Inter-Process Communication: Network Programming using TCP Java Sockets

Ali Fanian,

Isfahan University of Technology

INTERPROCESS COMMUNICATION

From Chapter 4 of Distributed Systems Concepts and Design

Outline: Communications Models

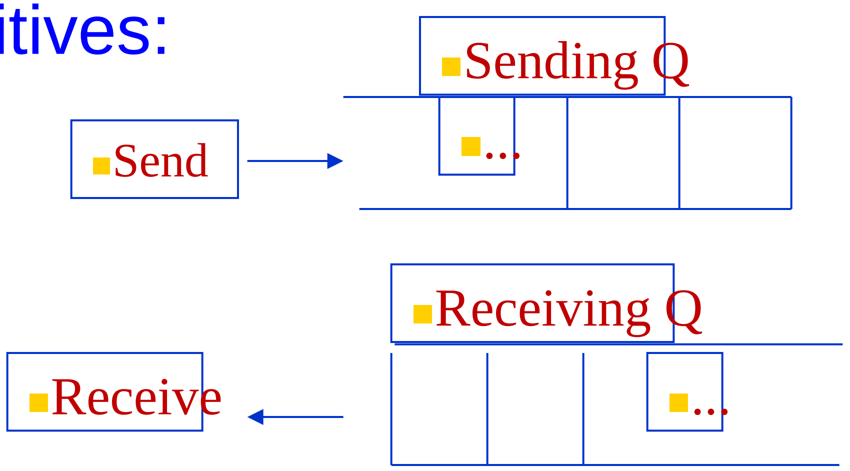
- Communication Models:
  - General concepts.
  - Message passing.
  - Distributed shared memory (DSM).
  - Remote procedure call (RPC) [Birrel et al.]
    - Light-weight RPC [Bershad et al.]
  - DSM case studies
    - IVY [Li et al.]
    - Linda [Carriero et al.]

Communication Paradigms

- 2 models
  - Message Passing (MP)
  - Distributed Shared Memory (DSM)
- Message Passing
  - Processes communicate by sending messages.
- Distributed Shared Memory
  - Communication through a “virtual shared memory”.

Message Passing

- Basic communication primitives:
  - Send message.
  - Receive message.
- Modes of communication:
  - Blocking versus non-blocking.
- Semantics:
  - Reliable versus unreliable.



non-Blocking Communication

- Non-blocking send: sending process continues as soon message is queued.
- Blocking or non-blocking receive:
  - Blocking:
    - Timeout.
    - Threads.
  - Non-blocking: proceeds while waiting for message.
    - Message is queued upon arrival.
    - Process needs to poll or be interrupted.

Reliability of Communication

- Unreliable communication:
  - “best effort” - send and hope for the best
  - No ACKs or retransmissions.
  - Application must provide its own reliability.
  - Example: User Datagram Protocol (UDP)
    - Applications using UDP either don’t need reliability or build their own (e.g., UNIX NFS and DNS (both UDP and TCP), some audio or video applications)

Reliability of Communication

- Reliable communication:
  - Processes have some guarantee that messages will be delivered.
  - Example: Transmission Control Protocol (TCP)
  - Reliability mechanisms:
    - Positive acknowledgments (ACKs).
    - Negative Acknowledgments (NACKs).
  - Possible to build reliability atop unreliable service

Interprocess communication

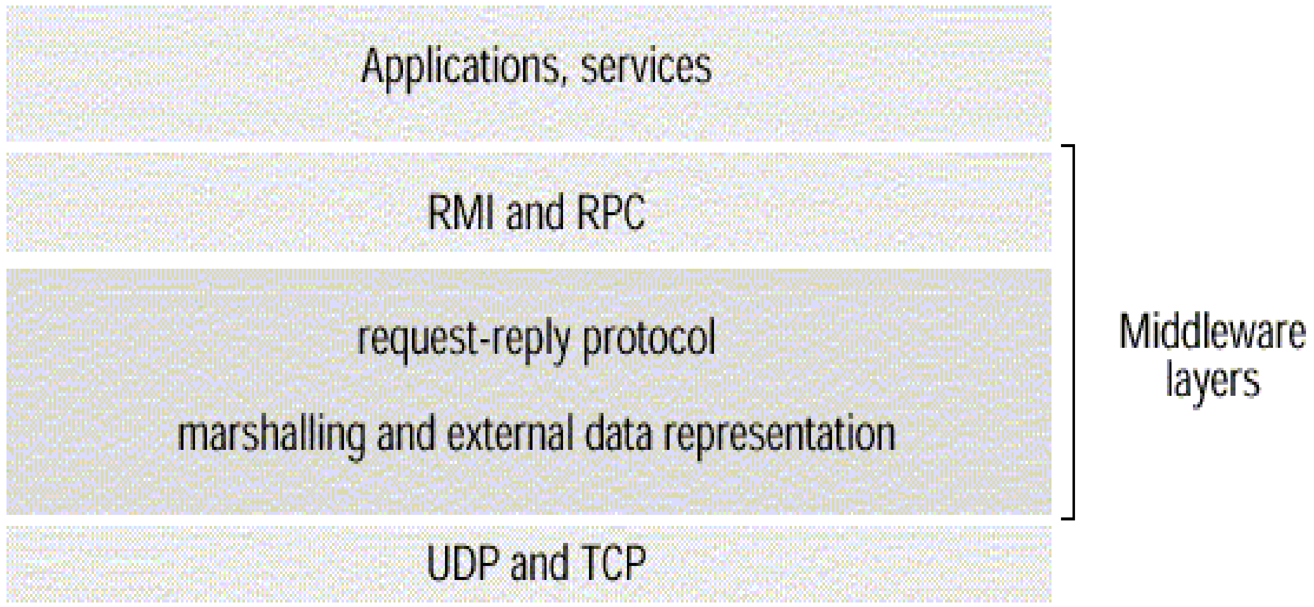
- There are some APIs for interprocess communication in the internet provides both datagram and stream communication.
- The two communication patterns that are most commonly used in distributed programs:
  - Client-Server communication
    - The request and reply messages provide the basis for remote method invocation (RMI) or remote procedure call (RPC).
  - Group communication

Interprocess communication

- Remote Method Invocation (RMI)
  - ➢ It allows an object to invoke a method in an object in a remote process.
    - ❖ E.g. CORBA and Java RMI
- Remote Procedure Call (RPC)
  - ➢ It allows a client to call a procedure in a remote server.

Interprocess communication

- This chapter is concerned with middleware.



- **Middleware layers**

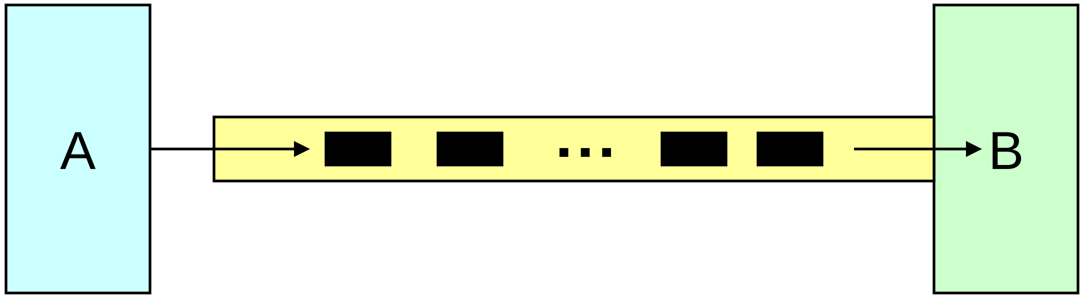
Interprocess communication

- Request-reply protocols are designed to support client-server communication in the form of either RMI or RPC.

