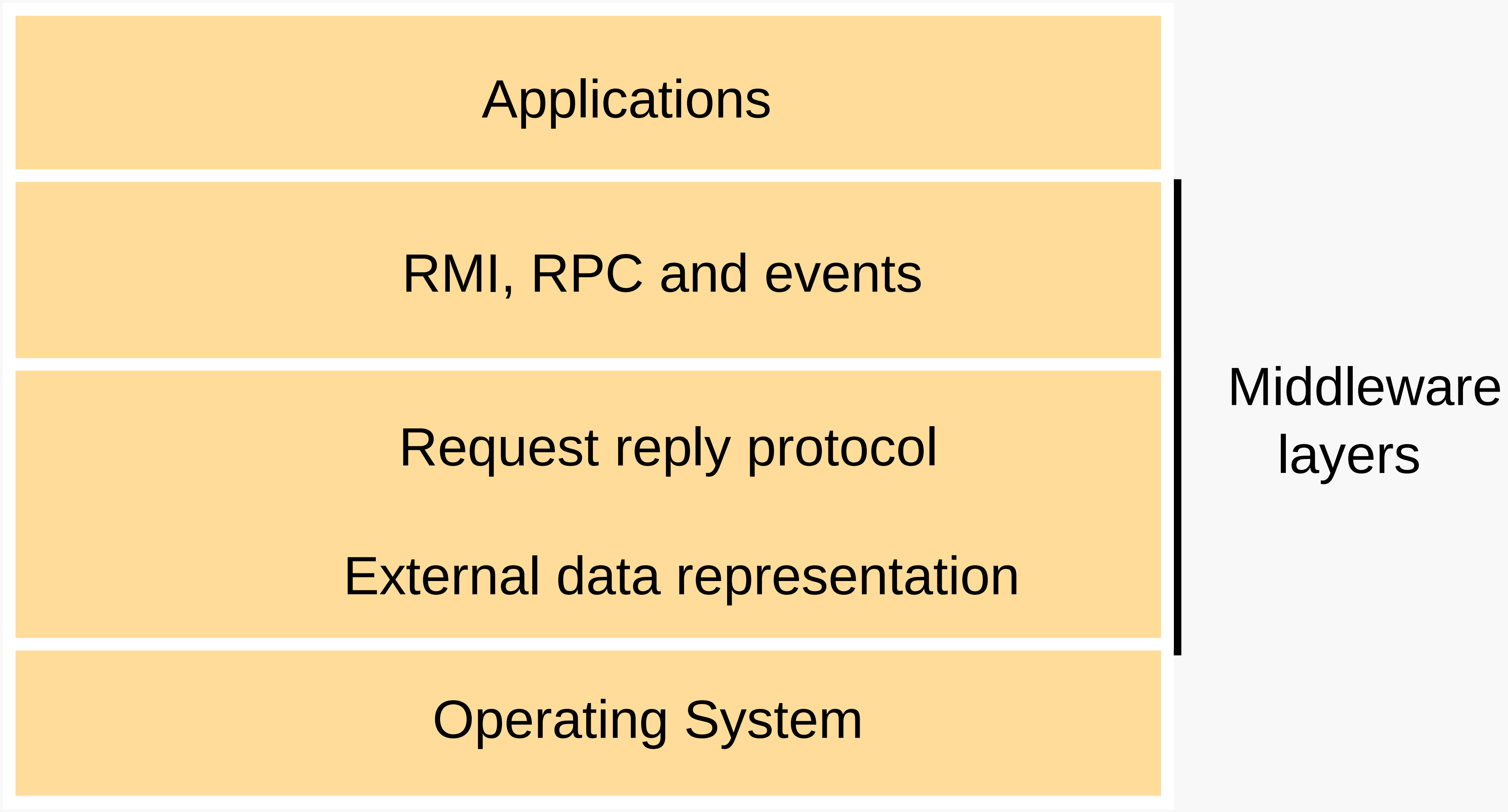
REMOTE PROCEDURE CALL


Applications
RMI, RPC and events
Request reply protocol
External data representation
Operating System
Middleware layers

Distributed Programming Model

Remote procedure call (RPC)	*call procedure in separate process*
Remote method invocation (RMI)	*extension of local method invocation in OO model* *invoke the methods of an object of another process*
Event-based model	*Register interested events of other objects* *Receive notification of the events at other objects*

Interfaces

- Interface
  - *The interface of a module specifies accessible procedures and variables that can be accessed from other modules*
  - *Inner alteration won’t affect the user of the interface*
- Interface in distributed system
  - *Can’t access variables directly from another process.*
  - *Input argument and output argument*
  - *Pointers can’t be passed as arguments or returned results*

Interface cases

- RPC’s Service interface
  - *specification of the procedures of the server, defining the types of the input and output arguments of each procedure*
- RMI’s Remote interface
  - *Specification of the methods of an object that are available for objects in other processes, defining the types of them.*
  - *may pass objects or remote object references as arguments or returned result*
- Interface definition languages
  - *program language, e.g. Java RMI*
  - *Interface definition languages (IDLs), are designed to allow objects implemented in different languages to invoke one another.*
- e.g. CORBA IDL (n1), DCE IDL and DCOM IDL

Interface Definition Language

These constraints have a significant impact on the specification of interface definition languages,

- *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.

CORBA IDL example

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

CORBA has a struct

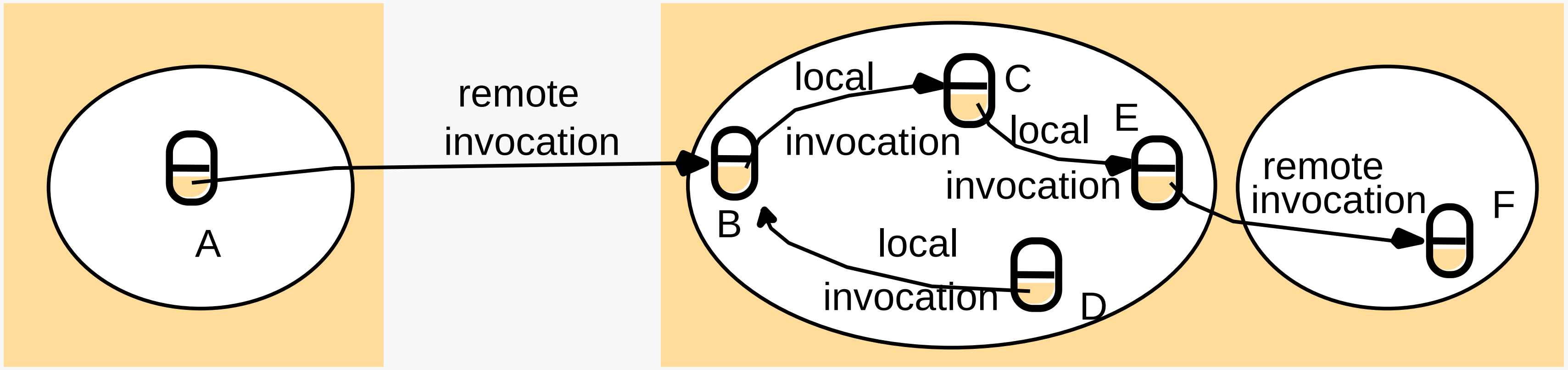
remote interface

remote interface defines methods for RMI

parameters are *in*, *out* or *inout*

- Remote interface:
  - *IDL is allowing objects implemented in different languages to invoke one another.*
  - *specifies the methods of an object available for remote invocation*
  - *an interface definition language (or IDL) is used to specify remote interfaces. E.g. the above in CORBA IDL.*
  - *Java RMI would have a class for Person, but CORBA has a struct*

