debug('Received From: %s:%s [%s]' % (addr+(message,)))
    except (KeyboardInterrupt, SystemExit) as e:
        LOG.info('Terminating client ...\n')
        break
# Termination -------------------------------------------------------------
# Clean-up the socket
__s.close()
LOG.debug('Socket closed ...')
LOG.info("Server terminated ...")
```

The output should look like this:

```
2016-08-19 16:27:59,413 DEBUG UDP socket declared ...
2016-08-19 16:27:59,413 DEBUG UDP socket reuse set ...
2016-08-19 16:27:59,413 DEBUG Enabled loop-back multi-casts ...
2016-08-19 16:27:59,413 DEBUG Set multi-cast time-to-live [4] hops ...
2016-08-19 16:27:59,413 DEBUG Socket bound to 239.1.1.1:53124
2016-08-19 16:28:05,828 DEBUG Received From: 172.17.161.232:54469 [Hello world!]
2016-08-19 16:28:10,833 DEBUG Received From: 172.17.161.232:54469 [Hello world!]
2016-08-19 16:28:15,838 DEBUG Received From: 172.17.161.232:54469 [Hello world!]
2016-08-19 16:28:20,841 DEBUG Received From: 172.17.161.232:54469 [Hello world!]
^C
2016-08-19 16:28:23,850 INFO Terminating client ...
2016-08-19 16:28:23,850 DEBUG Socket closed ...
2016-08-19 16:28:23,850 INFO Server terminated ...
```

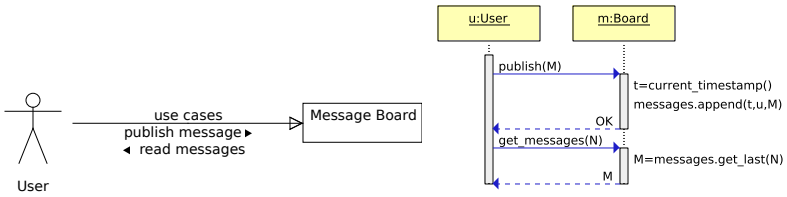


Figure 2.1: Message board scenario #1 use cases and interaction diagram

Programming Application layer protocol on top of UDP

Here we are ready to implement a protocol for an application, but first of all let's agree what sort of application we will program.

Let's say we want the public message board service to be shared with any number of users who want to publish or read messages. So we have two roles for network entities - the "Message Board" and the "User". The required scenarios are:

- Scenario:
  - Part 1: The message board service is responsible for serving user requests.
    - User may request to submit his message to public board
    - User may request the list of messages currently published on the board
  - Part 2: Complimentary requirement.
    - User may request to submit big message (for example, text and pictures, videos or archive attachments)

Part 1: Message Board Protocol

Designing the protocol

Scenario #1 is illustrated in Figure 2.1. As we can notice the "User" refers to "Message Board" and can request to publish his message or he may request all messages to view. As result, the "User" is always the first who initiates the communication, as there is no use case when "Message Board" initiates communication. Therefore, the client-server architecture here is the most suitable: one board can serve many users and, one user may refer to only one board at once (user cannot change the board during runtime). Basically, once the application gets started, it is connected to the specified board and is staying connected till application dies (next time it is started user may specify different board to refer to).

We have declared 2 operations between the previously shown entities. Both operations evolve transmitting some information between the entities. And we already know our communication medium - it's the UDP transport protocol (we have to refer to Socket API of Python). Now let's agree how this information is transmitted, in other words: let's declare the protocol.

By definition, we need a special sort of message, where we specify the type of action (publish or retrieve) that is associated to this message. In addition, dependent on the type there can be the corresponding "text message" that has to be published or the number of the messages to retrieve in case of retrieval action.

In our case, the special message is exactly a definition of a protocol data unit or PDU (maximal transportable piece of data in one transaction). Plus, we have to carry our data protocol data units over UDP; therefore,

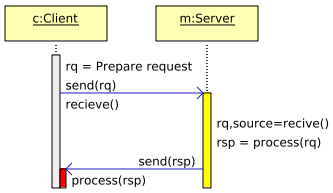


Figure 2.2: Sequence diagram of a simple Request-Response Protocol

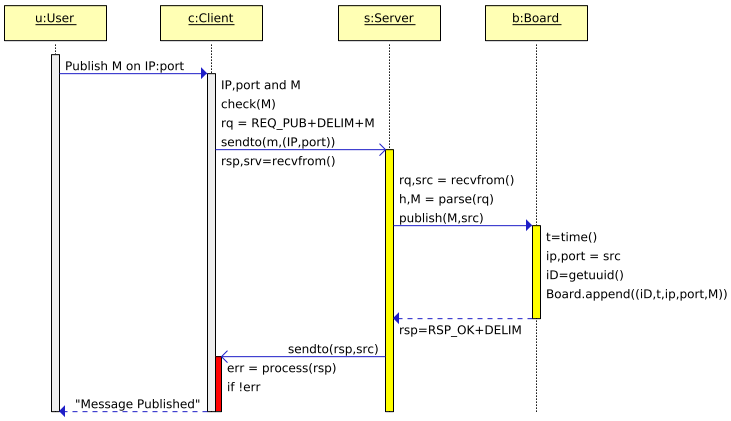


Figure 2.3: Sequence diagram of "Publish" routine

we should take into account the features of the UDP. The one that affects us most is the inability to send the big message reliably. So we either implement our own routines: for 1.) guaranteeing delivery 2.) preserving data. Alternatively we may just adapt our protocol data unit towards UDP. And in the first scenario we rely on the easier way and declare the maximal size of our PDU as maximal length of the UDP packet's payload (or the maximal size of the UDP packet minus it's header length: 2^16-32 bytes). In order to guarantee delivery we will introduce the server response messages: each request sent by a client has to be confirmed by the server. Client considers the request delivered only if corresponding confirmation is received from the server. The described pattern is usually referred as request-response pattern, refer to Figure 2.2.

Now knowing the characteristics of our PDU let's declare the control options or the way we inform the endpoint what do we want. In other words, let's introduce the header of our PDU. Now we can say the header is of fixed length (first N bytes) and in these N bytes, we store all the control options we need. For example, in the first byte we specify which sort of action is requested (0x0 for publishing and 0x1 for retrieval). However, if we do not reserve enough bytes in the header, it will be hard to extend the protocol with more options. Another disadvantage is that not used space is a wasted space in PDU's header and it could be used to carry the payload. The solution here would be the floating header of various size with special delimiter byte denoting the end of the header and the start of the payload.

Remark: In fact what bytes to we use for denoting a delimiter (if use floating header) or denoting actions is up to the developer. In our code examples we intentionally use characters only (for our own convenience in debugging - strings are readable if you refer to raw UDP packet's payload when monitoring network).

After we declared the PDU, its header and payload options, let's discuss the mentioned use cases, and how these might be reflected in our protocol:

- User wants to publish the message (control code “1”)
  The routines are quiet straight forward, refer to Figure 2.3:

  1. Client side:
     (a) the client reads the user’s message and server’s address (IP, port)
     (b) client cuts off the tail in case the message is bigger then our maximal PDU size (excluding PDU header length).
     (c) client wraps the message into request (adding the header containing the control code “1” - publish and “:” - terminating header)
     (d) client sends UDP packet with the request to the server on the address specified by the user
  2. Server side:
     (a) server receives new UDP packet and extracts the payload (the client’s request)
     (b) server checks the header and determines the operation that the client has requested
     (c) server proceeds with “publish” routine and gets the client’s message from the request
     (d) server issues publish routine of the message board, giving message and the client’s socket address as an argument
     (e) message board gets the current time stamp and stores the values: times tamp, client’s IP, client’s port, message
     (f) server prepares the response (adding header containing control code “0” - No Error and “:” terminating header)
     (g) server sends the UDP packet with response to client on the address the request was received from
  3. Client side:
     (a) client receives the UDP packet and extracts the server’s response
     (b) client checks the header and determines if control code reports “No Error”
     (c) client reports “Message Published” to user and terminates

- User wants to see N last published messages:
  This scenario is a bit more tricky, as we requesting multiple messages and the size of a result payload might easily exceed the maximal allowed payload of our PDU. Therefore we introduce two additional sub-routines, and two additional control codes for PDU header:

  – Get iD-s of last N messages published (control code “2”)
    * Client sends the request with desired number of last messages to read (N)
    * Server responses with the list of iDs of the corresponding messages
  – Get message by iD (control code “3”)
    * Client sends the request with the iD of the desired message
    * Server responses with the requested message

  The whole routine “User wants to see N last published messages” is then happening in multiple steps, where each step is request/response iteration:

  1. Client retrieves the list of iDs (of the last N messages published)
  2. For each of the received iD the client then fetches the message text in separate request

  When the list of last N messages is collected, these are shown to user, leaving an impression of single routine. The whole routine is illustrated in Figure 2.4.

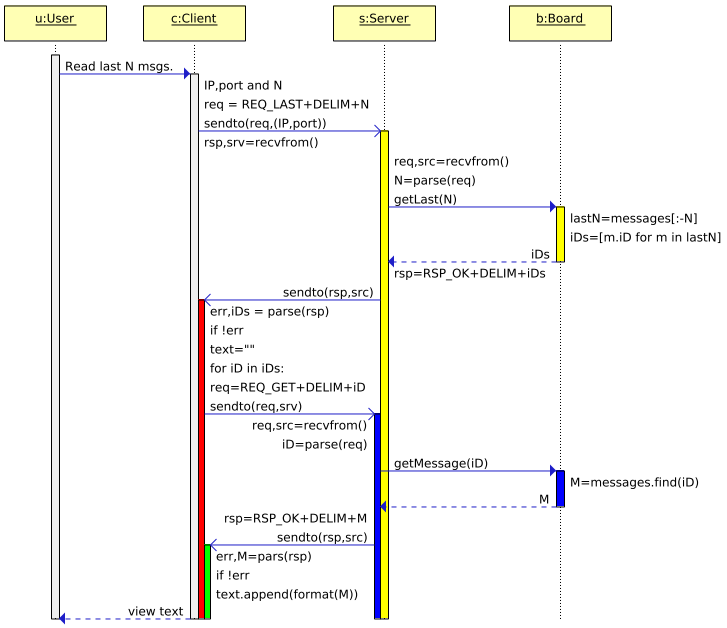


Figure 2.4: Sequence diagram of "Read Last N messages" routine

The resulting protocol is therefore as follows:

- request-response stateless protocol, refer to Figure 2.2
  - client sends a request to server and waits for a confirmation (a response)
  - request/response are not longer than one data unit (PDU)
- protocol data unit (PDU) size = maximal UDP payload size
- request/response PDU has floating header with ':' as delimiter, all the numerical values are string-encoded
  (the whole PDU data is then easily readable as string, for the sake of tracing the protocol).
- the request codes:
  - __REQ_PUBLISH = '1'
    Client wants to publish new message
    Payload is the text of the user's message (till the end of UDP packet)
    Example of the total message:
    * "1:Hello world!"
  - __REQ_LAST = '2'
    Client wants the iDs of the last N messages
    Payload is one integer N (string coded)
    Example of the total message:
    * "2:10"
  - __REQ_GET = '3'
    Client wants the message by iD
    Payload is one integer iD (string code)
    Example of the total message:

    - "3:4325204352"

- response codes:
  - __RSP_BADFORMAT = '1'
    Server could not parse the header of the request
    No payload
  - __RSP_MSGNOTFOUND = '2'
    Server could not find the message by the iD specified
    No payload
  - __RSP_UNKNCONTROL = '3'
    Server parsed the header, but there is no action for this control code
    No payload
  - __RSP_OK = '0'
    No error
    Payload is different dependent on the request:
    - __REQ_PUBLISH
      No payload
    - __REQ_LAST
      Payload is the list of integers (string coded, delimited by ':')
      Example of the total message:
      - "0:235432:242435243:3452425423"
    - __REQ_GET
      Payload is a requested message with all it's attributes: time stamp when published (float: unix epoch time), sender-socket's IP address (string) and port (int), actual message text (string). All attributes are string encoded and appended using ':' as delimiter.
      Example of the total message:
      - "0:12343241.3241:127.0.0.1:34234:Hello world!"

At this point the design of the protocol should be clear, let's proceed with the implementation.

Implementation of the protocol

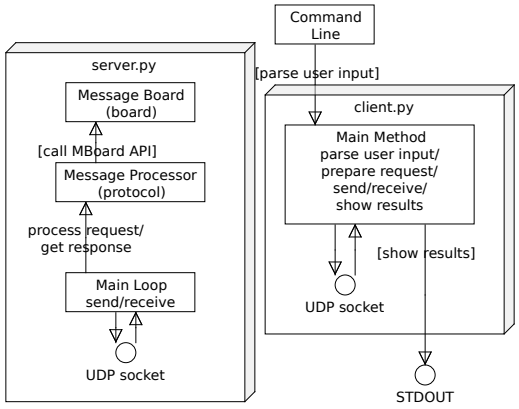


Figure 2.5: Message Board components diagram: Server-side (on the left), Client-side (on the right)

Implementing part 1 of our scenario is straightforward, we organized the code into 4 Python modules (refer to: 2.5):

- board.py
  Implements the MBoard API (the data structures for holding messages, the methods for publishing and retrieving the messages).
- protocol.py:
  Implements the MBoard protocol's server side: message processor (parsing request header, extracting payload, calling the MBoard API dependent on request, composing response). Also holds the protocol constants: control codes for requests, header delimiter, response codes.
- server.py:
  Implements the MBoard server: arguments processor, socket binding, main serving loop (receive, process, response).
- client.py:
  Implements the MBoard client: arguments processor, and methods for requesting the data according to routines: "publish message" or "view last N messages"

Client-side

Let's start first by the client implementation. The client Implements message board UDP client and it is a simple command-line application. Currently no interactive mode implemented. The application does the request and dies. In addition, the user has to provide the IP address of the server at least.

Default behavior is to fetch all the messages from the Board and show them to the user. Refer to --help to get the details about the options.

```
$python client.py -h
```

The current implementation considers all user input is by default in UTF-8, no additional encoding (providing non utf-8 characters in user's input may produce error)

Limitations (considered in protocol)

Protocol implements state-less message board protocol (MBoard protocol). In this protocol, we assume that the client never sends the big message (that would need multiple UDP packets). In case the client sent a long text, the only text contained in the packet will be the one respecting the size and the remaining will be lost. Therefore, if you want to preserve the long text, you have to take into account the UDP packet behavior and contained. This latter means that you have to split your message according to the authorized size that can be contained by UDP packets and send you message in multiple packets. However, the ID of the corresponding message are sent first, then the client asks messages one by one using IDs, refer to Figures 2.3 and 2.4.

Server-side

The server implements message board UDP server (server.py) and it is using static message board implementation (board.py) and implementation of message board protocol (protocol.py). The server is using simple state-less protocol, assuming client can only send one UDP packet per interaction (2 packets received from the same origin are processed in separate interactions).

The server does not assure if the packets arriving is in sequence from the same origin (each new arrived packet is considered as a new client). With a protocol like this it is impossible to sent messages in multiple

UDP packets from the same client; as the server may receive packets from different clients interleaved and does not organize packets into client sessions.

Having client sessions requires to store additional information about a client on server side (client sate). For example, in order to allow sending messages in multiple UDP packets, the partially received messages must be temporally stored by server (till server receives all the UDP packets belonging to this message). Keeping the information about client state on server side is the key idea of stateful protocols (see the next section for more details).

The server may be started from command line using:

```
$python server.py
```

The default behavior is to start listening on localhost address (127.0.0.1) and port 7777. Once application is started it runs forever, if not interrupted with Ctrl+C.. In order to see other options of the applications use refer to application's help:

```
$python server.py -h
```

Part 2: Stateful Protocol (Introducing sessions on UDP)

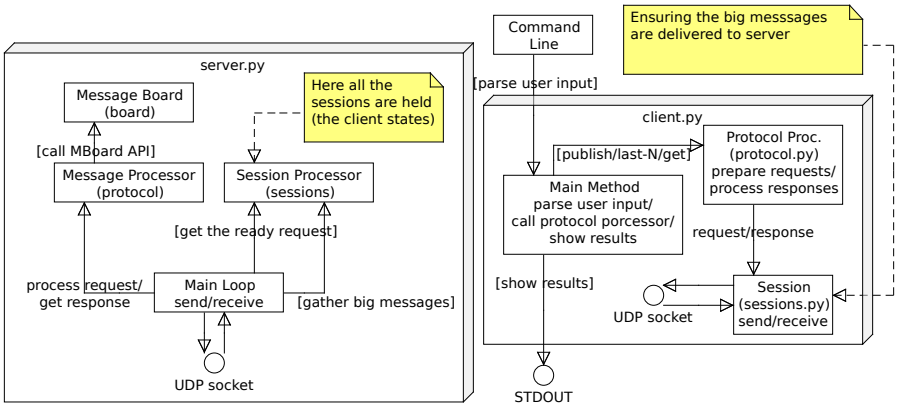


Figure 2.6: Message Board components diagram: Server-side (on the left), Client-side (on the right)

Concerning the part 2 of our example implementation, it will be divided into five main modules (client, protocol, session, server and board), refer to Figure 2.6.

> **Remark:** At this level we are expecting you can read diagrams we provided and make the analogy with source code of the example application. Further we will not elaborate our code and diagrams like in previous section, rather we expect you to capable of doing it by yourself, in case of questions we are there in the Seminars and Office hours to help you.

Design

In general the design did not change that much compared to part one, except that we introduced the module Sessions which take care of communication and brings the notion of stateful aspect to our system. In details (refer to Figure 2.7)[1]:

[1] In this diagram we refer to TCP model layers

1. In part 1 the MBoard protocol data units (containing requests/responses) were delegated directly to UDP and therefore we could not have big messages sent at once.

2. In part 2 the MBoard protocol data units (containing requests/responses) are delegated to session protocol to ensure delivery. And the sessions takes care of splitting the big message and sent it over UDP ensuring delivery.

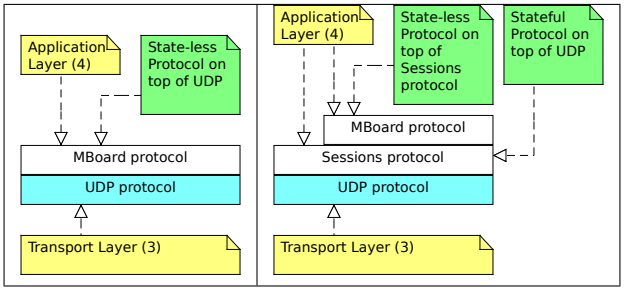


Figure 2.7: Protocol layers of Part 1 (on the left) and Part 2 (on the right)

The most intuitive way of ensuring the big message delivery is to split it into blocks and send them one by one, confirming delivery of each one. Once all blocks are delivered the final assembly happens, refer to Figure 2.8).

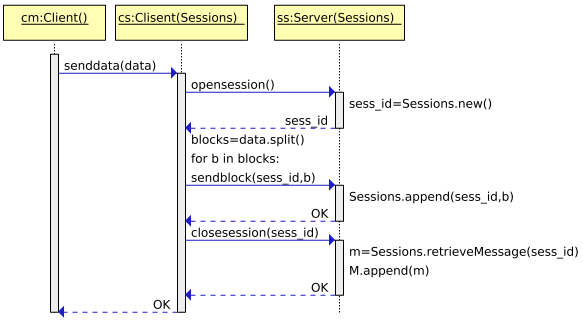


Figure 2.8: Simple illustration of the block delivery protocol

However on the server side the blocks of different sessions might appear interleaved as two clients may send two different big messages to the server in blocks. In this case server must understand what block belongs to to what client and store those into separate buffers (assembling big message). Therefore before starting sending we ask server to give us the session id for particular big message delivery, server keeps information of session id and the socket address of client requested the session. Client then uses session id in with each block it sends. Once full message is assembled the server threats the whole message as received from the client who has issued session.

The sequences diagram of the sessions protocol is illustrated in Figure 2.9 , the implementation is in the material folder provided [UDPApplication/sessions].

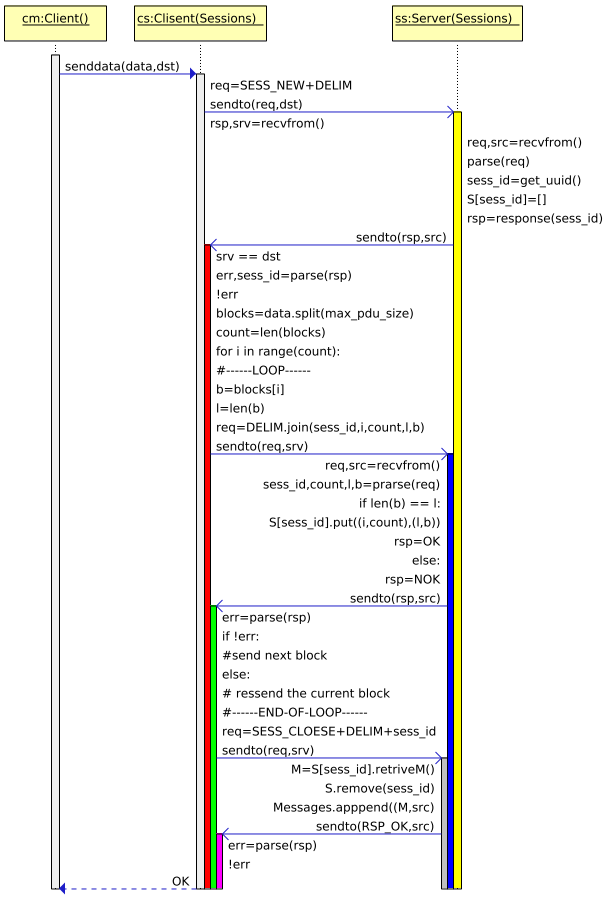


Figure 2.9: Sessions Protocol on UDP

TCP Protocol

We are expecting you that you assimilated most of the information introduced in the previous section on UDP. However, we will make some recall for important definitions as a reminder if necessary.

16. *Transmission Control Protocol (TCP) is a standard reliable data delivery that defines how to establish and maintain a connection or network conversation via which data can be exchanged. Moreover, TCP is one of the transport layers suggested in the OSI layer and it is used to create connection between remote machine by transporting and ensuring delivery of messages over the network and the Internet.*

Note: Summarizing features of TCP

- Connection oriented: An application requests a "connection" to destination and uses connection to transfer data – UDP does not use "connections" - each datagram is sent independently!
- Point-to-point: A TCP connection has two endpoints (no broadcast/multicast)
- Reliability: TCP guarantees that data will be delivered without loss, duplication or transmission errors
- Full duplex: Endpoints can exchange data in both directions simultaneously
- Reliable connection startup: TCP guarantees reliable, synchronized startup between endpoints (using "three-way handshake")
- Graceful connection shutdown: TCP guarantees delivery of all data after endpoint shutdown

Having in mind those features you may think of TCP as of a pipeline between 2 endpoints. The pipeline consists of 2 pipes with opposite flow. Both endpoints have 2 pipe valves Rx and Tx valve (refer to : Figure 2.10 on page 36. Having a pipeline like that we guarantee nothing gets lost during the delivery (otherwise the pipe is broken); additionally we guarantee the order of delivery (FIFO - first in first out). However we lose some features, as you can see the pipes associating two sockets only, if we had a third party "Socket C", then we would have needed an additional pipelines "A<->C" and "B<->C".

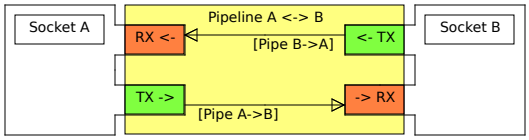


Figure 2.10: Illustrating TCP on pipeline example

Note: Broken pipe exception
Good feature of TCP pipes is the immediate report once the active pipe is broken. It happens in case there is a connectivity issues between the end-points (lost signal, broken cable, no gateway etc.). By analogy with cable powering the desktop lamp, there will be immediate feedback once the cable gets damaged - the lamp will stop shining.

In the next sections, we will step by step implement the simple messaging protocol between 2 endpoints relying on TCP.

TCP Socket

Before we start showing how to manipulate TCP socket, we have to introduce the states of TCP socket. TCP is a stateful Transport Layer (3) protocol. We have in fact introduced stateful protocols already and even implemented one on Application Layer (4)[2], having additional PDU headers, options and additional structures on both endpoints. In TCP it is all implemented in the OS (Berkly or POSIX socket interface are implemented in Linux and Mac, Microsoft has its own interface - Winsock which closely follows the POSIX standard). In general, in TCP a connection progresses through a series of states during its lifetime (refer to: Figure 2.11 on page 38)[3]. The states are as follows:

[2]In this example we refer to TCP model layers
[3]TCP/IP Illustrated, Volume 2: The Implementation by Gary R. Wright and W. Richard Stevens

- LISTEN: represents waiting for a connection request from any remote TCP socket address (IP and port).
- SYN-SENT: represents waiting for a matching connection request after having sent a connection request.
- SYN-RECEIVED: represents waiting for a confirming connection request acknowledgment after having both received and sent a connection request.
- ESTABLISHED: represents an open connection, data received can be delivered to the user. The normal state for the data transfer phase of the connection.
- FIN-WAIT-1: represents waiting for a connection termination request from the remote TCP, or an acknowledgment of the connection termination request previously sent.
- FIN-WAIT-2: represents waiting for a connection termination request from the remote TCP.
- CLOSE-WAIT: represents waiting for a connection termination request from the local user.
- CLOSING: represents waiting for a connection termination request acknowledgement from the remote TCP.
- LAST-ACK: represents waiting for an acknowledgment of the connection termination request previously sent to the remote TCP. (it also includes an acknowlegment of its connection termination request)
- TIME-WAIT: represents waiting for enough time to pass to be sure the remote TCP received the acknowledgement of its connection termination request.
- CLOSED: represents no connection state at all.

Note: CLOSED is fictional state because it represents the state when there is no TCP pipe, thus no connection.

In general, TCP connection progresses from one state to another in response to events. The events are:

- The user calls/Server receives incoming connection request (SYN sent/SYN received)
- Open, set socket into listening mode
- Send, sending the data, ensuring its delivery
- Receive, receiving the data, ensuring its delivery
- Close, putting from states ESTABLISHED or LISTENING into state CLOSED. In case of ESTABLISHED state we also wait for remote endpoint to confirm the closing, therefore the FIN_WAIT states. And in case no confirmation received we close the socket after certain time out.
- Abort, when socket in state LISTENING denies to accept new connection. In case of exceeding the maximal number of simultaneous sockets in state ESTABLISHED.
- Status, when socket in state ESTABLISHED sends keep-alive to inform remote socket of its presence (in case there pipe is idling with no active data transmission).
- Incoming segments / flags (SYN, ACK, RST, FIN, and timeout)

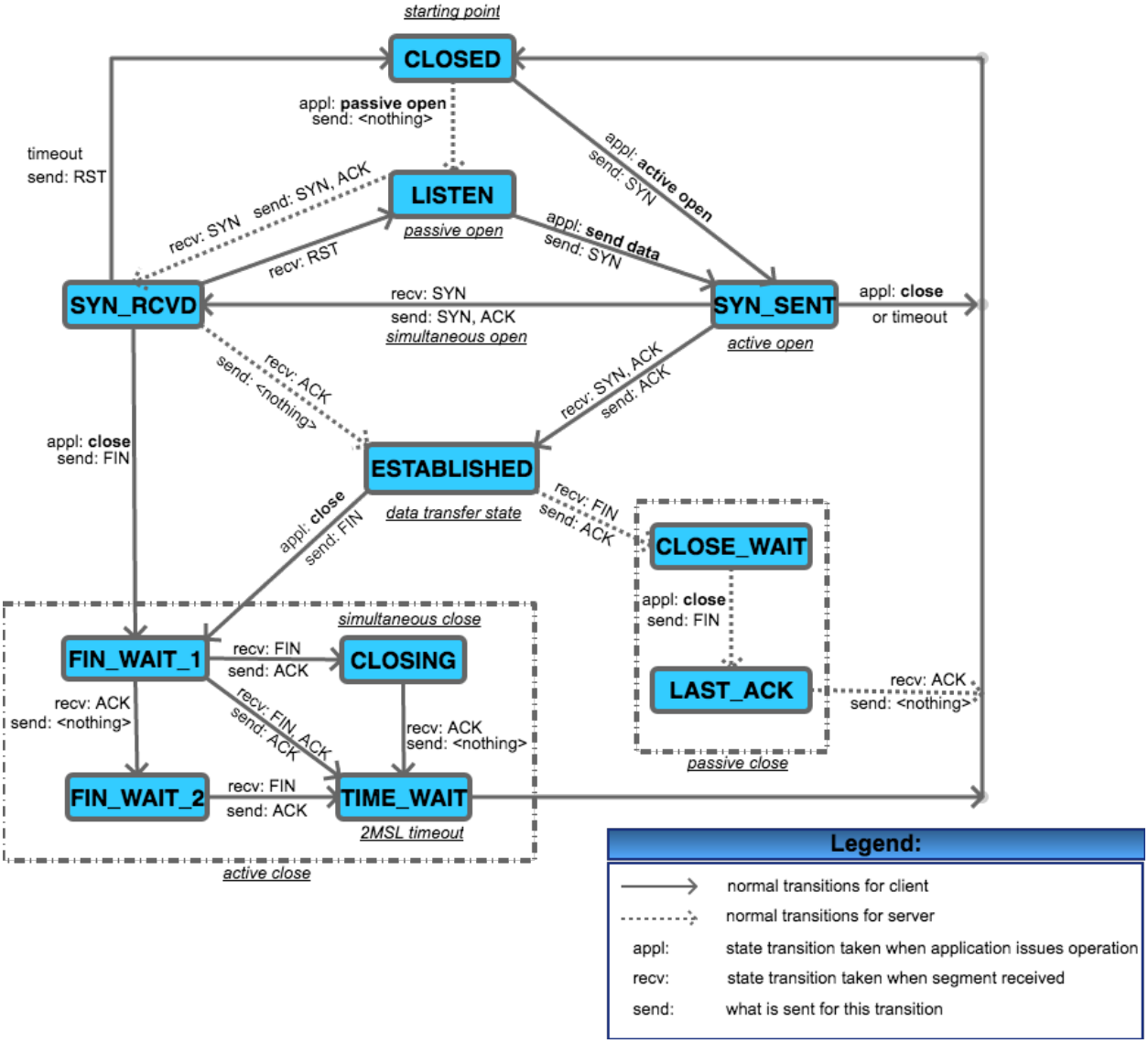


Figure 2.11: TCP State Transition Diagram

- SYN: Initiate connection
- ACK: Acknowledge received data
- FIN: Close a connection
- RST: Abort the connection in response to an error

The "creating" and "binding" operations of a TCP socket are similar to the ones of a UDP socket.

Creating socket

Here is an example code showing how to create the TCP socket and how to check what descriptor assigned to it by the OS:

```
# From socket module we import the required structures and constants.
from socket import AF_INET, SOCK_STREAM, socket
# Sleep function will be used to keep application running for a while
from time import sleep
# And we also will need the process id given by OS when we execute this code
from os import getpid
# Main method
if __name__ == '__main__':
    print 'Application started'
    print 'OS assigned process id: %d' % getpid()
    # Creating a TCP/IP socket
    s = socket(AF_INET, SOCK_STREAM)
    print 'TCP Socket created'
    print 'File descriptor assigned by OS: ', s.fileno()
    wait_secs = 60*5
    print 'Waiting %d seconds before termination ...' % wait_secs
    sleep(wait_secs)
    print 'Terminating ...'
```

Executing the code should print the following result, please note that letters X and Y will be denoting the process ID and the descriptor respectively and are specific to your environment (each time your run the code it will give different numbers):

```
Application started OS assigned process id: X
TCP Socket created
File descriptor assigned by OS:  Y
Waiting 300 seconds before termination ...
```

Now while the code is running for 5 minutes we have time to refer to the list of registered descriptors of the OS and check if we see our running application process with a corresponding descriptor:

1. in Linux systems the Kernel hold raw process information in folder-like structures (populated by Kernel). So the very primitive folder listing call is enough (the ls command). Issue the following command in shell, make sure in place of X you use the process ID reported by the executed Python code and in place of Y - the descriptor):

```
~$ ls -l /proc/X/fd/Y
```

   ... in the output you should now see the socket's descriptor that were assigned by the OS to a process number X:

```
lrwx------ 1 user user 64 Aug 18 13:13 Y -> socket:[129069931]
```

2. in MAC systems the unbound sockets are visible by lsof command, so it is enough to print all the TCP descriptors (lsof -i 4tcp) and filter the ones assigned to specific process ID (grep X). Issue the following command in shell, make sure in place of X you use the process ID reported by the executed Python code:

```
$ lsof -i 4tcp |grep X
```

   ... in the output you should now see the socket's descriptor that were assigned by the OS to a process number X:

```
Python    2062   user    3u  IPv4 0x90964583fac7dd81      0t0  TCP *:*
```

If you can see the descriptor of a socket you have created in your code, you may continue to the next step of binding a socket. The currently running application (if not terminated after 5 minutes) may be terminated manually (Ctrl+C if running in shell explicitly). You may notice that after the application was terminated all it's descriptors (including our TCP socket) are gone (closed automatically by the OS). Alternatively in your code you may issue close() method on socket before terminating the application (2 last lines of the reference code are modified):

```
    sleep(wait_secs)
    print 'Closing the TCP socket ...'
    s.close()
    print 'Terminating ...'
```

Please note that "creating" socket is yet not enough to start communicating, remember for UDP "creating" socket was like "constructing a new mailbox"; now for TCP an analogy to creating socket would be "mounting the valves for an upcoming pipeline" or "mounting a socket/plug of a 220V power cable".

Binding socket

As we know, after creating the socket, we have to provide a transmission endpoint; tell to the OS/kernel what IP address and TCP port we would like to use. As we saw in a previous chapter, binding a UDP socket to an address, made a UDP socket reachable from the network; However, binding TCP socket gives us the choice to choose either "connecting" or "listening from that socket" (by analogy "calling-out" or "waiting for a call" when using a phone).

The socket "listening" and "connecting" states will be explained later. The "binding" however is similar to a UDP socket.

In the following code we will create TCP socket and bind it to loop-back address and TCP port 7777:

```
from socket import AF_INET, SOCK_STREAM, socket
from os import getpid

if __name__ == '__main__':
    print 'Application started'
    print 'OS assigned process id: %d' % getpid()

    s = socket(AF_INET, SOCK_STREAM)
    print 'TCP Socket created'
    print 'File descriptor assigned by OS: ', s.fileno()
    # Binding the TCP/IP socket to loopback address and port 7777
    s.bind(('127.0.0.1',7777))
    print 'Socket is bound to %s:%d' % s.getsockname()
     # Wait for user input before terminating application
    raw_input('Press Enter to teminate ...')

    s.close()
    print 'Terminating ...'
```

After starting the application it will stay running till the user hits "Enter" button.

As you can see in the output, the corresponding TCP socket issued by Python process (PID X) is registered with transport address 127.0.0.1:7777 which means the running python code can now "listen" to new connections on that socket address (IP:port) or initiating new connection or "connecting" to a remote socket using local bound socket.

Now we know how to create and bind TCP sockets for the application to be able to communicate to the other instances in the IP network using TCP protocol. Before we proceed to "sending" / "receiving" the data over TCP, let's make sure we understand "listen" and "connect" operations as these are the key features of TCP sockets.

Listening

TCP Socket in the "listening" state is like an idling phone, expecting an incoming call. Once the call is received we literally should "pick-up a the phone" and start conversation. Let's first of all illustrate how TCP socket looks like in "listening" state, please refer to the code:

```
from socket import AF_INET, SOCK_STREAM, socket
from os import getpid
if __name__ == '__main__':
    print 'Application started'
    print 'OS assigned process id: %d' % getpid()
    s = socket(AF_INET, SOCK_STREAM)
    print 'TCP Socket created'
    print 'File descriptor assigned by OS: ', s.fileno()
    # Binding the TCP/IP socket to loopback address and port 7777
    s.bind(('127.0.0.1',7777))
    print 'Socket is bound to %s:%d' % s.getsockname()
    # Put socket into listening state
    backlog = 0 # Waiting queue size, 0 means no queue
    s.listen(backlog)
    print 'Socket %s:%d is in listening state' % s.getsockname()
    # Wait for user input before terminating application
     raw_input('Press Enter to teminate ...')
    s.close()
     print 'Terminating ...'
```

The output:

```
Application started
OS assigned process id: 24335
TCP Socket created
File descriptor assigned by OS:  3
Socket is bound to 127.0.0.1:7778
Socket 127.0.0.1:7778 is in listening state
Press Enter to terminate ...
```

Check the socket state using netstat command:

1. In Linux systems, from the shell

```
~$ netstat -lnpt | grep X
```

where X is the process ID printed by the executed Python code. As a result the following information will be shown (where X is PID reported by the executed Python code:

```
tcp 0 0 127.0.0.1:7777 0.0.0.0:* X/python
```

2. In Mac from the terminal

```
~$ netstat  -n -p tcp| grep X
```

where X is the port number used in the Python code (e.g. 127.0.0.1.7777). The results will be as follows:

```
tcp4       0      0  127.0.0.1.7777         *.*
```

3. In Windows in command line (cmd):

```
~$ netstat -p tcp -a
```

will give a list of all states of all TCP sockets including our one which should be listed as:

```
TCP 127.0.0.1:7777 HOSTNAME LISTENING
```

At this point we may proceed to an "accept" method that does-the pickup on the TCP socket in the "listening" state.

Accepting Connections (server-side)

Accepting the TCP connection by the TCP socket in listening state is like to pick-up a phone. Once the phone is picked-up the conversation may start. According to TCP state diagram the accept operation can only occur on TCP socket in listening state. TCP accept operation creates another socket on server just to communicate with the client that was requesting connection. Calling accept method will also block until the incoming TCP connection on corresponding socket has occurred. The code illustrating "listener-socket" accepting incoming connections and creating socket in established state:

```
from socket import AF_INET, SOCK_STREAM, socket
if __name__ == '__main__':
    print 'Application started'
    s = socket(AF_INET, SOCK_STREAM)
    print 'TCP Socket created'
    # Binding the TCP/IP socket to loop-back address and port 7777
    s.bind(('127.0.0.1',7777))
    print 'Socket is bound to %s:%d' % s.getsockname()
    # Put socket into listening state
    backlog = 0 # Waiting queue size, 0 means no queue
    s.listen(backlog)
    print 'Socket %s:%d is in listening state' % s.getsockname()
    client_socket,client_addr = s.accept()
    print 'New client connected from %s:%d' % client_addr
    print 'Local end-point socket bound on: %s:%d'\
    '' % client_socket.getsockname()
    # Wait for user input before terminating application
    raw_input('Press Enter to terminate ...')
    s.close()
    print 'Terminating ...'
```

... keeping the server-side code running we proceed to the client-side code.

Connecting (client-side)

For client side socket it is simply the connect operation we need to issue after creating the socket. The code is as follows:

```
from socket import AF_INET, SOCK_STREAM, socket
if __name__ == '__main__':
    print 'Application started'
    s = socket(AF_INET, SOCK_STREAM)
    print 'TCP Socket created'
    # No binding needed for client, OS will bind the socket automatically
    # when connect is issued
    server_address = ('127.0.0.1',7777)
    # Connecting ...
    s.connect(server_address)
    print 'Socket connected to %s:%d' % s.getpeername()
    print 'Local end-point is  bound to %s:%d' % s.getsockname()
    # Wait for user input before terminating application
    raw_input('Press Enter to terminate ...')
    s.close()
    print 'Terminating ...'
```

Questions:

1. What will happen to the running applications in case we hard-interrupt the connectivity (unplugging the Ethernet cable or hardware switching the wireless adapter off)?

TCP packet/header

17. *TCP packet is transport unit of the TCP protocol. It contains between 20 bytes till 60 bytes of TCP header and the payload.*

The TCP header contains the following:

- Source address (2 bytes) [identifies the sending port]
- Destination address (2 bytes) [identifies the receiving port]
- Sequence number (4 bytes) has a dual role:
  - if the SYN flag is set to (1), then this is an initial sequence number.
  - if the SYN flag is set to (0), then this is the accumulated sequence number of the first data byte of this segment for the current session.
- Acknowledgement number (if ACK set)(4 bytes)
- Data offset (1-st bit of 1 byte) [Specify the size of TCP header in 32-bit words.]
- Reserved (2-nd of 1 byte) [for future use and should be set to zero]
- Flags (3 bits: 3-d, 4-th and 5-th of 1 byte) [combination of those gives us the following flags:
  - NS (1 bit) – ECN-nonce concealment protection (experimental: see RFC 3540).

  - CWR (1 bit) – Congestion Window Reduced (CWR) flag is set by the sending host to indicate that it received a TCP segment with the ECE flag set and had responded in congestion control mechanism (added to header by RFC 3168).
  - ECE (1 bit) – ECN-Echo has a dual role, depending on the value of the SYN flag. It indicates:
    * If the SYN flag is set (1), that the TCP peer is ECN capable
    * If the SYN flag is clear (0), that a packet with Congestion Experienced flag set (ECN=11) in IP header received during normal transmission (added to header by RFC 3168). This serves as an indication of network congestion (or impending congestion) to the TCP sender.
  - URG (1 bit) – indicates that the Urgent pointer field is significant.
  - ACK (1 bit) – indicates that the Acknowledgment field is significant. All packets after the initial SYN packet sent by the client should have this flag set.
  - PSH (1 bit) – Push function. Asks to push the buffered data to the receiving application.
  - RST (1 bit) – Reset the connection
  - SYN (1 bit) – Synchronize sequence numbers. Only the first packet sent from each end should have this flag set. Some other flags and fields change meaning based on this flag, and some are only valid for when it is set, and others when it is clear.
  - FIN (1 bit) – No more data from sender]

- Windows size (4 bytes) [the size of the receiving window]
- Urgent pointer (if URG set) (2 bytes) [Indicate the last urgent data bytes if the URG is set.]
- Checksum (2 bytes) [used for error-checking of the header and data].
- Options (4 bytes). The length of this field is determined by the data offset field. Options have up to three fields:
  - Option-Kind (1 byte),
  - Option-Length (1 byte),
  - Option-Data (variable).

Receiving using TCP connection

Here we use again the same example with client and server, and we demonstrate how server starts receiving a client's message when there is a new client connected (does not have to be this way, and the server might start sending the data to client first, and then proceeds to receiving). For receiving we issue `recv(...)` with buffer length to collect the received data (this is similar to UDP recvfrom method), according to socket API:

Note:

- *`socket.recv(bufsize[, flags])`*
  - Receive data from the socket. The return value is a string representing the data received. The maximum amount of data to be received at once is specified by bufsize. See the Unix manual page recv(2) for the meaning of the optional argument flags; it defaults to zero

Keep in mind that send/receive we cannot do on the server's listener-socket, till it accepts new client by using `accept()` method, which will lead to the creation of a new socket for this particular connected client (we

call it client-socket). The client socket can then be used to send/receive to this particular client. And here we demonstrate sequence of 1.) creating socket 2.) binding it 3.) making it a listener-socket 4.) accepting new client and creating client-socket 5.) receiving a message using client-socket:

```
from socket import AF_INET, SOCK_STREAM, socket
if __name__ == '__main__':
    print 'Application started'
    s = socket(AF_INET, SOCK_STREAM)
    print 'TCP Socket created'
    # Binding the TCP/IP socket to loop-back address and port 7777
    s.bind(('127.0.0.1',7777))
    print 'Socket is bound to %s:%d' % s.getsockname()
    # Put socket into listening state
    backlog = 0 # Waiting queue size, 0 means no queue
    s.listen(backlog)
    print 'Socket %s:%d is in listening state' % s.getsockname()
    client_socket,client_addr = s.accept()
    print 'New client connected from %s:%d' % client_addr
    print 'Local end-point socket bound on: %s:%d'\
            '' % client_socket.getsockname()
    # Once the client is connected start receiving the data using its socket:
     recv_buffer_length = 1024
    message = client_socket.recv(recv_buffer_length)
    print 'Received %d bytes from %s:%d' % ( (len(message),)+client_addr )
    print 'Received message: \n%s' %  message
    # Wait for user input before terminating application
    raw_input('Press Enter to terminate ...')
    client_socket.close()
    print 'Closed the client socket'
    s.close()
    print 'Closed the server socket'
    print 'Terminating ...'
```

The output is then looking like:

```
Application started
TCP Socket created
Socket is bound to 127.0.0.1:7777
Socket 127.0.0.1:7777 is in listening state
...
```

Keeping the server's code running you may proceed with the client's code (next section). The client will then send the data and then the server continues to execute with the output:

```
...
New client connected from 127.0.0.1:59484
Local end-point socket bound on:
127.0.0.1:7777
Received 11 bytes from 127.0.0.1:59484
Received message:  Hello World!
Press Enter to terminate ...
```

Our message is successfully received. Now let's make sure we close both sockets (client-socket and listener-socket). For the TCP socket in listening or established state it is critical to enforce close() method before

terminating an application. If we not do so, the socket will stay there waiting for socket timeout till it gets cycled by the OS, due to statefulness of the TCP protocol (refer to Figure 2.11).

Sending a message using TCP connection

Sending the data using TCP is simplified a lot compared to UDP. Since there is a defined pipe between source and destination, and the pipe guarantees the delivery regardless of how much data we want to send (just we have to be sure, the remote endpoint is ready to receive that much). Therefore no additional addressing on send method is needed (socket is already connected to the endpoint). The code for sending the message:

```
from socket import AF_INET, SOCK_STREAM, socket
if __name__ == '__main__':
    print 'Application started'
    s = socket(AF_INET, SOCK_STREAM)
    print 'TCP Socket created'
    # No binding needed for client, OS will bind the socket automatically
    # when connect is issued
    server_address = ('127.0.0.1',7777)
    # Connecting ...
    s.connect(server_address)
    print 'Socket connected to %s:%d' % s.getpeername()
    print 'Local end-point is  bound to %s:%d' % s.getsockname()
     message = 'Hello World!'
    if s.sendall(message) == None:
        print 'Send %d bytes to %s:%d' % ( (len(message),)+s.getpeername())
    # Wait for user input before terminating application
    raw_input('Press Enter to terminate ...')
    s.close()
    print 'Terminating ...'
```

The output is then looking like:

```
Application started TCP Socket created
Socket connected to 127.0.0.1:7777
Local end-point is  bound to 127.0.0.1:58244
Send 11 bytes to 127.0.0.1:7777
Press Enter to terminate ...
Terminating ...
```

When we issue `sendall(...)` method on the socket object of the connected state the socket API then takes care of sending the data using TCP protocol. The data gets split and sent in blocks if necessary, and for each block sent the remote endpoint sends acknowledgement, ensuring the delivery. This way the TCP guarantees the delivery of the whole data. The sendall(...) method always returns None in case of successfully delivered data. In case of data-loss, it rises an Exception. There is also another method `send(...)` which gives more details regarding delivery, it just reports the number of bytes delivered. The application has to check if it was the right amount of bytes that was intended to be delivered (in this case we avoid throwing exceptions). According to socket API:

Note:

- *socket.send(string[, flags])*
  - Send data to the socket. The socket must be connected to a remote socket. The optional flags argument has the same meaning as for `recv()` above. Returns the number of bytes sent. Applications are responsible for checking that all data has been sent; if only some of the data was transmitted, the application needs to attempt delivery of the remaining data. For further information on this concept, consult the Socket Programming HOWTO.
- *socket.sendall(string[, flags])*
  - Send data to the socket. The socket must be connected to a remote socket. The optional flags argument has the same meaning as for `recv()` above. Unlike `send()`, this method continues to send data from string until either all data has been sent or an error occurs. None is returned on success. On error, an exception is raised, and there is no way to determine how much data, if any, was successfully sent.

Dealing with Big Amount of Data

Sending big message using TCP

```
from socket import AF_INET, SOCK_STREAM,  socket
if __name__ == '__main__':
    print 'Application started'
    # Creating a UDP/IP socket
    s = socket(AF_INET, SOCK_STREAM)
    # Sending the message (socket is still unbound)
    # We will use the break line to indicate the end of message
    # then the receiver may receive without knowing the full length
    # of the message he is receiving
    term = '\n'
    message = 'Hello world!'*7000+term
    destination = ('127.0.0.1',7777)
    # Connect to the server
    s.connect(destination)
    print 'Connected to the server %s:%d' % s.getpeername()
    print 'Local end-point bound on %s:%d' % s.getsockname()
    # Send the data
    s.sendall(message)
    print 'Sent message to %s:%d' % destination
    print 'Pay-load length %d bytes: [%s]' % (len(message),message)
    raw_input('Press Enter to terminate ...')
    print 'Closing the TCP socket ...'
    s.close()
    print 'Terminating ...'
```

Receiving big message using TCP

```
from socket import AF_INET, SOCK_STREAM, socket
if __name__ == '__main__':
    print 'Application started'
    # Creating a TCP/IP socket
    s = socket(AF_INET, SOCK_STREAM)
    # Binding the UDP/IP socket to address and port
    s.bind(('127.0.0.1',7777))
    print 'Socket is bound to %s:%d' % s.getsockname()
    # Turn the socket into listener-socket
    s.listen(0)
    print 'Socket is listening on %s:%d' % s.getsockname()
    # Wait for a client to connect
    client_socket,client_addr = s.accept()
    print 'Client connected from %s:%d' % client_addr
    # Receiving the message,
    # here we need to the socket API what is the size of the block
    # that we are ready to receive at once
    recv_buffer_length = 1024
    term = '\n'
    # Terminator indicating the message is over
    message = ''
    # This will grow in progress of receiving
    print 'Waiting for message ...'
    # Append message block by block till terminator is found
    while not message.endswith('\n'):
          m = client_socket.recv(recv_buffer_length)
        print 'Received block of %d from %s:%d'\
              '' % ( (len(m),) + client_socket.getpeername())
        message +=m
    print 'Total length %d bytes: [%s]' % (len(message),message)
    raw_input('Press Enter to terminate ...')
    client_socket.close()
    print 'Closed client socket...'
    s.close()
    print 'Closed the listener socket ...'
    print 'Terminating ...'
```

Reflection: as you can see the receiver knows exactly from what origin is the data received, as the TCP socket in connected state has only two endpoints. Therefore the `recv(...)` method does not return any additional information from where the data was received.

Ensuring Packet Arrival Order
TCP guarantees the data is delivered in the same order it was sent, therefore we do not need to program any additional routines or protocols to ensure the arrival order.

Note: The receiver has to know how much data there will be sent by sender also in case of TCP. In our example we chose to use the terminator to mark the end of message, however we could avoid using special terminator character and just add the total message length into first block of the message.

TCP socket shutdown routines

Shutdown is used in case we need to close the communication in one or both direction. As we mentioned before TCP socket has two transmitting pipes, the shutdown routine can close one of them or both. Usually shutdown is used to denote the end of transmission in one direction and start transmission in the other one. For example, when client did finish sending the data, the corresponding TX pipe is closed on client side and the client proceeds to receiving. Once client has closed the TX pipe, the server will receive empty buffer calling the recv(...), and understand that there is the end of received data. According to socket API:

- socket.shutdown(how)
  - Shut down one or both halves of the connection. If how is SHUT_RD, further receives are disallowed. If how is SHUT_WR, further sends are disallowed. If how is SHUT_RDWR, further sends and receives are disallowed. Depending on the platform, shutting down one half of the connection can also close the opposite half (e.g. on Mac OS X, shutdown(SHUT_WR) does not allow further reads on the other end of the connection).

The socket shutdown also does not close the I/O descriptors of the socket (the close(...) method does it).

Programming Application layer protocol on top of TCP

We will eventually implement the same example of MBoard protocol as in the previous chapter, but this time on top of TCP.

We may keep the same scenario with user willing to publish the message or to read all messages. Thanks to TCP it is not that important how big is the message the user wants to publish or how many messages are there already on server-side.

The following we leave one-to-one like it was declared in the MBoard of UDP version:

Note:
Let's say we want the public message board service to be shared with any number of users who want to publish or read messages. So we have two roles for network entities - the "message board service" and the "user".

Regarding scenarios we have changes though, definitely we do not need separate scenario for handling the big messages (like in UDP). Regarding the first scenario, we just say big messages are by default allowed (the only limits regarding the message length are now OS specific - maximal length of command line arguments, and on the server side - the amount of RAM to process the messages).

The resulting scenario is, refer to Figure :

- The message board service is responsible for serving user requests:
  - User may request to submit his message to public board
  - User may request the list of messages currently published on the board

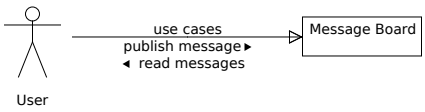


Figure 2.12: Use case of MBoard

Server-side

Regarding the protocol design, we will still use the request-response protocol (MBoard protocol), we just slightly adjust it taking into account the TCP features. The server side (server.py) we change it into TCP connection handler, once the server bound the socket, it is put into listening state and falls into loop. Within the loop we have the following processes:

1. Accept the client's connection (blocks until new client is connected)
2. Receive the data using client's socket till client issues shutdown on writing end (TX) of his socket.
3. Process the client's sent data (the request), prepare the response
4. Send the response data using client's socket
5. Close the client socket

You may have noticed we do not start accepting new client till we finished serving the previous one. Eventually it may happen that there will be new client asking for connection while the server is busy (serving the previous client as we still are using only one main loop with no additional threads). In order no to lose client we allow the listener socket to "remember" the clients who asked for connection, but whose connections is not yet accepted by the server. This is done using backlogging of the listener, according to socket API:

- socket.listen(backlog)
  - Listen for connections made to the socket. The backlog argument specifies the maximum number of queued connections and should be at least 0; the maximum value is system-dependent (usually 5), the minimum value is forced to 0.

Client-side

On client-side the sequence are mostly the same:

1. Process user input (get server's IP and port; options to publish message or read the last N messages)
   (a) In our implementation all the required values are asked from the user using command line arguments
2. Connect to server (create TCP socket, issue connect method)
3. Once connected to server:
   (a) Prepare the request data
   (b) Send data using connected socket
   (c) Shutdown writing pipe (TX) of the socket, when sending is over

(d) Receive the response

i. Show the results to the user

4. Disconnect the socket (issue close method)
5. Terminate the client's application

Changes in MBoard protocol

The routine "get last N messages" does not have to be split anymore into two subroutines. Therefore, we may use just one request/response iteration to get all the messages. New control code __REQ_GET_N_LAST (4) which instructs the server to give last N messages in one response.

Handling broken pipe exceptions

TCP is reliable enough and the data usually gets delivered, otherwise TCP reports socket error. This usually happens if send or receive is issued but at the same time the host gets disconnected from the network. These error reports may be taken into account when programming the application. In our case, for example on the server side when the send or receive is interrupted by socket error, we stop handling the client, close its socket, then socket is proceeded to the next client. On the other hand, if we have the same error on the client side, we just report communication error to the user and terminate.

Chapter 3

Threads

Introduction

During this lecture we will introduce threads, processes and synchronization among them. We start by introducing threads and its operations. Then, we will try to understand how to create and manipulate threads in python using examples and illustrations. In the second half we will introduce the network programming patterns using threads and processes. We conclude as usual showing the example of application illustrating how the obtained knowledge can be applied.

What is multitasking?

Before defining Multitasking let's define first the meaning of task in computer programming.

18. ***Task*** *is a generic term referring to an activity carried out by software that occurs independently from other activities in the system.*

Therefore, the multitasking can be defined as follows:

19. ***Multitasking*** *is the capability to perform more than one task at a time.*

A concept where multiple tasks are executed interleaved over certain period of time. The concept is an opposed to sequential execution - where the new task can not be started till the old is finished. Please note, that the term multitasking does not automatically mean executing tasks in parallel (concurrently). It means there is just more than one task progressing in time. The illusion of simultaneous execution is usually guaranteed letting the executor rapidly switch between the tasks (refer to Figure 3.1). In this case executor does a small portion of the first task, leaves it and proceeds with the second task etc. In fact, the described approach is widely applied in single executor environments:

- Instruction pipelining in obsolete single-core processors
- Cooperative multitasking in obsolete OS versions (Windows 3.x, 9x, etc.)

However, the amount of overall time required for completing for example three tasks is still the same for both sequential and multitasking concepts. Therefore, there is no actual gain in performance in multitasking, however in leaves an "impression" like the tasks are executed simultaneously. The only way to gain in

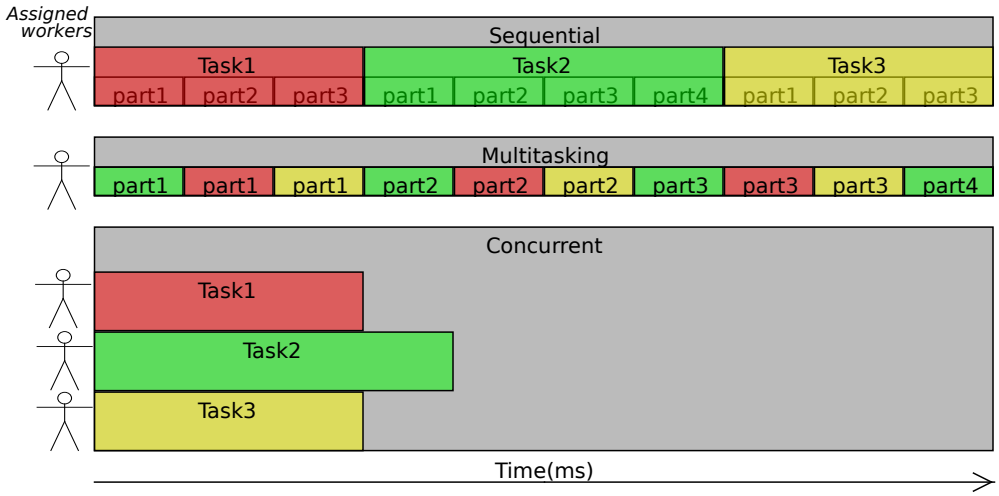


Figure 3.1: Task Execution Concepts

performance (have the tasks done faster) is to distribute the tasks among the multiple workers and let them execute. In this case there is "real" simultaneous execution of the tasks, and overall execution time is therefore reduced. The described concept is called "parallel" or "concurrent" and at this point we may introduce Threads and Process (correspond to workers in concurrent concept).

Processes and Threads

The terms Process and Thread are both related to the independent execution sequence (the worker assigned for a specific task) and are often confused. Processes are referred as threads and vice versa. Therefore we have to introduce and elaborate both terms before we actually start programming. In simple words: each time we start an application, the OS starts a separate process and allocates required resources (virtual address space, executable code, I/O descriptor, sockets, etc. what may be required by the application). Additionally inside the process one or many Threads are launched in order to execute the application's code. In case we have simple application with "main" method only, we will have only one "main" thread running inside our application process.

Starting a "main" thread is done automatically (without any calls to threading API) and therefore you might mistakenly think that there is no threads and the "main" method is run by the process directly. In case of multi-threaded applications the process will have several threads running and accessing the resources allocated for a process. The application has to be programmed taking into account the fact of possible concurrent use of resource, otherwise there is high risk for facing hardly traceable and irregular errors during application's runtime.

Even more "human friendly" explanation of processes and threads is given in George Coulouris book (see "Flies in a Jar" example):

"... An execution environment consists of a stoppered jar and the air and food within it. Initially, there is one fly – a thread – in the jar. This fly can produce other flies and kill them, as can its progeny. Any fly can consume any resource (air or food) in the jar. Flies can be programmed to queue up in an orderly manner to consume resources. If they lack this discipline, they might bump into one another within the jar – that is, collide and produce unpredictable results when attempting to consume the same resources in an unconstrained manner. Flies can communicate with (send messages to) flies in other jars, but none may escape from the jar, and no fly from outside may enter it. In this view, originally a UNIX process was a single jar with a single sterile fly within it."

Processes

A process is a passive entity such as a file on disk storage and it is also an active one. Besides, more processes can refer to the same program. This means, two instances of the same program have the same code section but with different current activities. From this perspective we can define Process as follows:

20. ***Process*** *is just a program in execution. It is modeled as a finite state machine which transits from state to state by executing a sequence of atomic actions.*

When it comes to the modern operating systems, it seems for the user like the OS is capable of running simultaneously the programs. This is not true, but they are using something called **timesharing** to manage multiple programs, which give this illusion of simultaneous execution.

21. *A* ***timesharing*** *operating system is that in which each task is given some time to execute and all tasks are given time so that all processes run seamlessly without any problem.*

As we defined before each running program counts as a **process** in UNIX terminology and also in Windows. Therefore, multiple copies of a program running counts as multiple processes. The processes will run by taking turns. This means, one process might run for a few milliseconds causing the OS to run till it is interrupted by the timer hardware. At this level, the OS saves the current state of the interrupted process in order to resume it later, then selects the next candidate process to give it a chance to run. This latter phenomena is called **context switch** (because the running CPU has switched from one process to another)**.** The cycle is repeated and any given process will get the Chance to run till eventually all processes are finished. The period of time when the process is allowed to run is called a **time slice** or **quantum**.

Moreover, the OS maintains a **process table**, where all current processes are listed. Each process will have state either **Run** or **Sleep**.

The Run state means that the current process is ready to run and the OS will choose on the processes in Run each time a turn ends. The Sleep state means that the process is waiting for an event to occur and we say that the process is **blocked**.

Threadsand

A thread is the entity within a process that can be scheduled for execution. All threads of a process share its virtual address space and system resources. Therefore threads occupy much less memory, plus it take less time to create then do processes.

The main goal of threads is to make it possible to write programs which run multiple tasks, and execute them in an efficient manner. Nowadays, the role of threads has become very important in applications programming such as in web servers development or GUI programs.

22. ***Thread*** *can be defined as a single sequential flow of control within a program. Sometimes called "lightweight" process because it runs within the context of a full-blown program and uses the resources allocated for that program and its environment.*

To resume our definition of threads, threads are the smallest units that can be scheduled in an operating system. Usually, they are contained in processes and there can be many of them within the same process. Moreover, as we stated before they share also all resources provided for the process in question.

There are two different kind of threads:

- Kernel threads

- User threads (or user-space threads)

For your information kernel threads are part of the operating system, meanwhile user threads are an extension of the function concept of a programming language (does not exit in the kernel).

Advantages of threading

We can resume the advantage of threading into four main ones:

- First, it allows programs to run faster on computer systems with the use of multiple CPUs, thanks to its concurrent execution aspect.
- Second, it help to maintain the program to be responsive in both cases: single or multiple CPU.
- Third, all the threads of a process can share the memory of the global variables allocated for that process and any change affect to them is noticeable by all the threads. (But thread can have also local variables)
- With threading it is possible to hide the time that it takes to send/receive messages (or synchronization) in parallel applications behind doing some useful activities simultaneously. Doing the communication in a separate thread does not block the CPU from performing some calculations, as an example.

Thread Managers

Thread manager is like a mini-operating system similar to a real operating system that maintains a table of processes, where thread system's thread manager maintains a table of threads. The moment a thread abandon the CPU or has its turn **Pre-empted**, then the thread manager looks in the table for another thread to activate.

23. ***Pre-emption*** *refers to the temporary interruption and suspension of a task, without asking for its cooperation, with the intention to resume that task later. This act is called a context switch and is typically performed by the pre-emptive scheduler, a component in the operating system authorized to pre-empt, or interrupt, and later resume tasks running in the system.*

A thread being **pre-empted** is the moment when an active thread within a given time slice is being interrupted by hardware the thread from a time cause control of the CPU and transfer it to the thread manager.

Another analogy with processes, a process is either in sleep mode or run mode, the same apply to threads. A thread is either ready to be given a turn to run, or is waiting for some event. As already mentioned, thread systems are either **kernel-level** or **user-level**.

Kernel-level thread managers

In kernel-level each thread is really like a process and the thread manager is like an OS. When different threads are set up by a given application they will take turns running like processes do. The moment a thread is activated, it has a specific time slice for running. When the time slice is over it will get pre-empted. Another case is when the thread reaches a point at which it has to wait for some other event to occurs before continuing, then it will voluntarily relinquish its time slice (this type of threads are used in Unix system and windows as well).

Remark: In kernel-level thread manager the fact that threads act exactly as a process, thus they will appear as a single running process when executing **ps** process command line in Unix.

User-level thread managers

User-level thread systems, on the other hand, are "private" to the application. Here the threads are not pre-empted; on the contrary, a given thread will continue to run until it voluntarily gives up control of the CPU, either by calling some "yield" function or by calling a function by which it requests a wait for some event to occur.

Remark: In User-level thread manager the fact that threads are private to the application, thus they will not appear as a single running process when executing **ps** process-command line in Unix but you will see the application to which they are associated with.

Note: Comparison
Kernel-level threads have the advantage that they can be used on multiprocessor systems, thus achieving true parallelism between threads.
On the other hand, User-level threads can allow one to produce code which is much easier to write, to debug, cleaner and clearer.

Python thread manager

Even though Python's threads mechanisms are built on top of the underlying platform's threads system, this is basically transparent to the application programmer. Which means that python threads are a blend of the kernel- and user-level approaches, but still more towards the user-level.

The interpreter keeps track on how long the current thread has been executing, in terms of the number of Python byte code instructions have executed. When it reaches a certain number, by default 10, another thread will be given a turn. Such a switch will also occur if a thread reaches an I/O statement.

Thus Python threads are pre-emptive. Internally, Python maintains a Global Interpreter Lock to ensure that only one thread has access to the interpreter at a time.

Threads modules in python

Python threads are accessible via two modules, **thread.py, threading.py**, and **multiprocessing.py**.

Note for your information: The thread module has been considered as deprecated for quite a long time. Users have been encouraged to use the threading module instead.

Thread module

The simplest and easy way to use a Thread is to instantiate it with a target function and call start_new_thread() to let it begin working.

