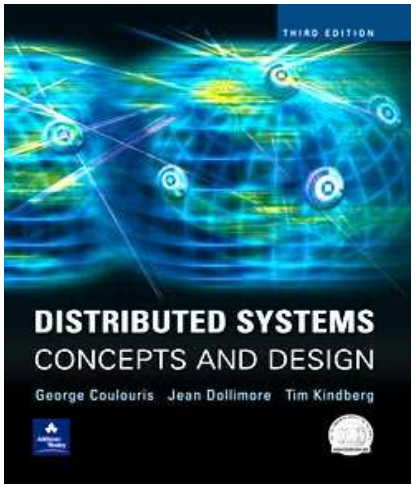
Chapter 5: Remote Procedure Call



From **Coulouris, Dollimore and Kindberg**

Distributed Systems:

Concepts and Design

Edition 3, © Addison-Wesley 2001

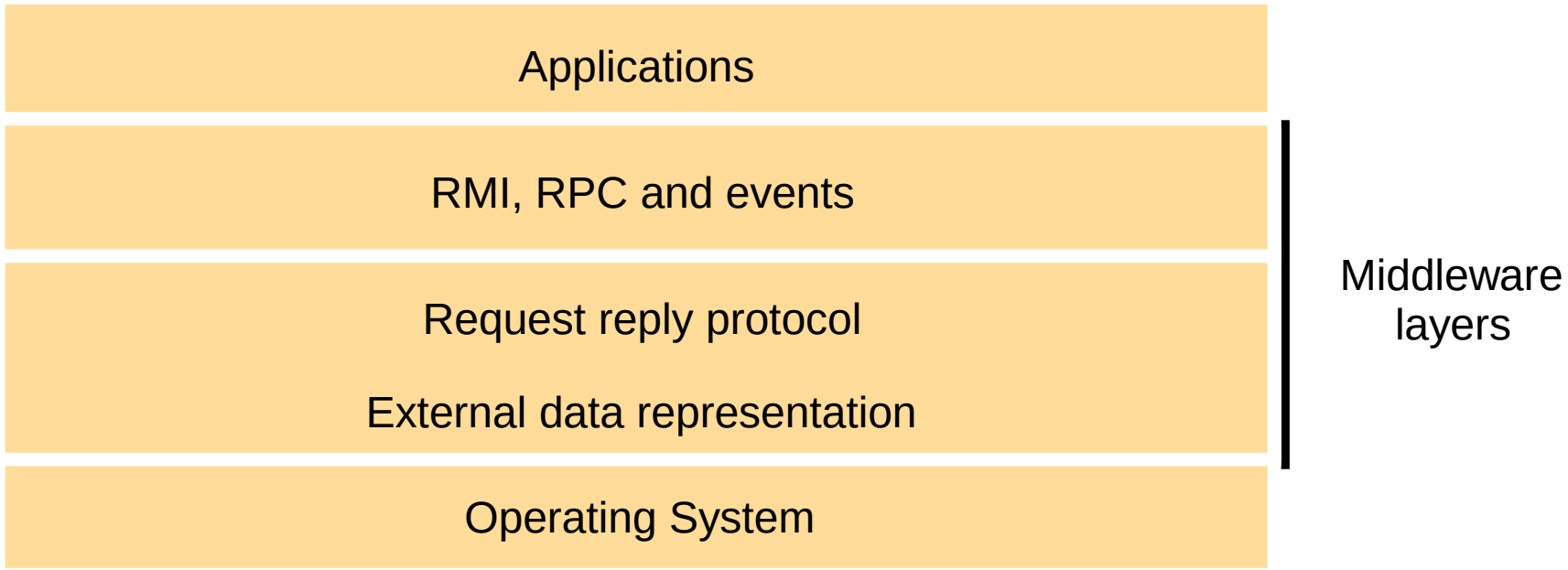
Programming Models for Distributed Application

- Remote procedure call – client calls the procedures in a server program that is running in a different process
- Remote method invocation (RMI) – an object in one process can invoke methods of objects in another process
- Event notification – objects receive notification of events at other objects for which they have registered
- These mechanism must be location-transparent.

