MONASH University
Information Technology

FIT3142 Distributed Computing 2015 Semester 2

Topic 3: Sockets, Remote Procedure Calls, Remote Object Invocation APIs

Dr Carlo Kopp
Clayton School of Information Technology
Monash University

Why Study Sockets, RPC, Remote Object Invocation APIs?

- These APIs sit beneath the middleware of all distributed systems, and critically impact function and performance;
- Foundation knowledge: Because the behaviour and performance of APIs determines many critical aspects of distributed system function and performance, understanding APIs is essential;
- Foundation knowledge: The limitations of API can limit what a distributed application can or cannot do;
- Practical skills: When coding distributed applications you will have to rely on middleware written around the underlying APIs and their limitations, or the APIs themselves;
- Practical skills: The throughput performance of a API can become a 'live or die" problem for a distributed application, making this an important item to understand;

Inter-Process Communications (Revision)

- IPC exists in many forms, but can be broadly divided into three categories:

1. *Shared Memory IPC that relies on the Virtual Memory system to map pages or segments between processes;*
2. *Message Passing IPC in which the kernel carries discrete messages (or signals) between processes;*
3. *Stream Oriented IPC in which the kernel provides a stream channel between two processes with FIFO properties;*

- Stream Oriented IPC is the model and the programming abstraction mostly employed in distributed systems;
- The two most commonly used stream oriented APIs are BSD Sockets and SVR4 STREAMs;