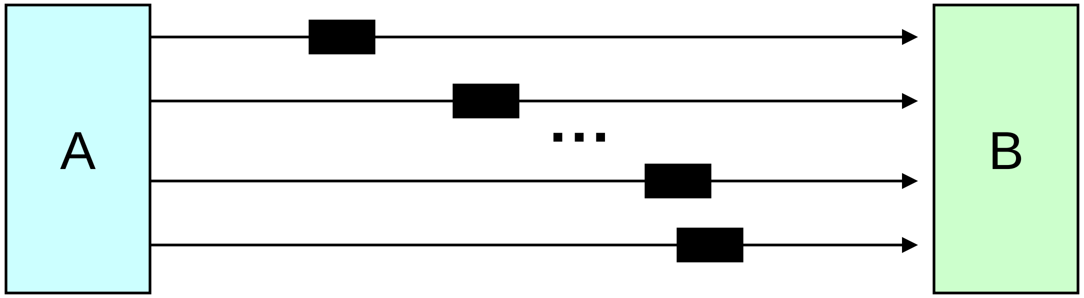
- Group multicast is a form of interprocess communication in which one process in a group of processes transmits the same message to all members of the group.

Communication Models

TCP Vs UDP Communication



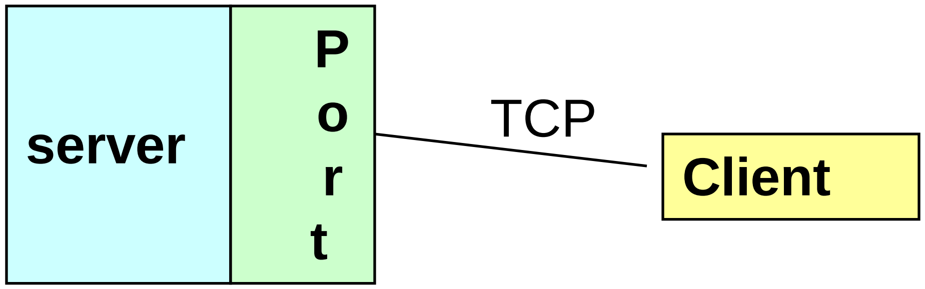
- Connection-Oriented Communication

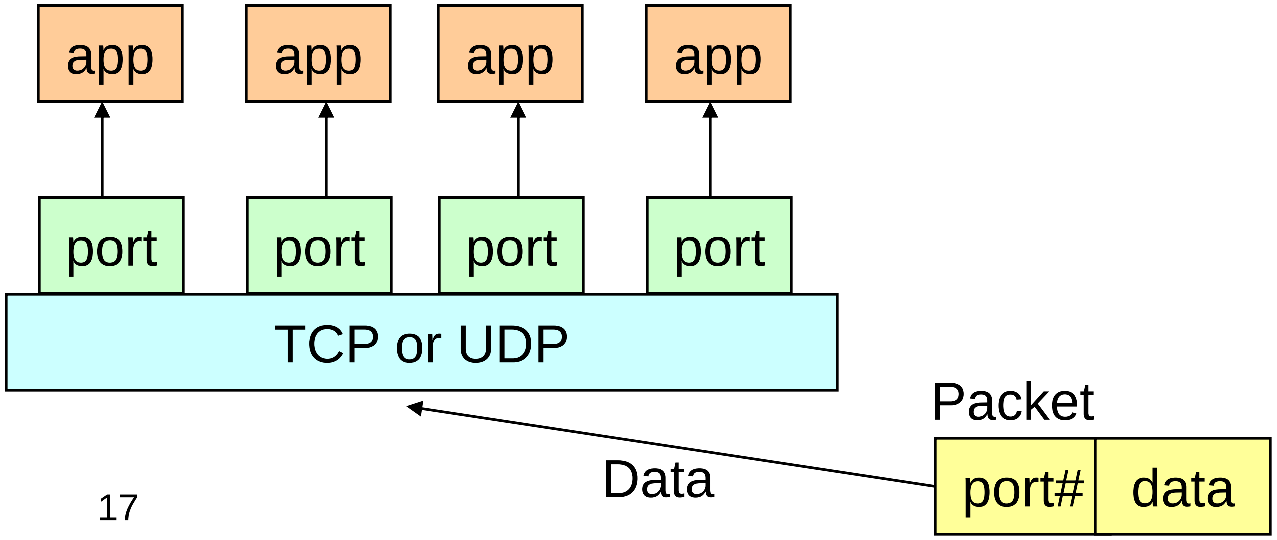


- Connectionless Communication

Understanding Ports

- The TCP and UDP protocols use *ports* to map incoming data to a particular *process* running on a computer.





Understanding Ports

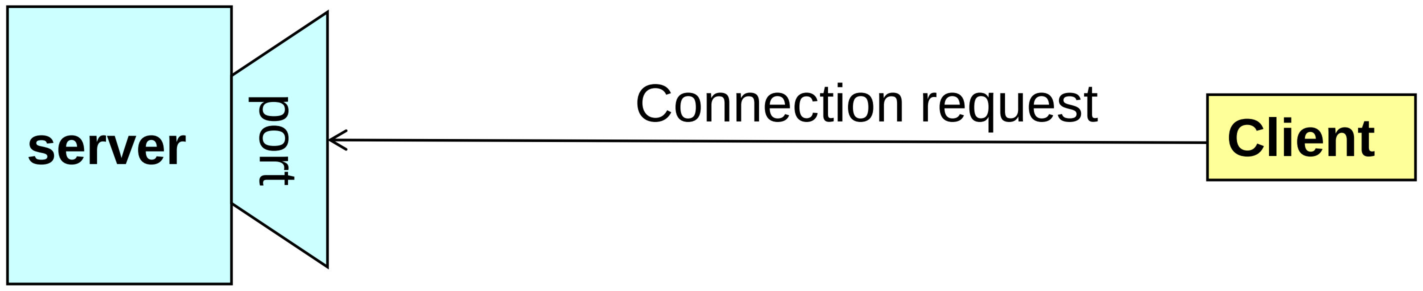
- Port is represented by a positive (16-bit) integer value
- Some ports have been reserved to support common/well known services:
  - ftp   21/tcp
  - telnet 23/tcp
  - smtp 25/tcp
  - login 513/tcp
- User-level processes/services generally use port number value >= 1024

Sockets

- Sockets provide an interface for programming networks at the transport layer
- Network communication using Sockets is very much similar to performing file I/O
  - In fact, socket handle is treated like file handle.
- Socket-based communication is programming language independent.
  - That means, a socket program written in Java language can also communicate to a program written in Java or non-Java socket program

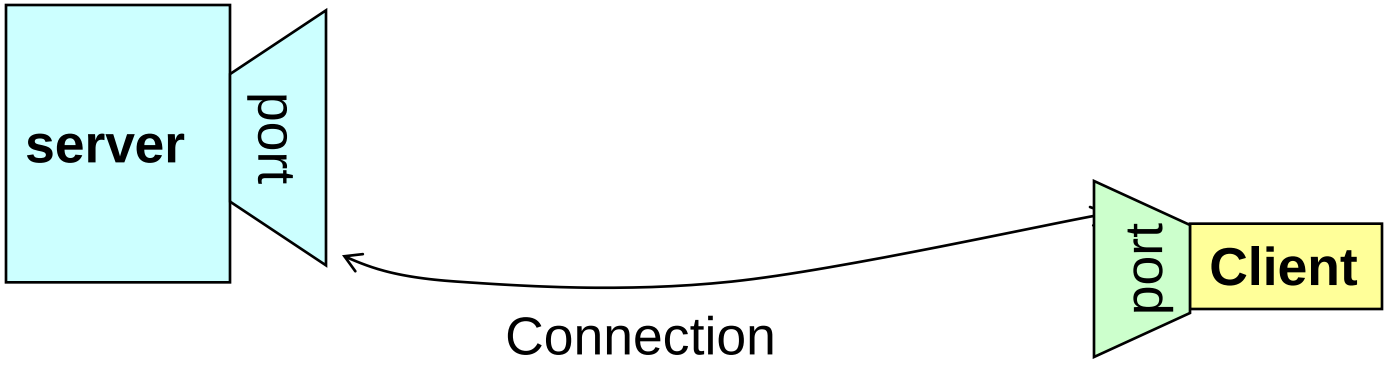
Socket Communication

- A server (program) runs on a specific computer and has a socket that is bound to a specific port. The server waits and listens to the socket for a client to make a connection request.