Role of Middleware

- Middleware Roles
  - provide high-level abstractions such as RMI
  - enable location transparency
  - free from specifics of
    - communication protocols
    - operating systems and communication hardware
  - interoperability

Figure 5.1
Middleware layers



Interfaces

- Interface – skeleton of public class
  - Interaction specification
  - useful abstraction that removes dependencies on internal details
  - examples
    - procedure interface
    - class interface
    - module interface
- Distributed system interfaces
  - What is the main difference from single processor situation?
    - processes at different nodes
    - can only pass accessible information

Service Interface (RPC)

- Service interface
  - specifies set of procedures available to client
  - input and output parameters
  - Remote Procedure Call
    - arguments are marshaled
    - marshaled packet sent to server
    - server unmarshals packet, performs procedure, and sends marshaled return packet to client
    - client unmarshals the return
    - all the details are transparent

Remote Interface (RMI)

- Remote interface
  - specifies methods of an object available for remote invocation
  - input and output parameters may be objects
  - Remote Method Invocation
    - communication actual arguments marshaled and sent to server
    - server unmarshals packet, performs procedure, and sends marshaled return packet to caller
    - client unmarshals return packet
    - common format definition for how to pass objects (e.g., CORBA IDL or Java RMI)

Interface Definition Language

- Interface Definition Language
  - notation for language independent interfaces
    - specify type and
    - kind of parameters
  - examples
    - CORBA IDL for RMI
    - Sun XDR for RPC
    - DCOM IDL
  - IDL compiler allows interoperability

CORBA IDL Example

```
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();
};
```

Figure 5.2

CORBA has a struct

remote interface

remote interface defines methods for RMI

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

- Remote interface:
  - 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*

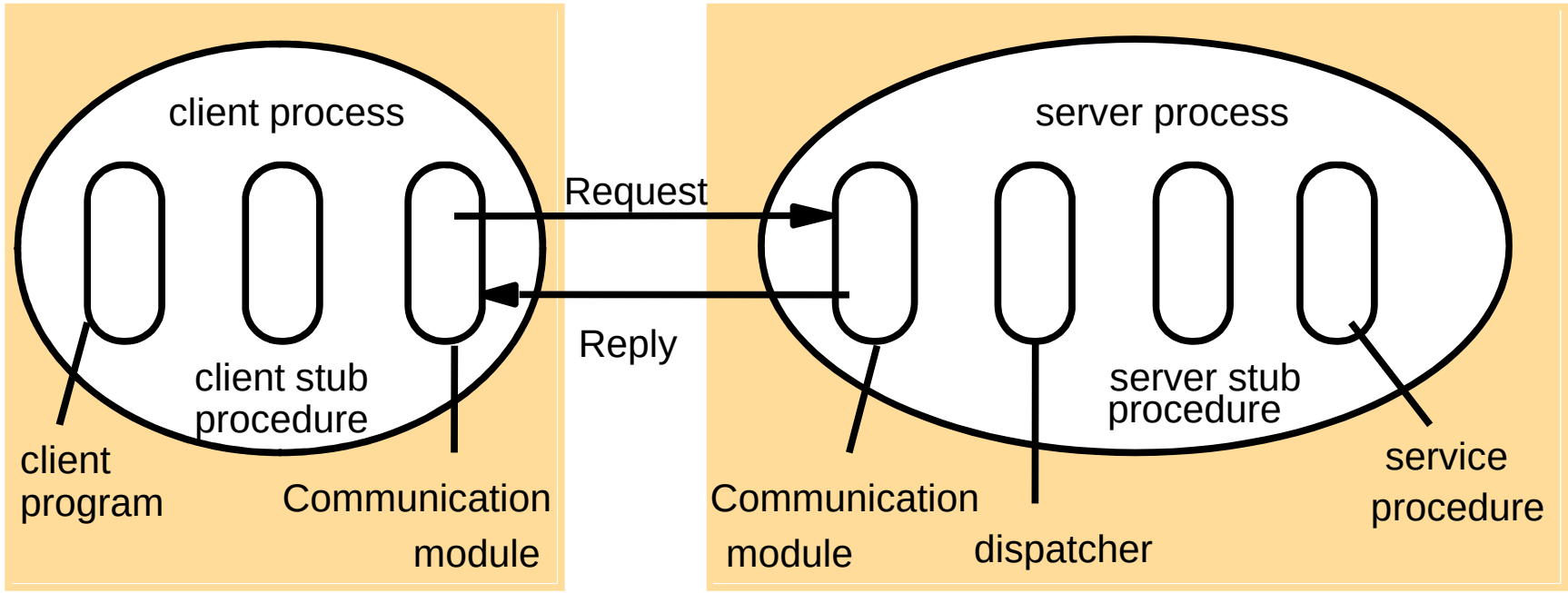
Remote Procedure Call Basics

- Problems with sockets
  - The read/write (input/output) mechanism is used in socket programming.
  - Socket programming is different from procedure calls which we usually use.
  - To make distributed computing transparent from locations, input/output is not the best way.

