Redes de Computadores (RCOMP)

Lecture 07

2017/2018

- UDP and TCP network applications development.

User Datagram Protocol (UDP)

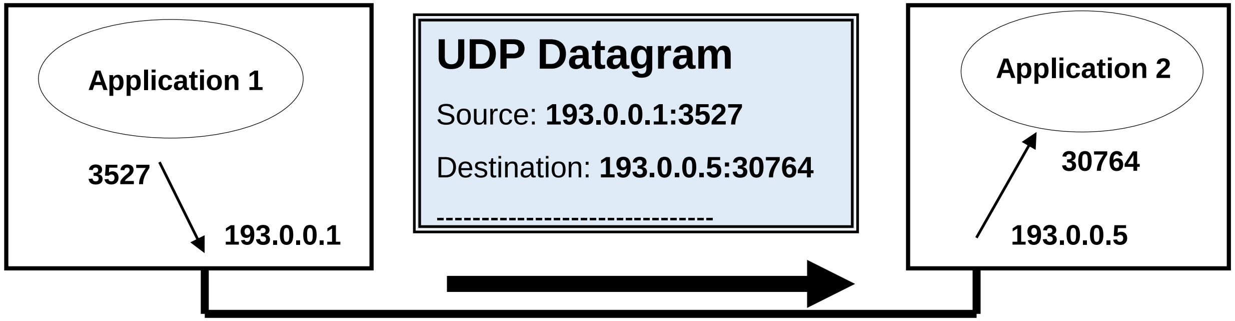
As the name point out, UDP is a datagram service, this means, it offers the transportation of variable size data blocks.

Being a service intended to be used by network applications, it also provides a way to identify individual applications.

Source and destination applications of each UDP datagram are identified by 16-bit numbers, known as port numbers.

Port numbers are node specific, nevertheless, a port number together with the node's IP public address will **universally and uniquely identify a network application**.

The image below represents a UDP datagram being sent from **Application 1** to **Application 2**:



Associating a port number to an application is called binding. This is something the application must request to the operating system. One thing the operating system will ensure is that there is only one application using each port number.

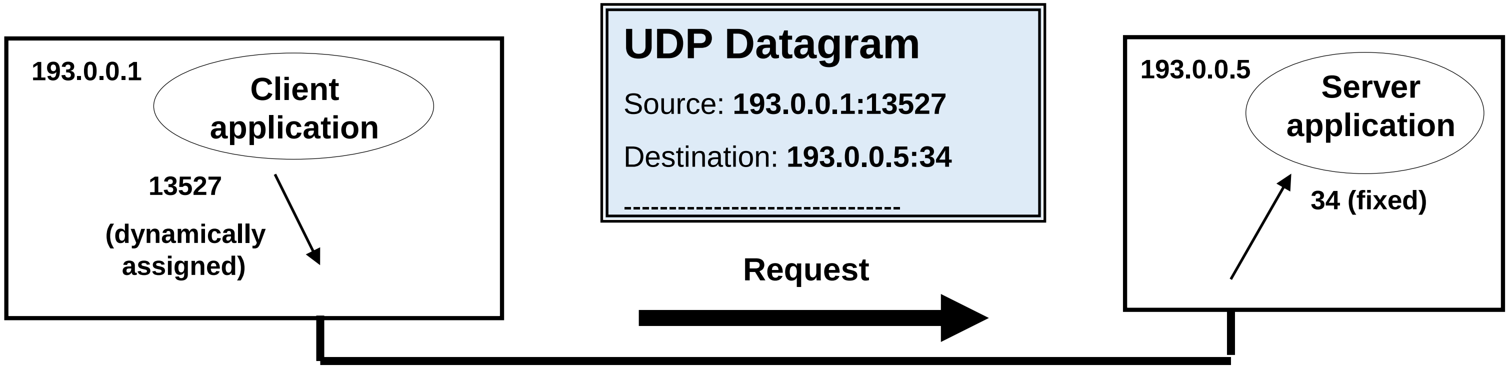
UDP network sockets

From the application development point of view, interactions with the network are based on the socket concept. UDP sockets must first be associated with local port numbers, only afterwards they will be ready to receive and send UDP datagrams.

Assigning a local port number (bind) to a UDP socket will be successful only if no other local application is currently using that same port number.

Yet, for some applications any available port number will do. A dynamically assigned port number can then be requested to the operating system (usually by binding to port number zero).

Most network applications use the client-server model. The server must use a fixed local port number (pre agreed with the client). On the other hand, the client may use any port number (dynamically assigned).