SVR4 STREAMS vs BSD Sockets

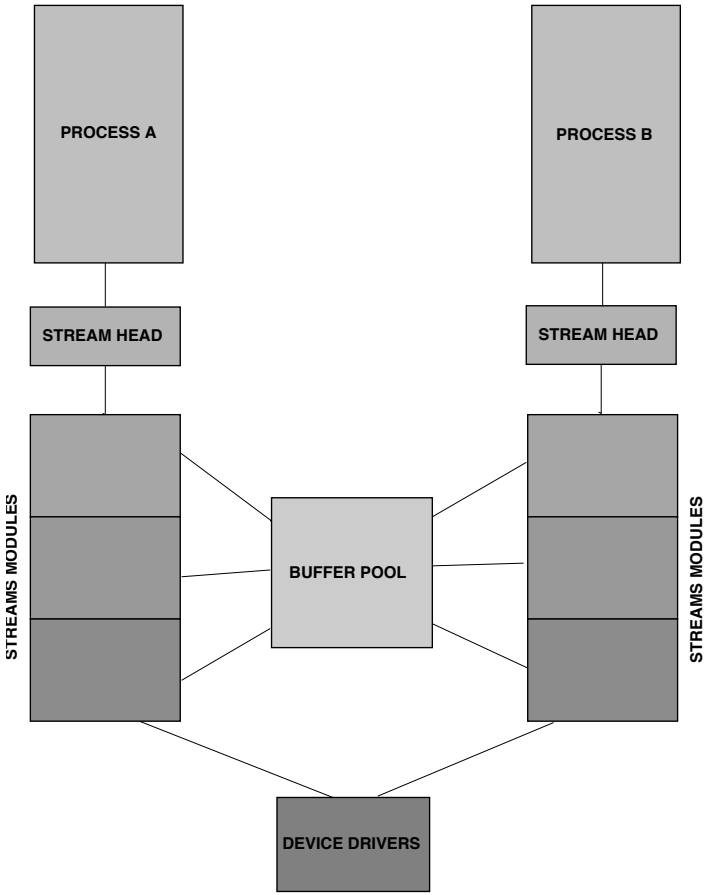


Fig.2.2 STREAMS Transport - Conceptual Model

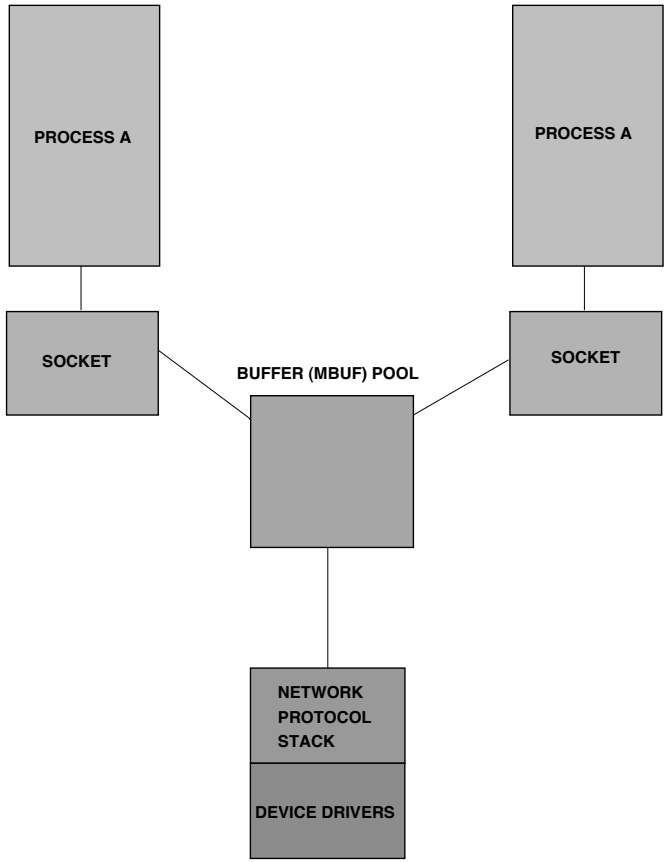
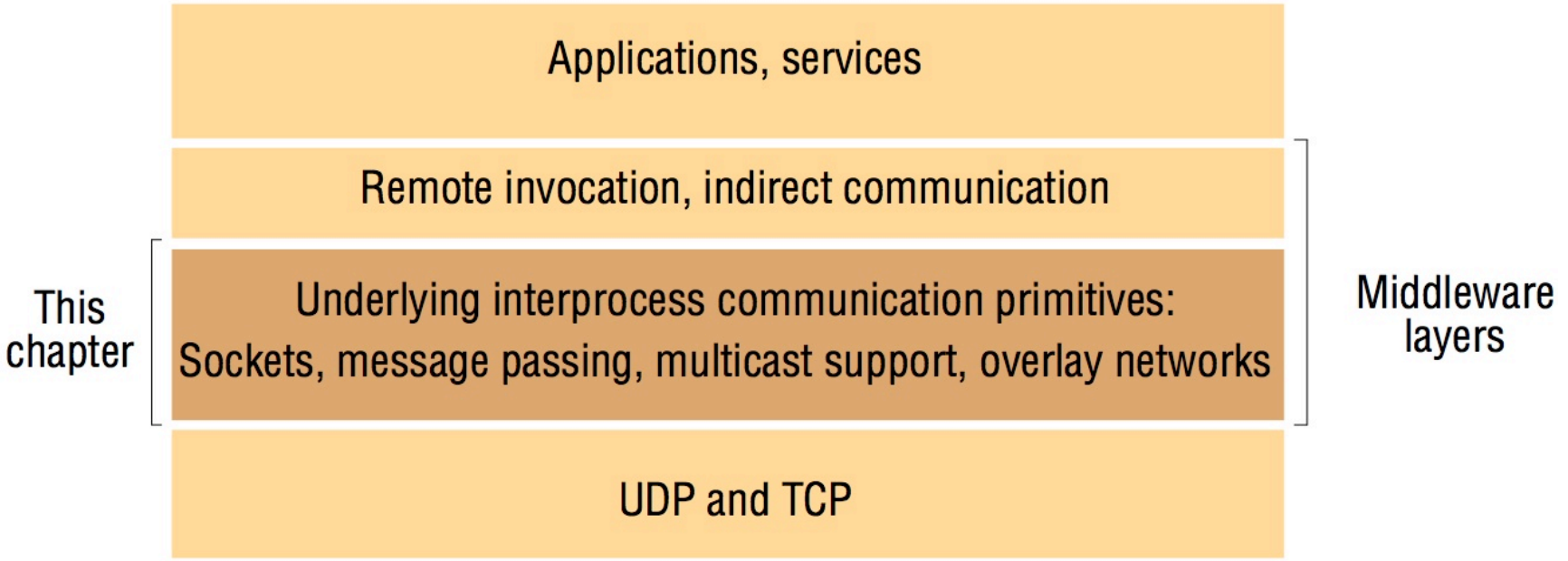


Fig 2.4 BSD Socket Transport - Conceptual Model

Stream Oriented IPC – Integration



Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5, © Pearson Education 2012

- **The "middleware" provides the API interface for the user program running in a process;**
- **The middleware interfaces to the protocol stack software, which is usually embedded in the operating system kernel;**

Example: Linux Socket Integration [Stallings Ch.17]

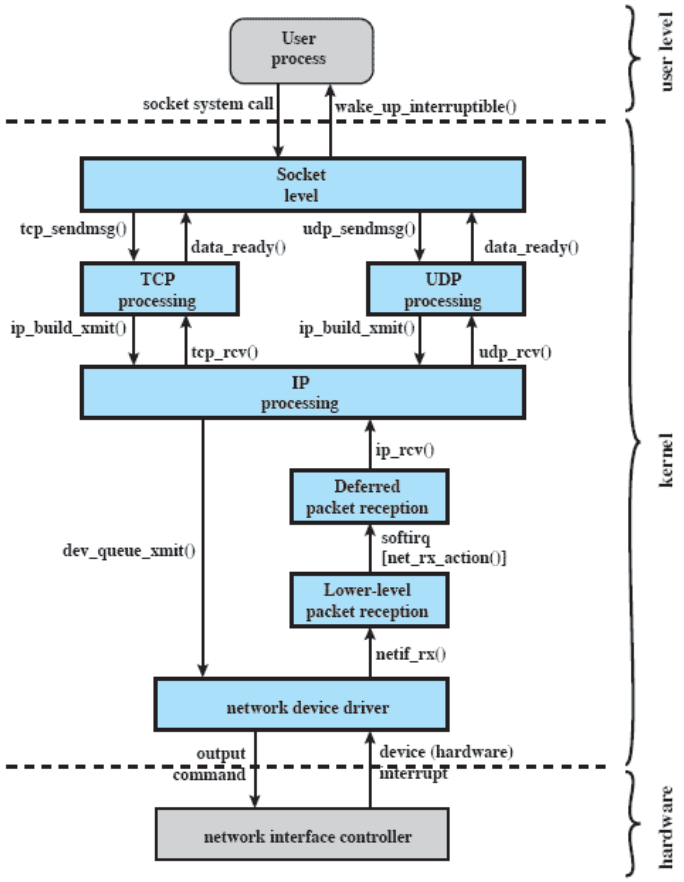


Figure 17.7 Linux Kernel Components for TCP/IP Processing

BSD SOCKET MODEL AND API

BSD Socket Application Programming Interface

- Opening a socket connection requires the creation of the socket with a `socket()` system call, binding a socket address to the socket with a `bind()` call and initiating the connection with a `connect()` call.
- The `socket()` call returns an index into the process file table, termed a file descriptor in Unix/Linux/BSD.
- Once the connection is open, the programmer may use both socket specific calls or the established Unix/Linux/BSD `read()` and `write()` system calls.
- The BSD Socket has become the defacto standard low level programming interface for networked IPC, although in most contemporary applications it is hidden below other protocols;
- Nearly all operating systems will provide a BSD socket API, although some will use the newer POSIX standard.