```
import thread

def User():
    """thread User function"""
    print 'Hello I am User'
    return

for i in range (4):
     thread.start_new_thread(User,())

raw_input("Click Enter to Terminate ...")
```

The output should look like this:

```
Click Enter to Terminate ...
Hello I am User
Hello I am User
Hello I am User
Hello I am User
```

Although it is very effective for low-level threading, but the thread module is very limited compared to the newer threading module.

Threading module

The latest version of threading module in python is very powerful. It has high-level support for threads in comparison with the thread module presented in the previous section.

The Threading module contains all the methods of the thread module and more. Here is a collection of examples of additional methods that are useful for our course:

- **`threading.activeCount()`**: Returns the number of thread objects that are active.
- **`threading.currentThread()`**: Returns the number of thread objects in the caller's thread control.
- **`threading.enumerate()`**: Returns a list of all thread objects that are currently active.

Moreover, the thread class implemented threading are as follows:

- **`run()`**: The run() method is the entry point for a thread.
- **`start()`**: The start() method starts a thread by calling the run method.
- **`join([time])`**: The join() waits for threads to terminate.
- **`isAlive()`**: The isAlive() method checks whether a thread is still executing.
- **`getName()`**: The getName() method returns the name of a thread.
- **`setName()`**: The setName() method sets the name of a thread.

Now let's start exploring how to use some of the features or functions of threading module. Here we will show most important ones but not everything; therefore, be curious and try to explore more features about the module by your own.

Declaring a thread:

Concerning using the module threading is very simple you can trigger the threads by call the function start(), the following example will demonstrate how to use it:

```
import threading

def User():
    """thread User function"""
    print 'Hello I am User'
    return
threads = []
for i in range (4):
     t=threading.Thread(target=User)
     threads.append(t)
     t.start()

raw_input("Click Enter to Terminate ...")
```

The output should look like this:

```
Hello I am User
Hello I am User
Hello I am User
Hello I am User
Click Enter to Terminate ...
```

Using Argument with thread:

After demonstrating how to start threads in threading module, let's try now to spawn a thread by passing argument that it will include in its printing action:

```
import threading

def User(counter):
    """thread User function"""
    print 'Hello I am User number: %s' % counter
    return
threads = []
for i in range (4):
     t=threading.Thread(target=User, args=(i,))
     threads.append(t)
     t.start()

raw_input("Click Enter to Terminate ...")
```

The output should look like this:

```
Hello I am User number: 0
Hello I am User number: 1
Hello I am User number: 2
Hello I am User number: 3
Click Enter to Terminate ...
```

Identifying the current thread:

At this stage, we would like not only to use threads but also identifying them. Thus, we will use arguments to identify or name the thread by choosing a label. Normally, each thread instance has a name with a default value associate to it that we can change as the thread is created. The action of naming the threads is very important and useful in server processes especially when you are handling multiple service threads in different operations.

```
import threading
import time

def User():
    print threading.currentThread().getName(), 'Arrived'
    time.sleep(2)
    print threading.currentThread().getName(), 'Leaving'

def session():
    print threading.currentThread().getName(), 'Starting'
    time.sleep(3)
    print threading.currentThread().getName(), 'Exiting'

t = threading.Thread(name='session', target=session)
u = threading.Thread(name='User', target=User)
u2 = threading.Thread(target=User)# use default name

u.start()
u2.start()
t.start()
```

The output should look like this: (in this output the name of the current thread is on each line except the line with "Thread-1" is the unnamed thread u2)

```
User Thread-1 Arrived
Arrived
 session Starting
Thread-1 Leaving
User Leaving
session Exiting
```

Logging debug:

Most of clean code or program we don't use print statement as a mean to debug. Thanks to the logging module, it supports embedding the thread name in every log message. The following code demonstrates how to make use of it:

```
import threading
import time
import logging

 logging.basicConfig(level=logging.DEBUG,
                           format='[%(levelname)s](%(threadName)-10s)%(message)s',
)
```

```
7
8  def User():
9      logging.debug('Arrived')
10     time.sleep(2)
11     logging.debug(Leaving)
12
13 def session():
14     logging.debug('Starting')
15     time.sleep(3)
16     logging.debug('Exiting')
17
18 t = threading.Thread(name='session', target=session)
19 u = threading.Thread(name='User', target=User)
20 u2 = threading.Thread(target=User)# use default name
21
22 u.start()
23 u2.start()
24 t.start()
```

The output should look like this:

```
[DEBUG] (User      ) Arrived
[DEBUG] (Thread-1  ) Arrived
[DEBUG] (session   ) Starting
[DEBUG] (User      ) Leaving
5 [DEBUG] (Thread-1  ) Leaving
[DEBUG] (session   ) Exiting
```

Daemon threads:

Till now all the examples we showed the programs waited till all the threads exit and complete their work. However, when a program spawns a thread as daemon that runs without blocking the main program from exiting.

The use of daemon thread is important when you want your thread to complete its task no matter what happened to the hosting program.

let's demonstrate how it works using the **`setDaemon()`** method:

```
1  import threading
2  import time
3  import logging
4
5   logging.basicConfig(level=logging.DEBUG,
6                       format='[%(levelname)s](%(threadName)-10s)%(message)s',
   )
7
8  def User1():
9      logging.debug('Arrived')
10     time.sleep(2)
11     logging.debug(Leaving)
12
13 d= threading.Thread(name='User1', target=User1)
```

```
d.setDaemon(True) #we set User1 to daemon thread

def User2():
    logging.debug('Arrived')
    logging.debug('Leaving')

t = threading.Thread(name='User2', target=User2)

d.start()
t.start()
```

The output should look like this:

```
[DEBUG] (User1      ) Arrived
[DEBUG] (User2      ) Arrived
[DEBUG] (User2      ) Leaving
```

you can notice from the output that User1 thread did not leave because it is a daemon thread. Since all of the non-daemon threads (including the main thread) exit before the daemon thread wakes up from its two second sleep.

In order to wait until a daemon thread has completed its work, we can use the **`join()`** method.

```
import threading
import time
import logging

 logging.basicConfig(level=logging.DEBUG,
                          format='[%(levelname)s](%(threadName)-10s)%(message)s',
)

def User1():
    logging.debug('Arrived')
    time.sleep(2)
    logging.debug(Leaving)

d= threading.Thread(name='User1', target=User1)
d.setDaemon(True) #we set User1 to daemon thread

def User2():
    logging.debug('Arrived')
    logging.debug('Leaving')

t = threading.Thread(name='User2', target=User2)

d.start()
t.start()

d.join()
t.join()
```

The output should look like this:

```
[DEBUG] (User1      ) Arrived
```

```
[DEBUG] (User2      ) Arrived
[DEBUG] (User2      ) Leaving
[DEBUG] (User1      ) Leaving
```

Locks in threads:

It is very important to prevent and control corruption of data when handling any communication. Thus, it is necessary to watch out the accesses to an object. In order to do so, we have to use **lock** object in our code.

24. ***Lock*** *is a synchronization primitive which is not owned by a particular thread when locked.*

A primitive lock is one of two states: "locked" or "unlocked" and it is created in the unlocked state. Moreover, it has two basic methods: **`acquire()`** and **`release()`**.

Let's now discuss some specific cases:

- In case the state is unlocked, then **`acquire()`** will changes the state to locked and returns immediately.
- in case the state is locked, then **`acquire()`** will block until a call to **`release()`** in another thread changes it to unlocked, then **`acquire()`** call resets it to locked and return.

> **Remark:** The **`release()`** method should only be called in the locked state; it changes the state to unlocked and returns immediately. If an attempt is made to release an unlocked lock, a **RuntimeError** will be raised.

Here is an illustration of lock object:

```
1  import threading
2  import time
3  import logging
4
5  logging.basicConfig(\
6       level=logging.DEBUG,\
7       format='(%(threadName)-9s) %(message)s',)
8
9  class Counter(object):
10     def __init__(self, start = 0):
11         self.lock = threading.Lock()
12         self.value = start
13
14    def increment(self):
15         logging.debug('Waiting for a lock')
16         self.lock.acquire()
17         try:
18             logging.debug('Acquired a lock')
19             self.value = self.value + 1
20             logging.debug('Value %d' % self.value)
21         finally:
22             logging.debug('Released a lock')
23             self.lock.release()
```

```
24
25      def decrement(self):
26          logging.debug('Waiting for a lock')
27          self.lock.acquire()
28          try:
29              logging.debug('Acquired a lock')
30              self.value = self.value - 1
31              logging.debug('Value %d' % self.value)
32          finally:
33              logging.debug('Released a lock')
34              self.lock.release()
35
36  def worker_a(counter):
37      for i in range(10):
38          logging.debug('Sleeping %0.02f', 1)
39          time.sleep(1)
40          counter.increment()
41      logging.debug('Done')
42
43  def worker_b(counter):
44      for i in range(10):
45          logging.debug('Sleeping %0.02f', 2)
46          time.sleep(2)
47          counter.decrement()
48      logging.debug('Done')
49
50  if __name__ == '__main__':
51      counter = Counter()
52      t_inc = threading.Thread(target=worker_a, args=(counter,))
53      t_dec = threading.Thread(target=worker_b, args=(counter,))
54      t_inc.start()
55      t_dec.start()
56      logging.debug('Waiting for worker threads')
57      t_inc.join()
58      t_dec.join()
59      logging.debug('Counter final value: %d', counter.value)
```

The output should look like this:

```
(Thread-1 ) Sleeping 1.00
(Thread-2 ) Sleeping 2.00
(MainThread) Waiting for worker threads
(Thread-1 ) Waiting for a lock
(Thread-1 ) Acquired a lock
(Thread-1 ) Value 1
(Thread-1 ) Released a lock
(Thread-1 ) Sleeping 1.00
(Thread-1 ) Waiting for a lock
(Thread-1 ) Acquired a lock
(Thread-1 ) Value 2
(Thread-1 ) Released a lock
(Thread-1 ) Sleeping 1.00
(Thread-2 ) Waiting for a lock
(Thread-2 ) Acquired a lock
```

```
(Thread-2 ) Value 1
(Thread-2 ) Released a lock
(Thread-2 ) Sleeping 2.00
....
```

Conditional Variables in threads:

Typical situation in thread programming is when we need to acquire a lock for a queue and pop one object from the queue. The queue is empty from objects but at the same time there is an initiated process of populating the queue with objects by another thread. In this situation we need to wait until the queue has at least one object. Then, we can of course put the thread into the sleeping mode for a while and retry checking the queue later (if there are new objects in the queue). However, this action does not guarantee us the ideal timing as the sleeping timeout does not necessarily happen at the same time when the queue gets populated with at least one object. In fact, what we want is that our sleeping thread gets woken up exactly at the time the queue gets first object inserted into it.

This is typical scenario for Conditional Variables and our first thread is waiting on condition "until queue is empty". The waiting thread then expects, it will get notification from someone (by itself the waiting thread will never wake up until interrupted). The second thread (the one populating the queue) knows that there might be other threads waiting for queue to get at least one element. Therefore once the first element is inserted, the populating thread notifies the waiting thread. The consumer thread in this case gets woken up exactly at the time the queue has new element inserted, and this is how exact timing is guaranteed between populating and consuming threads. Please note in this example we do not have any threads "idling free", which means iterating in loop: if the queue is empty - sleep, else - consume an element.

25. ***Conditions*** *are a wrapper around an underlying* ***Lock*** *that provide wait/notify functionality. Condition is used when threads are interested in waiting for something to become true (by issuing* ***`wait()`*** *method on conditional variable), and once its true, the thread has exclusive access to some shared resource. On the other hand, the* ***`notify()`*** *and* ***`notifyAll()`*** *are there to inform waiting threads about the fact that something it was waiting for became true.*

Here is the example of Conditional Variables in threads:

```
1  import logging
2  import threading
3  import time
4
5  logging.basicConfig(\
6       level=logging.DEBUG,\
7      format='%(asctime)s (%(threadName)-2s) %(message)s',)
8
9  class Queue:
10     def __init__(self):
11         self.__q = []
12         self.__cv = threading.Condition()
13
14     def put(self,e):
15         with self.__cv:
16             logging.debug('Added element')
17             self.__q.append(e)
18             if len(self.__q) == 1:
```

```
                # First element added - notify all waiting threads
                logging.debug('Queue is no more empty, notifying all'\
                              'waiting threads')
                self.__cv.notifyAll()

    def get(self):
        e = None
        with self.__cv:
            if len(self.__q) <= 0:
                logging.debug('Queue is empty, waiting ...')
                self.__cv.wait()
            e = self.__q.pop()
        return e

def consumer(q):
    """wait for the condition and use the resource"""
    logging.debug('Starting consumer thread')
    for i in range(5):
        e = q.get()
        logging.debug('Retrieved: %d' % e)
        time.sleep(1)

def producer(q):
    """set up the resource to be used by the consumer"""
    logging.debug('Starting producer thread')
    for i in range(5):
        q.put(i)
        logging.debug('Added: %d' % i)
        time.sleep(3)

q = Queue()
cons = threading.Thread(name='Consumer', target=consumer, args=(q,))
prod = threading.Thread(name='Producer', target=producer, args=(q,))
cons.start()
prod.start()
cons.join()
prod.join()
logging.debug('Terminating ...')
```

The output should look like this:

```
2016-10-03 09:27:35,002 (Consumer) Starting consumer thread
2016-10-03 09:27:35,005 (Consumer) Queue is empty, waiting ...
2016-10-03 09:27:35,010 (Producer) Starting producer thread
2016-10-03 09:27:35,010 (Producer) Added element
2016-10-03 09:27:35,010 (Producer) Queue is no more empty, notifying allwaiting threads
2016-10-03 09:27:35,011 (Producer) Added: 0
2016-10-03 09:27:35,011 (Consumer) Retrieved: 0
2016-10-03 09:27:36,012 (Consumer) Queue is empty, waiting ...
2016-10-03 09:27:38,014 (Producer) Added element
2016-10-03 09:27:38,014 (Producer) Queue is no more empty, notifying allwaiting threads
2016-10-03 09:27:38,014 (Producer) Added: 1
2016-10-03 09:27:38,015 (Consumer) Retrieved: 1
2016-10-03 09:27:39,016 (Consumer) Queue is empty, waiting ...
```

```
2016-10-03 09:27:41,016 (Producer) Added element
2016-10-03 09:27:41,016 (Producer) Queue is no more empty, notifying allwaiting threads
2016-10-03 09:27:41,017 (Producer) Added: 2
2016-10-03 09:27:41,017 (Consumer) Retrieved: 2
2016-10-03 09:27:42,018 (Consumer) Queue is empty, waiting ...
2016-10-03 09:27:44,017 (Producer) Added element
2016-10-03 09:27:44,018 (Producer) Queue is no more empty, notifying allwaiting threads
2016-10-03 09:27:44,018 (Producer) Added: 3
2016-10-03 09:27:44,018 (Consumer) Retrieved: 3
2016-10-03 09:27:45,019 (Consumer) Queue is empty, waiting ...
2016-10-03 09:27:47,019 (Producer) Added element
2016-10-03 09:27:47,020 (Producer) Queue is no more empty, notifying allwaiting threads
2016-10-03 09:27:47,020 (Producer) Added: 4
2016-10-03 09:27:47,020 (Consumer) Retrieved: 4
2016-10-03 09:27:50,021 (MainThread) Terminating ...
```

Events in threads:

Another possibility to synchronize the Threads in python is to use Event objects. For cultivating yourself about Events in threading check the link below:

http://www.bogotobogo.com/python/Multithread/python_multithreading_Event_Objects_between_Threads.php

Multiprocessing module

Python API for managing Processes is reduced in “multiprocessing” module and is very similar to the one of the “threading” module. This way it is much more easier for programmer to master the Multiprocessing API once he/she knows the Threading API. Surely these two APIs are not one-to-one similar, all the differences are quiet understandable once we understand the concepts of threads and processes.

The differences in Threading and Multiprocessing API are explained well in the tutorial below:

https://pymotw.com/2/multiprocessing/basics.html

Chapter 4

Shared state

Introduction

In the previous chapter, we introduced threads and thread programming models [7]. Apparently, Python threads are not good for parallel computing, as the Python's global interpreter lock (GIL) allows only one thread to execute the code at the time. However the waiting threads are allowed to perform system calls. Handling I/O descriptors are one of those calls (read/write on file or send/receive on socket) [4]. In other words Python threads may outperform on parallel computing but they are still good for parallel I/O handling. As for true parallel execution of the code, different module - *multiprocessing* is advised [5].

Therefore, this time we will employ the thread programming models to handle network related I/O routines in our applications. We will start by improving request-response protocol servers focusing on servers-side mostly. After defining the server-side multi-threading patterns for request-response protocols, we will switch to more advanced long-lived TCP connections and cover the application protocols that allow server to send asynchronous notifications to clients. At this level, we will apply multi-threading for both client and server. Then, we will cover these essential aspects of threads and network – we will introduce the shared state topic and illustrate on simple network application (real-time concurrent modification of a shared data structure).

Threads in network applications

We should remember that the essential idea behind threads or multiple processes is the ability to execute in separate flow. However, we figured out that when it comes to computing, the Python **`threading`** module is not very suitable compared to **`multiprocessing`**. In addition, if we need more efficient computations, we have to deploy all the cores of our CPU equally and this is exactly the task of `multiprocessing` module in Python. In case we just need to do many I/O routines simultaneously (like reading many files in parallel), the `threading` module of Python is quite suitable and can be recommended for handling these routines.

Simple example

Sending and receiving the data using the network sockets are also I/O tasks. Therefore, it is not a bad idea to handle each particular socket in a separate thread. For example, having an application to download four files, we will eventually win in performance if we do download all four simultaneously as opposed to downloading all four sequentially. In the following code, we use Pythons **`urllib`** module to handle HTTP

```
from tempfile import mktemp
import logging
import urllib
from time import time
logging.basicConfig(level=logging.DEBUG,\
                         format='%(asctime)s (%(threadName)-2s) %(message)s',)

def download(url):
     logging.info('Downloading %s' % url)
     tmpfilepath= mktemp()
     urllib.urlretrieve(url,tmpfilepath)
     logging.info('Finished downloading %s' % url)

if __name__ == '__main__':
     urls = [
          'https://cdn.kernel.org/pub/linux/kernel/v4.x/testing/linux-4.9-rc1.tar.xz',
          'https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.8.2.tar.xz',
          'https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.7.8.tar.xz',
          'https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.4.25.tar.xz'
     ]

     logging.debug('Application start ...')
     start = time()
     for u in urls:
          download(u)
     t = time() - start
     logging.debug('Terminating, total time spent %d sec ...' % t)
```

Figure 4.1: HTTP Download manager, single threaded

protocol (to download the files using HTTP), the first example is the sequential version (refer to Figure 4.1); and the next one (refer to Figure 4.2) illustrates the same application using threads. Here, we have used a *Delegation* or *Manager-Workers* thread programming model where the main method assigns a download task (the *download* method with a URL argument) to a worker thread.