Sending and receiving UDP datagrams

The UDP datagram service is not connection-oriented, nor offers any kind of delivery guarantee. Each datagram is handled individually, for each datagram to be sent, a destination IP address and a destination port number must be provided by the sender.

For a sent datagram to be received, on the specified destination IP address a UDP application should be running and listening on the specified destination port number.

Because UDP is not connection-oriented, broadcasting is possible. By specifying the broadcast address of an IPv4 network, or the generic broadcast address (255.255.255.255), as destination address, all nodes will receive a datagram copy on the specified destination port.

Reception will be usually synchronous. The reception operation halts the application execution (process or thread) until a datagram is received.

Once an application receives a UDP datagram, it also obtains the source IP address and source port number. In the case of a server they will be required later to send the reply back to the correct client.

As already mentioned, UDP offers no delivery guarantee. Worst than that, the sender gets no feedback, hence it will not know if the datagram has ever arrived at the destination or not.

There is one exception to this lack of feedback: when the target node is operational, and the destination port number is not in use by any application, the target node sends back an ICMP message **destination port unreachable**. Because this message contains the original UDP header (port numbers included), the source node will be able to relate it to the application and raise an API error on the sender socket.

Transmission Control Protocol (TCP)

TCP protocol provides a significantly higher quality service when compared to UDP. TCP creates logical bidirectional communication channels between applications located on different network nodes. These logical channels, commonly referred to as TCP connections, provide guarantees on data delivery and data sequence.

Each TCP connection is for exclusive use by the two applications that created it. It’s a dedicated communication channel in which third parties cannot intervene.

Data sending and receiving on TCP connections is performed in flux, there’s no data blocks concept, all data is sent and received on a byte-by-byte continuous flow.

Data transactions via TCP are possible only after a successful connection establishment phase.

To establish a TCP connection between two applications, each has to undertake a different role:

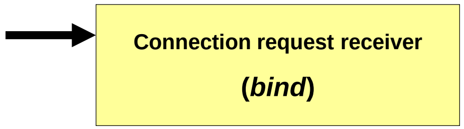
- One application accepts the TCP connection (usually the TCP Server)

- The other application requests the TCP connection (usually the TCP Client)

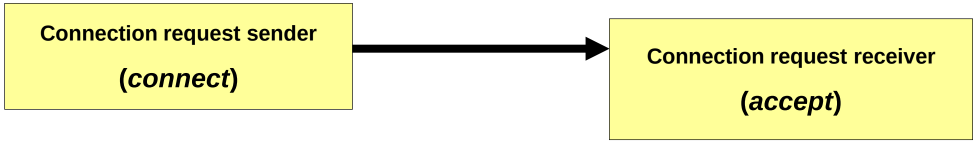
TCP connection establishment

Establishing a TCP connection requires coordinated efforts from two applications, one of them takes the connection request receiver role. First, it will bind the TCP socket to a fixed TCP port number, previously agreed with the issuer of the connection request. Then it waits for a connection.

The other application can now send a TCP connection establishment request. It must know the destination IP address (the node where the first application is running) and the destination port number (the first application's local port).

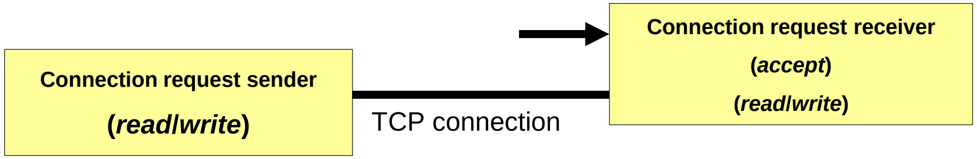


The receiving application associates a TCP port number to the TCP socket (bind). The application is then available to receive connection requests on that TCP port number.



The sender application makes a connection establishment request (connect) to the IP address of the receiving node and port number used by the receiving application.

The receiving application accepts the connection request (accept), the TCP connection between the two applications is now created.



On the receiver side, the acceptance of the connection (accept) has created a new socket associated to the TCP connection, the original socket remains open to receive other connection requests.

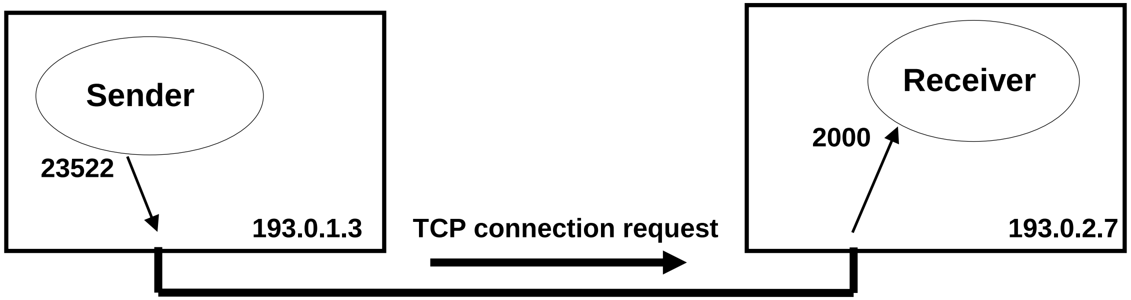
The two ends of the TCP connection are accessible to both applications, at the sender on the socket used to make the connection request, at the receiver, on the new socket created when the connection request was accepted.