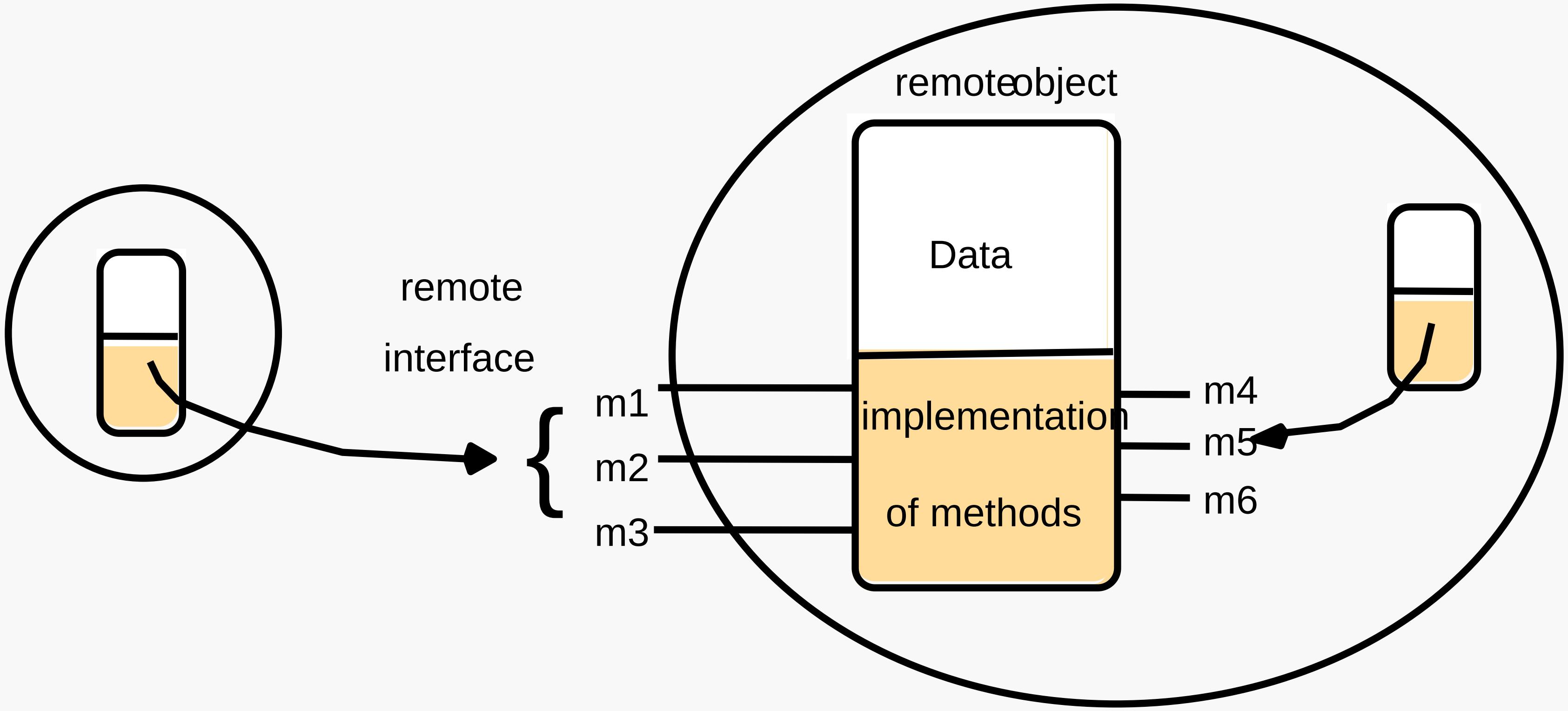
Distributed object model



- each process contains objects, some of which can receive remote invocations, others only local invocations.
- Method invocations between objects in different processes are known as remote method invocations, otherwise local method invocations.
- objects that can receive remote invocations are called *remote objects, e.g. B & F*
- objects need to know the *remote object reference* of an object in another process in order to invoke its methods. E.g. C must have a reference to E
- the *remote interface* specifies the methods of and object that are available for invocation by remote objects in other processes. E.g. B and F must have remote interfaces

Distributed Object Model cont….

- Two fundamental concepts:
  - *Remote object reference: other objects can invoke the methods of a remote object if they have access to its remote object reference.*
  - *Remote interface: Every object has a remote interface in which methods can be invoked remotely.*



Distributed Object Model Cont….

- Actions in DS Object Systems:
  - *When an action leads to the instantiation of a new object, that object is normally live within the process where instantiation is requested.*
- Garbage Collection:
  - *Achieved by local garbage collectors and added module that carries out a form of distributed garbage collection.*
- Exceptions:
  - *The process containing the remote object may have crashed or may be too busy to reply, or the invocation or result message may be lost.*

Remote Procedure Calls

- Procedures on remote machines can be called as if they are procedures in the local address space.

- The underlying RPC system then hides important aspects of distribution, including the encoding and decoding of parameters and results, the passing of messages and the preserving of the required semantics for the procedure call.

Design Issues

- Style of programming– programming with interfaces;
- Call Semantics;
- Key issue of transparency and how it relates to remote procedure calls.

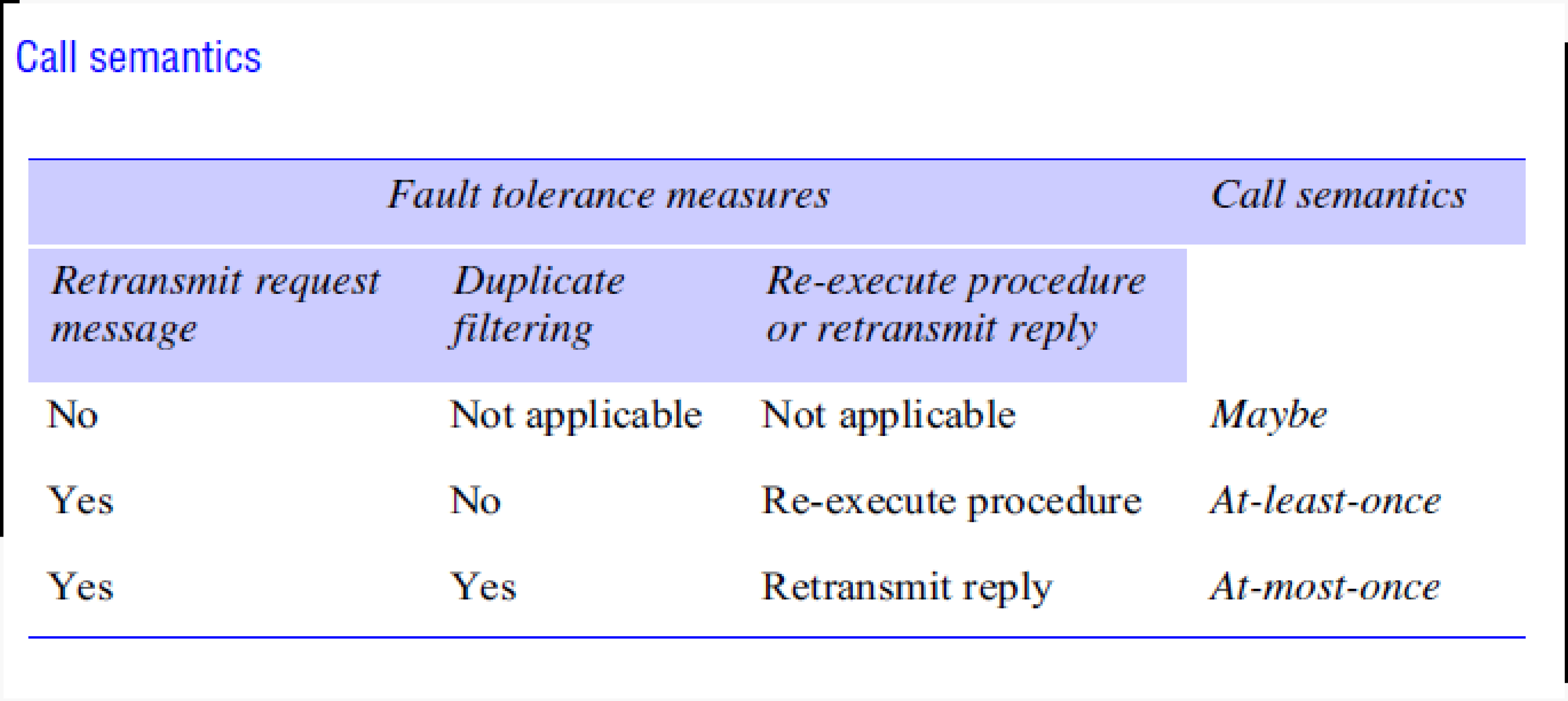
Distributed RPC

The definition of service interfaces is influenced by the distributed nature of theunderlying infrastructure:

- It is not possible for a client module running in one process to access the variables in a module in another process.

- The parameter-passing mechanisms used in local procedure calls – for example, call by value and call by reference not suitable

- Another difference between local and remote modules is that addresses in one process are not valid in another remote one



Call semantics

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

- *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 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.