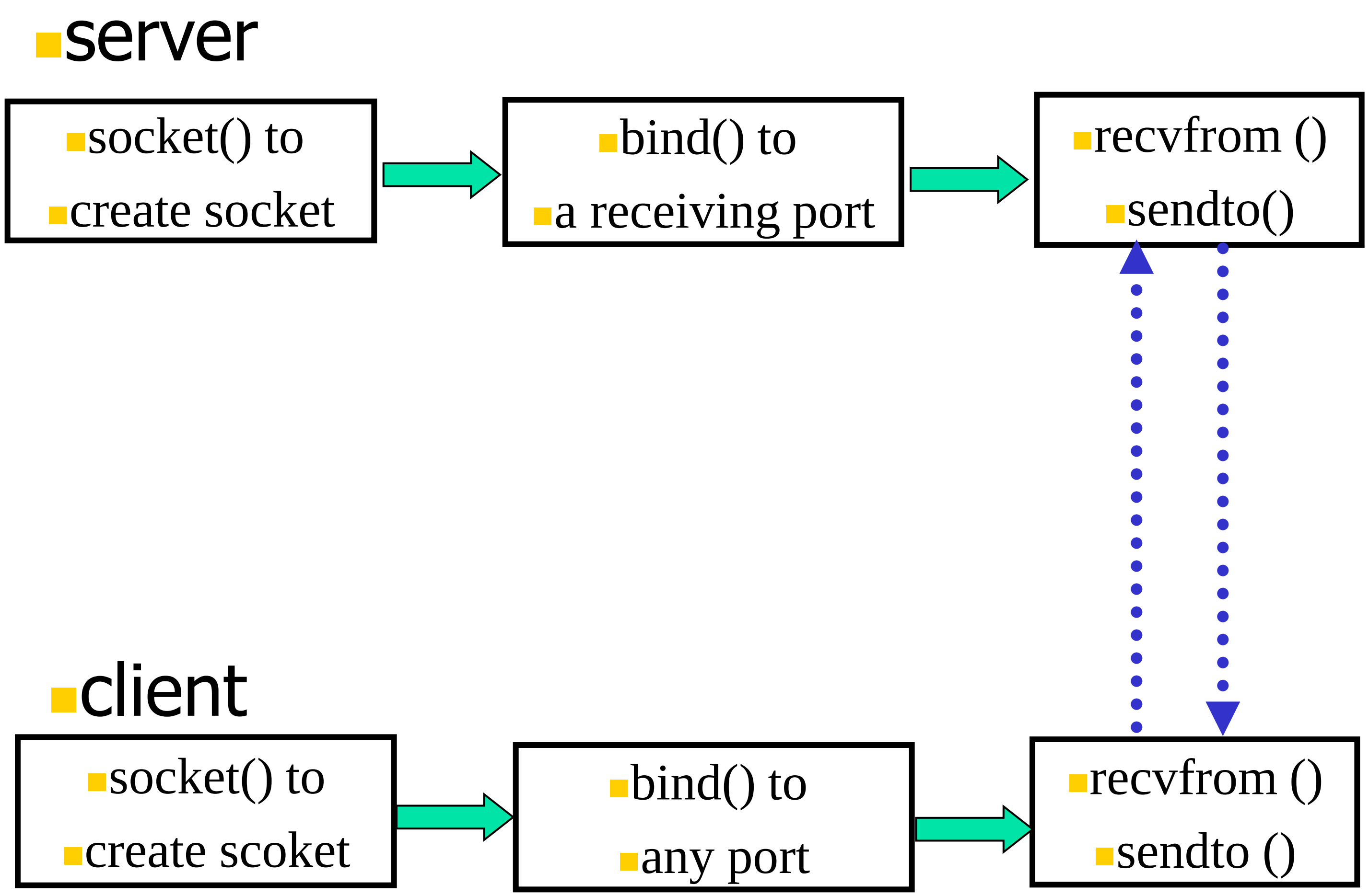


Socket Communication

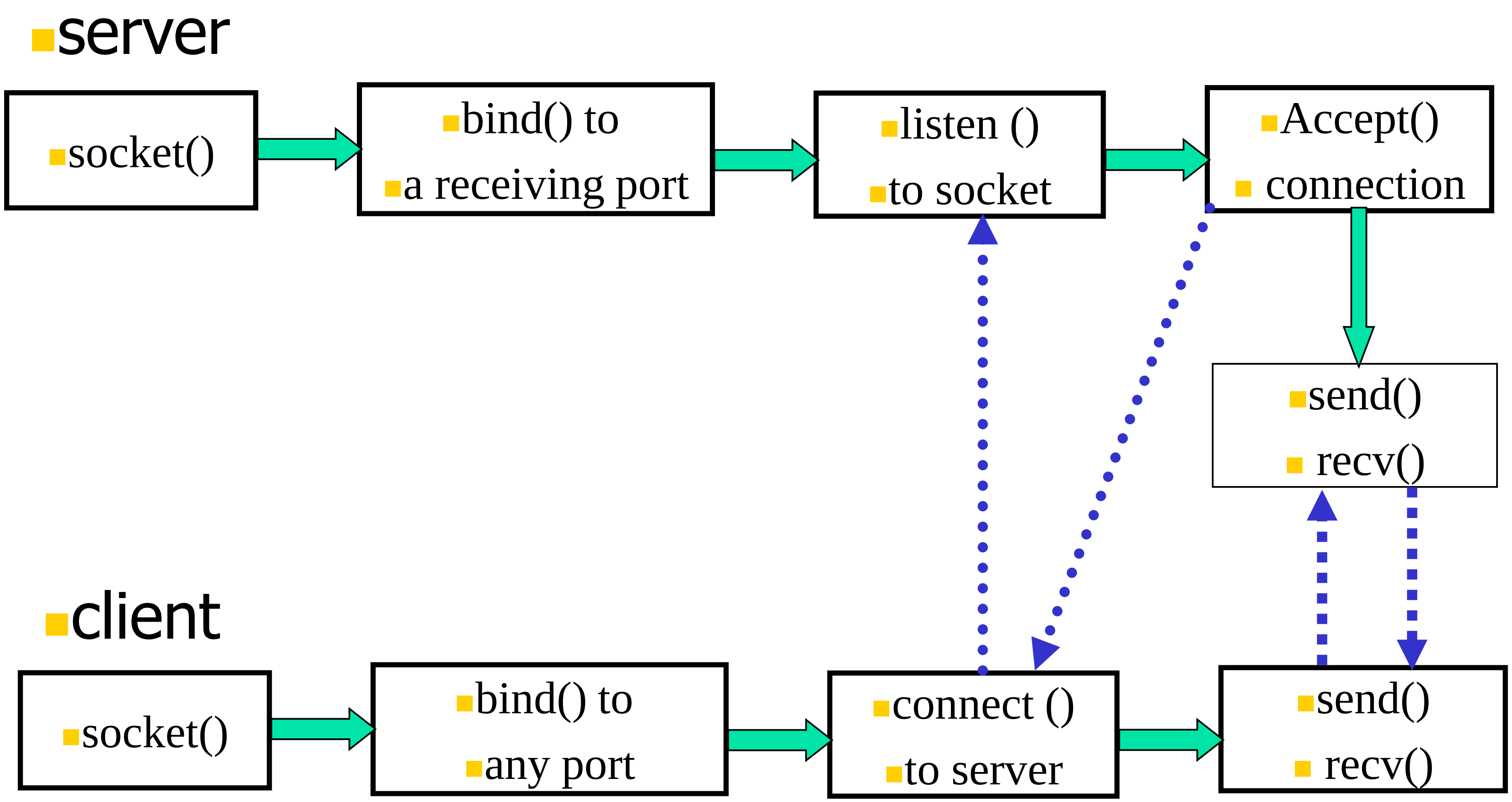
- If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket so it can continue to communicate the client.



