Concurrent & Distributed Systems

inter-process communication

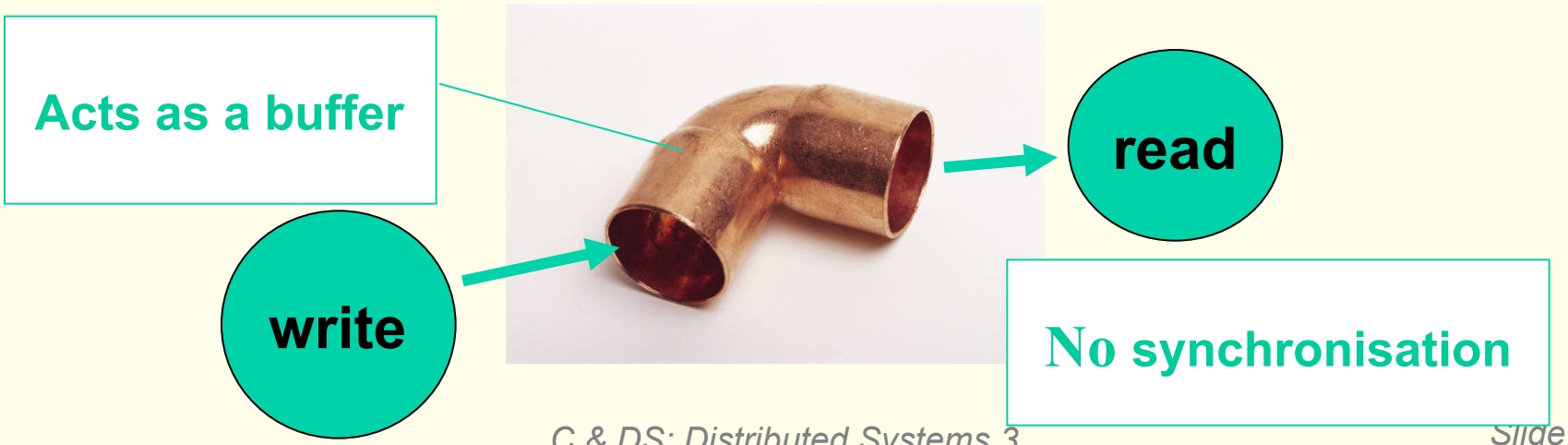
- **files**
- **sockets**
- **message passing**
- ***remote procedure calls***
  - **we will probably not finish this today**
- **remote method invocation**
- **(CORBA)**

remote procedure call

- in this lecture we shall examine:
  - Remote Procedure Call (RPC)
  - Marshalling and un-marshalling of data
  - Sun RPC (case study)
- References
  - “Power Programming with RPC”, John Bloomer, O’Reilly & Associates, Inc. A nutshell book. (whole book)
  - “Distributed Operating Systems”, Andrew Tanenbaum, Prentice Hall, §2.4.

remote procedure call: *motivation*

- although the model of programming with streams is convenient it suffers from one flaw: it is I/O-based.
  - not all programs fall into the category of read from a stream, write to a stream - the model forced on us using sockets (or pipes).



remote procedure call

- we have seen how this is possible in “traditional programming” where all procedures are held within the same address space.
- in a distributed system, we need a way to call procedures in different **address spaces**, and probably on **separate machines**.
- in order to do this we use *Remote Procedure Call* (RPC).
  - RPC allows procedures to be executed on remote machines.

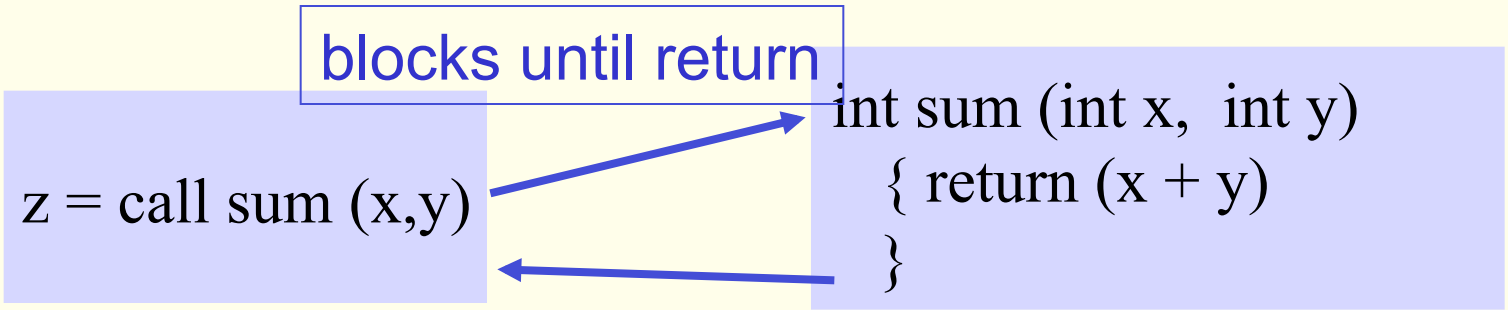
RPC

- the concept of RPC was first proposed by Birrell and Nelson in their seminal paper published in 1984:
  - Birrell, A. D. and Nelson, B. J. “Implementing Remote Procedure Calls”, Transactions on Computer Systems, volume 2, issue 1, pp. 39-59, 1984.
- RPC is a nice programming model:
  - It combines synchronisation and communication.
  - No message passing or I/O is visible to the programmer.
  - Everything looks exactly like a normal procedure call.