Socket Function Prototypes

- `int socket(int domain, int type, int protocol);`
- Where the domain can be IPV4, IPV6 or UNIX (local to host);
- `int bind(int sockfd, const struct sockaddr *my_addr, socklen_t addrlen);`
- Where the arguments define the interface;
- `int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);`
- Where `*serv_addr` is the address of the server being connected to;
- Once the stream is established between two processes on two hosts, traffic can be sent or received using `send()`, `recv()`, `write()`, `read()` system calls;
- The socket is shut down using a `close()` call.

Programming with the `socket()` API (I)

- Sockets can be used for IPC between two processes, that reside on one or two hosts, which is useful for debugging code; sockets assume the client-server model, so one process is always a server process, that responds to requests from a client process;
- This requires that the server process be running before the client can access it;
- There are necessary include files for the socket API, that define the API interface types:
- `#include <stdio.h>`
- `#include <sys/types.h>`
- `#include <sys/socket.h>`
- `#include <netinet/in.h>`

Server programming with the `socket()` API (II)

- The necessary starting point is always to set up a server process, which must set up a receiving socket;
- Several steps are necessary:

1. Use the `socket()` system call to create a socket;
2. The socket needs an address so the client can find it, and this is done by binding an address to the socket using the `bind()` system call – for the Internet a *port number* is used;
3. To activate the socket so it listens for incoming clients, a `listen()` system call is required;
4. An incoming client connection must be accepted using the `accept()` system call;

- Socket programming in Java or other languages typically employs a language specific wrapper layered over the C language system calls!

Server programming with the `socket()` API (III)

- The next step is always to set up a client process, which must set up a sending socket;
- Several steps are necessary:

1. Use the `socket()` system call to create a socket;
2. The socket needs to be connected to the server process, and this is done by connecting to the server socket using the `connect()` system call – for the Internet a host *IP address* and *port number* must be used;
3. Once the connection has been accepted by the server, data can be sent and received using `read()` and `write()` system calls;

- Socket programming in C language is the simplest means available for IPC over network connections, but also exposes all of the details of the connection;

Domain parameters for the `socket()` API

- The `int socket(int domain, int type, int protocol)` call has three parameters (example MacOSX 10.10), `domain`:

1. `PF_LOCAL` Host-internal protocols, formerly called `PF_UNIX`,
2. `PF_UNIX` Host-internal protocols, deprecated, use `PF_LOCAL`,
3. `PF_INET` Internet version 4 protocols,
4. `PF_ROUTE` Internal Routing protocol,
5. `PF_KEY` Internal key-management function,
6. `PF_INET6` Internet version 6 protocols,
7. `PF_SYSTEM` System domain,
8. `PF_NDRV` Raw access to network device

- NB local host `PF_LOCAL/PF_UNIX` versus network host `PF_INET/PF_INET6`!

Type parameters for the `socket()` API

- The `int socket(int domain, int type, int protocol)` call has some important parameters (example MacOSX 10.10);
- Defines for the `int type` argument:

1. `SOCK_STREAM` sequenced, reliable byte streams (TCP)
2. `SOCK_DGRAM` connectionless, unreliable (UDP)
3. `SOCK_RAW` internal network protocols and interfaces
4. `SOCK_SEQPACKET` sequenced, reliable, two-way connection-based data transmission path for datagrams of fixed maximum length
5. `SOCK_RDMNB` not implemented

- NB the `int protocol` argument is usually by default set to 0, although some systems may support a range of protocols.

Caveats for the `socket()` API

- The socket() API may be the simplest API for providing stream (TCP) or datagram (UDP) IPC, but it also requires that the programmer understand choices in protocols as well as the port addressing scheme;
- The programmer has to manage the state of the connection, and handle all of the errors;
- The API permits seamless use of socket connections local to the host (if Unix/BSD/Linux OS), or to remote hosts;
- The socket appears to the programmer as an `int sockfd` file descriptor, which is one layer of abstraction below the `file' used by the libc stdio *nix / C language abstraction;
- The `fwrite()` and `fread()` stdio calls are wrappers for the `int write()` and `int read()` *nix/POSIX system calls, and operate on a `FILE` struct that presents as a file;

Client-Server – HTTP over Sockets

- HTTP (Hypertext Transfer Protocol) is the most widely used protocol on the W3 and is a good example of a protocol built on top of the BSD Socket API;
- When a browser (client) intends to make a request of a web server (server), it opens a socket connection over the Internet to the web server;
- The browser then sends a HTTP Method message to the web server, for instance:

```
GET /mypath/to/myfile/blogs.html HTTP/1.0
```