May be Semantics

- With *maybe* semantics, the remote procedure call may be executed once or not at all.
- Maybe semantics arises when no fault-tolerance measures are applied and can suffer from the following types of failure:
  - ***omission failures if the request or result message is lost;***
  - ***crash failures when the server containing the remote operation fails.***
- *Maybe* semantics is useful only for applications in which occasional failed calls are acceptable.

At-least-once Semantics

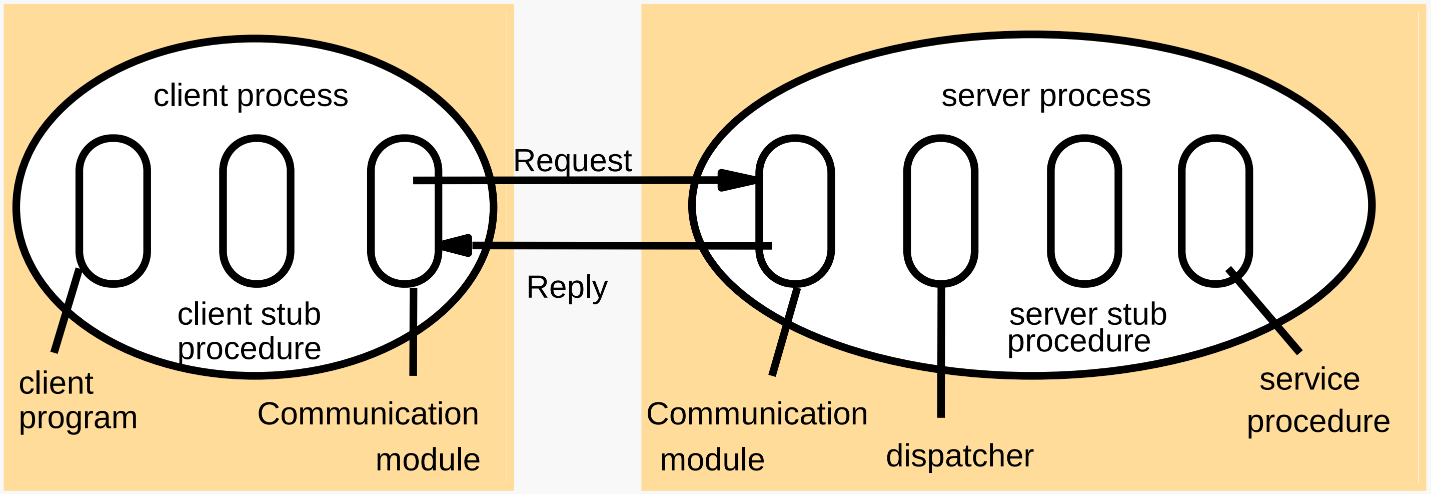
- With *at-least-once* semantics, the invoker receives either a result, in which case the invoker knows that the procedure was executed at least once, or an exception informing it that no result was received
- *At-least-once* semantics can be achieved by the **retransmission of request messages**, which masks the omission failures of the request or result message.
- *At-least-once* semantics can suffer from the following types of failure:
  - *crash failures when the server containing the remote procedure fails;*
  - *arbitrary failures – in cases when the request message is retransmitted, the remote server may receive it and execute the procedure more than once,* ***possibly causing wrong values to be stored or returned.***

At-most-once Semantics

- With *at-most-once* semantics, the caller receives either a result, in which case the caller knows that the procedure was executed exactly once, or an exception informing it that no result was received, in which case the procedure will have been executed either once or not at all.
- Mechanism: **retransmit until acknowledged, with duplicate suppression**
- Appropriate for non-idempotent operations

Implementation of RPC

- Building blocks
  - *Communication module*
  - *Client stub procedure (as proxy in RMI): marshalling, sending, unmarshalling*
  - *Dispatcher: select one of the server stub procedures*
  - *Server stub procedure: unmarshalling, calling, marshalling*



Steps take place during an RPC

- A client invokes a *client stub* procedure, passing parameters in the usual way. The client stub resides within the client's own address space.
- The client stub *marshals* the parameters into a message. Marshalling includes converting the representation of the parameters into a standard format and copying each parameter into the message.
- The client stub passes the message to the transport layer, which sends it to the remote server machine.
- On the server, the transport layer passes the message to a *server stub*, which unmarshals the parameters and calls the desired server routine using the regular procedure call mechanism.
- When the server procedure completes, it returns to the server stub (e.g., via a normal procedure call return), which marshals the return values into a message. The server stub then hands the message to the transport layer.
- The transport layer sends the result message back to the client transport layer, which hands the message back to the client stub.
- The client stub unmarshals the return parameters and execution returns to the caller.

Sun RPC case study

- Designed for NFS
  - *Uses at-least-once semantics.*
- The Sun RPC system provides an interface language called XDR and an interface compiler called *rpcgen*, which is intended for use with the C programming language.
- XDR - Interface definition language
  - *Interface name: Program number, version number*
    - Program number: from central authority to allow every program to have its own unique number.
    - Version number: changes when procedure signature changes.
    - Both are passed through the request message.
- A procedure definition specifies a procedure signature and a procedure number.
- The procedure number is used as a procedure identifier in request messages.

- 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.

- The procedure signature consists of the result type, the name of the procedure and the type of the input parameter.

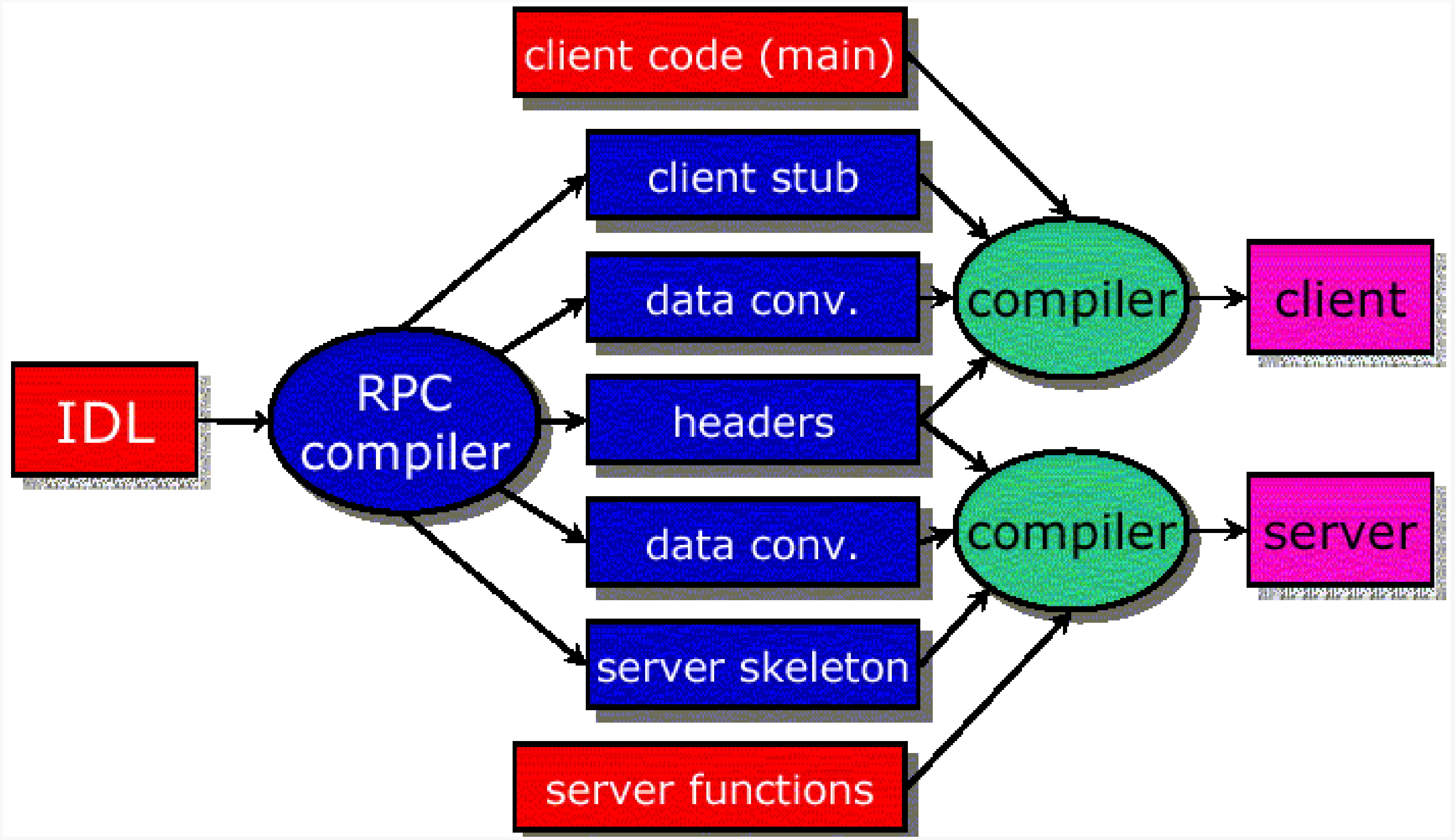
File Interface in SUN XDR

```
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; 1
    Data READ(readargs)=2;   2
  }=2; // ver no.
} = 9999; //Prg. No.
```