Client- Server communication (UDP)



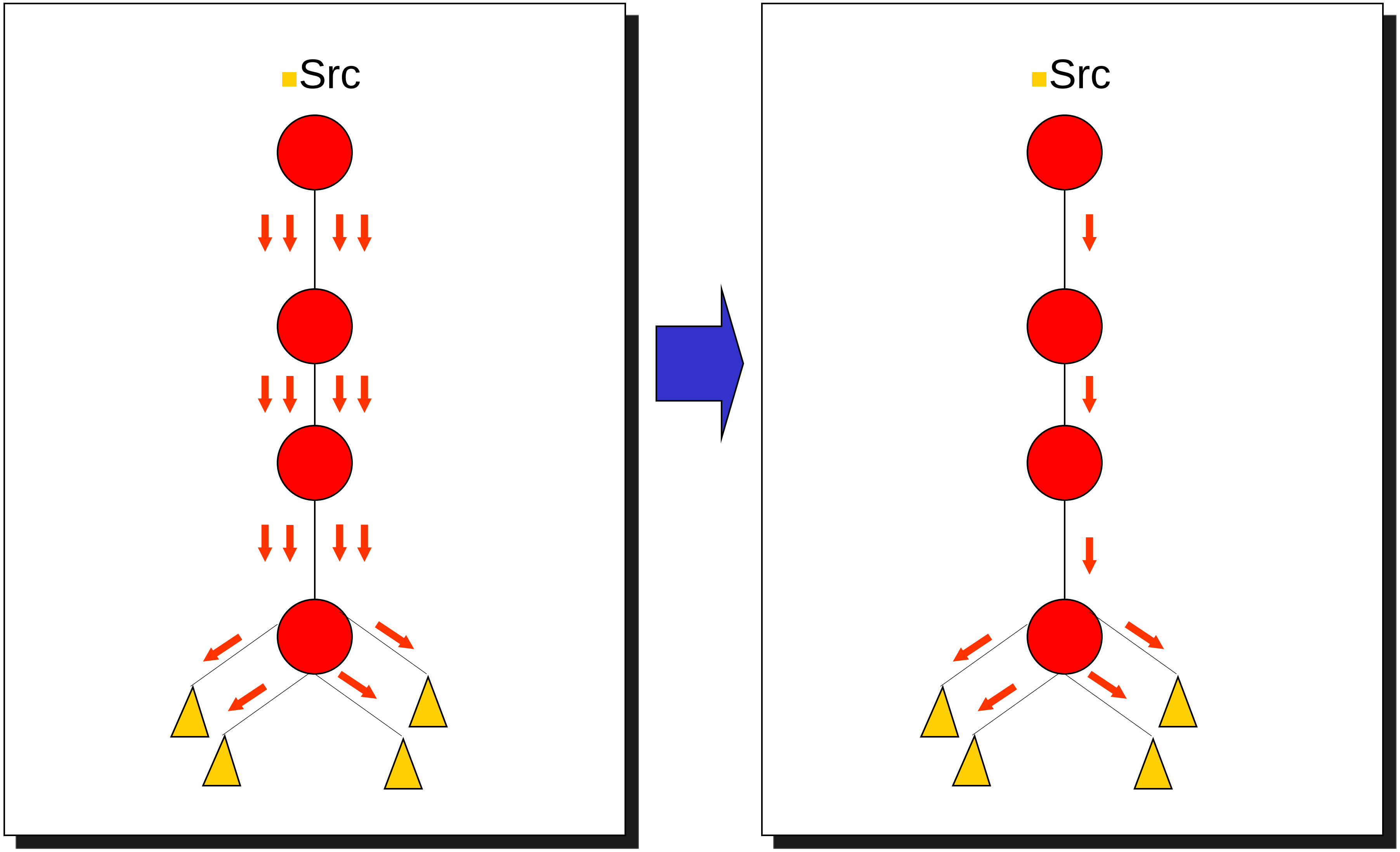
Client- Server communication (TCP)



TCP Stream Communication

- Use of TCP
  - Many services that run over TCP connections, with reserved port number are:
    - HTTP (Hypertext Transfer Protocol)
    - FTP (File Transfer Protocol)
    - Telnet
    - SMTP (Simple Mail Transfer Protocol)

Multicast = Efficient Data Distribution



Why Not Broadcast or Unicast?

- Broadcast:
  - Send a copy to every machine on the net
  - Simple, but inefficient
    - All nodes must process packet even if they don’t care
      - Wastes *more* CPU cycles of slower machines
    - Network loops lead to “*broadcast storms*”
- Replicated Unicast:
  - Sender sends a copy to each receiver in turn
  - Receivers need to register or sender must be pre-configured
  - Reliability => per-receiver state, separate sessions/processes at sender

IP Multicast model: RFC 1112

- Message sent to multicast "group" (of receivers)
  - *Senders need not be group members*
  - A group identified by a single "group address"
    - Use "group address" instead of destination address in IP packet sent to group
  - Groups can have any size;
  - Group members can be located anywhere on the Internet
  - Group membership is *not explicitly known*
  - Receivers can join/leave at will

IP Multicast Addresses

- Class D IP addresses
  - 224.0.0.0 – 239.255.255.255

Group ID

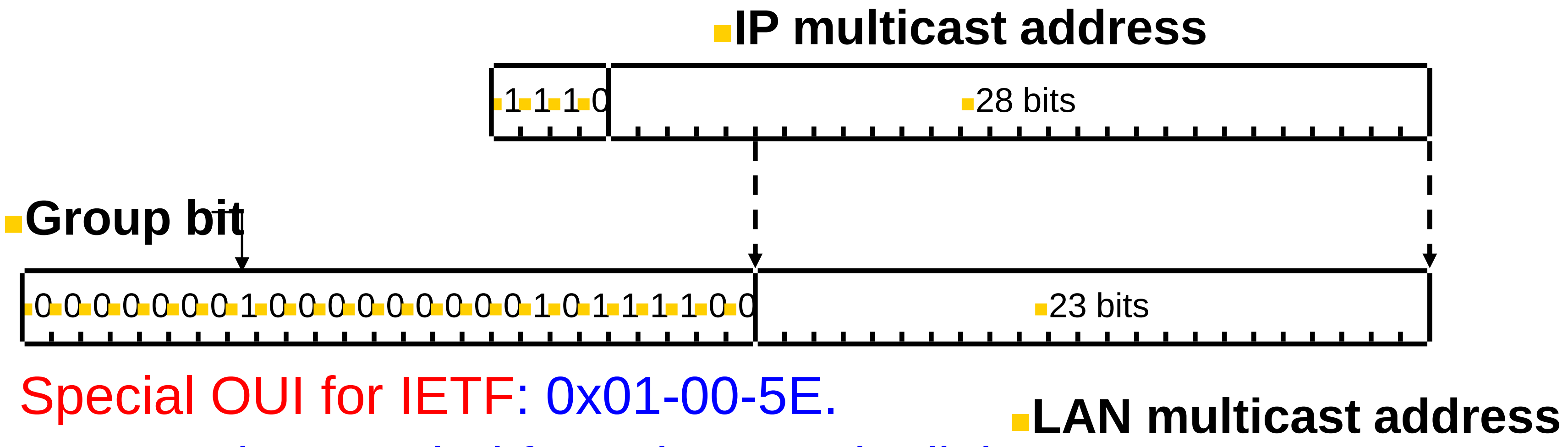
- Address allocation:
  - Well-known (reserved) multicast addresses, 224.0.0.x and 224.0.1.x
  - Each multicast address represents a *group of arbitrary size, called a "host group"*
- There is no structure within class D address space like subnetting => flat address space

IP Multicast Service — Receiving

- Two new operations
  - Join-IP-Multicast-Group
  - Leave-IP-Multicast-Group
- Receive multicast packets for joined groups via normal IP-Receive operation