Sending and receiving data over TCP connections is performed in byte stream, common generic functions like those used in the read and write operations on files and pipes are supported.

TCP – Dedicated channels

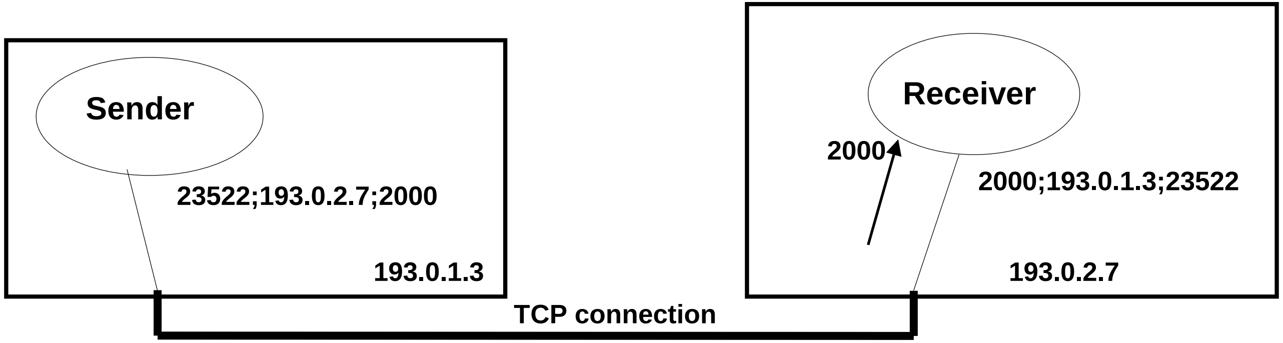
A TCP connection is a dedicated communication channel between two applications, for all effects it’s equivalent to a bidirectional pipe.

Before the TCP connection establishment, involved sockets are associated with local port numbers only.



After TCP connection establishment, each connected socket will also have associated with it the remote IP node address and the remote port number.

The receiver has now two different sockets associated to the same local port number. Although only the connected socket has a remote address associated.



Incoming TCP data is made available on a connected socket if: data destination port matches the local port number for the socket **and** the source IP node address matches the associated remote address **and** the source port number matches the associated remote port number.

UDP – Datagrams sending and receiving

Because UDP is connectionless, once a local port number is assigned to the socket (bind), then UDP datagrams can be received and sent.

Sending and receiving operations are managed through queues (FIFO), this is especially meaningful when receiving datagrams. From the instant a local port number is assigned to the UDP socked, it starts receiving and queueing datagrams.

As such, when an application requests for a datagram receiving, in fact, is requesting for the socket input queue reading. Although applications can overcome this behaviour, if the queue is empty, then the application will be blocked, wait until a datagram is received.

UDP datagrams are sent by calling a method or *system-call* that requires a data block, the data block size and the destination address (destination IP node address + destination UDP port number).

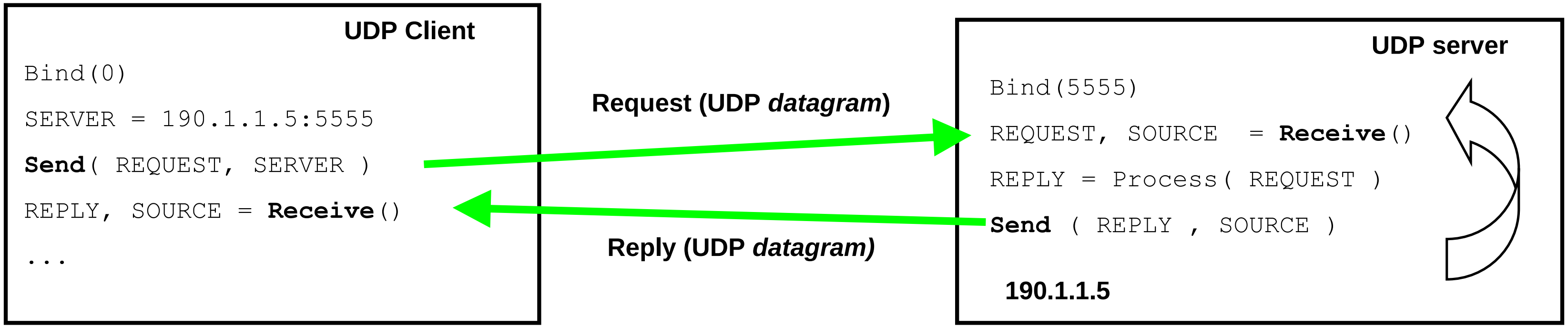
UDP datagrams are received by calling a method or *system-call* that returns a data block, the received data block size and the source address (source IP node address + source UDP port number). If the socket input queue is empty, the operation usually blocks the process or thread until a datagram is received.