RPC

- when a process on machine A calls a procedure on machine B, the thread of computation on A is suspended until the procedure returns from machine B.

- information can be passed to the callee by the caller and back when the procedure returns.



RPC: *reality (1)*

- RPC sounds easy in practice.
- however, it is complicated for a number of reasons:
- procedures reside on different machines.
  - this means we cannot simply jump to the start of the procedure.
  - we need to use network communication techniques to interact with the remote machine.

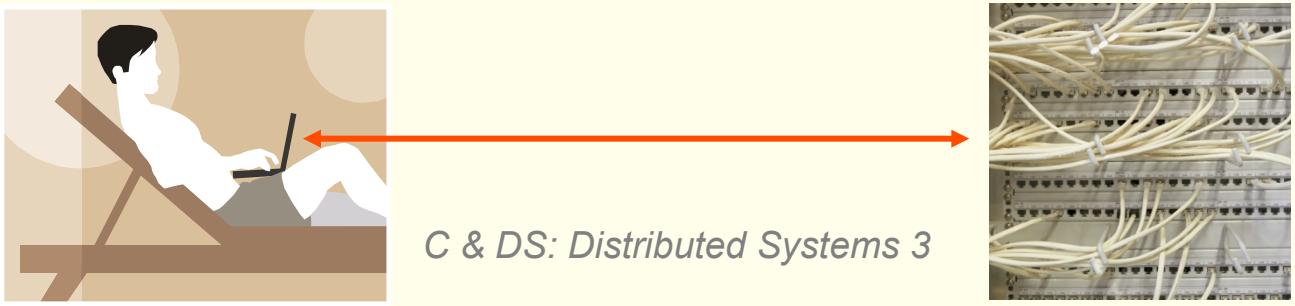
?

RPC: *reality (2)*

- procedures reside in different address spaces.
  - This means that we cannot pass pointers from caller to callee because a pointer is only valid in one address space.
- parameters and results need to be passed across the network.
- machines can crash. What happens if you call a remote procedure and the remote machine crashes before returning?

RPC: *requirements*

- the goal of an RPC mechanism is to make remote procedure call look like a call to a local procedure.
  - in other words, we wish RPC to transparent.
  - calling a remote procedure should be no different to calling a local one.
- the caller and the called procedure should not have to know that anything special is going on.



RPC: *implementation (1)*

- consider the standard *read* system call.
  - when a normal read system call is made, the parameters being passed to *read* are put on the run-time stack and a trap is made into the kernel.
- now imagine that we want to perform a *read* system call, but we want the call to take place on a remote machine.
  - we want the mechanism to look the same as for the local call.
- instead of calling the *read* procedure directly, we call a ***stub***.

RPC *implementation (2)*

- the ***stub*** for *read* looks exactly like the normal *read*, except its implementation is entirely different.

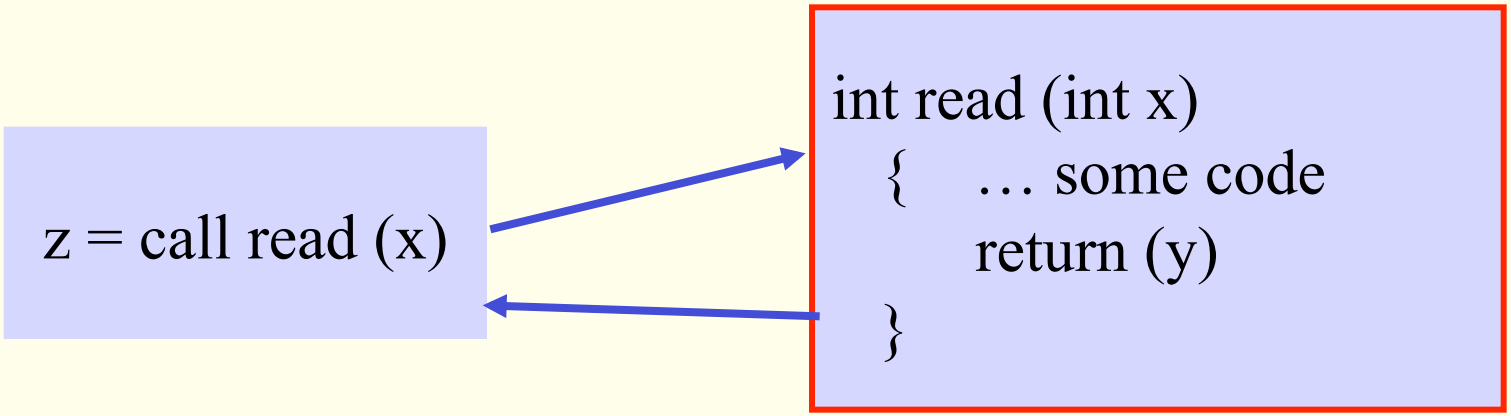
- the ***stub*** will put the parameters in a message and ask the kernel to send the message to the remote machine (server).