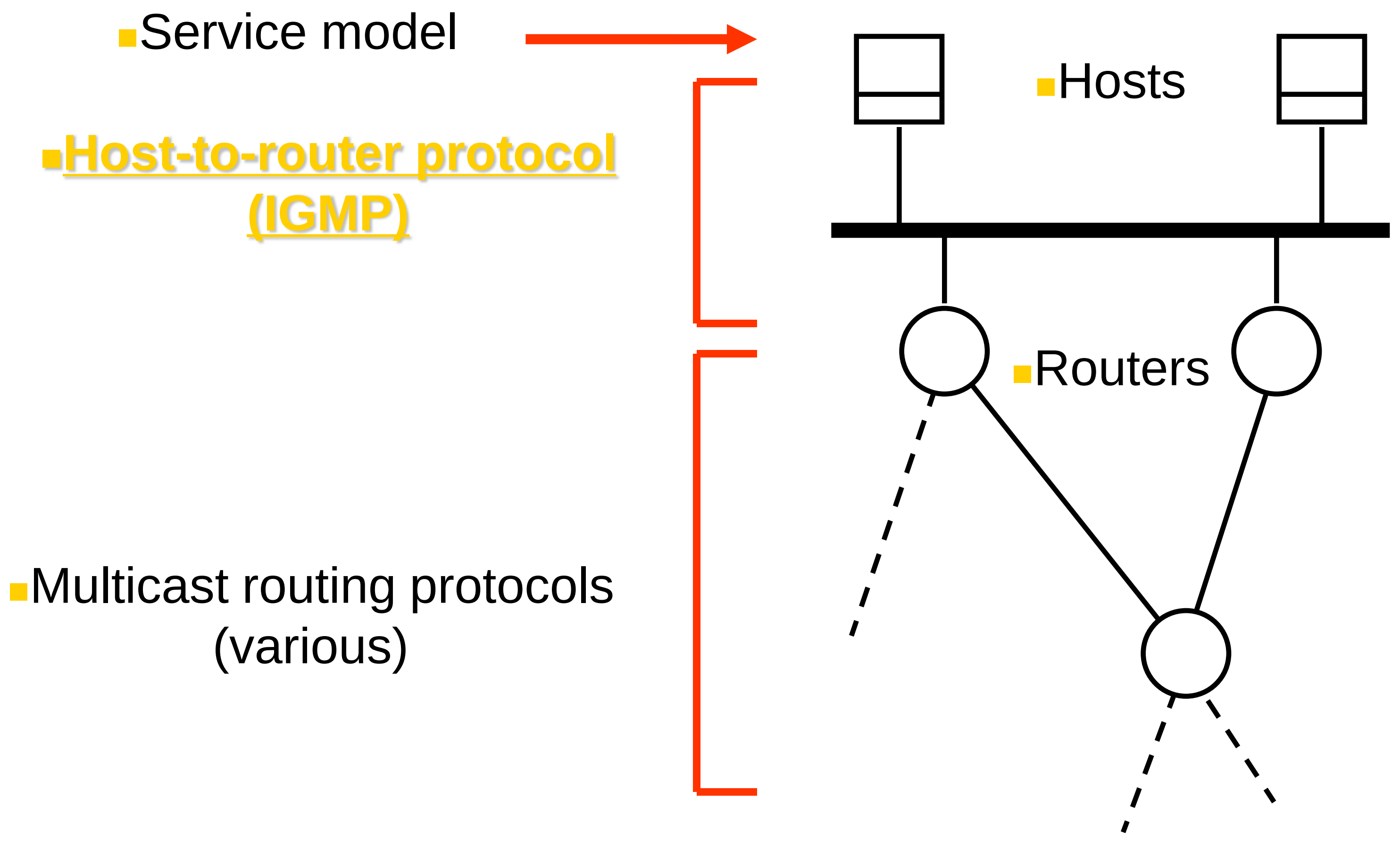
Using Link-Layer Multicast Addresses

- Ethernet and other LANs using 802 addresses:
  - Direct mapping! Simpler than unicast! No ARP etc.
  - 32 class D addrs may map to one MAC addr



- Special OUI for IETF: 0x01-00-5E.
- No mapping needed for point-to-point links

IP Multicast Architecture



Service model
Host-to-router protocol (IGMP)
Multicast routing protocols (various)
Hosts
Routers

External Data Representation

- The information stored in running programs is represented as data structures, whereas the information in messages consists of sequences of bytes.

- Irrespective of the form of communication used, the data structure must be converted to a sequence of bytes before transmission and rebuilt on arrival.

External Data Representation

- Marshalling
  - Marshalling is the process of taking a collection of data items and assembling them into a form suitable for transmission in a message.
- Unmarshalling
  - Unmarshalling is the process of disassembling a collection of data on arrival to produce an equivalent collection of data items at the destination.

External Data Representation

- Three approaches to external data representation and marshalling are:
    - CORBA
    - Java’s object serialization
    - XML

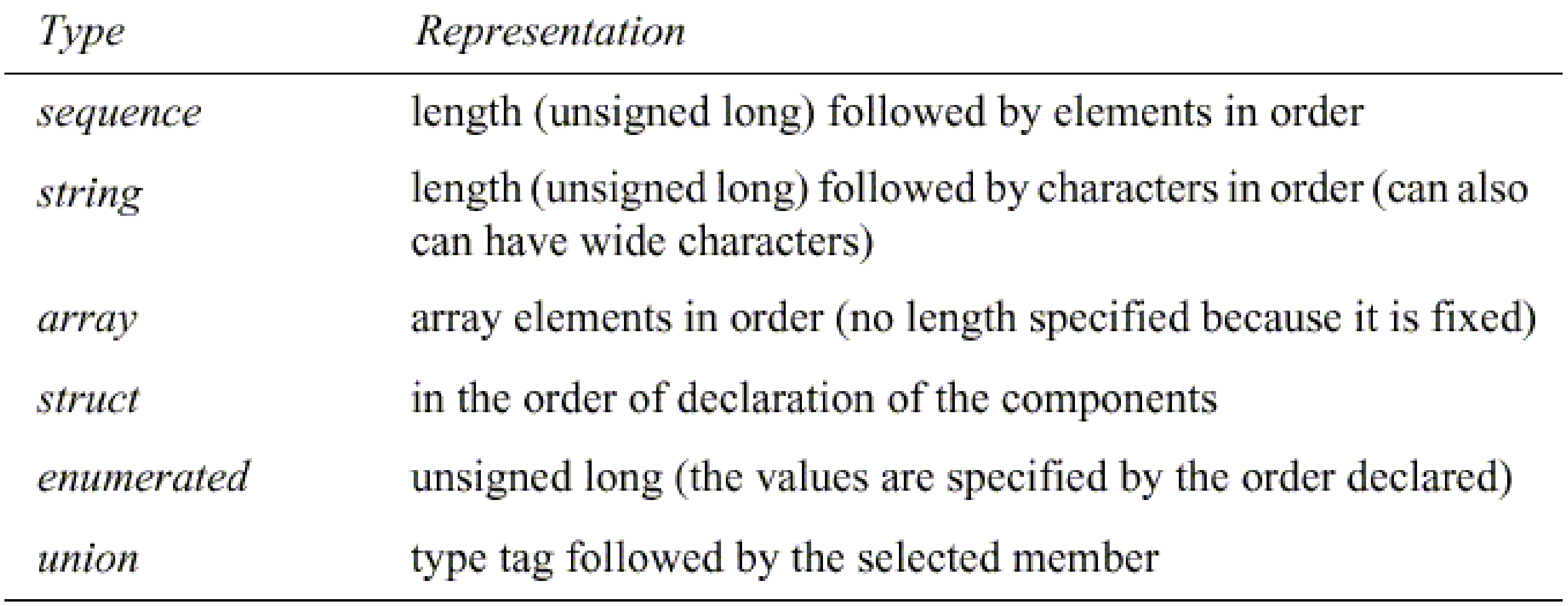
External Data Representation

- Marshalling and unmarshalling activities is usually performed automatically by middleware layer.

CORBA Common Data Representation (CDR)

- CORBA Common Data Representation (CDR)
  - CORBA CDR is the external data representation defined with CORBA 2.0.
  - It consists 15 primitive types:
    - Short (16 bit)
    - Long (32 bit)
    - Unsigned short
    - Unsigned long
    - Float(32 bit)
    - Double(64 bit)
    - Char
    - Boolean(TRUE,FALSE)
    - Octet(8 bit)
  - Composite type are shown in Figure 8.