- The socket connection is then closed, while the server processes the method request;
- Once processing is complete, the web server opens a socket connection to the client, and responds with a message, header and MIME encoded body: `HTTP/1.0 404 Not Found`

Client-Server – HTTP over Sockets [200 OK]

- **HTTP Response:** `HTTP/1.0 200 OK`
- **HTTP Header:** `Last-Modified: Tue, 12 Jul 2011 21:59:59 GMT`
- **HTTP Body:**

```
Content-Type: text/html
Content-Length: 512
<!DOCTYPE doctype PUBLIC "-//w3c//dtd html 4.0
   transitional//en"> <html> <head> <meta http-
   equiv="Content-Type" content="text/html;
   charset=UTF-8"> <meta name="Author"
   content="Carlo Kopp"> <meta name="GENERATOR"
   content="Mozilla/4.55 [en] (X11; U; Linux 2.4.2
   i386) [Netscape]"> <title>Carlo Kopp's Homepage</
   title> </head> <body
```

Client-Server – HTTP over Sockets

```
… More page content …
<center><!-- E N D T R A I L E R T A B L E --></
   center> </span><!-- M4 Macro --> </body> </html>
```

- Once the Body is transferred, the socket connection is then closed;
- HTTP is widely used to support other mechanisms used in distributed computing;
- Secure HTTP (SHTTP) employs a more complex connection mechanism due to the use of TLS or SSL encryption layers;
- As HTTP lacks mechanisms to handle multiple servers concurrently, it is a good example of a basic client server protocol.

REMOTE PROCEDURE CALLS

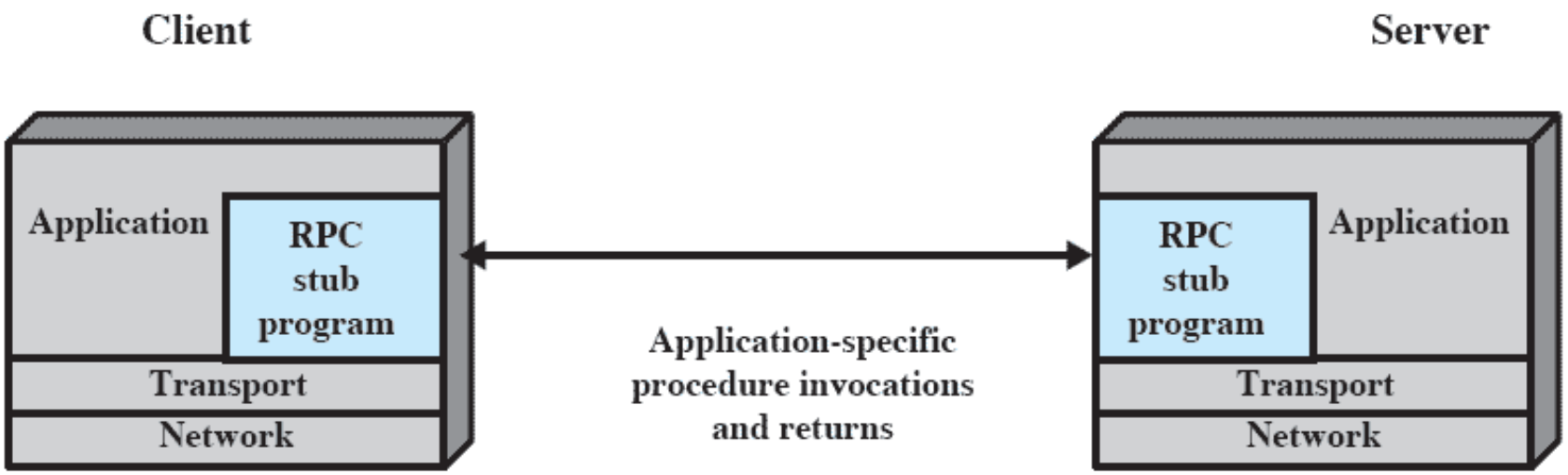
Remote Procedure Call API (RPC)

- Stream oriented network API interfaces provide an unstructured channel for byte or message oriented data transfers, which carry data – this is the most basic abstraction possible, involving `read()` and `write()` calls;
- The next level of abstraction is that of a *remote procedure call* in which a procedure (i.e. function call) may be executed locally on a host, or on a remote server host;
- The client process will make a request upon a server process, which involves a procedure identifier (name) and some list of arguments; the server then returns the results of the call;
- This is a structured API that bounds the transfers to very specific messages – calls and returns of values;
- ONC RPC (RFC1831) protocol is the most widely used *remote procedure call* API;

Remote Procedure Call API (ONC RPC)

- ***Open Network Computing Remote Procedure Call (ONC RPC RFC1831)*** **protocol was developed by Sun Microsystems and widely used for Network File System (NFS) remote mounting of disk storage devices;**
- ***Distributed Computing Environment / Remote Procedure Calls (DCE/RPC)*** **was developed in competition, and later used in the Microsoft DCOM and .NET Remoting;**
- **ONC RPC is structured around the use of "stubs", which are blocks of code that provide the RPC API interface, and are compiled into an application binary program;**
- **When RPC transfers data, such as arguments or return values, the "endianess" of the data matters; this is managed with the XDR (eXternal Data Representation) protocol, which ensures data always has the correct "endianess";**

Remote Procedure Calls [Stallings Ch.16]