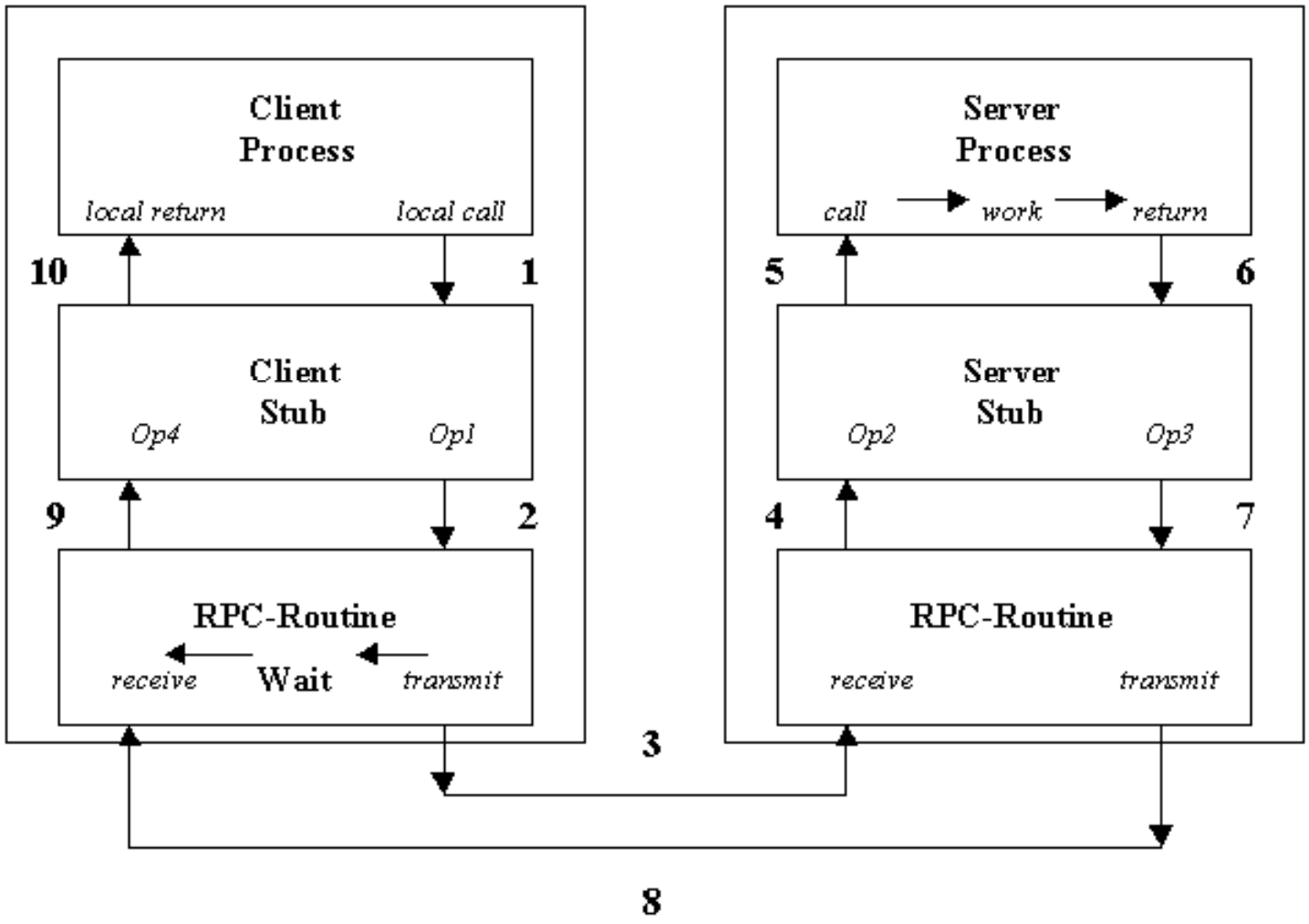
Remote Procedure Call Basics

- A procedure call is a standard abstraction in local computation.
- Procedure calls are extended to distributed computation in Remote Procedure Call (RPC) as shown in Figure 5.7.
  - A caller invokes execution of procedure in the callee via the local stub procedure.
  - The implicit network programming hides all network I/O code from the programmer.
  - Objectives are simplicity and ease of use.

Figure 5.7
Role of client and server stub procedures in RPC



Details of RPC



Remote Procedure Call Basics

- The concept is to provide a transparent mechanism that enables the user to utilize remote services through standard procedure calls.
- Client sends request, then blocks until a remote server sends a response (reply).
- **Advantages**: user may be unaware of remote implementation (handled in a stub in library); uses standard mechanism.
- **Disadvantages**: prone to failure of components and network; different address spaces; separate process lifetimes.