CORBA Common Data Representation (CDR)



Type	*Representation*
sequence	length (unsigned long) followed by elements in order
string	length (unsigned long) followed by characters in order (can also can have wide characters)
array	array elements in order (no length specified because it is fixed)
struct	in the order of declaration of the components
enumerated	unsigned long (the values are specified by the order declared)
union	type tag followed by the selected member

- **CORBA CDR for constructed types**

Client-Server Communication

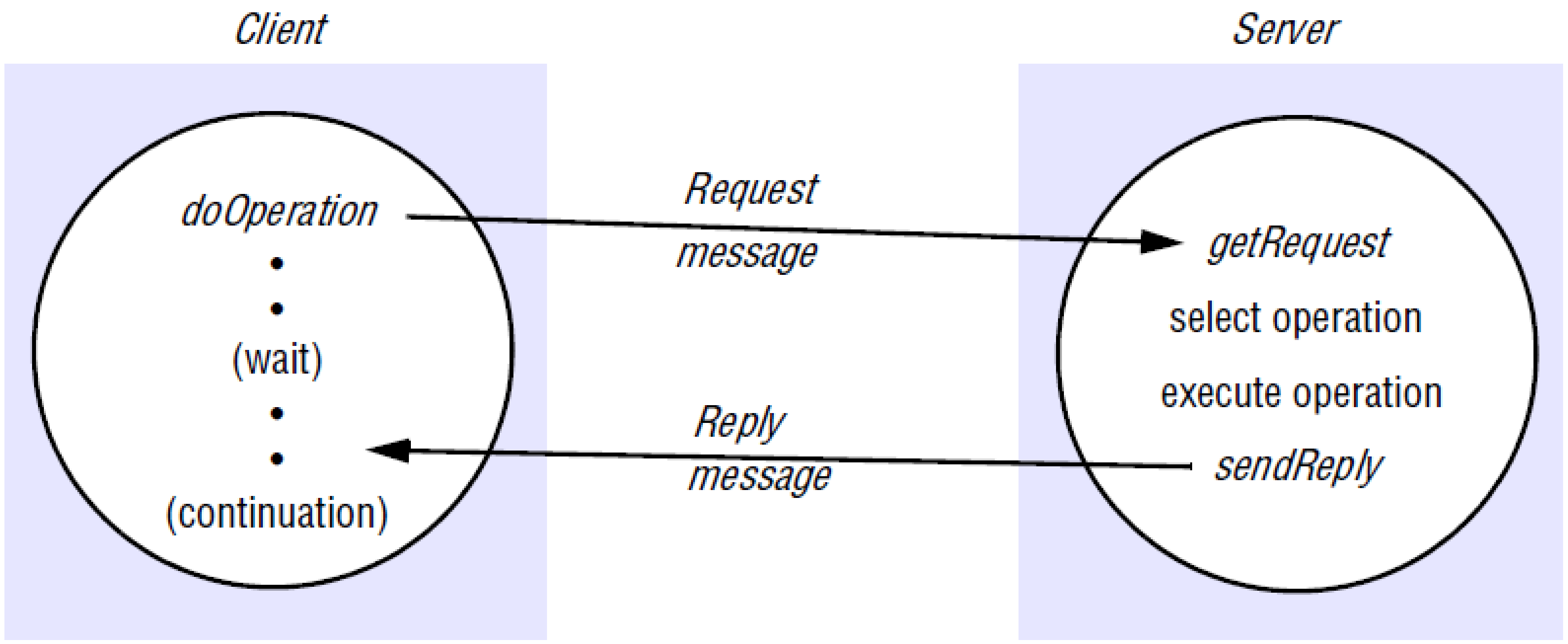
- The client-server communication is designed to support the roles and message exchanges in typical client-server interactions.
- In the normal case, request-reply communication is synchronous because the client process blocks until the reply arrives from the server.
- Asynchronous request-reply communication is an alternative that is useful where clients can afford to retrieve replies later.

Client-Server Communication

- It is better this model is implemented over UDP datagrams
  - Client-server protocol consists of request/response pairs, hence no acknowledgements at transport layer are necessary
  - Avoidance of connection establishment overhead
  - No need for flow control due to small amounts of data are transferred

Client-Server Communication

- The request-reply protocol was based on a trio of communication primitives: doOperation, getRequest, and sendReply shown in the followin

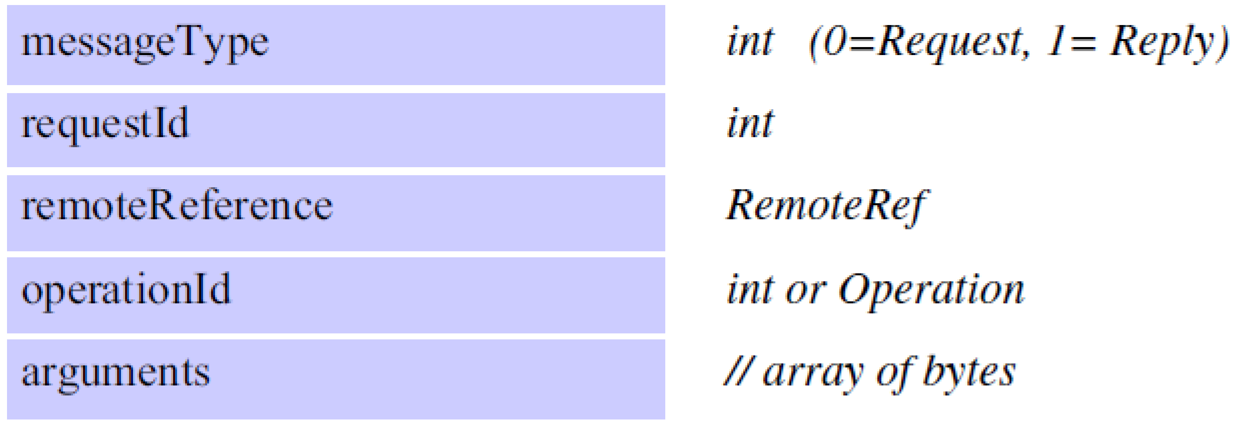


Client-Server Communication

- The designed request-reply protocol matches requests to replies.
- If UDP datagrams are used, the delivery guarantees must be provided by the request-reply protocol, which may use the server reply message as an acknowledgement of the client request message

Client-Server Communication

- The information to be transmitted in a request message or a reply message is shown in the following figure



messageType	*int (0=Request, 1= Reply)*
requestId	*int*
remoteReference	*RemoteRef*
operationId	*int or Operation*
arguments	*// array of bytes*

- **Request-reply message structure**

Client-Server Communication

- In a protocol message
  - The first field indicates whether the message is a request or a reply message.
  - The second field request id contains a message identifier.
  - The third field is a remote object reference .
  - The forth field is an identifier for the method to be invoked.

Client-Server Communication

- Message identifier
  - A message identifier consists of two parts:
    - A requestId, which is taken from an increasing sequence of integers by the sending process
    - An identifier for the sender process, for example its port and Internet address.

Client-Server Communication

- Failure model of the request-reply protocol
  - If the three primitive doOperation, getRequest, and sendReply are implemented over UDP datagram, they have the same communication failures.
    - Omission failure
    - Messages are not guaranteed to be delivered in sender order.