```
from tempfile import mktemp
import logging
import urllib
from time import time
from threading import Thread

logging.basicConfig(level=logging.DEBUG,
                    format='%(asctime)s (%(threadName)-2s) %(message)s',
                    )

def download(url):
    logging.info('Downloading %s' % url)
    tmpfilepath= mktemp()
    urllib.urlretrieve(url,tmpfilepath)
    logging.info('Finished downloading %s' % url)

if __name__ == '__main__':

     urls = [
          'https://cdn.kernel.org/pub/linux/kernel/v4.x/testing/linux-4.9-rc1.tar.xz',
          'https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.8.2.tar.xz',
          'https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.7.8.tar.xz',
          'https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.4.25.tar.xz'
    ]

    logging.debug('Application start ...')
    threads = []
    start = time()
    for u in urls:
        t = Thread(target=download,args=(u,))
        threads.append(t)
        t.start()
    for t in threads:
        t.join()
    t = time() - start
    logging.debug('Terminating, total time spent %d sec ...' % t)
```

Figure 4.2: HTTP Download manager multi- threaded

```
from socket import AF_INET, SOCK_STREAM, socket, SHUT_WR
if __name__ == '__main__':
    print 'Application started'
    s = socket(AF_INET, SOCK_STREAM)
    print 'TCP Socket created'
     s.bind(('127.0.0.1',7777))
     s.listen(1)
     client_socket,client_addr = s.accept()
    print 'New client connected from %s:%d' % client_addr
     print 'Local end-point socket bound on: %s:%d'\
           '' % client_socket.getsockname()
    # Simulate service offer by sending one byte to client
    client_socket.send('1')
    client_socket.shutdown(SHUT_WR)
    print 'Service offered, client is using a service ...'
    # Wait for the client to close connection
    # This one will block till client sends something or disconnects
    client_socket.recv(1)
    print 'Client finished using service'
    # Here we assume client is gone
    client_socket.close()
    print 'Client disconnected'
    # Wait for user input before terminating server application
    raw_input('Press Enter to terminate ...')
    s.close()
    print 'Terminating ...'
```

Figure 4.3: Single threaded server, one client at most (singlethreaded_server_one_client.py)

Client-server architectures

Now let's take a look how the thread programming models are applied for server side applications. We will start with more simple example of request-response protocol. As we saw in previous chapter, the pattern has one server and some clients. The clients connect and make requests to the server. The server accepts the connections and replies to the received messages. Let's first of all take a look at reference implementation:

Request-Response protocols

Let's take the server side created during the TCP lecture, this example has a limitation when two clients try to get connected at the same time – the server will interact only with one of them. The second one will be rejected (with TCP connection reset). Even if we increase the backlog size – the second client still has to wait in a queue till the server has finished processing the request of the first client. In the next code (refer to Figure 4.3) there is a server capable of serving only one client. And the next one (refer to Figure 4.4) the same server is capable of serving many clients, however the client requests are still processed sequentially.

Testing the servers with client's simulator

For better understanding of what happens to our clients let's perform a simple simulation. Suppose multiple clients are connecting to the server simultaneously. Once a client gets connected it expects the server to start dealing with it immediately, otherwise the client gets bored and leaves. The code in Figure 4.5 illustrates the corresponding simulator. Let's see how the server will perform with 5 clients simultaneously. We will

```
from socket import AF_INET, SOCK_STREAM, socket, SHUT_WR
from socket import error as soc_err

def handle_client(client_socket):
    try:
        print 'New client connected from %s:%d' % client_addr
        print 'Local end-point socket bound on: %s:%d'\
        '' % client_socket.getsockname()
        # Simulate service offer by sending one byte to client
        client_socket.send('1')
        client_socket.shutdown(SHUT_WR)
        print 'Service offered, client %s:%d is using a service ...'\
                '' % client_addr
        # Wait for the client to close connection
        # This one will block till client sends something or disconnects
        client_socket.recv(1)
        print 'Client %s:%d finished using service' % client_addr
        # Here we assume client is gone
    except soc_err as e:
        if e.errno == 107:
            print 'Client %s:%d left before server could handle it'\
            '' %  client_addr
        else:
            print 'Error: %s' % str(e)
    finally:
        client_socket.close()
    print 'Client %s:%d disconnected' % client_addr

if __name__ == '__main__':
    print 'Application started'
    s = socket(AF_INET, SOCK_STREAM)
    s.bind(('127.0.0.1',7777))
    s.listen(1)
    print 'Socket %s:%d is in listening state' % s.getsockname()
    print 'Falling to serving loop, press Ctrl+C to terminate ...'
    try:
        while 1:
            client_socket = None
            print 'Awaiting new clients ...'
            client_socket,client_addr = s.accept()
            handle_client(client_socket)
    except KeyboardInterrupt:
        print 'Ctrl+C issued closing server ...'
    finally:
        if client_socket != None:
            client_socket.close()
        s.close()
    print 'Terminating ...'
```

Figure 4.4: Single threaded server, many clients (singlethreaded_server_many_clients.py)

run the simulator's code as it is (Figure 4.5), having each client keeping the server busy for 5 seconds, and in case client cannot get immediate service, it waits for 15 seconds and leaves. The simulation results are shown in Figure 4.6 corresponding server's log is shown in Figure 4.7. As you can see, the server could handle 4 clients, and did not manage to handle last one before it disconnected. The reason is of course the sequential handling of the clients. The first client got service immediately and kept server busy for 5 seconds. The remaining 4 clients were still waiting loosing the seconds till timeout. Server managed to handle another 3 clients losing 5 seconds for each, the last client however was gone already (15 seconds timeout).

Question: What will be the results if the client can wait only for 5 seconds to get service, and each client keeps server busy for 30 seconds ?

Global Variables:

- DEFAULT_TIMEOUT_SEC = 5
- DEFAULT_STAY_CONNECTED_SEC = 30

Adding thread models to the server side

In order to make our server code more flexible in handling many clients simultaneously, we have to introduce threads. Next, the previous server code will be rewritten to handle multiple clients connections using threads. To do so, we have two options:

- Dealing with each connection in a separate thread (thread-per-connection)
- Dealing with each request in a separate thread (thread-per-request)
  - * in case there are many requests sent using one connection

Thread-per-connection using different thread models

- Delegation (Manager-Workers):
  - The connection handling is explicitly delegated to a new thread. Each time we call the *handle_client* method, we call it in a separate execution flow, releasing the *main* execution flow to continue accepting new clients. In other words: once the new client's connection is accepted by main server loop, the client's socket is delegated to a separate Thread for handling. The thread is then started with *handle_client* method and *client_socket* parameter. With the model like that we create a new thread each time new client connects (guaranteeing each client gets handled immediately), therefore it is also referred as ***thread-per-socket***.
    - As you may have noticed the amount of the threads created is defined by amount of incoming clients, therefore it can vary from zero to a huge number (in case of that many incoming clients).
- Producer-Consumer pattern
  - Indirect delegation using queue of client connections. The threads try to get Each time we accept new client's connection, we put it into client connections queue. Handling the client's connection is also done in a separate thread. The threads however are not created *on-demand,* instead a predefined set of threads is created once the application is started. With this model the amount of threads is not changed, the amount of clients that are handled simultaneously is defined by a number of predefined threads and the size of the queue. The model is therefore also referred as ***pre-fork***.

```
import logging logging.basicConfig(level=logging.DEBUG,
        format='%(asctime)s (%(threadName)-2s) %(levelname)s %(message)s',)
from socket import socket, AF_INET, SOCK_STREAM, SHUT_WR
from socket import error as soc_err
from socket import timeout as soc_err_timeout
from time import sleep from threading import Thread
DEFAULT_CLIENTS = 5
DEFAULT_SERVER_HOST = '127.0.0.1'
DEFAULT_SERVER_PORT = 7777
DEFAULT_STAY_CONNECTED_SEC = 5
DEFAULT_TIMEOUT_SEC = 15
DEFAULT_RECV_BUFSIZE = 1

def connect_toserver(socket_addr, timeout, stay_connected):
    s = socket(AF_INET,SOCK_STREAM)
    s.settimeout(timeout)
    try:
        logging.debug('Trying to connect to %s:%d' % socket_addr)
        s.connect(socket_addr)
        logging.info('Connected, waiting for service ...')
        # Just let's say if recieved at least something let's count it
        # server dealing with us
        m = s.recv(DEFAULT_RECV_BUFSIZE)
        if len(m) > 0:
            logging.info('Service available! '\
                         'Keeping it busy for %d sec ...' % stay_connected)
        # Simulate service usage by waiting for a while before disconnecting
            sleep(stay_connected)
            logging.info('Finished using a service ...')
            s.shutdown(SHUT_WR)
    except soc_err_timeout:
        logging.error('Timeout waiting for service!')
    except soc_err as e:
        if e.errno == 107:
            logging.error('Server closed connection, error %s ...' % str(e))
        else:
            logging.error('Can\'t connect to %s:%d, error %s'\
                      '' % (socket_addr+(str(e),)) )
    except KeyboardInterrupt:
        logging.info('Ctrl+C issued, terminating')
    finally:
        s.close()
    logging.info('Disconnected ...')

if __name__ == '__main__':
    logging.info('Application start')
    logging.info('%s clients will connect to %s:%s ...'\
            '' % (DEFAULT_CLIENTS, DEFAULT_SERVER_HOST, DEFAULT_SERVER_PORT))
    logging.debug('Connection timeout %s sec' % DEFAULT_TIMEOUT_SEC)
    logging.debug('Stay connected for %s sec' % DEFAULT_STAY_CONNECTED_SEC)
    threads = [] # Prepare clients
    for i in range(DEFAULT_CLIENTS):
        t = Thread(target=connect_toserver,
                   args=((DEFAULT_SERVER_HOST,int(DEFAULT_SERVER_PORT)),
                         int(DEFAULT_TIMEOUT_SEC),\
                         int(DEFAULT_STAY_CONNECTED_SEC)))
        threads.append(t)
    logging.debug('Client threads created ...')
    map(lambda x: x.start(),threads) # Start all
    logging.debug('Client threads started ...')
    map(lambda x: x.join(),threads)
    logging.debug('Client threads dead ...') # Wait for all
    logging.info('Terminating ...')
```

Figure 4.5: Clients simulator (using threads) (simulating_many_connections.py)

```
56:54,257 (MainThread) INFO Application start
56:54,261 (MainThread) INFO 5 clients will connect to 127.0.0.1:7777 ...
56:54,261 (MainThread) DEBUG Connection timeout 15 sec
56:54,262 (MainThread) DEBUG Stay connected for 5 sec
56:54,264 (MainThread) DEBUG Client threads created ...
56:54,268 (Thread-1) DEBUG Trying to connect to 127.0.0.1:7777
56:54,268 (Thread-1) INFO Connected, waiting for service ...
56:54,268 (Thread-1) INFO Service available! Keeping it busy for 5 sec ...
56:54,269 (Thread-2) DEBUG Trying to connect to 127.0.0.1:7777
56:54,269 (Thread-3) DEBUG Trying to connect to 127.0.0.1:7777
56:54,270 (Thread-3) INFO Connected, waiting for service ...
56:54,270 (Thread-4) DEBUG Trying to connect to 127.0.0.1:7777
56:54,270 (Thread-4) INFO Connected, waiting for service ...
56:54,270 (MainThread) DEBUG Client threads started ...
56:54,271 (Thread-5) DEBUG Trying to connect to 127.0.0.1:7777
56:54,276 (Thread-2) INFO Connected, waiting for service ...
56:54,278 (Thread-5) INFO Connected, waiting for service ...
56:59,270 (Thread-1) INFO Finished using a service ...
56:59,271 (Thread-1) INFO Disconnected ...
56:59,271 (Thread-2) INFO Service available! Keeping it busy for 5 sec ...
57:04,272 (Thread-2) INFO Finished using a service ...
57:04,273 (Thread-2) INFO Disconnected ...
57:04,273 (Thread-3) INFO Service available! Keeping it busy for 5 sec ...
57:09,279 (Thread-3) INFO Finished using a service ...
57:09,279 (Thread-3) INFO Disconnected ...
57:09,280 (Thread-4) INFO Service available! Keeping it busy for 5 sec ...
57:09,291 (Thread-5) ERROR Timeout waiting for service!
57:09,292 (Thread-5) INFO Disconnected ...
57:14,283 (Thread-4) INFO Finished using a service ...
57:14,284 (Thread-4) INFO Disconnected ...
57:14,284 (MainThread) DEBUG Client threads dead ...
57:14,284 (MainThread) INFO Terminating ...
```

Figure 4.6: Client Simulator output

```
Application started Socket
127.0.0.1:7777 is in listening state
Falling to serving loop, press Ctrl+C to terminate ...
Awaiting new clients ...
New client connected from 127.0.0.1:40366
Local end-point socket bound on: 127.0.0.1:7777
Service offered, client 127.0.0.1:40366 is using a service ...
Client 127.0.0.1:40366 finished using service
Client 127.0.0.1:40366 disconnected
Awaiting new clients ...
New client connected from 127.0.0.1:40367
Local end-point socket bound on: 127.0.0.1:7777
Service offered, client 127.0.0.1:40367 is using a service ...
Client 127.0.0.1:40367 finished using service
Client 127.0.0.1:40367 disconnected Awaiting new clients ...
New client connected from 127.0.0.1:40368
Local end-point socket bound on: 127.0.0.1:7777
Service offered, client 127.0.0.1:40368 is using a service ...
Client 127.0.0.1:40368 finished using service
Client 127.0.0.1:40368 disconnected Awaiting new clients ...
New client connected from 127.0.0.1:40369
Local end-point socket bound on: 127.0.0.1:7777
Service offered, client 127.0.0.1:40369 is using a service ...
Client 127.0.0.1:40369 finished using service
Client 127.0.0.1:40369 disconnected
Awaiting new clients ...
New client connected from
127.0.0.1:40370 Local end-point socket bound on:
127.0.0.1:7777 Client 127.0.0.1:40370 left before server could handle it
Client 127.0.0.1:40370 disconnected
Awaiting new clients ...

^CCtrl+C issued closing server ...
Terminating ...
```

Figure 4.7: Single threaded servers output

1. accept client's connection (creating client's socket)
2. receive the client's request (reading it from the client's socket)
3. prepare the response (processing clients request)
4. send the response to client (writing it to the client's socket)
5. close client's connection

Table 4.1: Client's Connection Handling Sequence

* As *pre-fork* model is having fixed amount of threads associated through a queue, it is easily portable from threading to multiprocessing - having predefined number of processes started early to handle the upcoming client requests.

Next we will take our single threaded server and make two improved versions out of it:

- using delegation pattern (illustrated in File [*multithreaded_server_delegation_pattern.py*])
- using producer-consumer pattern (illustrated in File [*multithreaded_server_prefork_pattern.py*])

Let's now measure performance of those versions having simulator settings as follows:

Global Variables:

- DEFAULT_TIMEOUT_SEC = 5
- DEFAULT_STAY_CONNECTED_SEC = 30

The simulators output looks same for both server implementations: all the client got service in time (refer to Figure 4.10 and 4.11). The output of the *thread-per-socket* model is illustrated in Figure 4.8 and the *pre-fork* model in Figure 4.9. We can see from server logs that in the first model the server created additional threads each time client was connected. In second model, eight threads were created at early point of application startup. Each thread was assigned to check for new sockets in the queue (which is empty in the beginning). Each time the main server loop accepts new connection, the corresponding socket is put into queue and then the waiting threads are invoked.

Question: What will happen if we set the following global variable in the *pre-fork* server's code (simulator stays with the same settings):

- PREFORK_THREADS = 4

Thread-per-request using different thread models

In our previous examples, we did not apply actual load (no request/response were sent), we were just simulating load by keeping the connection (that was opened by a client) alive. In our real applications (like Message Board and File Server), we have actual requests like "publish message", "get messages" or "upload file". Hence the client handling sequence on server side is as shown in Table 4.1.

Let's suppose we have a separate thread taking care of the client's connections. Once the client is connected (step 1) the corresponding socket is either delegated explicitly to a thread to handle or it is stored in a queue

```
18:21:29,396 (MainThread) Application started
18:21:29,396 (MainThread) Socket 127.0.0.1:7777 is in listening state
18:21:29,396 (MainThread) Falling to serving loop, press Ctrl+C to terminate ...
18:21:29,396 (MainThread) Awaiting new clients ...
18:21:41,577 (Thread-1) New client connected from 127.0.0.1:40429
18:21:41,577 (Thread-1) Local end-point socket bound on: 127.0.0.1:7777
18:21:41,577 (MainThread) Awaiting new clients ...
18:21:41,578 (Thread-1) Service offered, client 127.0.0.1:40429 is using a service ...
18:21:41,580 (Thread-2) New client connected from 127.0.0.1:40430
18:21:41,580 (MainThread) Awaiting new clients ...
18:21:41,580 (Thread-2) Local end-point socket bound on: 127.0.0.1:7777
18:21:41,581 (Thread-2) Service offered, client 127.0.0.1:40430 is using a service ...
18:21:41,584 (Thread-3) New client connected from 127.0.0.1:40431
18:21:41,584 (MainThread) Awaiting new clients ...
18:21:41,584 (Thread-3) Local end-point socket bound on: 127.0.0.1:7777
18:21:41,584 (Thread-3) Service offered, client 127.0.0.1:40431 is using a service ...
18:21:41,593 (Thread-4) New client connected from 127.0.0.1:40432
18:21:41,593 (MainThread) Awaiting new clients ...
18:21:41,593 (Thread-4) Local end-point socket bound on: 127.0.0.1:7777
18:21:41,594 (Thread-4) Service offered, client 127.0.0.1:40432 is using a service ...
18:21:41,594 (Thread-5) New client connected from 127.0.0.1:40433
18:21:41,594 (MainThread) Awaiting new clients ...
18:21:41,595 (Thread-5) Local end-point socket bound on: 127.0.0.1:7777
18:21:41,595 (Thread-5) Service offered, client 127.0.0.1:40433 is using a service ...
18:22:11,608 (Thread-1) Client 127.0.0.1:40433 finished using service
18:22:11,610 (Thread-2) Client 127.0.0.1:40433 finished using service
18:22:11,612 (Thread-3) Client 127.0.0.1:40433 finished using service
18:22:11,624 (Thread-5) Client 127.0.0.1:40433 finished using service
18:22:11,624 (Thread-4) Client 127.0.0.1:40433 finished using service
18:22:18,012 (MainThread) Ctrl+C issued closing server ...
18:22:18,013 (MainThread) Terminating ...
```

Figure 4.8: Log output of Multi-threaded server using thread-per-socket

```
12:07:21,206 (MainThread) Application started
12:07:21,207 (MainThread) Socket 127.0.0.1:7777 is in listening state
12:07:21,207 (MainThread) Falling to serving loop, press Ctrl+C to terminate ...
12:07:21,207 (MainThread) Awaiting new clients ...
12:07:28,237 (MainThread) Added item <socket._socketobject object at 0x7f58f1406c20>
12:07:28,238 (MainThread) Queue is no more empty, notifying all waiting threads
12:07:28,238 (MainThread) Awaiting new clients ...
12:07:28,238 (Thread-1) Got item <socket._socketobject object at 0x7f58f1406c20>
12:07:28,238 (Thread-1) New client connected from 127.0.0.1:42769
12:07:28,238 (Thread-1) Local end-point socket bound on: 127.0.0.1:7777
12:07:28,238 (Thread-1) Service offered, client 127.0.0.1:42769 is using a service ...
12:07:28,245 (MainThread) Added item <socket._socketobject object at 0x7f58f1406c90>
12:07:28,245 (MainThread) Queue is no more empty, notifying all waiting threads
12:07:28,245 (MainThread) Awaiting new clients ...
12:07:28,245 (Thread-2) Got item <socket._socketobject object at 0x7f58f1406c90>
12:07:28,245 (Thread-2) New client connected from 127.0.0.1:42770
12:07:28,245 (Thread-2) Local end-point socket bound on: 127.0.0.1:7777
12:07:28,246 (Thread-2) Service offered, client 127.0.0.1:42770 is using a service ...
12:07:28,249 (MainThread) Added item <socket._socketobject object at 0x7f58f1406d00>
12:07:28,249 (MainThread) Queue is no more empty, notifying all waiting threads
12:07:28,249 (MainThread) Awaiting new clients ...
12:07:28,250 (Thread-3) Got item <socket._socketobject object at 0x7f58f1406d00>
12:07:28,251 (Thread-3) New client connected from 127.0.0.1:42771
12:07:28,251 (Thread-3) Local end-point socket bound on: 127.0.0.1:7777
12:07:28,251 (Thread-3) Service offered, client 127.0.0.1:42771 is using a service ...
12:07:28,253 (MainThread) Added item <socket._socketobject object at 0x7f58f1406d70>
12:07:28,253 (MainThread) Queue is no more empty, notifying all waiting threads
12:07:28,253 (MainThread) Awaiting new clients ...
12:07:28,254 (Thread-4) Got item <socket._socketobject object at 0x7f58f1406d70>
12:07:28,254 (Thread-4) New client connected from 127.0.0.1:42772
12:07:28,254 (Thread-4) Local end-point socket bound on: 127.0.0.1:7777
12:07:28,254 (Thread-4) Service offered, client 127.0.0.1:42772 is using a service ...
12:07:28,258 (MainThread) Added item <socket._socketobject object at 0x7f58f1406de0>
12:07:28,258 (MainThread) Queue is no more empty, notifying all waiting threads
12:07:28,258 (MainThread) Awaiting new clients ...
12:07:28,258 (Thread-5) Got item <socket._socketobject object at 0x7f58f1406de0>
12:07:28,258 (Thread-5) New client connected from 127.0.0.1:42773
12:07:28,258 (Thread-5) Local end-point socket bound on: 127.0.0.1:7777
12:07:28,259 (Thread-5) Service offered, client 127.0.0.1:42773 is using a service ...
12:07:58,270 (Thread-1) Client 127.0.0.1:42773 finished using service
12:07:58,277 (Thread-2) Client 127.0.0.1:42773 finished using service
12:07:58,284 (Thread-4) Client 127.0.0.1:42773 finished using service
12:07:58,284 (Thread-3) Client 127.0.0.1:42773 finished using service
12:07:58,287 (Thread-5) Client 127.0.0.1:42773 finished using service
12:08:09,075 (MainThread) Ctrl+C issued closing server ...
...
```

Figure 4.9: Log output of Multi-threaded server using pre-fork