Remote Procedure Call Basics

- Differences with respect to message passing:
  - Message passing systems are peer-to-peer while RPC is more master/slave.
  - In message passing the calling process creates the message while in RPC the system create the message.
- Semantics of RPC:
  - Caller blocks.
  - Caller may send arguments to remote procedure.
  - Callee may return results.
  - Caller and callee access different address spaces.

RPC Issues

- **Uniform call semantics** - Calling semantics must be same whether procedure is implemented locally or remotely.
  - **Exactly once**: It is hard to achieve in practice.
  - **At most once**: It requests RPC only once (no reply may mean that no execution tooks place)
  - **At least once**: It keeps requesting RPC until valid response arrives at client.
  - RPC is inappropriate for nonidempotent operations.
- **Type checking** - Level of static type checking used for local calls should be the same as that for remote calls.

RPC Issues

- **Full parameter functionality** - All basic data types should be allowed; this includes but is not limited to: primitive types, structured types, and user defined types.
- **Concurrency control and exception handling** - Although not a direct portion of RPC, the programming language that provide RPC must support these services.

RPC Issues

- **Distributed binding** - Programming language must be able to compile, bind and load distributed program onto the network
- **Orphan Computations**
  - This involves recovery from failed RPC
  - It can use extermination to find and abort orphan computations.
  - It can use expiration to terminate a computation that exceeds its expected lifetime.

Case Study : Sun RPC

- It is designed for client-server communication over Sun NFS network file system.
- UDP or TCP can be used. If UDP is used, the message length is restricted to 64 KB, but 8 - 9 KB in practice.
- The Sun XDR is originally intended for external data representation.
- Valid data types supported by XDR include **int**, **unsigned int**, **long**, **structure**, **fixed array**, **string** (null terminated char *), **binary encoded data** (for other data types such as lists).

Interface Definition Language

- The notation is rather primitive compared to CORBA IDL or JAVA as shown in Figure 5.8.
  - Instead of no interface definition, a **program number** and a **version number** are supplied.
  - The procedure number is used as a procedure definition.
  - Single input parameter and output result are being passed.

Figure 5.8
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;
};
```

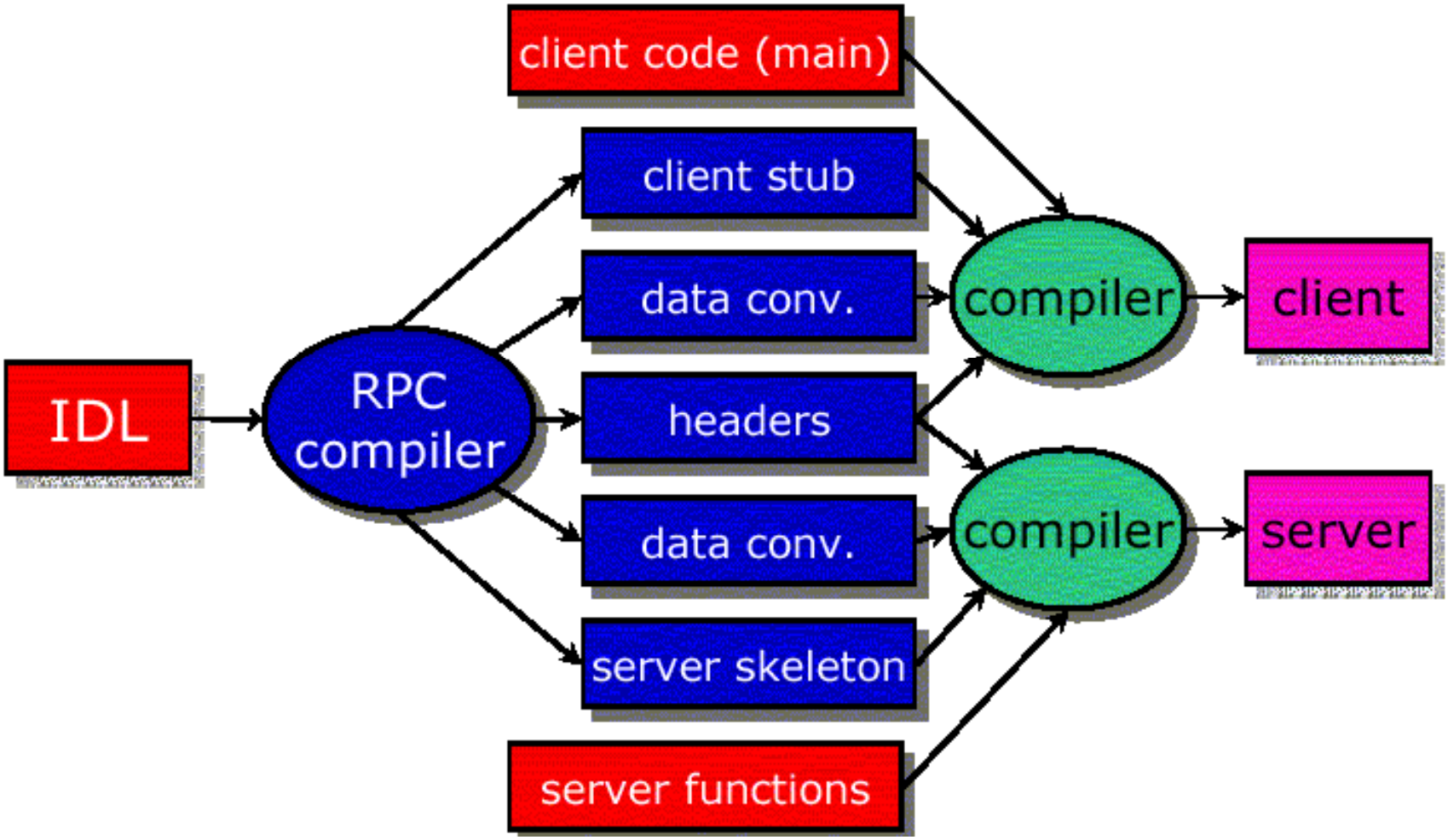
```
struct readargs {
    FileIdentifier f;
    FilePointer position;
    Length length;
};