RPC Interface Compiler



- *Rpcgen* – The interface compiler **rpcgen** can be used to generate the following from interface definition.
  - ***client stub*** *procedures*
  - ***server main*** *procedure, dispatcher and server stub procedures*
  - ***XDR marshalling and unmarshalling*** *procedures used by dispatcher and client, server stub procedures.*

- *Binding*- Local binding service called the PortMapper.
  - *Each instance of port mapper records the prg. No., ver. No. and port no. in use by each service locally.*
  - *The port mapper program maps Remote Procedure Call (RPC) program and version numbers to transport-specific port numbers.*
  - *The port mapper program makes dynamic binding of remote programs possible.*
  - *Server: register ((program number, version number), port number)*
  - *Client: request port number by (program number, version number)*

- ***Authentication***
  - *Each request contains the credentials of the user, e.g. uid and gid of the user*
  - *Access control mechanism can be built on server.*
    - according to the credential information will get access to procedures.

Implementing RPC

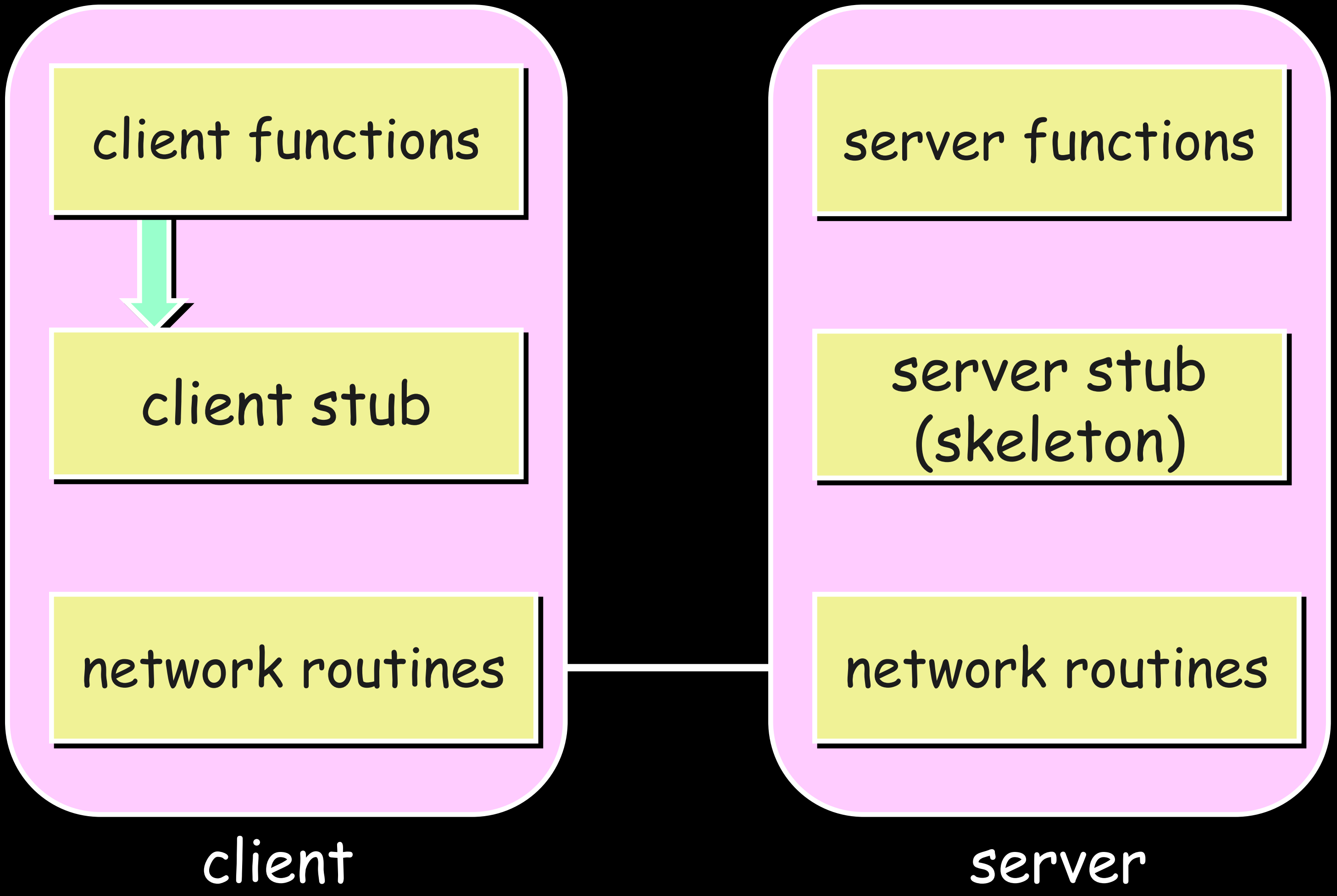
The trick:

Create **stub functions** to make it appear to the user that the call is local

Stub function contains the function's interface

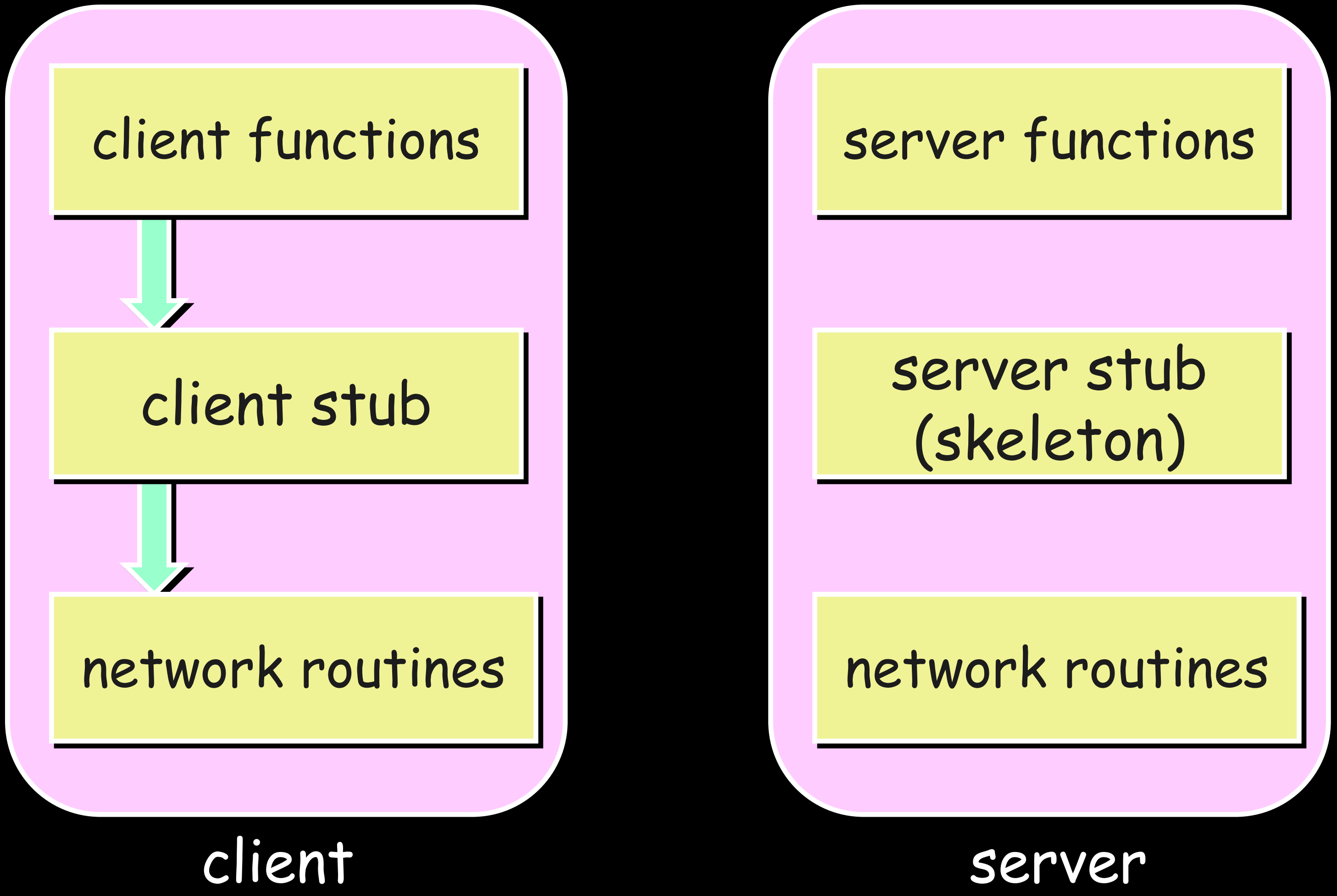
Stub functions

1. Client calls stub (params on stack)



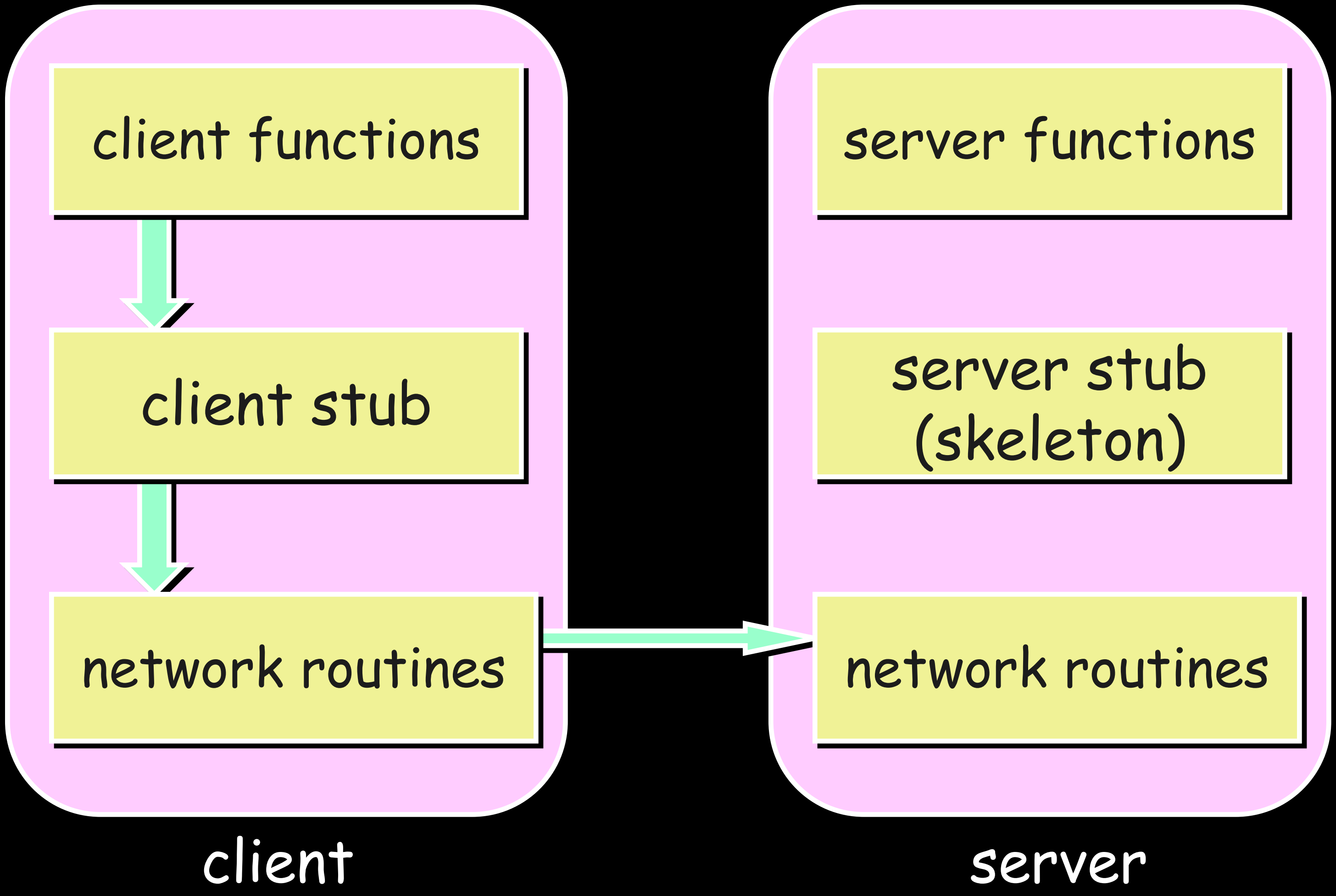
Stub functions

2. Stub marshals params to net message



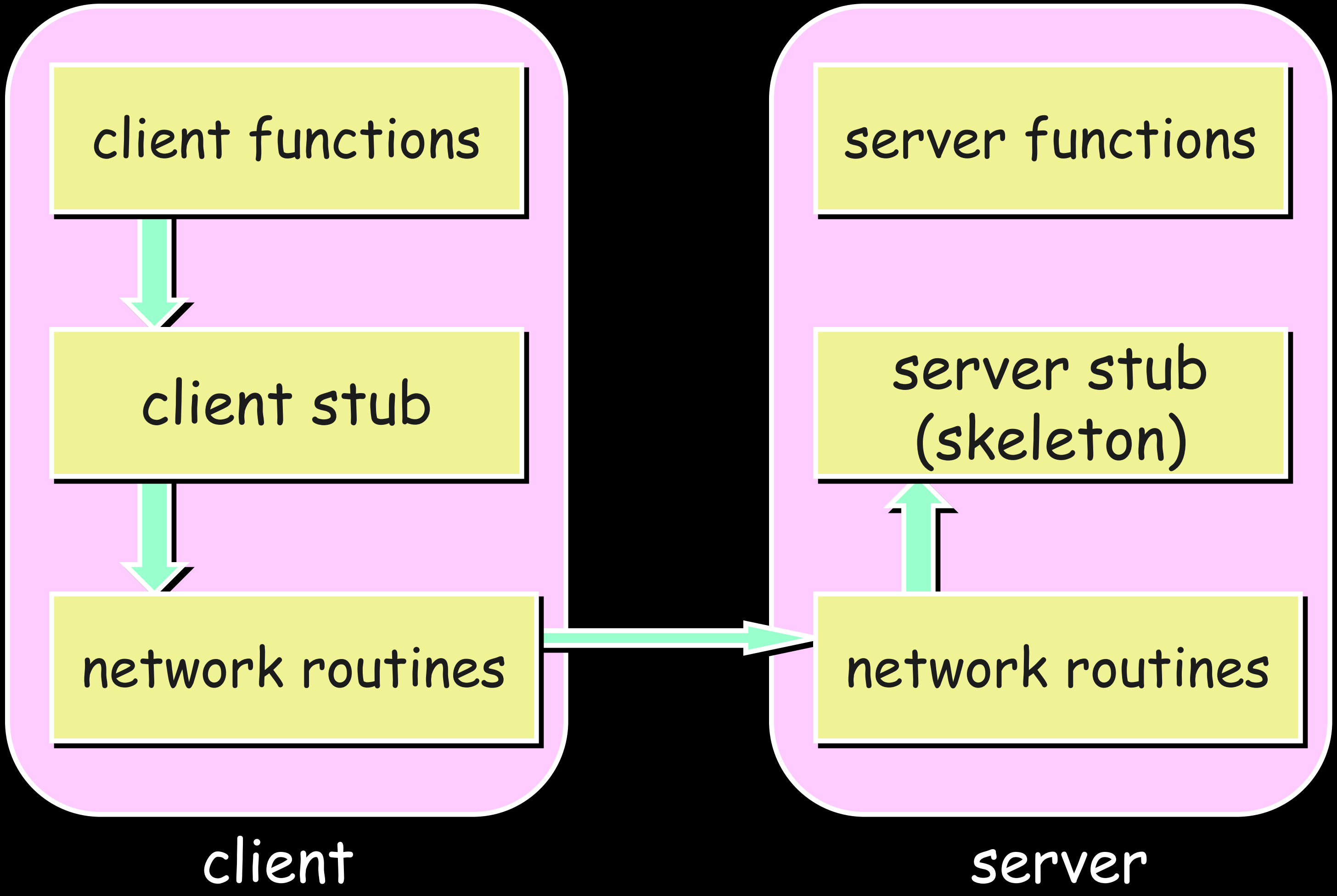
Stub functions

3. Network message sent to server



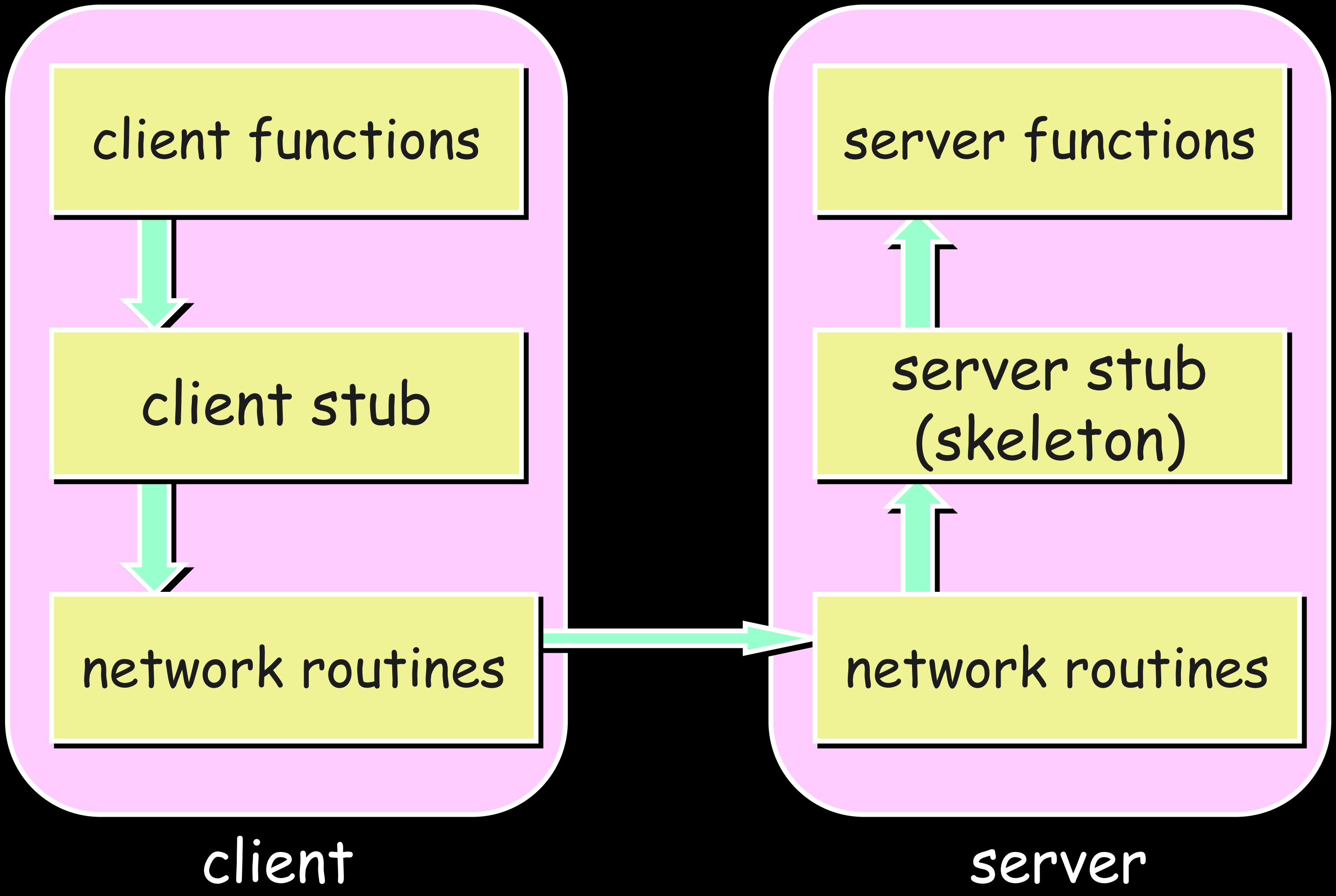
Stub functions

4. Receive message: send to stub



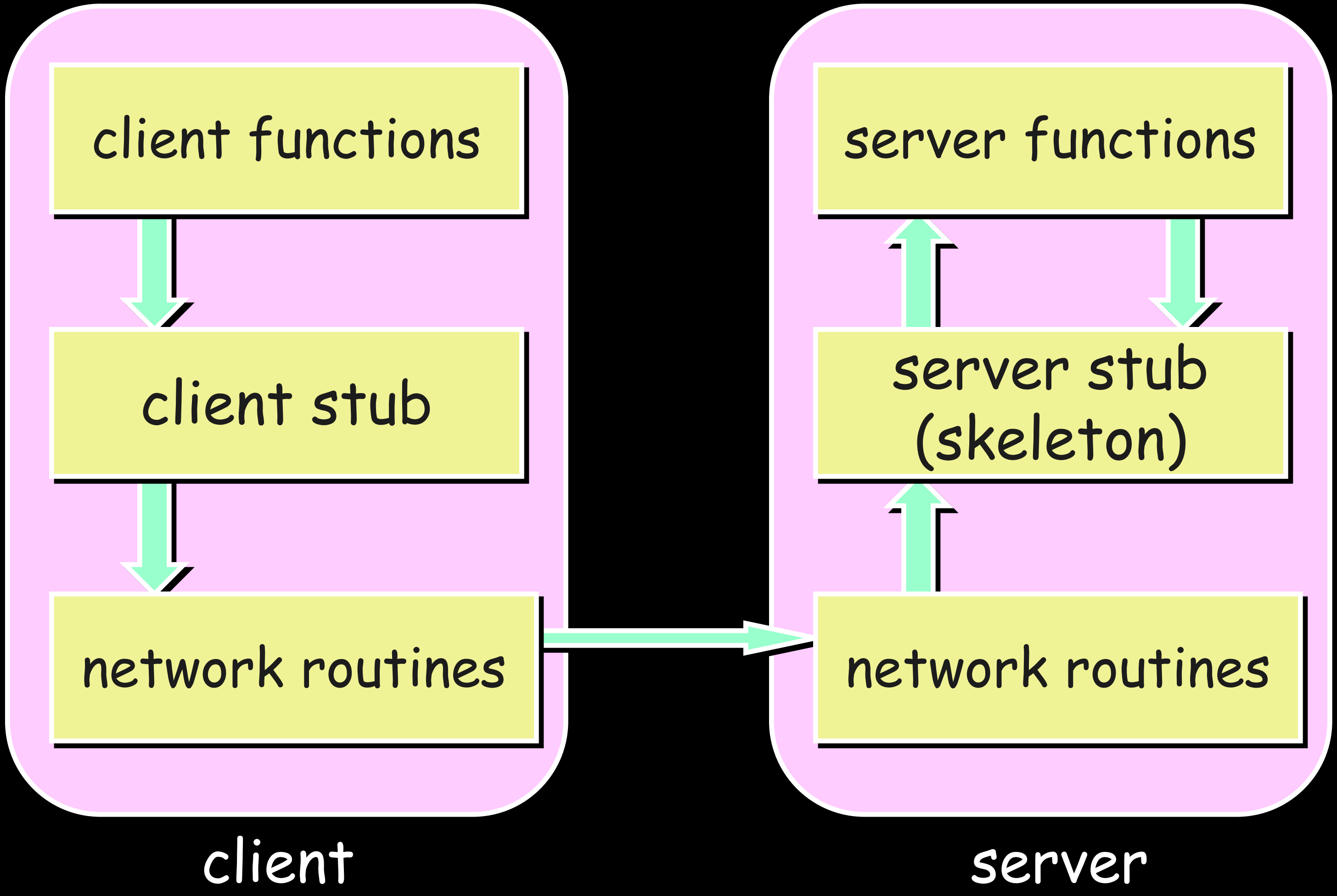