(b) Remote Procedure Calls

Remote Procedure Calls [Stallings Ch.16]

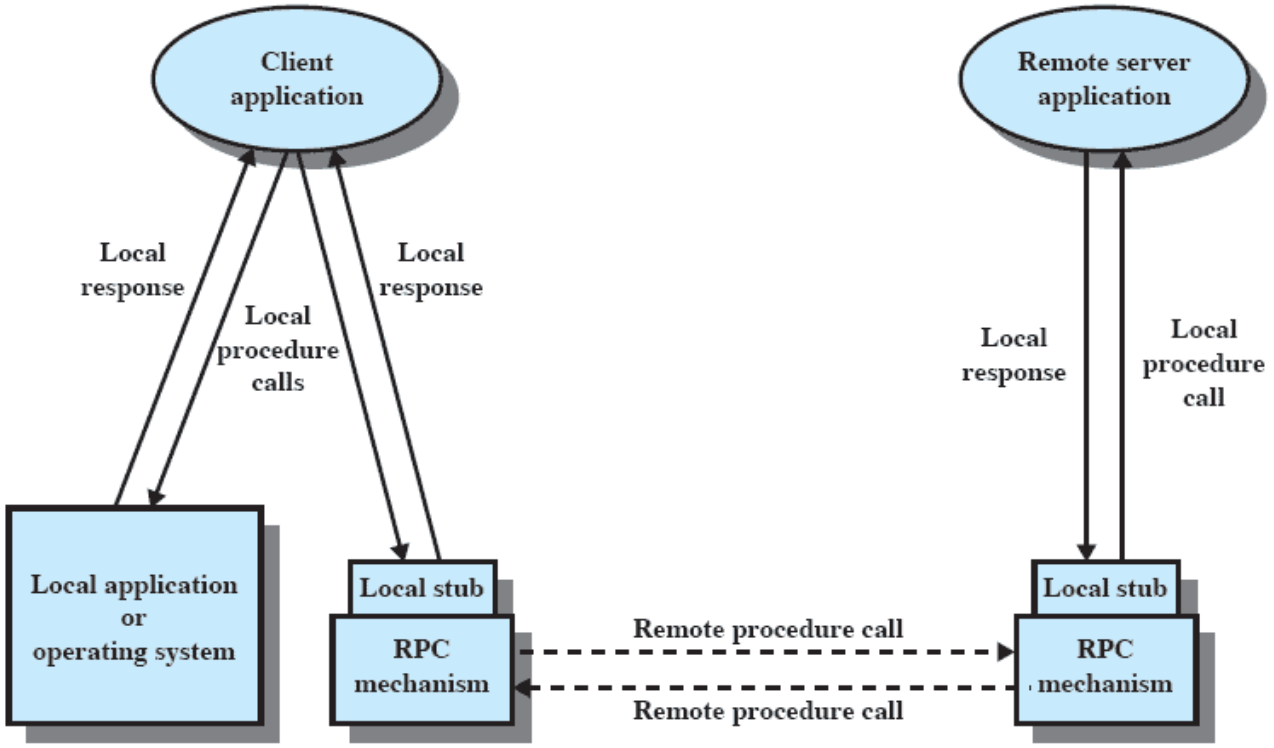


Figure 16.12 Remote Procedure Call Mechanism

ONC RPC Model (Kopp 1994)

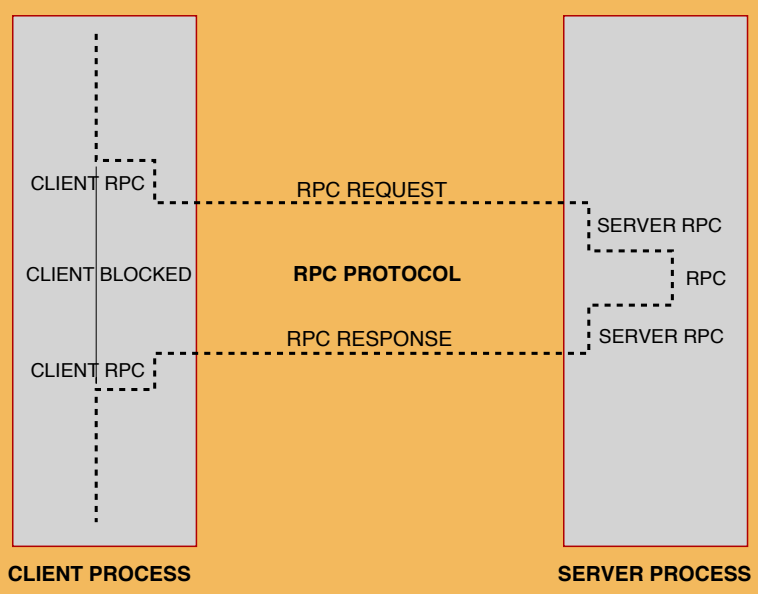


FIG.1 REMOTE PROCEDURE CALLS – THREAD OF EXECUTION

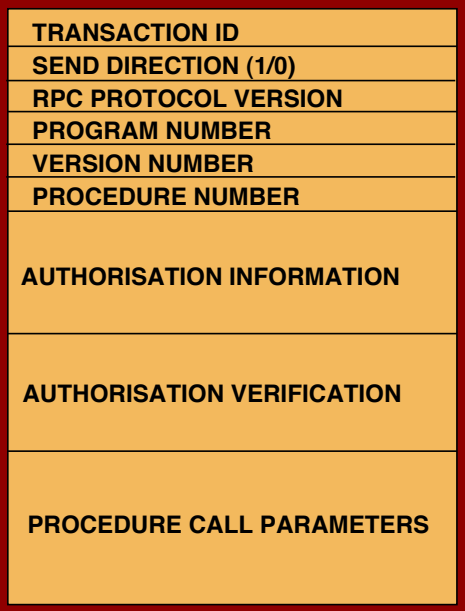


FIG.2 REMOTE PROCEDURE CALL PROTOCOL HEADER

ONC RPC Model (Kopp 1994)