```
18:21:41,569 (MainThread) INFO Application start
18:21:41,570 (MainThread) INFO 5 clients will connect to 127.0.0.1:7777 ...
18:21:41,570 (MainThread) DEBUG Connection timeout 5 sec
18:21:41,570 (MainThread) DEBUG Stay connected for 30 sec
18:21:41,570 (MainThread) DEBUG Client threads created ...
18:21:41,575 (Thread-1) DEBUG Trying to connect to 127.0.0.1:7777
18:21:41,576 (Thread-1) INFO Connected, waiting for service ...
18:21:41,577 (Thread-1) INFO Service available! Keeping it busy for 30 sec ...
18:21:41,579 (Thread-2) DEBUG Trying to connect to 127.0.0.1:7777
18:21:41,580 (Thread-2) INFO Connected, waiting for service ...
18:21:41,580 (Thread-2) INFO Service available! Keeping it busy for 30 sec ...
18:21:41,583 (Thread-3) DEBUG Trying to connect to 127.0.0.1:7777
18:21:41,584 (Thread-3) INFO Connected, waiting for service ...
18:21:41,584 (Thread-3) INFO Service available! Keeping it busy for 30 sec ...
18:21:41,591 (Thread-4) DEBUG Trying to connect to 127.0.0.1:7777
18:21:41,592 (Thread-4) INFO Connected, waiting for service ...
18:21:41,593 (MainThread) DEBUG Client threads started ...
18:21:41,594 (Thread-4) INFO Service available! Keeping it busy for 30 sec ...
18:21:41,594 (Thread-5) DEBUG Trying to connect to 127.0.0.1:7777
18:21:41,594 (Thread-5) INFO Connected, waiting for service ...
18:21:41,595 (Thread-5) INFO Service available! Keeping it busy for 30 sec ...
18:22:11,607 (Thread-1) INFO Finished using a service ...
18:22:11,608 (Thread-1) INFO Disconnected ...
18:22:11,609 (Thread-2) INFO Finished using a service ...
18:22:11,610 (Thread-2) INFO Disconnected ...
18:22:11,611 (Thread-3) INFO Finished using a service ...
18:22:11,612 (Thread-3) INFO Disconnected ...
18:22:11,623 (Thread-5) INFO Finished using a service ...
18:22:11,624 (Thread-5) INFO Disconnected ...
18:22:11,624 (Thread-4) INFO Finished using a service ...
18:22:11,624 (Thread-4) INFO Disconnected ...
18:22:11,624 (MainThread) DEBUG Client threads dead ...
18:22:11,625 (MainThread) INFO Terminating ...
```

Figure 4.10: Simulator output testing Multi-threaded server (thread-per-socket)

```
12:07:28,236 (MainThread) INFO Application start
12:07:28,236 (MainThread) INFO 5 clients will connect to 127.0.0.1:7777 ...
12:07:28,236 (MainThread) DEBUG Connection timeout 5 sec
12:07:28,236 (MainThread) DEBUG Stay connected for 30 sec
12:07:28,237 (MainThread) DEBUG Client threads created ...
12:07:28,237 (Thread-1) DEBUG Trying to connect to 127.0.0.1:7777
12:07:28,239 (Thread-1) INFO Connected, waiting for service ...
12:07:28,239 (Thread-1) INFO Service available! Keeping it busy for 30 sec ...
12:07:28,244 (Thread-2) DEBUG Trying to connect to 127.0.0.1:7777
12:07:28,245 (Thread-2) INFO Connected, waiting for service ...
12:07:28,246 (Thread-2) INFO Service available! Keeping it busy for 30 sec ...
12:07:28,248 (Thread-3) DEBUG Trying to connect to 127.0.0.1:7777
12:07:28,253 (Thread-4) DEBUG Trying to connect to 127.0.0.1:7777
12:07:28,253 (Thread-3) INFO Connected, waiting for service ...
12:07:28,254 (Thread-3) INFO Service available! Keeping it busy for 30 sec ...
12:07:28,254 (Thread-4) INFO Connected, waiting for service ...
12:07:28,255 (Thread-4) INFO Service available! Keeping it busy for 30 sec ...
12:07:28,257 (MainThread) DEBUG Client threads started ...
12:07:28,257 (Thread-5) DEBUG Trying to connect to 127.0.0.1:7777
12:07:28,258 (Thread-5) INFO Connected, waiting for service ...
12:07:28,259 (Thread-5) INFO Service available! Keeping it busy for 30 sec ...
12:07:58,267 (Thread-1) INFO Finished using a service ...
12:07:58,268 (Thread-1) INFO Disconnected ...
12:07:58,276 (Thread-2) INFO Finished using a service ...
12:07:58,276 (Thread-2) INFO Disconnected ...
12:07:58,283 (Thread-4) INFO Finished using a service ...
12:07:58,283 (Thread-4) INFO Disconnected ...
12:07:58,284 (Thread-3) INFO Finished using a service ...
12:07:58,284 (Thread-3) INFO Disconnected ...
12:07:58,287 (Thread-5) INFO Finished using a service ...
12:07:58,287 (Thread-5) INFO Disconnected ...
12:07:58,288 (MainThread) DEBUG Client threads dead ...
12:07:58,288 (MainThread) INFO Terminating ...
```

Figure 4.11: Simulator output testing Multi-threaded server (pre-fork)

for a thread to pickup later. In both scenarios the steps 2,3,4 and 5 are done by a separate thread and only step 1 is left for main thread. We assume that the steps 2,3 and 4 are the most time consuming and therefore it is good to handle them in a separate thread. The actual request handling (in step 3) may be very fast, but slow network might be slowing down the whole client handling (steps 2 and 4). In other scenarios, the request handling (step 3) might be a CPU intensive task and the actual request/response data might be very tiny. In this case, we will have a very fast send/receive (steps 2,4) and slow request handling (step 3). Example would be an integer prime factorization task[1], where clients request integer to factorize and the server that does the factorization. In this example most of the time is spent on actual computing and not on I/O (sending and receiving a couple of integers is doable fast even in slow network connections). Therefore we can assign the request handling to a separate execution flow as opposed to connection handling.

Next, we demonstrate the code examples of the corresponding pattern implementation compared to single threaded implementation. File [*singlethreaded_server_heavy_request.py*] illustrates the code of a simple request handling server. The request/response payload we simulate with fixed size random string (36 byte UUID). As you can see the request and response here are short in size therefore the time spent on delivery will be minimal. In *handle_request* method we simulate the load with intentionally added *sleep*, hence the time spent in request handling requests may vary (dependent on DEFAULT_REQ_PROCESS_DELAY global variable). There is no real work done by server with the requests in the *handle_request* method, once the *sleep* method returns, the server replies with the same string in its response. File [*simulating_many_requests.py*] illustrates the code of corresponding simulator. Here again we create multiple treads that start sending the requests simultaneously and they leave in case the server could not handle request in reasonable time.

We did run the client request simulator having the following settings:

- DEFAULT_CLIENTS = 5
- DEFAULT_TIMEOUT_SEC = 10

The output of the server is illustrated in file [*singlethreaded_server_heavy_request-log.txt*], simulator output is illustrated in file [*singlethreaded_server_heavy_request_simulator-log.txt*]. As you can see only 2 requests were successfully processed by a server and 3 clients left with no response received.

In order to make a server able to process the requests in time we indeed will apply the same threading patterns: either delegation or producer-consumer. This time the separate thread executes the *handle_request* method with request as an argument.

Combining thread-per-socket and thread-per-request

Several patterns can be combined in one server application, for example the pipeline model can be employed if separate the tasks (receiving, processing, sending), therefore we have a fixed number of workers assigned to groups “receivers”, “handlers” and “senders”. Each task assumes some input and output, and we use three different queues to collect “connections”, “requests” and “responses”. The worker dependent on his group is responsible for taking input task from one queue and putting a result into second one.

The corresponding model is implemented in [*multithreaded_server_combined_pipeline.py*]

The corresponding output is shown in file [*multithreaded_server_combined_pipeline-log.txt*] the corresponding simulator output is shown in file [*multithreaded_server_combined_pipeline_simulator-log.txt*].

Another combination could be the fixed number of group managers and on-demand workers. In this case there are four fixed threads (main, request-manager, receiver-manager, sender-manager), each manager can create additional threads for handling the corresponding tasks. The example code is shown in file [*multi-threaded_server_combined_delegation.py*].

The corresponding output is shown in file [*multithreaded_server_combined_delegation-log.txt*] the corresponding simulator output is shown in file [*multithreaded_server_combined_delegation_simulator-log.txt*].

Combining threading and multiprocessing

Considering previous example it is clearly visible the two kinds of thread jobs:

- an I/O job - reading/writing to sockets
- processing job - handling requests (which may CPU intensive task)

As we already know Python's threads are good for handling I/O but for parallelizing CPU intensive tasks they do not suit very well. The threads are virtual and in fact only one OS-thread is created for Python interpreter itself (many-to-one model). Therefore the Python threads will never be scheduled to different CPU cores, and in order to benefit from multi-core CPU the Python multiprocessing module has to be used. In our previous example we can just replace the request handling from Thread based to Process based producer-consumer pattern. The file [*multiprocess_server_combined_pipeline.py*] illustrates the modified code of combined pipeline (multiple producer-consumers attached) with Process based request handling. In this example the request-handler threads are running in separate processes, the request-receiver threads are still in the main process. Passing the input/output tasks between the thread groups is done using the Queue (this time using multiprocessing Queue which supports interprocess communication). Having producer-consumer model for multiprocessing is therefore obvious:

- Producer-Consumer model has a fixed set of producers and consumers (fixed amount of jobs)
- Creating a new Processes on demand is expensive, it is better to create a fixed amount of processes at the beginning and then to reuse them
- The number of Processes in a multi-process application is usually not exceeding the number of CPU cores (or CPUs)
- Tasks are not assigned explicitly, but through a queue
  - Porting the Python code from producer-consumer multi-threading to multi-processing is just a matter of replacing the queue implementation

The corresponding output is shown in file [*multiprocess_server_combined_pipeline-log.txt*] the corresponding simulator output is shown in file [*multiprocess_server_combined_pipeline_simulator-log.txt*].

Examples of object oriented implementation

The object oriented design of the server side application can be applied even before introducing threads. In case we have server implementation like the one shown in Figure 4.4. The corresponding handler method is always a subject of corresponding client's socket. Therefore the socket can be isolated completely inside a *ClientHandler* class. What remains is the server initialization and the main serving loop. In fact, both are subjects of one running server, therefore it is wise to isolate them into a separate *Server* class. The example of OOP implementation is shown in Figure 4.12

Implementing the multi threaded version of the same server using OOP is even more simpler. Our *ClientHandler* class suits perfectly for thread-per-socket model (delegation) as it is already a subject of a connected client (through client's socket). Now lets just make it live independent of the master thread once it is created. In the Figure 4.13 we see the changes made: the `ClientHandler` becomes the subclass of Python's `Thread` class overriding its `run()` method. The Server class we do not need to touch, thanks to OOP design for the Server creating the `ClientHandler` remains the same, just this time handling the client is done by a separate thread.

Concerning the client side there will be no changes.

```
from socket import AF_INET, SOCK_STREAM, socket, SHUT_WR
from socket import error as soc_err

class ClientHandler():
    def __init__(self,client_socket,client_addr):
        self.__client_socket = client_socket
        self.__client_address = client_addr

    def handle(self):
        try:
            print 'New client connected from %s:%d' % self.__client_address
            print 'Local end-point socket bound on: %s:%d'\
            '' % self.__client_address
            self.__client_socket.send('1')
            self.__client_socket.shutdown(SHUT_WR)
            print 'Service offered, client %s:%d is using a service ...'\
               '' % self.__client_address
            self.__client_socket.recv(1)
            print 'Client %s:%d finished using service' % self.__client_address
        except soc_err as e:
            if e.errno == 107:
                print 'Client %s:%d left before server could handle it'\
                '' %  self.__client_address
            else:
                print 'Error: %s' % str(e)
        finally:
            self.__client_socket.close()
        print 'Client %s:%d disconnected' % self.__client_address

class Server():
    def listen(self,sock_addr,backlog=1):
        self.__sock_addr = sock_addr
        self.__backlog = backlog
        self.__s = socket(AF_INET, SOCK_STREAM)
        self.__s.bind(self.__sock_addr)
        self.__s.listen(self.__backlog)
        print 'Socket %s:%d is in listening state' % self.__s.getsockname()

    def loop(self):
       print 'Falling to serving loop, press Ctrl+C to terminate ...'
        try:
            while 1:
                client_socket = None
                print 'Awaiting new clients ...'
                client_socket,client_addr = self.__s.accept()
                c = ClientHandler(client_socket,client_addr)
                c.handle()
        except KeyboardInterrupt:
            print 'Ctrl+C issued closing server ...'
        finally:
            if client_socket != None:
                client_socket.close()
            self.__s.close()

if __name__ == '__main__':
    print 'Application started'
    s = Server()
    s.listen(('127.0.0.1',7777))
    s.loop()
    print 'Terminating ...'
```

Figure 4.12: Single threaded server OOP implementation

```
import logging logging.basicConfig(level=logging.DEBUG,\
                    format='%(asctime)s (%(threadName)-2s) %(message)s',)
from socket import AF_INET, SOCK_STREAM, socket, SHUT_WR
from socket import error as soc_err
from threading import Thread

class ClientHandler(Thread):
    def __init__(self,client_socket,client_addr):
        Thread.__init__(self)
        self.__client_socket = client_socket
        self.__client_address = client_addr

    def run(self):
        self.__handle()

    def handle(self):
        self.start()

    def __handle(self):
        try:
            logging.info( 'New client connected from %s:%d'\
                          '' % self.__client_address )
            self.__client_socket.send('1')
            self.__client_socket.shutdown(SHUT_WR)
            logging.debug( 'Service offered, client %s:%d is using a '\
                           'service ...' % self.__client_address )
            self.__client_socket.recv(1)
            logging.info( 'Client %s:%d finished using service' \
                          '' % self.__client_address )
        except soc_err as e:
            if e.errno == 107:
                logging.warn( 'Client %s:%d left before server could handle it'\
                              '' %  self.__client_address )
            else:
                logging.error( 'Error: %s' % str(e) )
        finally:
            self.__client_socket.close()
        logging.info( 'Client %s:%d disconnected' % self.__client_address )

class Server():

    def listen(self,sock_addr,backlog=1):
        self.__sock_addr = sock_addr
        self.__backlog = backlog
        self.__s = socket(AF_INET, SOCK_STREAM)
        self.__s.bind(self.__sock_addr)
        self.__s.listen(self.__backlog)
        logging.debug( 'Socket %s:%d is in listening state'\
                       '' % self.__s.getsockname() )

    def loop(self):
        logging.info( 'Falling to serving loop, press Ctrl+C to terminate ...' )
        handlers = []
        try:
            while 1:
                client_socket = None
                logging.info( 'Awaiting new clients ...' )
                client_socket,client_addr = self.__s.accept()
                c = ClientHandler(client_socket,client_addr)
                handlers.append(c)
                c.handle()
        except KeyboardInterrupt:
            logging.warn( 'Ctrl+C issued closing server ...' )
        finally:
            if client_socket != None:
                client_socket.close()
            self.__s.close()
        map(lambda x: x.join(),handlers)

if __name__ == '__main__':
   logging.info( 'Application started' )
   s = Server()
    s.listen(('127.0.0.1',7777))
    s.loop()
    logging.info ( 'Terminating ...' )
```


Figure 4.13: Multi threaded server OOP implementation

Shared state

In order to get the idea of the shared state, we should first make sure we understand the meaning of the state in general. It easy to get it if you imagine a white-board drawing application (like famous windows paint application). Eventually, it gives you the drawing facility - white pane and the tools appllicable for drawing. Considering the implementation, the drawing itself is just a matrix of RGB pixel values (the triples (0-255,0-255,0-255)) that are representing the colors. The process of drawing is then reduced to alternation of pixel values by coordinates in the picture. Speaking about the state here - the whole drawable area is the state, all the pixel values contribute to it. Once, we change at least one value - the state of the whole drawable area gets changed.

Everything looks easy once we are sure that there is only one entity alternating the drawable area at each separate moment (like one user is drawing in windows paint). The things get more complicated when we allow many users to take part in the drawing process in our drawing application. Hence, we have to take into account the following:

1. Real-time drawing. The connected users want to see all changes immediately. Once some changes are introduced by one user or many users, the others should be notified immediately.
2. Collision resolving. Relevant actions need to be taken, when concurrent modification of the same pixel region by different users occurs at the same time.

Collision resolving in distributed systems shared state can roughly be divided into two:

- centrally coordinated – most natural in clent-server applications, where the application server takes the role of making decisions on collision resolution with certain policy
- collaborative coordination – certain consensus algorithms are used for reaching collaborative agreements on shared state

In any case the collision resolving leads to the problem of global ordering of distributed events and how, based on local states to determine the global state? This in turn gives rise to the problem of synchronization of physical clocks of distributed system components.

Short-lived vs. Long-lived TCP connections

Regarding request-response protocols, we have seen two delivery strategies when the application layer protocol uses TCP for transport:

- Each request-response transaction is a separate TCP connection
  - the connection gets closed after response is received
  - the new request means new TCP connection
  - in case of short requests/responses the average lifetime of the TCP connection is insignificant
  - therefore the name “short-lived” TCP connections
- Many request-responses interactions are done using the same connection
  - connection usually being established at application’s startup
  - connection stays open until application terminates

  - the request-response interactions use only one existing connection
  - the application therefore never creates more than one connection
  - average connection life-time is defined by application's runtime and may be long
  - therefore the name "long-lived" TCP connections

Next let's consider the HTTP and SSH protocols and how the corresponding TCP connections looks like.

[class demonstration HTTP vs. SSH]

In Linux, execute the following command:

```
1 while [ true ]; do netstat -tpn | grep -E ':22 |:80 '; sleep 1; clear; done;
```

This one will loop forever (kill using Ctrl+C) and each iteration it will print the action TCP connection on ports 80 and 22. The delay between iterations is set to one second, and before printing new results we clean the previous.

Next let's visit some news portal like *postimees.ee*.

Question:

- How many connections were required to load the title page ?
- How long did they stay open?

Next let's connect to remote shell of the host *newmath.ut.ee* (using UT credentials). Let's take a look at the TCP connections created and answer the same questions.

Advantages and Disadvantages

- Short lived TCP
  - Advantages
    * Connection is made only when we need to send/receive the data
    * The send/receive tasks can be balanced to multiple connections
    * Good for load balancers. In case of HTTP, the static read-only content gets distributed faster as each HTTP site usually has a number of replicas behind the load balancer proxy.
    * Good for network hand-overs. In case we change the Wi-Fi network while loading HTTP resources like *postimees.ee* only a small number of requests may get lost (and even they can be re-requested once we get connected to a new Wi-Fi network).
  - Disadvantages
    * Transport layer overhead. In case of HTTP the TCP handshake happens before each HTTP request. In case we implement sessions on top of HTTP, then the session metadata is added to each request, increasing the protocol overhead.
    * Client centric: client has to contact the server before server can send any updates. No way server can initiate the connection with the client.
    * Varying number of connections. We do not know how many connections the application will need. Each connection needs a TCP port and we only have 2^16 ports (some of them are already reserved by services).

            - For the routers serving private network it is even more worse. The ports opened for outgoing connections by individual machines in private network are reduced on one external IP of the router. Many machines in private network opening many outgoing connection might easily exhaust the router's ports.

- Long-lived
  - Advantages
    - No TCP overhead - only one TCP handshake is required
    - No protocol overhead. One connection is alway one user, no additional session metadata required.
    - Server always has a number of client connections idling (even if clients do not actively use them). Server may use those idling connections for sending notifications to clients.
    - Less connections needed. One application -> one connection, or one application+user -> one connection.
      - Less ports required
  - Disadvantages
    - Connection is kept open all the time, even we do not transfer any data
    - Impossible to load-balance the single session (one established connection)
      - We can only load balance many different connections (many individual sessions)
    - Impossible to handover the established TCP connection, as the source IP of the client changes when we join to new IP network. The established TCP pipe is still bound to old IP. We cannot interchange the endpoints of established TCP, but we have to disconnect and re-establish TCP pipe.

Make clients get the updates immediately

In this section, we demonstrate the message board implementations using long-lived TCP connections, which allows the clients to get notifications once the server gets new message published. As a result, the clients fetch the new messages almost immediately after the message get published.

The implementation of the Message Board using short-lived TCP that we have tried in 2 (TCP Application) was fetching the messages from the server manually by the user. The protocol allowed only one request-response transaction per connection (each new transaction was done using separate connection).

Heavily modified version of Message Board is illustrated in files [*long_lived_tcp_message_board_server.py*] and [*long_lived_tcp_message_board_client.py*]. The new version uses long-lived TCP connections, client gets connected early at application startup. The connection stays open till client application terminates it. Request-response transactions are done over the same connection (only one connection is required). The client application is interactive: user may publish the message multiple times per session. The new messages are automatically fetched from the server once the corresponding notification is received. The server is sending the notifications to connected clients once new message is published. The server keeps track of fetched messages for each client to avoid double-sending the messages.

Concurrent modification

Imagine the simple "guess word" game where multiple users guess word by proposing letters one by one. Usually in such game, the players propose letters one after another and the moderator ensures that they never propose letters simultaneously. Next, we demonstrate network multi-player implementation of this game. The

code may be found in files [*shared_state_guessword_game_server.py*] and [*shared_state_guessword_game_client.py*]. As you can notice, we mostly reused the network code of the previous example, as the way long-lived TCP connection are handled here is mostly the same. The connection is established once the user joins the game, and the user may join even in the middle of game session. Next, during the game process user can see immediately how the letters are guessed by the other players. User can propose his guesses too. In case two users propose the same letter as a guess - the server scores the fastest. There is no strict ordering how the players are guessing - the fastest one may score more points. Once the game is over - the users have to propose new word to guess. The server starts new game using the first proposed word. The key element of the server code is the *Game* structure containing all the routines for handling concurrent actions.

Chapter 5

Remote Procedure Calls (RPC) and Distributed Objects (DO)

Remote Procedure Calls

During this lecture we will introduce Remote Procedure Calls (RPC) and Distributed Objects (DO). Therefore, we will divide our lecture into two main sections one for RPC and the second one is for DO.

26. ***RPC*** *is a powerful technique for constructing distributed, client-server based applications. The RPC philosophy relies on extending the conventional or local procedure of calling. As a result, the called procedure does not exist in the same address space where the calling procedure is. Therefore, the two processes can be on the same system or on different systems that are connected via the network connection. Moreover, the use of RPC allows programmers of distributed applications to avoid details of the interface with the network. Thanks to the transport independence of RPC, it isolates the application from the physical and logical layer of the data communication mechanism. Hence, RPC allows the application to use a variety of transports and enhance the client/server model computing power.*

How Does it Work?

From the programming perspective RPC is equivalent to a function call. This means that when an RPC is made the calling arguments are passed to the remote procedure and the caller waits for a response to be returned from the remote procedure.

The Figure 5.1 illustrates the flow of activity occurring when an RPC call is made between two network systems. As you can see the client makes a procedure call that sends a request to the server and waits. The thread is blocked from the processing till either a reply is received or time-out occurs. When the request arrives the server calls a dispatch routine to perform the requested service. Next, a return reply is sent to the client. The client program continues after the RPC call is completed.

The entire trick in making the remote procedure calls work is in the creation of **stub function**, which gives the user the illusion as if the call is made locally. However, on client side the stub function contains a protocol of sending and receiving messages over the network.

Breaking down all the steps in the RPC procedure (Figure 5.2) will result in following:

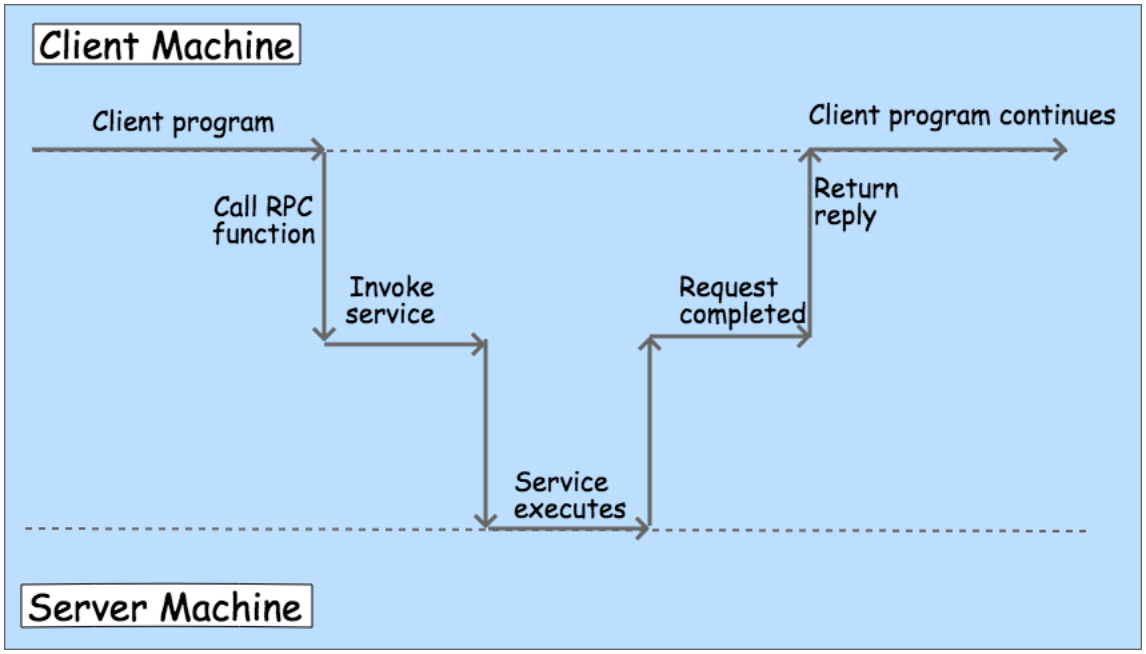


Figure 5.1: RPC Mechanism

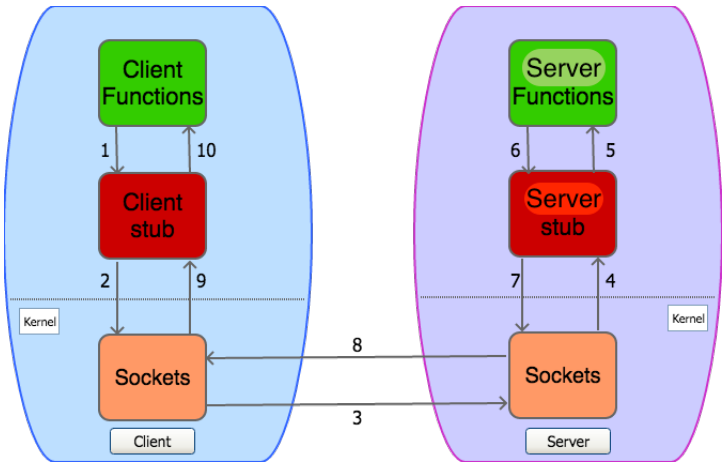


Figure 5.2: RPC Execution Steps

1. The client calls a local procedure (client stub). We described it above as a stub function. The client stub processes the parameters to the remote procedure and builds one or more network messages. The process of packaging all the arguments into network message is called **marshalling** (assembling) – it requires serializing all the data elements into a flat array-of-bytes format.
2. The client stub sends the network messages to the remote server through the kernel using sockets interface.
3. The sockets transfer the message to the remote system using some protocol.
4. The server stub (or **skeleton**) receives the messages on the server side. It **unmarshals** the arguments from the messages and preprocesses them (data conversion) if needed.
5. The server stub calls the server function and parses it with the received argument.

6. The moment the server function has finished, it returns to the server stub with the outputs.
7. The server stub preprocesses (data conversion) the outputs and marshals them into one or more messages to send them to the client stub.
8. The messages are sent back across the network to the client stub.
9. The client stub reads the messages from the local kernel.
10. The client stub preprocesses the returned data (if needed) and returns the results to the client function.
11. The client code continues its execution.

Programming RPC

The RPC was initially attributed to OS and it was the OS who handled them. For this reason, many popular programming languages were not designed with a built-in syntax for RPC. Hence, to enable the use of RPC using programming languages a separate compiler to generate the client and server stub functions was introduced. This compiler takes its input from a programmer-specified definition such as an **interface definition language** (**IDL**).

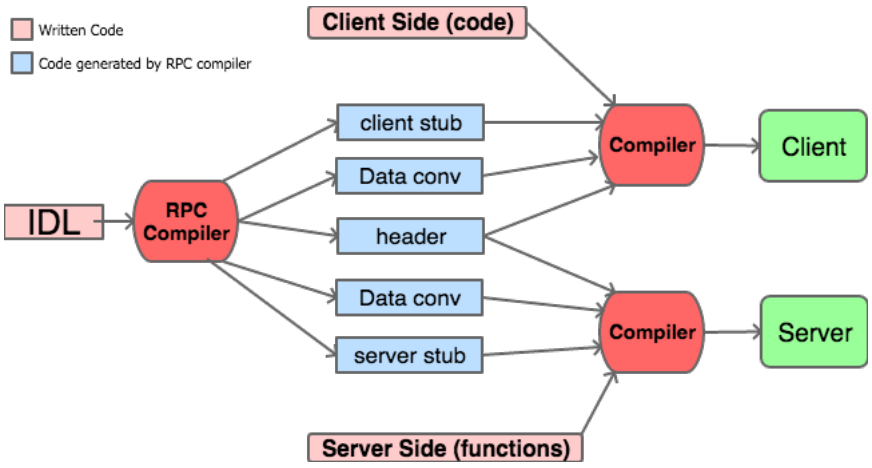


Figure 5.3: RPC Compilation Steps

The IDL usually looks similar to function prototype declaration (create enumeration of the set of functions alongside with input and return parameters). The moment RPC compiler is in action, the client and the server programs can be compiled and linked with the suitable stub functions (Figure 5.3). Note that the client side has to be modified to initialize the RPC mechanism in order to handle RPC failures in case these happen.

RPC Advantages

The advantages of using RPC can be resumed as follows:

1. The system can be independent of transport provider. Since the generation of server stub code is done automatically, it makes it available over any transport layer protocol (UDP or TCP).

2. No worries on getting a unique transport address any more. The server can bind to any available port and then register that port with an RPC name server. The client will contact this name server to find the port number that corresponds to the program it needs. All this will be invisible to the programmer.
3. Applications on client side only need to know a single transport address: the one of the name server that is responsible for telling the application where to connect for a given set of server functions.
4. The function-call model can be used instead of the send/receive (read/write) interface provided by sockets. Users do not have to deal with marshalling parameters neither actions of parsing them out on the other side.

RPC Disadvantages

Despite the many advantages that RPC gives to us there are still tasks where RPC will introduce rather regression in performance. In sample codes below we will discover which sort of tasks the RPC underperforms compared to other application layer protocols on top of TCP or UDP.

RPC Implementations

RPC implementations are usually platform independent and even language independent. Client may be implemented in Python on Windows and the server in C on Unix, *etc.* Here we mention a couple of corresponding libraries:

- XML-RPC, and it's successor SOAP (using HTTP based transport of XML encoded calls)
- JSON-RPC and it's derivative JSON-WSP (using HTTP based transport of JSON encoded calls)
- ICE framework (full-scale communication engine providing RPC on top of proprietary protocols)
  - supports communication over TCP, UDP, TLS and WebSockets
- CORBA, classics of RPC using broker architecture

There are number of RPC implementations that are platform specific but still support different languages:

- D-Bus, interprocess communication framework for Linux (libdbus), and it's several ports:
  - GDbus, glib based implementation used in GTK+ applications (in GNOME desktop)
  - QtDbus, Qt based implementation used in Qt applications
  - sd-bus heavily rewritten libdbus aiming at performance boost (as part of systemd framework)
  - kdbus another replacement of libdbus
- MSRPC Microsoft implementation of RPC (first used in Windows NT to support Windows Server domains)
- DCOM higher level framework on top MSRPC similar to CORBA (provides communication between software products of Microsoft)
- .NET Remoting and it's successor WCF, .NET API for building distributed application on Windows platform

Language specific:

- Java RMI, pure Java implementation of RPC
- RPyC pure Python implementation of RPC
- DRb, Ruby ...
- *etc* ...

Other frameworks that are not focused on but offer the RPC:

- Google Web Toolkit, web development framework offering wide range of features for web-devs, including asynchronous RPC
- Apache Avro, RPC framework for Apache's Hadoop and it's derivatives like Apache Spark.

As you can see the RPC is applied a lot, and it's original was used as a base for many frameworks we use nowadays, let's proceed with example application using RPC.

RPC Application

In our example application we will rely on XML-RPC implementation for Python:

- The `SimpleXMLRPCServer` module, providing server side bindings[3]
- The `xmlrpclib` module, providing client side bindings [6]

Both are based on the same generic specification of XML-RPC [8] which was aiming to achieve simplicity of RPC usage in particular. Adding the Python features made it even more simple – due to Python's dynamic nature. As a result, the XML-RPC Client in Python is skipping the IDL part completely (as it is done automatically in runtime).

Let's consider the example of client/server application we used to reference a lot: the Message Board. This time we design it to rely on RPC features and implement both server and client using Python's XML-RPC modules.

The design

The implementation is shown in Figures 5.4 and 5.5. The full-length code is illustrated in files [*rpc_mboard_server.py*] and [*rpc_mboard_client.py*].

Distributed Objects

We known the objects in terms of OOP paradigm are the class instances, and by default are referred from the scoped of the same application runtime (from the same address space). The concept "remote objects" or "distributed objects" allows objects to be referred from outside of the the current application's address spaces. Moreover the objects can be distributed across the different application running on different machines. This paradigm is the further developed concept of RPC, and is just adding objects support to existing features of RPC. Therefore Distributed Objects are often referred as "Second Generation" RPC.

With distributed objects we are extending the basicconcepts of object-oriented programming to the distributed systems world. With this move certain things certainly become more complex. What it means is summarised in the following table:

```
from SimpleXMLRPCServer import SimpleXMLRPCServer
from SimpleXMLRPCServer import SimpleXMLRPCRequestHandler
from time import time

class MessageBoard():

    def __init__(self):
        self.__m_board = {} # For storing published messages
        self.__m_uuid = 0   # For generating unique iDs

    def __get_uuid(self):
        uuid = self.__m_uuid
        self.__m_uuid += 1
        return uuid

    def publish(self,msg,source=('',-1)):
        ip,port = source
        t = time()
        uuid = self.__get_uuid()
        self.__m_board[uuid] = (uuid, t, ip, port, msg)
        return uuid

    def last(self,n=0):
        ids = map(lambda x: x[:2], self.__m_board.values())
        ids.sort(key=lambda x: x[1])
        return map(lambda x: x[0],ids[n*-1:])

    def get(self,m_id):
        return self.__m_board[m_id][1:]

# Restrict to a particular path
class MboardRequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

if __name__ == '__main__':
    mboard = MessageBoard()
    server_sock = ('127.0.0.1',7777)

    # Create XML-RPC server
    server = SimpleXMLRPCServer(server_sock,\
                                requestHandler=MboardRequestHandler)
    server.register_introspection_functions()

    # Register all functions of the Mboard instance
    server.register_instance(mboard)

    try:
        server.serve_forever()
    except KeyboardInterrupt:
        print 'Ctrl+C issued, terminating ...'
    finally:
        server.shutdown()       # Stop the serve-forever loop
        server.server_close()   # Close the sockets
    print 'Terminating ...'
```

Figure 5.4: Message Board Server implementation using Python's SimpleXMLRPCServer module

```
from xmlrpclib import ServerProxy

def mboard_client_main(args):
    m = args.message if len(args.message) > 0 else''
     # Message to publish
        if m == '-':
            LOG.debug('Will read message from standard input ...')
            # Read m from STDIN
            m = stdin.read()
        LOG.debug('User provided message of %d bytes ' % len(m))

    # Processing arguments
    # 2.) If -l was provided
    # Parse integer
    n = int(args.last)
    # Last n messages to fetch
    n = n if n > 0 else 0
    LOG.debug('Will request %s published messages'\
              '' % ('all' if n == 0 else ('last %d' % n)))
    # RPC Server's socket address
    server = (args.host,int(args.port))
    try:
        proxy = ServerProxy("http://%s:%d" % server)
    except KeyboardInterrupt:
        LOG.warn('Ctrl+C issued, terminating')
        exit(0)
    except Exception as e:
        LOG.error('Communication error %s ' % str(e))
        exit(1)
    LOG.info('Connected to Mboard XMLRPC server!')
    methods = filter(lambda x: 'system.' not in x, proxy.system.listMethods())
    LOG.debug('Remote methods are: [%s] ' % (', '.join(methods)))

    ids = []
    msgs = []

    try:
        if len(m) > 0:
            proxy.publish(m)
            LOG.info('Message published')
        ids += proxy.last(n)
        msgs += map(lambda x: proxy.get(x), ids)
    except Exception as e:
        LOG.error('Communication error %s ' % str(e))
        exit(1)
    except KeyboardInterrupt:
        LOG.debug('Crtrl+C issued ...')
        LOG.info('Terminating ...')
        exit(2)

....
```

Figure 5.5: Message Board Client implementation using Python's xmlrpclib module

Objects	Distributed objects	Description of distributed object
Object references	Remote references	Globally unique reference for a distributed object;may be passed as a parameter
Interfaces	Remote interfaces	Provides an abstract specification of the methods that can be invoked on the remote object; specified using an interface definition language (IDL)
Actions	Distributed actions	Initiated by a method invocation, potentially resulting in invocation chains; remote invocations use RMI
Exceptions	Distributed exceptions	Additional exceptions generated from the distributed nature of the system, including message loss or process failure
Garbage collection	Distributed garbage collection	Extended scheme to ensure that an object will continue to exist if at least one object reference or remote object reference exists for that object, otherwise, it should be removed. Requires a distributed garbage collection algorithm.

In short we can state, that with object-oriented programming we are concerned with objects, classes and their inheritence; while with distributed object programming we deal with encapsulation, data abstraction and design methodologies.

With distributed objects certain complexities are added, in particular:

- **Inter-object communication** is to be handled. Usually this is done with the help for remote method invocation (RMI), but other communication paradigms can be used as well.
- **Additional services** come into play. Most obvious examples are naming, security and transaction services.
- **Life-cycle management** is needed for cration, migration and deletion of distributed objects.
- **Activation** and **deactivation** of distributed objects are needed to be handled as the number of distributed objects can become very large. Also node availabilities needs to be handled.
- **Persistance** is an important property that needs to be handled in case of distributed objects – the way of object survival in case of server restarts, as an example, but also across all its cycles like acitvation and deactivations, system failures etc.

Distributed Objects Frameworks

As the whole paradigm is just a next version of RPC, the most of the frameworks we listed in RPC section are actually supporting the Distributed Objects as well: CORBA, DCOM, JavaRMI, DRb *etc.*

In our sample application we will rely on pure Python implementation of Distributed Objects paradigm:

- **Pyro** - Python Remote Objects framework aiming at the simplicity of implementing distributed applications, therefore most of the initialization related routines are done automatically.

Thanks to Python's dynamic nature the whole development is even easier.

Pyro

27. ***Pyro*** *is a framework that allows the users on a network to share nearly every resource and responsibility that might be distributed between different parties.*

Pyro is capable of handling many resources. For example, we can resume its responsibilities into the following categories:

- Computational resources (Hardware);
- Informational resources (data management);
- Business logic expertise.

How does it work?

Pyro permits one application to serve objects to another application. In most cases, these applications are running on top of a network (loopback on one machine works also). Pyro gives the capability to make a remote objects on a **Pyro server** to become attached to a proxy object on a **Pyro client**. From the moment these two objects are attached, the client application is able to treat the remote object as if it was a true local object.

Besides, once a specific object is being served by a certain machine - one or multiple machines can use it. Plus, the served machine can simultaneously serve other objects as well. Moreover, the Pyro client is capable of getting connected to many machines, where each one has it own resources.

Another interesting aspect of Pyro is the usage of the **Pyro Name Server**, which can run anywhere as far as it is accessible via TCP/IP by all the Pyro clients and servers. In case we are on a local LAN, broadcast is used to find a Name server node without specifying any extra parameters. Note that, if you have a firewall in you network then you have to use some IP addresses or domains to find the Name server. After all, using Pyro does not require Name Server. Moreover, you can create your own by using the Pyro libraries.

Pyro Application

Here we demonstrate a simple chatting application. You can notice the client side being very minimalistic, as all the objects are created on server side and then exposed to the client using Pyro URI. The server code is also minimal, containing only the chat logics, and no protocol or network related code at all (as it is being handled by Pyro). The code is illustrated in Figures 5.6 and 5.7.

From objects to components

With the development of RMI the practical implementation by practitioners designing real distributed applications noticed certain patterns that happen again and again with every project they developed. They have come up with an idea of: **component-based approaches** – a natural evolution from distributed object computing. We will first look what are the concerns with RMI and then will see the concepts on how the componet-based approach can solve these.

Issues with object-oriented middleware

The first concern with object-oriented approach is that object interfaces do not describe what the implementation of an object depends on [2]. We can therefore state this issue as:

Implicit dependencies – internal (encapsulated) behaviour of an object is hidden – think remote method invocation or other communication paradigms... – not apparent from the interface

- there is a clear requirement to specify not only the interfaces offered by an object but also the dependencies that object has on other objects in the distributed configuration

The second concern is with the need to master many low-level details. The issue is within the

Interaction with the middleware – too many relatively low-level details associated with the middleware architecture

Therefore we have a clear need to:

- simplify the programming of distributed applications,
- to present a clean separation of concerns between code related to operation in a middleware framework and code associated with the application,
- to allow the programmer to focus exclusively on the application code.

Lack of separation of distribution concerns: Application developers need to deal explicitly with non-functional concerns related to issues such as security, transactions, coordination and replication – largely repeating concerns from one application to another

- the complexities of dealing with such services should be hidden wherever possible from the programmer

No support for deployment: objects must be deployed manually on individual machines – can become a tiresome and error-prone process, in particular with large-scale deployments with lots of objects spread over different sites with a large number of (potentially heterogeneous) nodes

- Middleware platforms should provide intrinsic support for deployment so that distributed software can be installed and deployed in the same way as software for a single machine, with the complexities of deployment hidden from the user

To resolve the issues above **component based middleware** has been emerging as a new general paradigm for designing distributed systems.

Essence of components

Software components can be defined as follows:

> **Software component** is a unit of composition with contractually specified interfaces and explicit context dependencies only

With component based middleware all dependencies are also represented as interfaces. Each component is specified in terms of a contract, which includes a set of provided interfaces and a set of required interfaces. The provided interfaces are the ones that the component offers as a service to other components. The required interfaces are the dependencies that this component has in terms of other components that must be present and connected to this component for it to function correctly. Therefore, every required interface must be bound to a provided interface of another component. This makes up a software architecture consisting of components, interfaces and connections between interfaces.

Many component-based approaches offer two styles of interface:

- Interfaces supporting remote method invocation, as in CORBA and Java RMI,
- Interfaces supporting distributed events (as described in [2] Chapter 6).

Component-based system programming is concerned with (a) development of components as well as with (b) composition of components. In general terms, with this paradigm shift the programmer is moving from software development to software assembly.

Components and distributed systems

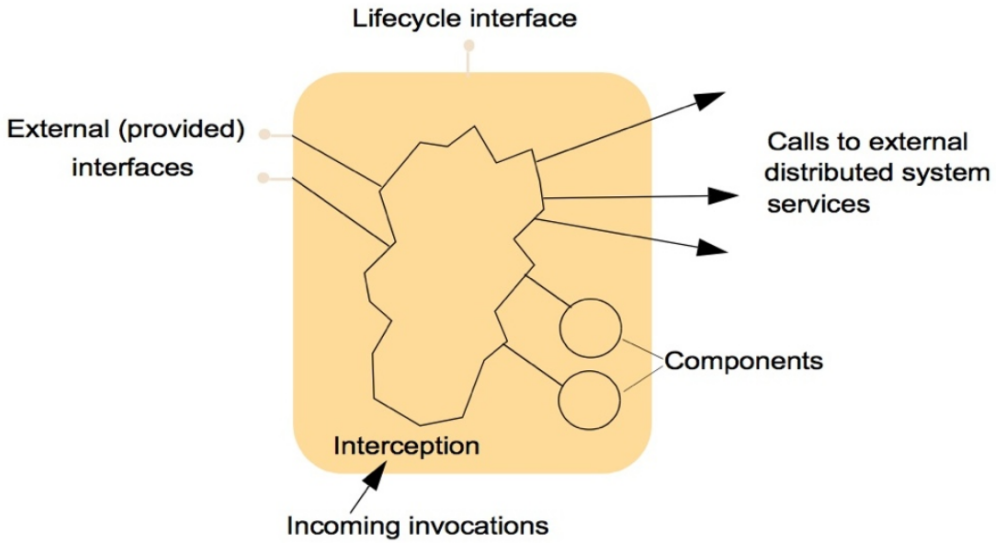
Containers:

Containers support a common pattern often encountered in distributed applications, which consists of:

- a front-end (perhaps web-based) client,
- a container holding one or more components that implement the application or business logic,
- system services that manage the associated data in persistent storage.

Components deal with application concerns, container deals with distributed systems and middleware issues (ensuring that non-functional properties are achieved).

Figure 8.12 [2] The structure of a container



The container does not provide direct access to the components but rather intercepts incoming invocations and then takes appropriate actions to ensure the desired properties of the distributed application are maintained. Middleware supporting the container pattern and the separation of concerns implied by this pattern is known as an ***application server.***

This style of distributed programming is in widespread use in industry today: – we list here some of the application servers that are around:

Technology	*Developed by*	*Further details*
WebSphere Application Server	IBM	www.ibm.com
Enterprise JavaBeans	SUN	java.sun.com
Spring Framework	SpringSource (a division of VMware)	www.springsource.org
JBoss	JBoss Community	www.jboss.org
CORBA Component Model	OMG	[Wang *et al.* 2001JOnAS]
JOnAS	OW2 Consortium	jonas.ow2.org
GlassFish	SUN	glassfish.dev.java.net
zope.component	zope.org	zopecomponent.readthedocs.io/en/latest/

Support for deployment

Component-based middleware provides support for the **deployment of *component configuration.***

- Components are deployed into containers,
- Deployment descriptors are interpreted by containers to establish the required policies for the underlying middleware and distributed system services.

Container therefore includes a number of components that require the same configuration in terms of distributed system support. Deployment descriptors are typically written in XML with sufficient information to ensure that:

- components are correctly connected using appropriate protocols and associated middleware support,
- the underlying middleware and platform are configured to provide the right level of support to the component configuration,
- the associated distributed system services are set up to provide the right level of security, transaction support, *etc.*

Python implementations of component-based architectures

One of the component-based architectures based entirely on python is zope, which seems actually just a web-server written in python. But actually, it is based on a full component-based architecture setup, **zope Application Server** implementd in zope.component.

Read more about zope at https://zopecomponent.readthedocs.io/en/latest/.

There is a small tutorial on how to write your own first python component-based architecture application at https://regebro.wordpress.com/2007/11/16/a-python-component-architecture/. An extensive book on zope.component architecture can be found at A Comprehensive Guide to Zope Component Architecture A Comprehensive Guide to Zope Component Architecture.

Note: installing zope.component on Linux system can be done with the command:
```
$ pip install zope.component
```