Stub functions

5. Unmarshal parameters, call server func



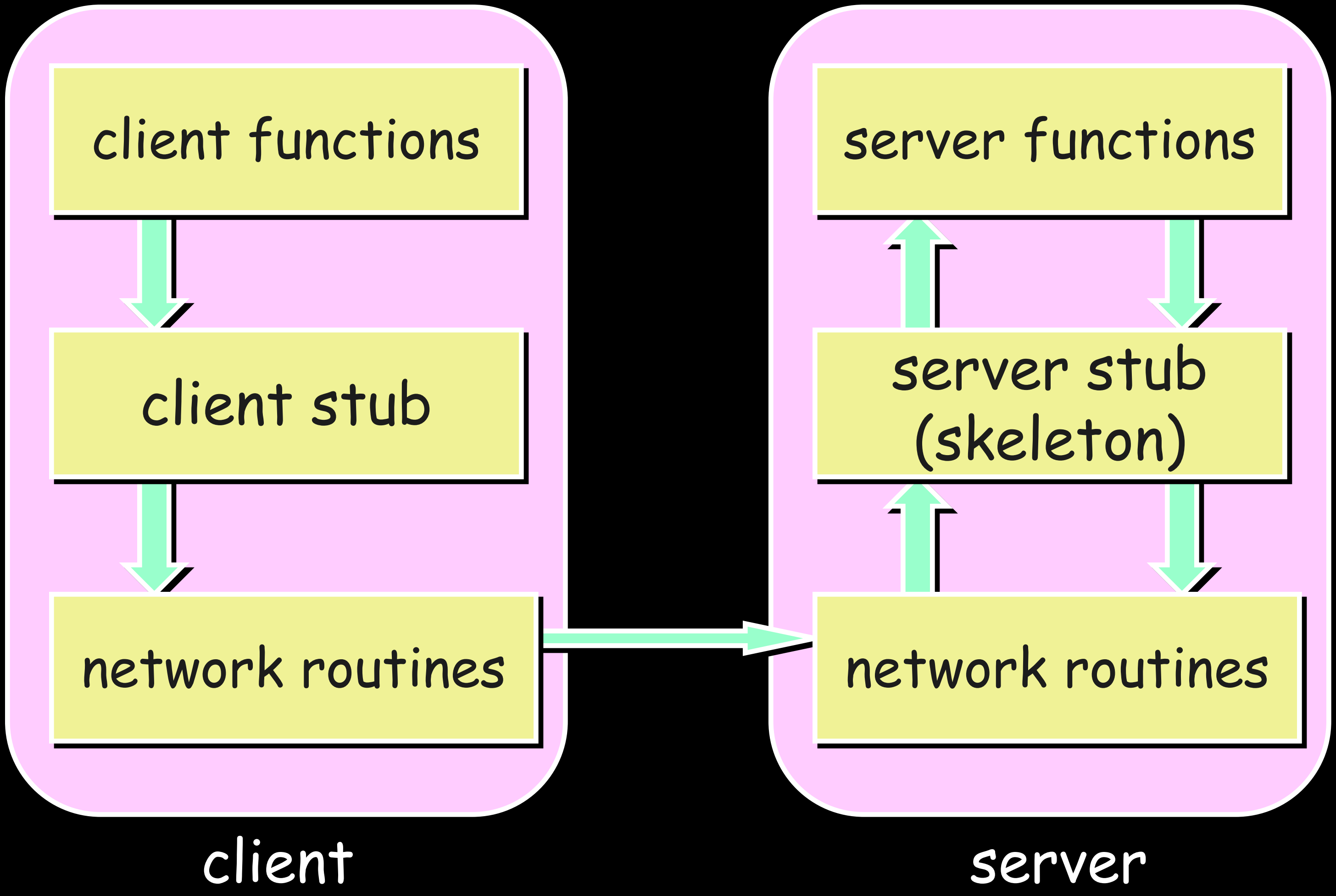
Stub functions

6. Return from server function



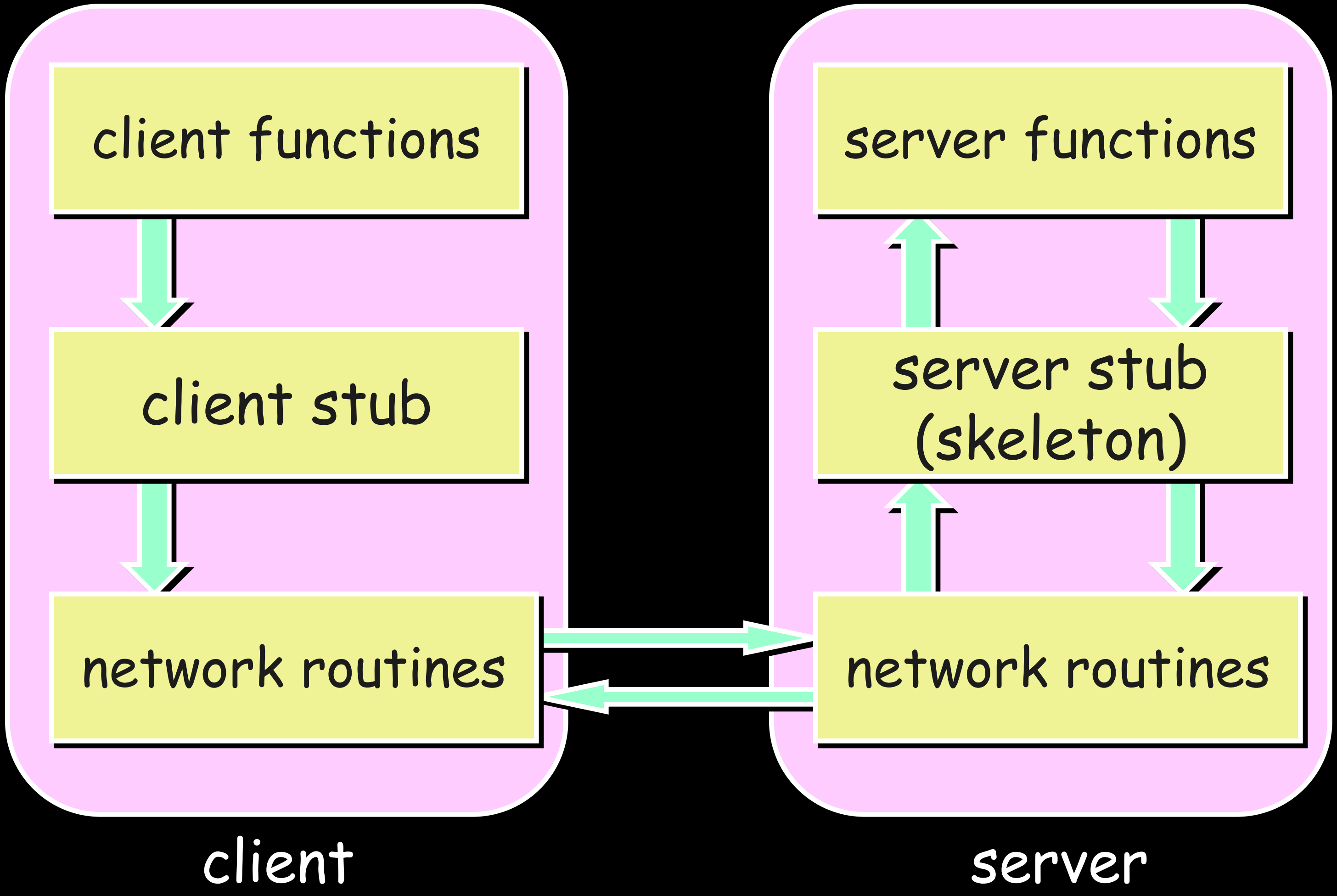
Stub functions

7. Marshal return value and send message



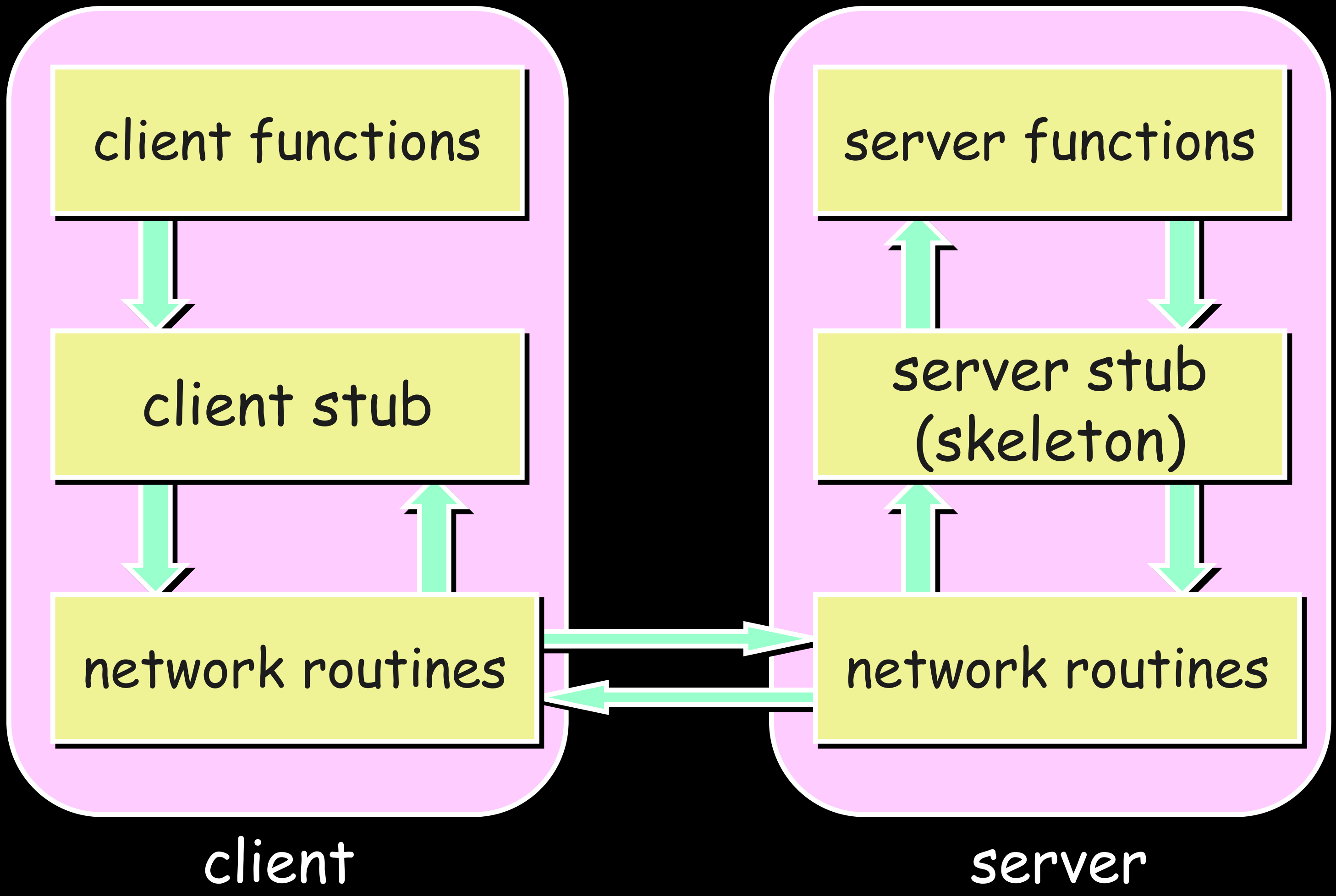
Stub functions

8. Transfer message over network



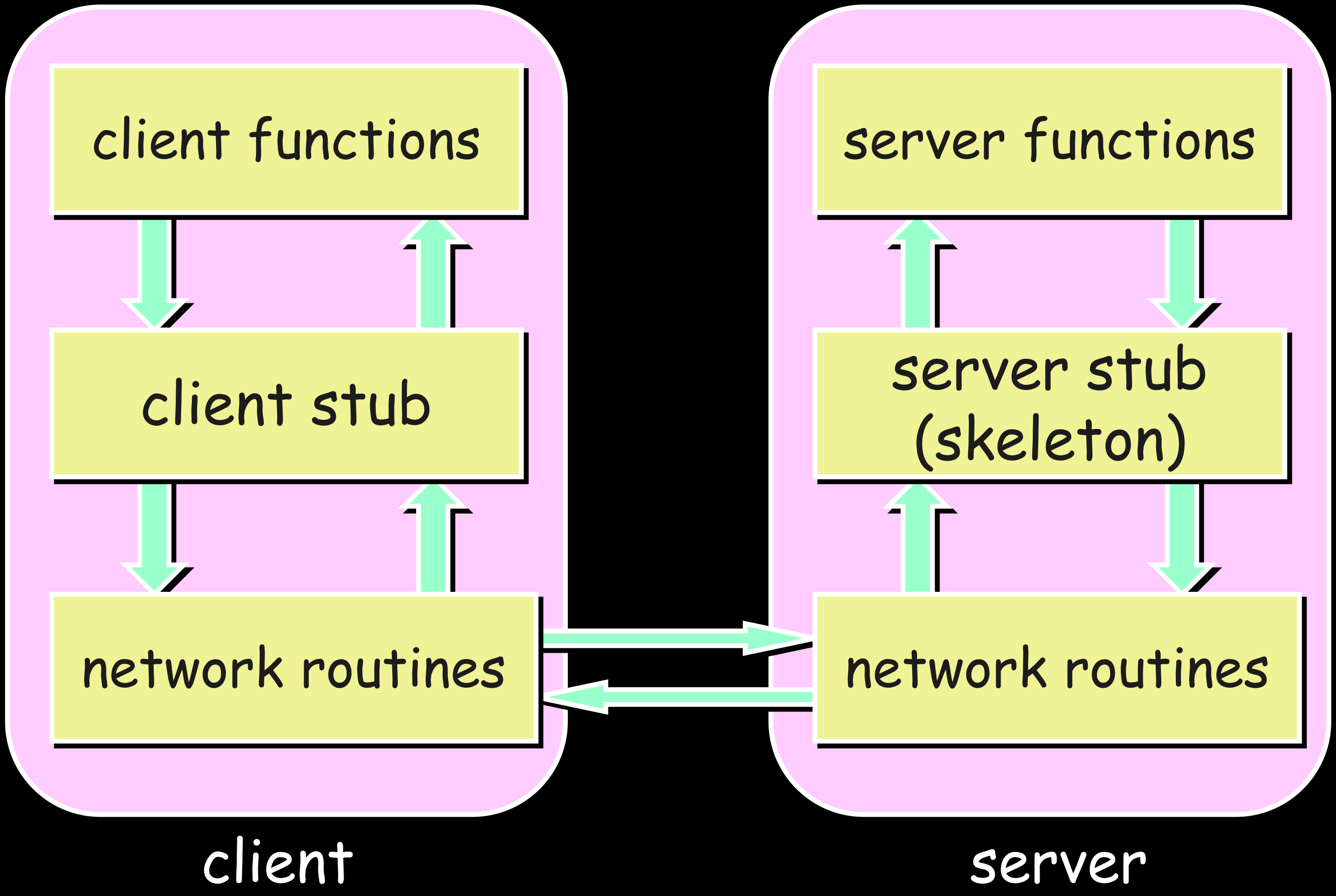
Stub functions

9. Receive message: direct to stub



Stub functions

10. Unmarshal return, return to client code



Internal Details of Sun RPC

- Initialization
  - *Server runs: register RPC with port mapper on server host (**rpcinfo –p**)*
  - *Client runs: **clnt_create** contacts server's port mapper and establishes TCP connection with server (or UDP socket)*
- Client
  - *Client calls local procedure (client stub: **sumproc_1**), that is generated by rpcgen. Client stub packages arguments, puts them in standard format (XDR), and prepares network messages (marshaling).*
  - *Network messages are sent to remote system by client stub.*
  - *Network transfer is accomplished with TCP or UDP.*

Internal Details of Sun RPC

- Server
  - *Server stub (generated by rpcgen) unmarshals arguments from network messages. Server stub executes local procedure (**sumproc_1_svc**) passing arguments received from network messages.*
  - *When server procedure is finished, it returns to server stub with return values.*
  - *Server stub converts return values (XDR), marshals them into network messages, and sends them back to client*
- Back to Client
  - *Client stub reads network messages from kernel*
  - *Client stub returns results to client function*