UDP – Unreliable services

UDP services are very simple but they provide no guarantee whatsoever, is up to applications (the application protocol) solving problems that may arise.

There is no grantee a sent datagram will ever reach the destination, neither any feedback to the sender concerning the delivery success. Due to unhandled delivery issues, poorly designed application protocols can easily compromise transactions.

The following diagram represents the standard interaction between a UDP client and server:



UDP Client

```
Bind(0)
SERVER = 190.1.1.5:5555
Send( REQUEST, SERVER )
REPLY, SOURCE = Receive()
...
```

UDP server

```
Bind(5555)
REQUEST, SOURCE  = Receive()
REPLY = Process( REQUEST )
Send ( REPLY , SOURCE )
```

In this case, problems with UDP’s lack of reliability will arise on the client side. The issue is the client trusts there will be a reply from the server. If no reply is received the client blocks forever in the Receive call.

No reply being received by the client can be due to any of the following three causes: the request was lost, the server failed to reply (crashed or is offline), or the reply was lost.

UDP clients – fault tolerance

UDP is connectionless and unreliable, within the client-server model these characteristics pushes the design to **stateless and idempotent servers**. For a stateless and idempotent UDP server, a request or reply loss has no impact.

As already seen, for a UDP client, the scenario is different because it gets dependent on a reply arrival. The key to solve this client lock issue, is setting a reply receive timeout instead of waiting forever. There're several ways to achieve this:

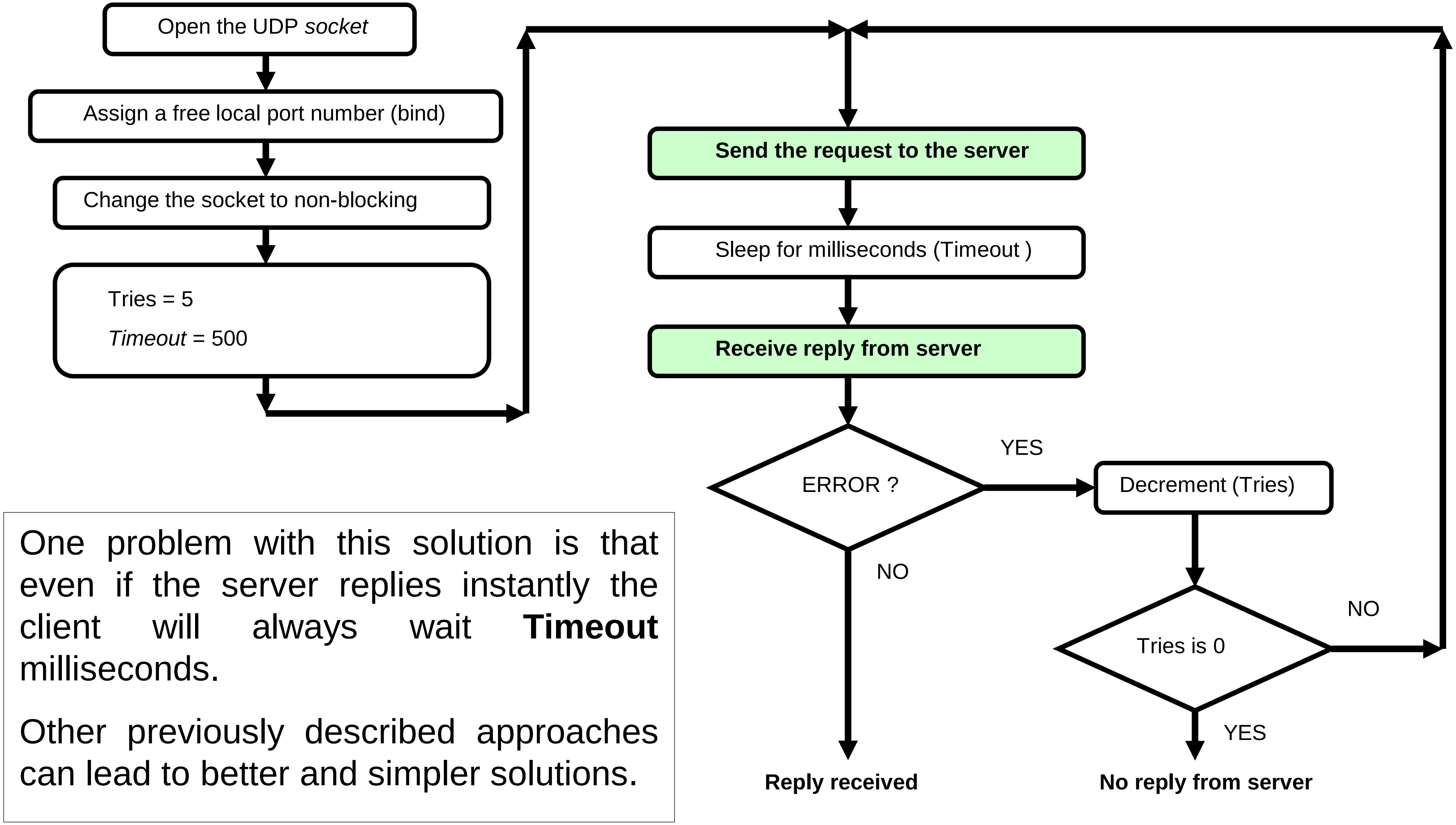
- **Non-blocking sockets** – the behaviour off a socket can be changed from blocking to non-blocking, by doing so, any operation on the socket that cannot be executed immediately (would block process or thread), will return immediately with an error.