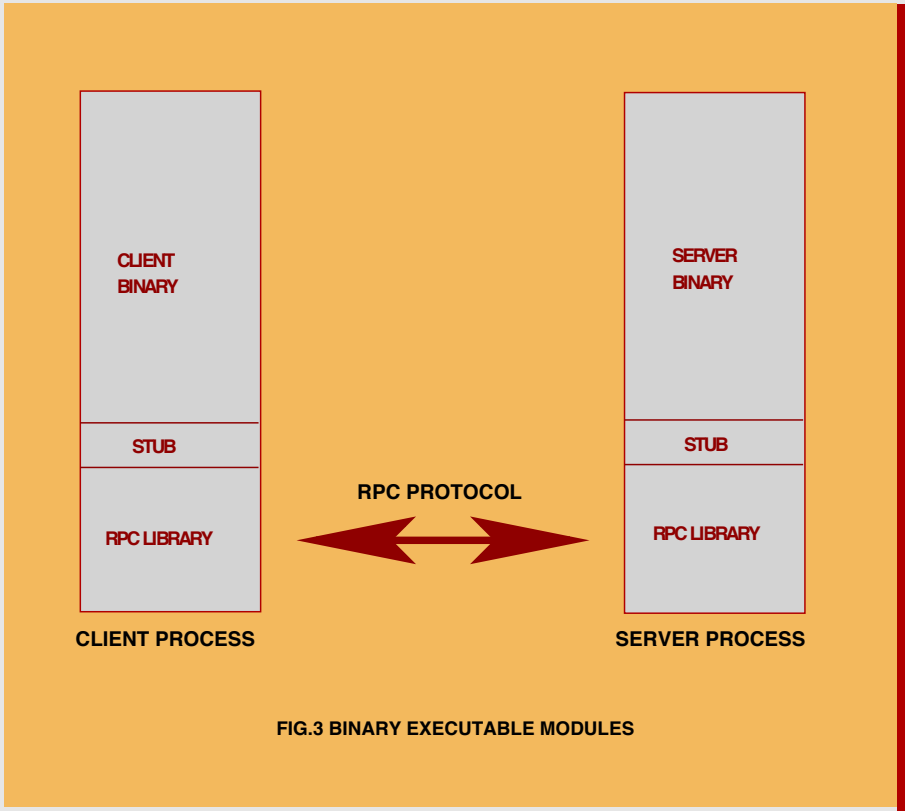


FIG.3 BINARY EXECUTABLE MODULES

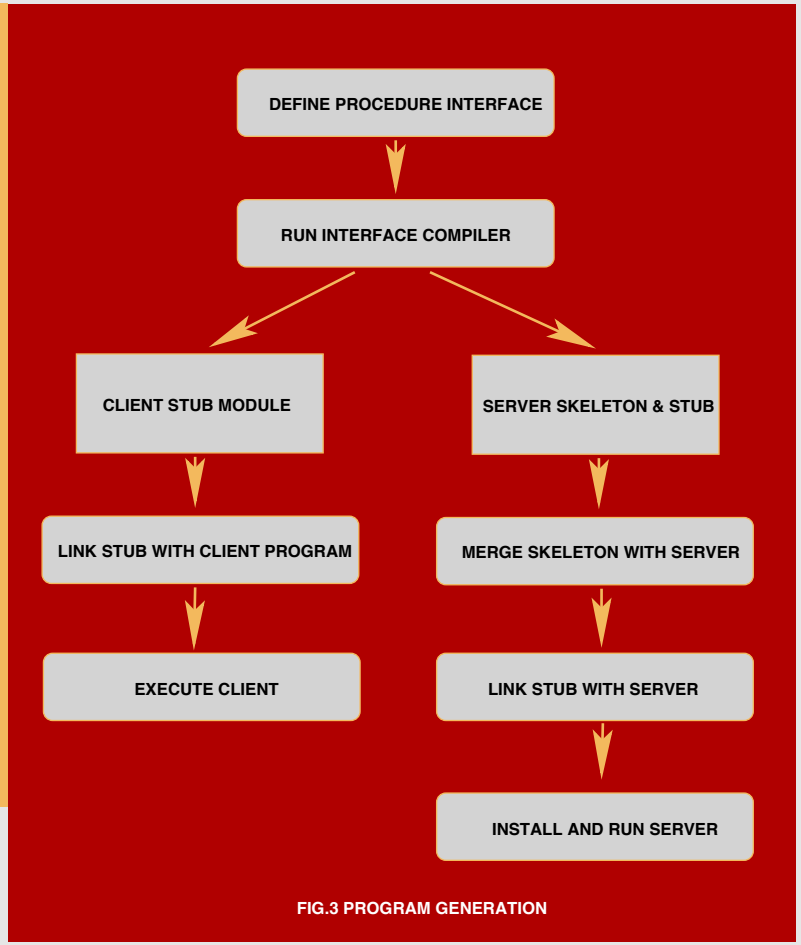


FIG.3 PROGRAM GENERATION

REMOTE OBJECT INVOCATION

Invoking Remote Objects

- The socket IPC API provides for unstructured transfers of data, while remote procedure call APIs are designed for remote execution of subroutine code;
- Neither of these abstractions is well aligned with Object Oriented (OO) languages, such *as C++, C#, Objective C, Java, Python*, where objects combine internal procedures and data;
- The preferred approach has been to develop OO APIs for such languages, using protocols that support the remote invocation and management of objects;
- The general approach used in such APIs follows a similar model to that used with remote procedure calls;
- A number of schemes have been developed since the 1990s to provide this style of API, support depending on the language and operating system employed;

Some Remote Object Invocation Schemes

- The popularity of OO languages resulted in ~30 schemes for remote object invocation, the most widely used are:
- *CORBA (Common Object Request Broker Architecture)* was developed for the C++ language and Unix operating system, even though other languages and platforms are supported;
- *Windows Communication Foundation (WCF)* evolved from *DCOM (Distributed Component Object Model)*, developed by Microsoft to compete with CORBA, using DCE/RPC;
- *Java Remote Method Invocation (Java RMI)* running over *Java Remote Method Protocol (JRMP)* is Java specific scheme;
- *SOAP (Simple Object Access Protocol) evolved from the earlier* XML-RPC scheme and uses XML encoded messages usually over HTTP transport; SOAP is used for web services and also in the Open GRID protocol suite;

CORBA Model (OMG 2000)

- CORBA extends the RPC model from simple procedures to complete objects including data;
- The intent was to provide a client-server scheme which was suitable for OO languages;
- CORBA provides interfaces for the C, C++, Java, COBOL, Smalltalk, ADA, LISP, and Python languages.
- CORBA is primarily built for client-server applications.

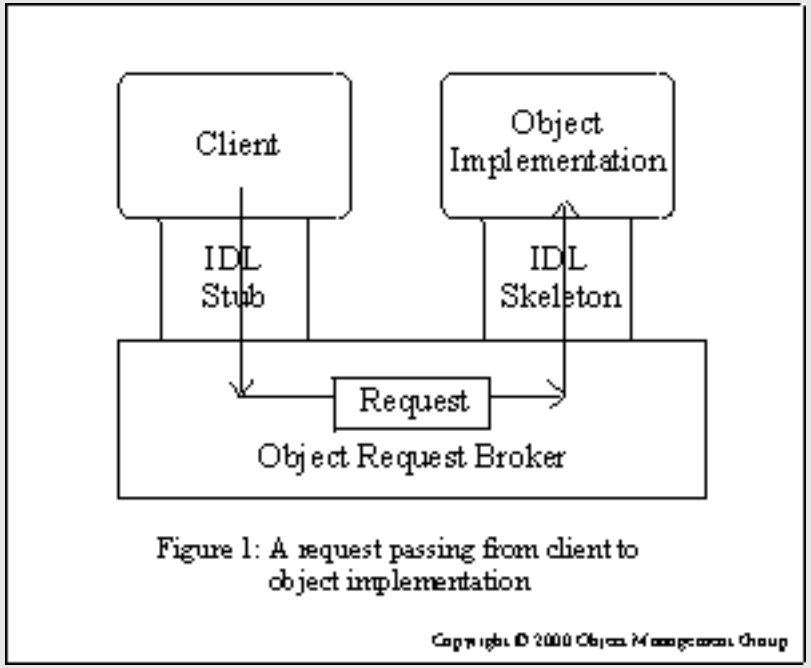


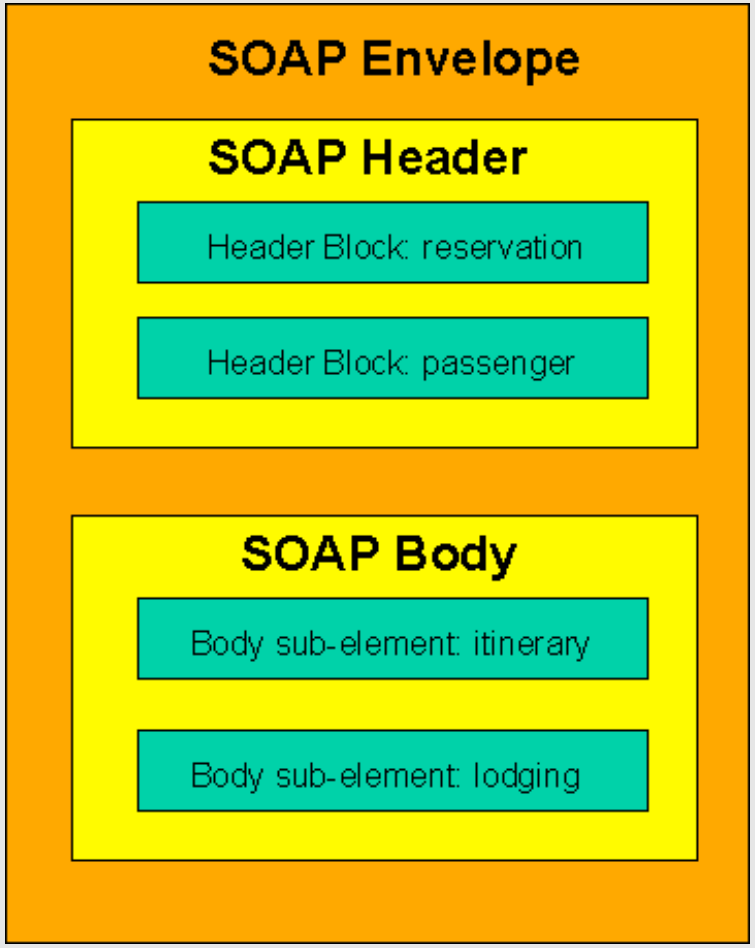
Figure 1: A request passing from client to object implementation