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

Sun RPC

- 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:
  - **portmapper** records program number, version number, and port number.
  - If there are multiple instance running on different machines, clients make **multicast** remote procedure calls by broadcasting them to all the port mappers.

RPC Interface Compiler



Sun RPC

- Authentication:
  - Additional fields are provided in the request-reply message.
  - Server program should check the authentication and then execute.
  - Different authentication protocols can be supported - none, UNIX style, shared key, Kerberos style.
  - A field in RPC header indicates which style is used.

Example (Sun RPC)

- long sum(long) example
  - client localhost 10
  - result: 55
- Need RPC specification file (sum.x)
  - defines procedure name, arguments & results
- Run (interface compiler) rpcgen sum.x
  - generates sum.h, sum_clnt.c, sum_xdr.c, sum_svc.c
  - sum_clnt.c & sum_svc.c: Stub routines for client & server
  - sum_xdr.c: XDR (External Data Representation) code takes care of data type conversions

RPC IDL File (sum.x)

```
struct sum_in {
  long arg1;
};
struct sum_out {
  long res1;
};
program SUM_PROG {
  version SUM_VERS {
     sum_out SUMPROC(sum_in) = 1; /* procedure number = 1*/
  } = 1;                          /* version number = 1 */
} = 0x32123000;                   /* program number */
```

Example (Sun RPC)

- Program-number is usually assigned as follows:
  - 0x00000000 - 0x1fffffff defined by SUN
  - 0x20000000 - 0x3fffffff defined by user
  - 0x40000000 - 0x5fffffff transient
  - 0x60000000 - 0xffffffff reserved

RPC Client Code (rsum.c)

```
#include "sum.h"
main(int argc, char* argv[]) {
  CLIENT* cl; sum_in in; sum_out *outp;
  // create RPC client handle; need to know server's address
  cl = clnt_create(argv[1], SUM_PROG, SUM_VERS,   "tcp");
   in.arg1 = atol(argv[2]); // number to be squared
   // Call RPC; note convention of RPC function naming
   if ( (outp = sumproc_1(&in, cl)) == NULL)
      err_quit("%s", clnt_sperror(cl, argv[1]);
   printf("result: %ld\n", outp->res1);
}
```

RPC Server Code (sum_serv.c)

```
#include "sum.h"
sum_out* sumproc_1_svc (sum_in *inp, struct svc_req *rqstp)
{ // server function has different name than client call
  static sum_out out; // why is this static?
  int i;
  out.res1 = inp->arg1;
  for (i = inp->arg1 - 1; i > 0; i--)
    out.res1 += i;
  return(&out);
}
// server's main() is generated by rpcgen
```

Compilation Linking

```
rpcgen sum.x
cc -c rsum.c -o rsum.o
cc -c sum_clnt.c -o sum_clnt.o
cc -c sum_xdr.c -o sum_xdr.o
cc -o client rsum.o sum_clnt.o sum_xdr.o
cc -c sum_serv.c -o sum_serv.o
cc -c sum_svc.c -o sum_svc.o
cc -o server sum_serv.o sum_svc.o sum_xdr.o
```

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

Thank You