- **Sockets with timeout** – settling a timeout for the socket, if the operation over the socket takes longer than the timeout, the operation will abort with an error.

- **Threads or processes** – create a separate thread or process to perform the blocking operation, thus, the main thread or process is not blocked. The created thread or process can also be terminated after a period of time (timeout).

- **Sockets monitoring** – some languages (e.g. C) provide functions capable of detecting when a socket is ready for reading. The select function in C language allows this and also the settling of a timeout.

A fault tolerant UDP client – Non-blocking Socket


Open the UDP *socket*
Assign a free local port number (bind)
Change the socket to non-blocking
Tries = 5
Timeout = 500
Send the request to the server
Sleep for milliseconds (Timeout)
Receive reply from server
ERROR ?
YES
NO
Decrement (Tries)
Tries is 0
NO
YES
Reply received
No reply from server

One problem with this solution is that even if the server replies instantly the client will always wait **Timeout** milliseconds.

Other previously described approaches can lead to better and simpler solutions.

UDP – Sending to the broadcast address

This is something impossible to do with a connection oriented protocol like TCP, hence, it may be one reason for using in some cases UDP instead of TCP.

When an IP packet is sent to a broadcast address, all nodes will receive a copy. By sending an UDP datagram to a broadcast address has the same effect, so all nodes will receive a datagram copy in the specified destination port number.

In IPv4, a network’s broadcast address is the last address of the network (the node address zone filled with ones). However, using this address in an application source code is a bad idea as that application would only work on that specific network.

A network specific broadcast address would have to be provided on runtime, for instance, through a configuration file. There’s though o simpler option, for broadcast on the local network the generic broadcast address (255.255.255.255) can be used instead. It means broadcast on the local network, whatever it my be.

One important use of UDP broadcast is on servers discovery. By sending a request to the broadcast address, clients can reach servers without previously knowing their node addresses.

This ability is used in several local network environments like for instance Windows NetBIOS to locate servers on the Network Neighbourhood.

UDP – Sending to broadcast – applications discovery

Even if afterwards TCP is used, UDP broadcast can be used in first place to find applications (get IPv4 node addresses of applications available on the local network).
In a client-server architecture, two approaches can be used:

Server announcement – periodically the server sends a broadcast UDP datagram to a settled destination port number announcing its presence, by doing so, every client on the network can learn its IP node address.
Clients will start by listening UDP datagrams in the settled port number and building a list of available servers, including their IP node addresses.

Request for servers – the client sends a broadcast UDP datagram to a settled port number asking for servers.
Servers on the local network will reply, allowing the client to collect a list of available servers and their IP node addresses.

In either approach, the client is able to build a list of available servers IP addresses. It can then pick one, and then communicate with it, either by using UDP or TCP.

UDP – The datagram size issue

Beyond the lack of reliability and feedback in UDP datagrams delivery, UDP applications face another constraint:

How much information can be placed inside each UDP datagram?