Web Services - Simple Object Access Protocol

- **Communications between distributed applications in WS employ primarily HTTP, or other "basic" well established protocols such as FTP, SMTP etc;**
- **Messaging is based on SOAP (Simple Object Access Protocol), which is most commonly transmitted over HTTP or HTTPS in secure environments;**
- **SOAP provides similar functionality to CORBA, but is inherently more verbose due to the use of XML;**
- **W3C*: "A SOAP message is fundamentally a one-way transmission between SOAP nodes, from a SOAP sender to a SOAP receiver, but SOAP messages are expected to be combined by applications to implement more complex interaction patterns ranging from request/response to multiple, back-and-forth 'conversational' exchanges."***

W3C Simple Object Access Protocol

- Example SOAP message (W3C) for airline reservation application in WS environment;
- SOAP message has "Envelope" containing a SOAP "Header" with identifying information and "Body" with actual message payload;
- The SOAP message is formatted in XML language.



Example SOAP Message

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Header>
<m:reservation xmlns:m="http://travelcompany.example.org/reservation" env:role="http://www.w3.org/2003/05/soap-
      envelope/role/next" env:mustUnderstand="true">
<m:reference>uuid:093a2da1-q345-739r-ba5d-pqff98fe8j7d</m:reference>
<m:dateAndTime>2001-11-29T13:20:00.000-05:00</m:dateAndTime> </m:reservation>
<n:passenger xmlns:n="http://mycompany.example.com/employees" env:role="http://www.w3.org/2003/05/soap-
      envelope/role/next" env:mustUnderstand="true">
<n:name>Åke Jógvan Øyvind</n:name> </n:passenger> </env:Header>
 <env:Body> <p:itinerary xmlns:p="http://travelcompany.example.org/reservation/travel">
<p:departure>
 <p:departing>New York</p:departing>
<p:arriving>Los Angeles</p:arriving>
<p:departureDate>2001-12-14</p:departureDate>
<p:departureTime>late afternoon</p:departureTime>
 <p:seatPreference>aisle</p:seatPreference> </p:departure>
 <p:return> <p:departing>Los Angeles</p:departing> <p:arriving>New York</p:arriving>
      <p:departureDate>2001-12-20</p:departureDate>
<p:departureTime>mid-morning</p:departureTime> <p:seatPreference/> </p:return> </p:itinerary> <q:lodging
      xmlns:q="http://travelcompany.example.org/reservation/hotels"> <q:preference>none</q:preference> </
      q:lodging> </env:Body> </env:Envelope>
```

W3C: Simple Object Access Protocol RPC

- To invoke a SOAP RPC, the following information is needed:

1. The address of the target SOAP node.
2. The procedure or method name.
3. The identities and values of any arguments to be passed to the procedure or method together with any output parameters and return value.
4. A clear separation of the arguments used to identify the Web resource which is the actual target for the RPC, as contrasted with those that convey data or control information used for processing the call by the target resource.
5. The message exchange pattern which will be employed to convey the RPC, together with an identification of the so-called "Web Method" (on which more later) to be used.
6. Optionally, data which may be carried as a part of SOAP header blocks.

Web Services in Java

- Java has a defined interface for Web Services, released initially in the 2002 JAX-RPC document;
- The model is based on the RPC concept, where a client can communicate with a remote process on a server host, and invoke procedure calls as required;
- The JAX-RPC scheme is based on the SOAP interface, and is intended to be standards complaint;
- The intent of the JAX-RPC scheme is to provide Java developers with an interface which is easily used and well validated.

References/Reading

1. Ch.1; Ch.3; Ch.4; Coulouris, Dollimore, Kindberg and Blair, *Distributed Systems: Concepts and Design*, Edition 5, © Addison-Wesley, 2012.
2. Pawel Plaszczak and Richard Wellner, Jr.; *Grid Computing*, Elsevier, 2005.
3. http://www.globus.org/
4. http://www.csse.monash.edu.au/~carlo/SYSTEMS/IPC-Introduction-0595.htm
5. http://www.csse.monash.edu.au/~carlo/SYSTEMS/RPC-Intro-0796.htm
6. http://www.csse.monash.edu.au/~carlo/WALNUT/msc.thesis.ck.pdf
7. http://www.javacoffeebreak.com/articles/rmi_corba/index.html
8. http://java.sun.com/javase/technologies/core/corba/index.jsp
9. http://www.cs.utexas.edu/~wcook/Drafts/2006/WSvsDO.pdf
10. http://www.eg.bucknell.edu/~cs379/DistributedSystems/rmi_tut.html
11. http://www.csse.monash.edu.au/~carlo/SYSTEMS/Cluster-Practical-1299.html
12. http://www.omg.org/gettingstarted/corbafaq.htm