Remote Procedure Call

- the style of programming promoted by RPC – programming with interfaces;
- the call semantics associated with RPC;
- the key issue of transparency and how it relates to remote procedure calls

Programming With Interfaces

- Most programming languages organize a program as a set of module
- Communication between modules can be by means of
  - Procedure call between them
  - Direct access to the their variables
- Interface: it specifies the procedure and the variables that can be accessed from other modules
- Modules are implemented so as to hide all the information about them except that which is available through its interface
  - The implementation of a module may be changed without affecting the users of the module

Interface in distributed systems

- In a distributed program, the module can run in separate process.
- Service Interface
  - The specification of the procedures offered by a server
  - Defining the types of the arguments of each of the procedures

benefits of interface

- Programmers are concerned only with the abstraction offered by the service interface
- Programmers do not
  - Need be aware of implementation details
  - Need to know the programming language
  - Need to know underlying platform used to implement the service

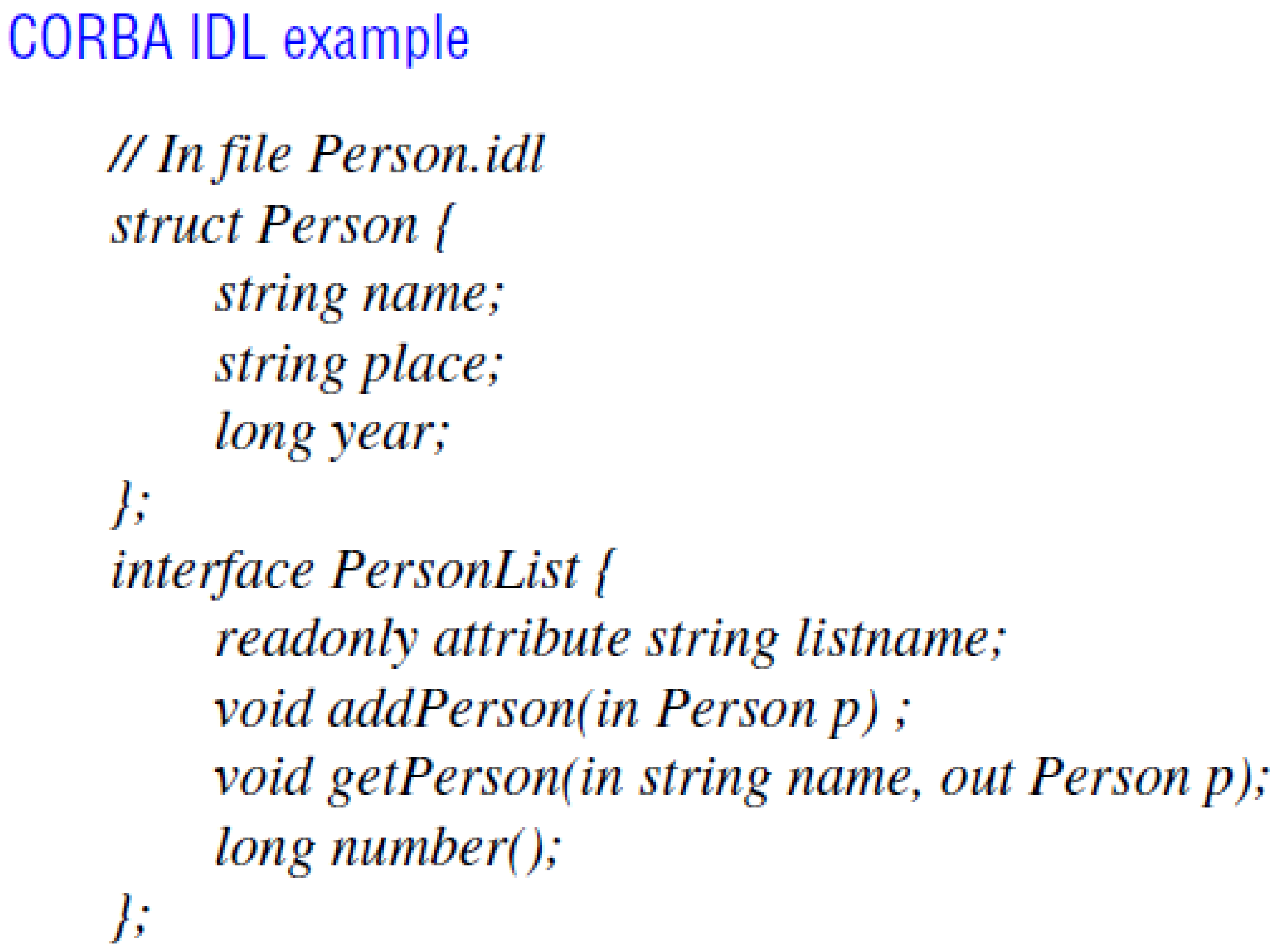
Influence of DS to SI

- Client can not access to the variables in a module in another process
- Call by reference is not supported
  - Addresses in on process are not valid in another remote one
- The parameters in the interface in DS are specified by input, output or both

Interface Definition Language(IDL)

- An RPC mechanism can be integrated with a particular programming language
  - if it includes an adequate notation for defining interfaces
  - allowing input and output parameters to be mapped onto the language’s normal use of parameters
- *Interface definition languages (IDLs) are designed to* allow procedures implemented in different languages to invoke one another
- An IDL provides a notation for defining interfaces in which each of the parameters of an operation may be described as for input or output in addition to having its type specified

Simple Example of CORBA IDL



CORBA IDL example

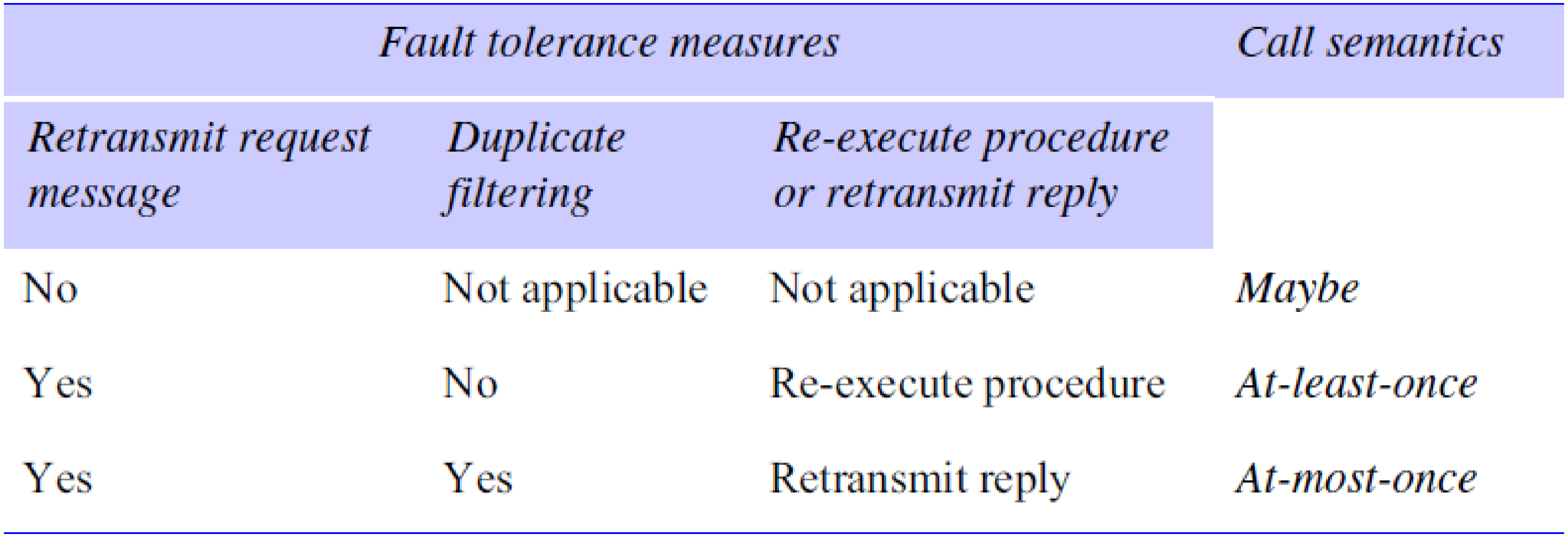
```
// In file Person.idl
struct Person {
        string name;
        string place;
        long year;
};
interface PersonList {
        readonly attribute string listname;
        void addPerson(in Person p) ;
        void getPerson(in string name, out Person p);
        long number();
};
```

RPC Call Semantics

- Request-replay protocol can be implemented in different ways to provide different delivery guarantees
  - Retry request message
    - Controls whether to retransmit the request message until either a reply is received or the server is assumed to have failed
  - Duplicate filtering
    - Controls when retransmissions are used and whether to filter out duplicate requests at the server
  - Retransmission of the results
    - Controls whether to keep a history of result messages to enable lost results to be retransmitted without re-executing the operations at the server

Call Semantic

- Combinations of these choice lead to a variety of possible semantic for the reliability of remote invocations as seen by the invoker



Fault tolerance measures			*Call semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

Call Semantics

- Maybe call semantics
  - After a RPC time-out (or a client crashed and restarted), the client is not sure if the RP may or may not have been called.
  - This is the case when no fault tolerance is built into RPC mechanism.
  - Clearly, maybe semantics is not desirable.