Theoretically an IPv4 datagram can be up to 65535 bytes long, thus the volume of data a UDP datagram could hold should be that value subtracted by the size of the IPv4 header (20 to 60 bytes) and the size of the UDP header (8 bytes).

Though fragmentation is not currently used, so, this is only theoretical. RFC 791 (IPv4) settles every IPv4 node must be able to handle IPv4 datagrams with up to 576 bytes.

The safe way to ensure the excessive size of a UDP datagram will not compromise its delivery, is avoiding its content length to be above 512 bytes (RFC 791).

When transactions volume is above 512 bytes there are two options:

- Forget about UDP datagrams and use a TCP connection instead.

- Split data into several UDP datagrams.

UDP – multiple datagram transactions

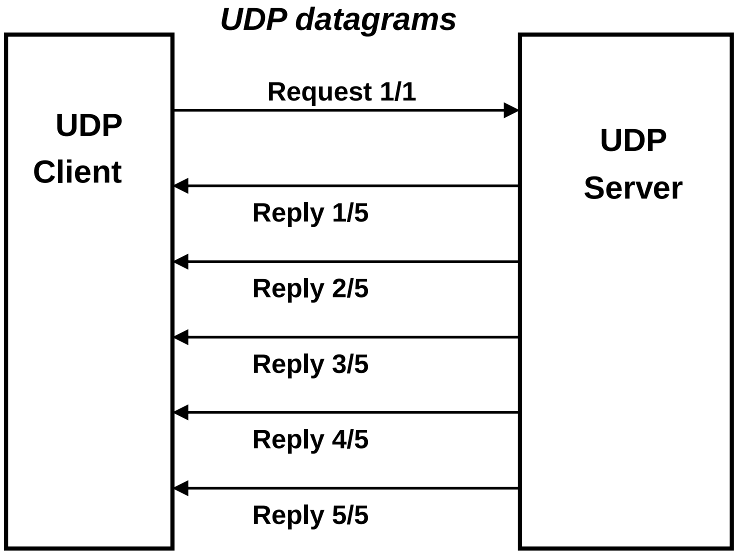
Usually when more than 512 bytes of data are to be exchanged the best option would be TCP and not UDP.
UDP does not guarantee the delivery of datagrams nor the sequence of delivery, thus, splitting a data block in a set of datagrams will require the applications to implement several control proceedings.

As minimum procedures, the receiver of a set of datagrams must be informed of the total number of datagrams, also each datagram must have a sequence number:

The application protocol will also have to handle error recovery in more or less sophisticated form:

- the failure in receiving a datagram may abort the entire transaction and a new initial request must be made.

- the client may be able to request the retransmission of only the missing datagrams.



UDP socket – association to remote addresses

Even tough UDP is connectionless, a UDP socket can be associated to a remote IP address and remote port number.

In C language this is achieved by using the **connect** system call, the same function used to establish TCP connections, but that's the only resemblance. With UDP connections don't exist, for instance the connect function can be used to associate the socket to some remote address and later to another remote address, this would be impossible it TCP where connections exist.

Associating a UDP socket to a remote address affects both sending and receiving datagrams:

SENDING – in each sending operation, the remote address specification is no longer required, the already associated remote address will always be used.

RECEPTION – the socket will receive datagrams if, and only if, they are coming from the associated remote address (filtering).

Associating a remote address to UDP sockets has no other effect beyond these described behaviours, all UDP characteristics like the lack of reliability persist.

These associations can, however, be useful for some applications especially regarding filtering in datagrams reception. Multi-process UDP servers can take advantage of this.

TCP connections – Sending and receiving data

TCP has some significant advantages: ensures the reliable delivery of data in the order it was sent without any size limitations. Sending and receiving data throughout a TCP connection is byte oriented, this raises some particular issues:

There must be an exact match between the number of bytes requested to be read on one connection's end and the number of bytes written on the other connection's end.

- If more bytes are requested to be read than those written, the reading operation will block, waiting for the remaining bytes.
- If less bytes are requested to be read than those written, the reading won't block, however, the unread bytes will emerge in the next reading.

Ensuring a perfect match (read and write byte synchronization) is a mission for the application protocol. The application protocol must unambiguously define all data exchanges between applications, including amounts of data for each transaction.

TCP connection – Sending and Receiving – The application Protocol

An application protocol is a set of rules two network applications must follow so they can talk to each other with no ambiguities. It will define proceedings for each application and data exchanges contents to take place in each proceeding.

In the case of a TCP connection, it's up to the application protocol to ensure a perfect read and write synchronization on both ends of the connection.