```
import Pyro4

@Pyro4.expose class User(object):
    '''User class extending PyRO object.
    instances of this one can be exposed over network     '''
    def __init__(self,name,chat):
        self.name = name
        self.private_msgs = [] # for receiving private messages
        self.last_common = 0 # for knowing last viewed public message
        self.chat = chat # parent chat

    def post(self,msg):
        '''Public post'''
        self.chat.post(msg,self.name)

    def post_private(self,msg,to):
        '''Private post'''
        self.chat.post_private(msg,to,self.name)

    def get(self):
        '''Get all public posts for me, not viewed'''
       msgs = self.chat.get(self.last_common)
        self.last_common += len(msgs)
        return msgs

    def get_private(self):
        '''Get all private posts for me, not viewed'''
        msgs = [m for m in self.private_msgs]
        self.private_msgs = []
        return msgs

    def whoami(self):
        return self.name

@Pyro4.expose class Chat(object):
    '''Chat class extending PyRO object.
    instances of this one can be exposed over network     '''
    def __init__(self):
        self.users = {}
        self.msgs = []

    def register(self,me):
        '''Register user by name, return PyRO objects URI'''
        user = User(me,self)
        self.users[me] = user
        u_uri = daemon.register(user)
        return str(u_uri)

    def post(self,msg,name='Unknown'):
        '''Post public message'''
        self.msgs.append('From %s : %s' % (name,msg))

    def post_private(self,msg,to,name='Unknown'):
        '''Post private message'''
        if to not in self.users.keys():
            return          self.users[to].private_msgs.append('From %s : %s' % (name, msg))

    def get(self,idx=0):
       '''Get all public messages starting from idx'''
        return self.msgs[idx:]

    def get_users(self):
        '''Get all users online'''
        return self.users.keys()

if __name__ == '__main__':

    # make a Pyro daemon
    daemon = Pyro4.Daemon(host='127.0.0.1',port=7777)
    # Create chat, expose to network using PyRO
    chat = Chat()
    uri = daemon.register(chat)
    print "The chat URI is:",uri
    # Serves forever ...
    daemon.requestLoop()
```


Figure 5.6: Simple Chat server using Pyro

```
import logging
FORMAT = '%(asctime)-15s %(levelname)s %(message)s'
logging.basicConfig(level=logging.DEBUG,format=FORMAT)
LOG = logging.getLogger()
import Pyro4 from argparse
import ArgumentParser
from sys import exit

# Constants ------------------------------------------------------------------
___NAME = 'PChat'
___VER = '0.2.0.0'
___DESC = 'Simple Chat Client (Pyro version)'
___BUILT = '2016-10-27' ___VENDOR = 'Copyright (c) 2016 DSLab'
# Private methods ------------------------------------------------------------
def __info():
    return '%s version %s (%s) %s' % (___NAME, ___VER, ___BUILT, ___VENDOR)

if __name__ == '__main__':
    parser = ArgumentParser(description=__info(),
                            version = ___VER)
    parser.add_argument('-u','--uri',\
                        help='Chat server Pyro URI ',\
                        required=True)
    parser.add_argument('-n','--name',\
                        help='Username',\
                        required=True)
    args = parser.parse_args()
    name = args.name
    chat_uri = args.uri

    try:
        chat = Pyro4.Proxy(chat_uri)
    except Exception as e:
        LOG.error('Cannot connect to chat by URI: %s' % chat_uri)
        exit(1)

    try:
        my_uri = chat.register(name)
        me = Pyro4.Proxy(my_uri)
    except Exception as e:
        LOG.error('Cannot connect to my remote object by URI: %s' % my_uri)
        exit(2)

    while True:
        LOG.info('My messages:\n' '\n'.join(me.get()))
        input_msg = raw_input('Message to send: ')
        if len(input_msg) > 0:
            me.post(input_msg)
```

Figure 5.7: Simple Chat client using Pyro

Chapter 6

Indirect communication

In this lecture, our objective is to understand one of the communication paradigm which is indirect communication. Hence, we will try to go through the following key topics:

- key properties of indirect communication
- space and time uncoupling
- group communication
- publish and subscribe systems
- message queues

28. ***Indirect communication*** *is defined as communication between entities in a distributed system via an* ***intermediary*** *with no direct* ***coupling*** *between the sender and the* ***receiver*** *or* ***receivers***

Before we proceed with indirect communication techniques, lets remind what techniques of direct communication we did cover up to now:

Direct communication techniques:

With direct communication we have two distinguishable communication parties irregardless of how the communication is organized. This is the unbreakable limit of the direct communication - the explicit delivery between two parties, hence all these techniques are so "end-to-end" pattern oriented. Surely we can have multi-party communication pattern organized by one of the up-mentioned techniques, but it will be just several "end-to-end" patterns incorporated.

Disadvantages of direct communication techniques

Think of an example application of message board. In order for the application to function the server has to be reachable. Eventually if the server is not reachable - we cannot publish neither fetch any messages. But why is it that critical? Why can't we just leave the message as "pending for publish" (server would then publish the message once it is back on-line). We could implement it through client retrying the publish attempt till the server is finally reachable and can process the request. However we will fail if the client will also go off-line before the server is back, leaving the goal then not reached – the critical message was not published. In order for "pending for publish" to work we would want some intermediary between the message board and the client – the third party who can keep pending requests alive in case either sever or client happen to be off-line.

Another issue is that in case of direct communication the message board server has to be specified explicitly before we can publish or fetch messages. The first step we do is "connect" to the server (and even in case of UDP the address of the server typically has to be given). Can we do it more conveniently for the client side? What if the client does not specify any server and just issues the publish or fetch. Here we have multiple options how to achieve it:

- Static list of servers
  - Predefined list of Message Board servers hard-coded in each client
  - DNS name for Message Board server with multiple matching IP addresses
- Dynamic list of servers
  - Broadcast/Multicast announcement of Message Board server presence
  - Message Board server deployed on host with dynamic DNS client (updating DNS name each time the IP address changes)
  - Message Board server implemented using RCP and is exposed using RCP name server

Even if we have something like that we will solve only the issue of server's reachability (eventually at least one server will be reachable in case the client is on-line). However we wanted also the servers to be interchangeable – the client should not publish to a particular server, instead the client can publish to any server. In this case we need more intelligent servers, that should be able to synchronize the published content. Now this means that the servers have to communicate with each other as well, and here we are back with the reachability issue (for servers this time). How will the servers all know about each other's presence (in order to synchronize)? Having a finite number of servers on static IPs reachable all the time? Who wants a service taking N fixed hosts that will run forever (burning kilowatts of energy even if not used)? We are in the era of minimizing costs of everything, including the infrastructure and power consumption. Our services have to be dynamic and corresponding features became a "must have" for good service:

- Automatically increase the number of servers in case of growing demand
- Reduce the number of servers in case of demand decreasing

Coming back to our example, in order to bring the up-mentioned "dynamic features" to our architecture, we would need to have a sort of announcement or discovery of dynamic servers. Both clients and serves have to be able to understand that a new server has joined or one has left. We could rely on dynamic DNS, however this would only help in case the servers are in public space. What if we want to host the servers also in private space (without the need for manual configuring the Proxy or NAT)? Indeed for this case we would again refer to three-component pattern: where there is a third party helping the clients and servers find each other. This third-party has however no other task but providing the intermediate discovery and communication for the other architecture components. Once we have introduce an intermediate (we can call it broker) the clients are not anymore contacting the servers directly, but contacting the broker to publish or fetch the messages. The servers do not interact with clients directly, but are interacting with the broker (processing the pending requests).

With architecture like this it looks like we can achieve the specified goals, barely introducing a third party between clients and servers, hence we end up with the paradigm of "indirect communication".

Indirect communication

Indirect communication is good for cases when the users connect and disconnect very often or in the case of event dissemination where receivers may be unknown and change often. In addition, it is suitable for the

scenarios where we have so many users involved or the change is anticipated. However, it has also a few disadvantages: for example performance overhead due to the level of indirection, difficulties in management because of lack of direct coupling and finally it is hard to achieve real time properties or security. Next let's discuss the properties of indirect communication and defining the terms.

Key properties

Most of the cases described in previous sections have a certain name in the scope of indirect communication, let's list them once again from the term definition perspective.

Space uncoupling

Space uncoupling means that the sender does not know or need not to know the identity of the receiver and vice versa.

This is exactly the case where client can submit the request without careing which server will process it (in particular – the client does not have to mention the server's IP or Host name). And the other way round – the server does not care about clients' transport endpoint, server is rather request-oriented as opposed to connection-oriented. In this case we do not address the request explicitly to an addressee — to whom then should we send? The options are: to send it to everyone (group communication using multicast or broadcast) or to delegate it to an intermediate party (message bus, broker).

Time uncoupling

In time uncoupling, the sender and the receiver can have independent lifetimes. In other words the client can submit the request without making sure if the server is on-line or not. The server does not have to stay on-line 24/7.

This is the case with "pending requests" discussed above (which is why it is well suitable with "request-response" kind of protocols). It is not critical to submit the request to the server explicitly, request may be delegated to an intermediate layer (message bus, broker) in the form of a message. Broker then delegates the message further to a server, and after the request is processed by the server the response is delegated to the broker in the same way. In order to get a response the client has to contact the broker. In case the server is off-line at the time the client submits the request, the broker will keep the request till the server gets back on-line to process it. In case the client is off-line by the time the server sends the response, the broker will keep it till the client is back on-line to receive the response.

Space and time coupling

The architectures can vary based on the features incorporated. Here we show some of them classified into to groups based on implemented features, refer to Table 6.1:

Asynchronous communication

It looks like the paradigm of "asynchronous communication" is very close to the concept of "time-uncoupling". Asynchronous communication allows the sender to send a message and then continue (no blocking). The response is then handled in a separate (receiver) thread, and it does not necessarily have to happen right

	Time-coupled	Time-uncoupled
Space-coupled	Typical direct communication: client and server both have to be on-line and reachable. Client refers to a server explicitly, establishing a TCP connection or sending an UDP packet. Examples: FTP, HTTP, SMTP, IMAP, etc. (most of the well known protocols).	Explicit communication (no intermediates) between sender and receiver and yet one of them may be off-line during communication. Example: an off-line "state file" that can be read/written by several processes in order to exchange the messages. In this case a message can be placed for "future processes" to consider.
Space-uncoupled	A sort of communication where sender/receiver are not addressed explicitly, but still both have to be on-line and reachable. Example: IP broadcast or multicast, here the packet is sent to "everyone" assuming the actual receiver is somewhere among the on-line peers.	Typical indirect communication that we discuss in this lecture. Sender/Receiver do address each other directly, also they may be off-line during communication. Examples: Group Communication, Publish-Subscribe, Message Queues etc.

Table 6.1: Space and Time uncoupling in distributed systems

after sending the request. The sender may even be off-line for a while between sending the request and receiving the response, which may lead to a conclusion that asynchronous communication is time-uncoupled. In fact it is not completely the case, because not all of the properties of "time-uncoupling" are preserved, like shown in Table 6.2.

Therefore we may say the asynchronous communication is no completely time-uncoupled, but "loosely" time-coupled.

Next let's discuss the corresponding technologies of the indirect communication and try out some frameworks by implementing our famous message board application on top of them.

Group communication

One of advantage of group communication is in offering service whereby a message is sent to a group and then this message is delivered to all the members. In addition, it is characterized by the fact that the sender has no awareness of the identity of the receivers and it represents an abstraction over multicast communications. Besides, from an implementation point of view it gave you the ability to do it via IP multicast or any equivalent overlay network by managing group membership, detecting failures, providing reliability and ordering guarantees.

- JGroups, Appia, NeEM - Java implementations of reliable multicast communication
- Spread Toolkit - platform neutral group communication framework, supports C/C++, Java, Python
- PGM experimental protocol

Asynchronous Communication	Time-uncoupled
The request/response (or send/receive) activities are time-uncoupled in the scope of one session. Receive action does not necessarily occur after sending. Regarding the overall existence of sender and receiver, they still have to meet in the same time slot for the transfer to happen. If one of them is off-line the message delivery is impossible. Example: 1. Server is on-line, Client is on-line 2. Client sends message to the Server and continues with other routines (not waiting for a reply) 3. Server receives the request and processes it 4. Server sends reply to Client triggering Client's "on-receive" callback	It should be possible to deliver the message even if one of the endpoints is off-line. Example: 1. Sever is off-line, Client is on-line 2. Client sends message to the Server and goes off-line 3. Sever comes on-line and still receives the message from the Client who is already off-line. 4. Server processes the request and replies to the Client even if the Client is off-line 5. Client should receive the reply once it is back on-line (even if the Server is off-line by that time).

Table 6.2: Asynchronous Communication (direct) and Time-uncoupling (indirect)

Publish-subscribe systems

First let's discuss the Publish-Subscribe systems (also referred as Distributed event-based systems). As have already seen in previous lectures, many protocols map naturally onto request-response or remote invocation paradigm (we had our MessageBoard application employing a protocol which was very easily replaced with XML-RPC calls). Moreover, both request-response as well remote invocation paradigms are in fact "event driven", as the communication is triggered by certain kind of event. Consider the MessageBoard application, where we had all the protocol built around "publish message" event, which we say is primal. There was secondary event though "get messages", and we call it secondary because it was only needed for the cases where the Server couldn't notify the Clients about new published messages (Lecture #3). In fact we solved this issue in (Lecture #4) evolving the long-lived TCP with asynchronous notifications: all the Clients were notified by Server in case new message was published. By introducing the design like that we did in fact use the "publish-subscribe" (or "event driven") paradigm. In our case we had only one kind of event "publish message" and all the Clients were automatically subscribed to the "publish event" one once they got connected. General definition of the publish-subscribe system says:

29. *A publish-subscribe system is a system where publishers publish structured events to an event service and subscribers express interest in particular events through subscriptions which can be arbitrary patterns over the structured events.*

There is a number of frameworks implementing publish-subscriber pattern and offering corresponding API. In case the framework supports "distributed events" (publish-subscribe over network), it provides a middleware service in addition to API library. The middleware is dealing with all the network related tasks, the API refers to middleware in case event has to be delivered to a remote host. Next we will list some of the frameworks and focus on Python implementations in particular. In the end we will show the MessageBoard application ported onto one of them.

Distributed events frameworks

Also called “distributed publish-subscribe”, “event service” or “broker network”. As we have said before the idea is quite simple: a “publish-subscribe” paradigm is in fact “event driven”. We can emit events and we can subscribe for certain events, assuming once event is emitted a certain action will be triggered.

- Centralized (Broker Service)
  - CORBA Event Service
  - OSE Framework
- Distributed (Broker Network)
  - Apache Kafka
  - DDS (Data Distribution Service)

None of the above mentioned are in fact pure Python, and rather platform and language neutral middlewares with Python interfaces.

Message queues

Also called “message brokers”, middleware is build around central entity storing the queues that can be shared/subscribed by different entities. Allows building different communication paradigms, which are well illustrated by following link (RabbitMQ tutorial):

[https://www.rabbitmq.com/getstarted.html]

- Python supported frameworks:
  - snakemq - implements own message brokers (not compatible with AMQP neither ZeroMQ)
  - pika python library (AMQP client library, compatible with RabbitMQ server)
  - pyzmq python library (ZeroMQ python bindings)
  - celery (distributed task queue implementation, compatible with multiple message brokers)

Bibliography

[1] Connelly Barnes. Integer factorization algorithms. http://www.connellybarnes.com/documents/factoring.pdf, 2004. [Online; accessed 18-October-2016].

[2] George F Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design*. pearson education, 2005.

[3] Python Software Foundation. Python basic xml-rpc server. https://docs.python.org/2/library/simplexmlrpcserver.html, 2016. [Online; accessed 27-October-2016].

[4] Python Software Foundation. Python higher-level threading interface. https://docs.python.org/2/library/threading.html, 2016. [Online; accessed 18-October-2016].

[5] Python Software Foundation. Python process-based threading interface. https://docs.python.org/2/library/multiprocessing.html, 2016. [Online; accessed 18-October-2016].

[6] Python Software Foundation. Python xml-rpc client api. https://docs.python.org/2/library/xmlrpclib.html, 2016. [Online; accessed 27-October-2016].

[7] Cameron Hughes. *Parallel and distributed programming using C*. Addison-Wesley, Boston, 2004.

[8] Inc. Scripting News. Xml-rpc. http://xmlrpc.scripting.com/, 2016. [Online; accessed 27-October-2016].