Call Semantics

- At-least-once call semantics
  - With this call semantics, the client can assume that the RP is executed at least once (on return from the RP).
  - Can be implemented by retransmission of the (call) request message on time-out.

Call Semantics

- At-most-once call semantics
  - When a RPC returns, it can assumed that the remote procedure (RP) has been called exactly once or not at all.
  - Implemented by the server's filtering of duplicate requests (which are caused by retransmissions due to IPC failure, slow or crashed server) and caching of replies (in reply history, refer to RRA protocol).

RPC Mechanism

- How does the client know the procedure (names) it can call and which parameters it should provide from the server?
- Server interface definition
  - RPC interface specifies those characteristics of the procedures provided by a server that are visible to the clients.
  - The characteristics includes: names of the procedures and type of parameters.
  - Each parameter is defined as input or output.

RPC Mechanism

- In summary, an interface contains a list of procedure signatures - the names and types of their I/O arguments (to be discussed later).
- This interface is made known to the clients through a server process binder (to be discussed later).

RPC Mechanism

- How does the client transfer its call request (the procedure name) and the arguments to the server via network?
- Marshalling and communication with server:
  - For each remote procedure call, a (client) stub procedure is generated and attached to the (client) program.
  - Replace the remote procedure call to a (local) call to the stub procedure.

RPC Mechanism

- The (codes in the) stub procedure marshals (the input) arguments and places them into a message together with the procedure identifier (of the remote procedure).

- Use IPC primitive to send the (call request) message to the server and wait the reply (call return) message (DoOperation).

RPC Mechanism

- How does the server react the request of the client? From which port? How to select the procedure? How to interpret the arguments?
- Dispatching, Un-marshalling, communication with client:
  - A dispatcher is provided. It receives the call request message from the client and uses the procedure identifier in the message to select one of the server stub procedures and passes on the arguments.

RPC Mechanism

- For each procedure at the server which is declared (at the sever interface) as callable remotely, a (server) stub procedure is generated.
- The task of a server stub procedure is to un-marshal the arguments, call the corresponding (local) service procedure.

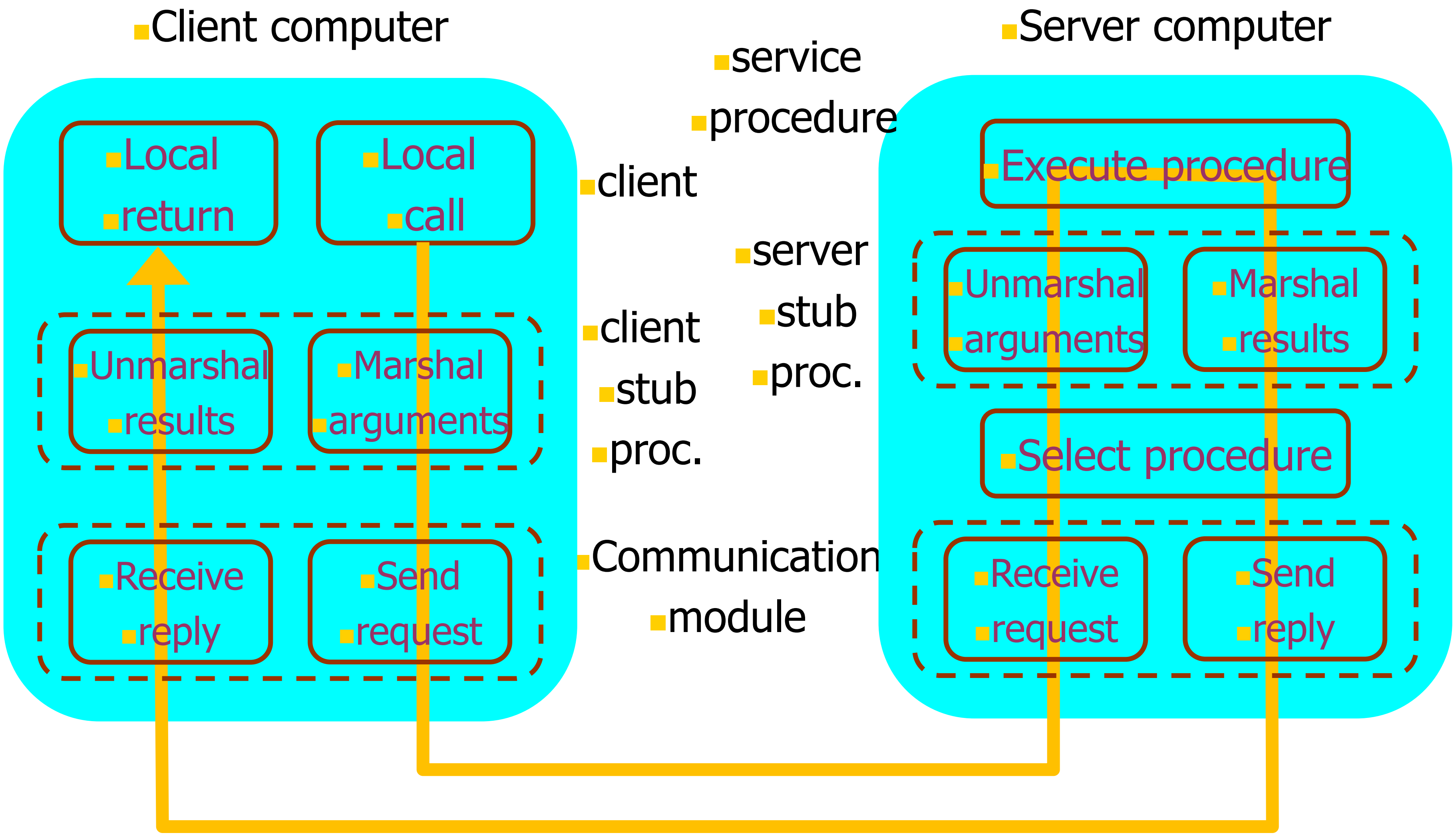
RPC Mechanism

- How does the server transmit the reply back?
- On return, the stub marshals the output arguments into a reply (call return) message and sends it back to the client.

RPC Mechanism

- How does the client receive the reply?
- The stub procedure of the client unmarshals the result arguments and returns (local call return). Note that the original remote procedure call was transformed into a (local) call to the stub procedure.

RPC Mechanism



Case Studies: SUN RPC

- Interface definition language: XDR
  - a standard way of encoding data in a portable fashion between different systems;
- Interface compiler: rpcgen
  - A compiler that takes the definition of a remote procedure interface, and generates the client stubs and the server stubs;
- Communication handling: TCP or UDP
- Version: RPCSRC 3.9 (4.3BSD UNIX)
  - A run-time library to handle all the details.

Case Studies: SUN RPC

- Most languages allow interface names to be specified, but Sun RPC does not
- instead of this, a program number and a version number are supplied
  - The program numbers can be obtained from a central authority to allow every program to have its own unique number
  - The version number is intended to be changed when a procedure signature changes
  - Both program and version number are passed in the request message

Case Studies: SUN RPC

- **A procedure definition specifies a procedure signature and a procedure number.** The procedure number is used as a procedure identifier in request messages.
  - The procedure signature consists of the result type, the name of the procedure and the type of the input parameter
- **Only a single input parameter is allowed. Therefore, procedures requiring** multiple parameters must include them as components of a single structure.
- **The output parameters of a procedure are returned via a single result.**

Case Studies: SUN RPC

Files interface in Sun XDR

```
const MAX = 1000;
typedef int FileIdentifier;
typedef int FilePointer;
typedef int Length;
struct Data {
        int length;
        char buffer[MAX];
};
struct writeargs {
        FileIdentifier f;
        FilePointer position;
        Data data;
};

struct readargs {
        FileIdentifier f;
        FilePointer position;
        Length length;
};

program FILEREADWRITE {
        version VERSION {
                void WRITE(writeargs)=1;
                Data READ(readargs)=2;
        }=2;
} = 9999;
```