There are three basic approaches that can be used isolated or combined together:

- Fixed-size blocks – if a fixed-size message is established by the application protocol for each context, then reading those messages won't be a problem. Though this solution may result in network overhead if the real amount of transferred useful data is far less than the fixed-size.

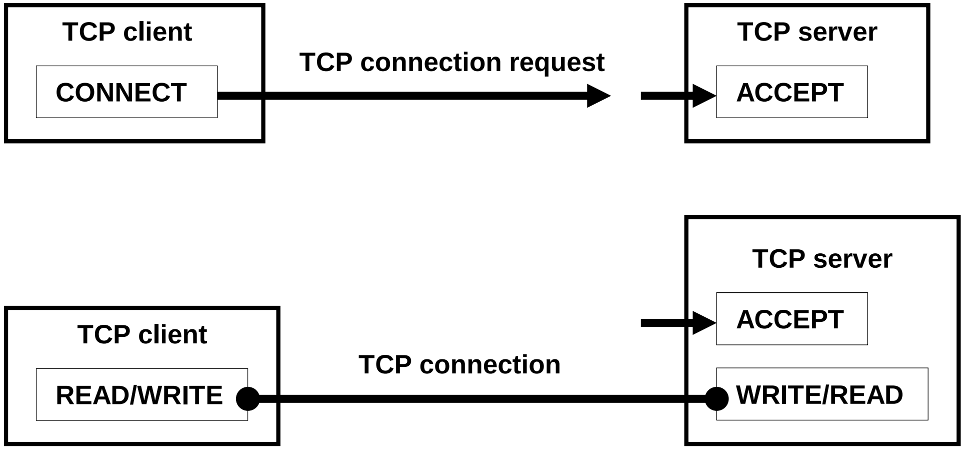
- Explicit block size statement – the sender start by informing about the size of the following message. With this knowledge, the receiver can then request the reading of the exact amount of bytes. For instance, in HTTP, the Content-Length header line informs the message's reader about the body size.

- End of block marker – the application protocol establishes an end of message mark. The receiver reads one byte at a time and stops when it gets the end mark. It's easy to implement, but it must be guaranteed the mark will not occur inside the message itself. In HTTP this is used twice, the message header (plain text) uses the CR/LF sequence to mark the end of each line, and the CR/LF/CR/LF sequence (one empty line) is used to mark the header's end.

TCP servers

To start with, a TCP server is an application that accepts TCP connection requests in a port number defined by the application protocol.
Once the TCP server accepts a connection request from a client, a new TCP connection is established. On the server side, **a new socket is created** representing the connection with that client.

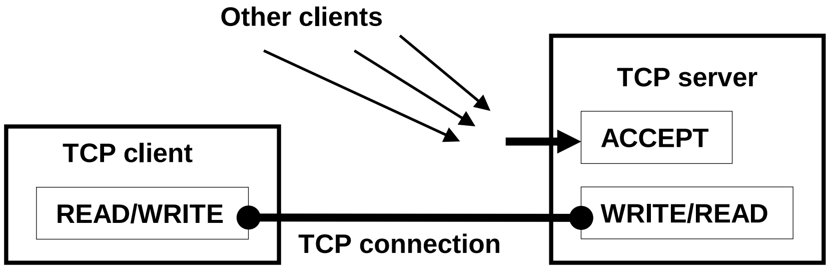
The TCP connection between the client and the server is a permanent dedicated channel that can only be used by those two applications. It will be available until one of them decides to close it.



The application protocol defines the sequence of messages, and messages contents to be exchanged between the client and server over the TCP connection. Because it´s a permanent and dedicated channel, the TCP connection allows a session of successive interactions, not just a single request and a single reply like in a typical UDP client-server.

TCP multi process servers

After accepting a client's TCP connection the server must nevertheless remain available to accept other connection requests from further clients.

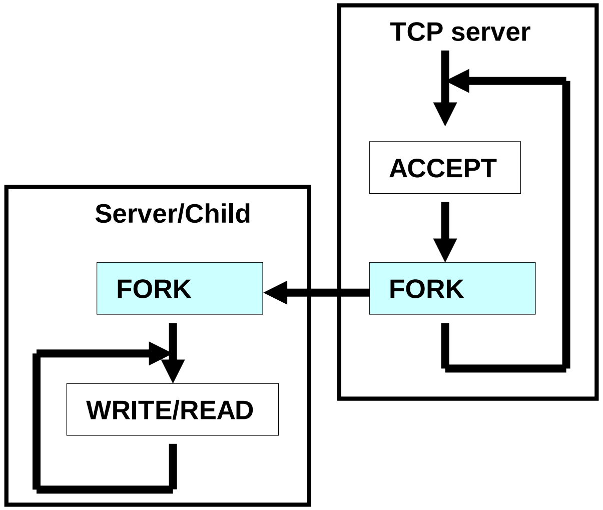


This means the server must be able to dialogue with the connected client (using the application protocol) and, at the same time, it must be available to accept new connection requests.

Thus, it has to handle with two sockets. Must keep accepting new clients on the initial socket, and must exchanging information with the connected client over the new socket.

There are several ways to address this problem, in C/Unix, one of the most popular is creating a child process to handle the connected client.

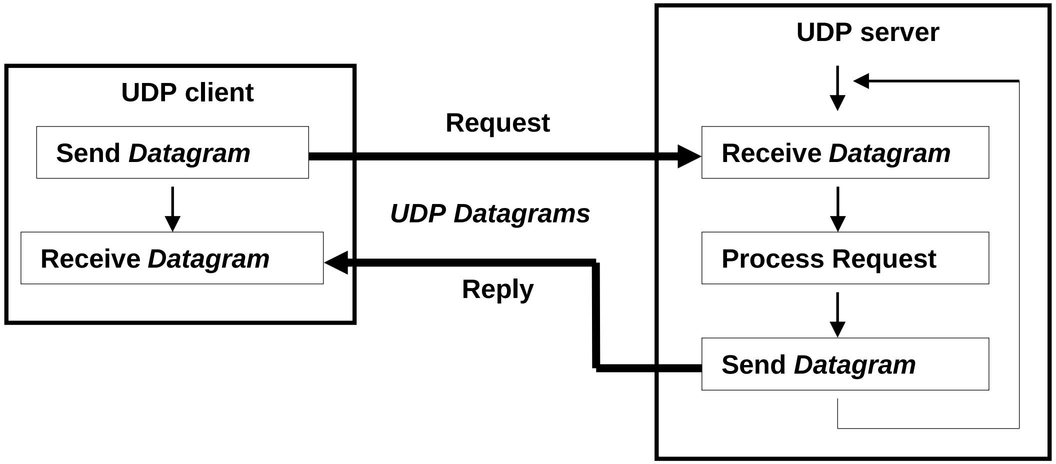
This solution has the advantage of creating a new independent process for each client connection, thus, each client will have a dedicated child process totally available for it.



UDP servers

UDP servers are usually rather simpler than TCP servers. This is because, due to UDP limitations, UDP servers are mostly stateless and idempotent. A simple UDP server, receives a UDP datagram on a settled port number, this datagram holds a request, the server then processes the request and sends back a reply to the client, also inside a UDP datagram.

Because there's no connection, each request is handled by the server as unique and not related with other requests. When the server receives the request it stores the source address for later use when sending the reply. Once the reply is sent the server forgets all about that client and jumps to the next client request.



There's no connection or session between the client and server, the server receives the requests in the order they arrive (or are stored at the input queue), a request is handled by the server only after previous requests are processed and replied.

Implementing a session oriented application protocol over UDP is not impossible, however, the server will be somewhat complex. The server would have to store different communication backgrounds, one for each client, when a request is to be processed the server would have to select the correct background based on the client's address (IP address + port number).

Asynchronous reception

Because a network application has no direct control over network data arrival timing, network data reception must regarded as inherently asynchronous. As seen, some applications simply stop and wait until data arrives, this is called synchronous reception.

Synchronous reception is not always acceptable, for instance, if the application:

- has several sockets and is not aware in which of them data will arrive first.

- the application has other things to do and can’t stay blocked waiting for data arrival.

Solutions for asynchronous reception:

- non blocking sockets or timeout – periodically the application tries to receive data to see if it’s available. This may become more efficient if combined with a data availability alert mechanism (e.g.: SIGIO in Unix systems).

- *threads* or processes – for each socket a *thread* or a process is created to receive data, only that thread or process will be blocked waiting for data.

- specific function to monitor a set of sockets (e.g. select in C language) - the select function can be used to monitor a set of given sockets, and unlock when there is data to be received in one of them. With select a timeout can also be specified.

Application protocol

For computer networks in general, a protocol is a set of rules aiming the unambiguous exchange of information between two communicating entities over a network. When communicating entities are applications (placed at OSI layer seven, the application layer) these protocols are called application protocols.

A protocol should be the most formal and exact possible specification for:

- Dialogue phases (state diagram).
- All message contents used in different phases of dialogues.
- All dialogues and possible actions, their objectives and possible results.
- Proceedings for error detection and recovery.

Application protocols should be flexible enough to allow new features, yet keeping backward compatibility:

- Desirably, the message’s first element should identify the protocol version and then the message type.
- The general message format should be flexible enough to be able to hold new more specific formats.