- following the send, the ***client stub*** waits to receive a reply message from the server.

RPC *implementation (3)*

- on the remote side, the kernel receives the incoming message and passes it to a *server stub* which is blocked waiting to receive a message.

- the *server stub* unpacks the parameters from the message and calls the server procedure *(read)* in the normal manner.



RPC *implementation (4)*

- when the remote *read* procedure returns, it returns to the server-side stub that called it.
  - any results produced by the procedure will be on the run-time stack (on the server computer!).

- the *server-side stub* packs the results into a reply message and calls the kernel to send the message to the client.

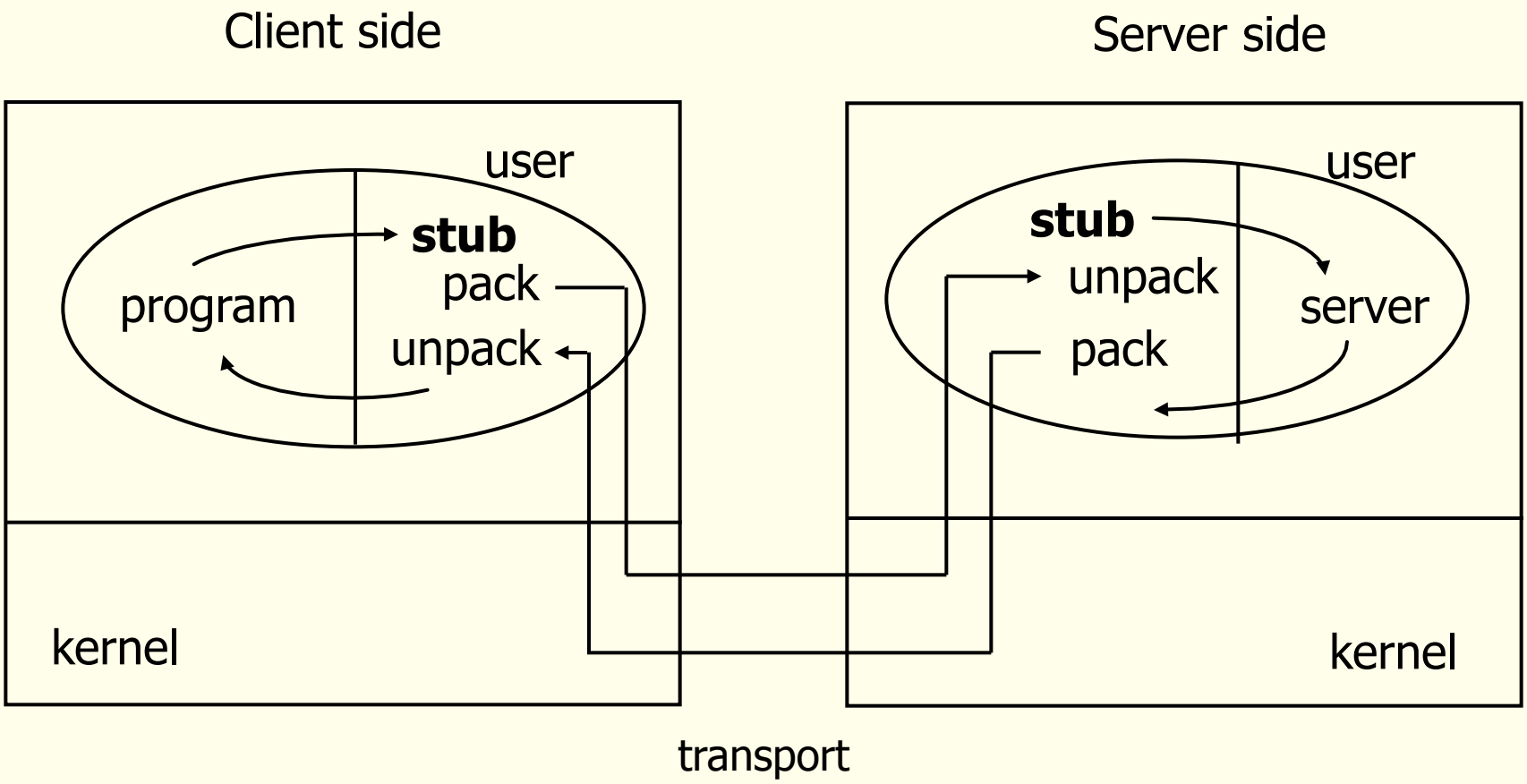
RPC *implementation (5)*

- the kernel *on the caller side* receives the reply message and passes it to the blocked ***client-side stub*.**

- the ***client-side stub*** unpacks the results from the message (onto the stack) and returns to the caller in the normal manner.

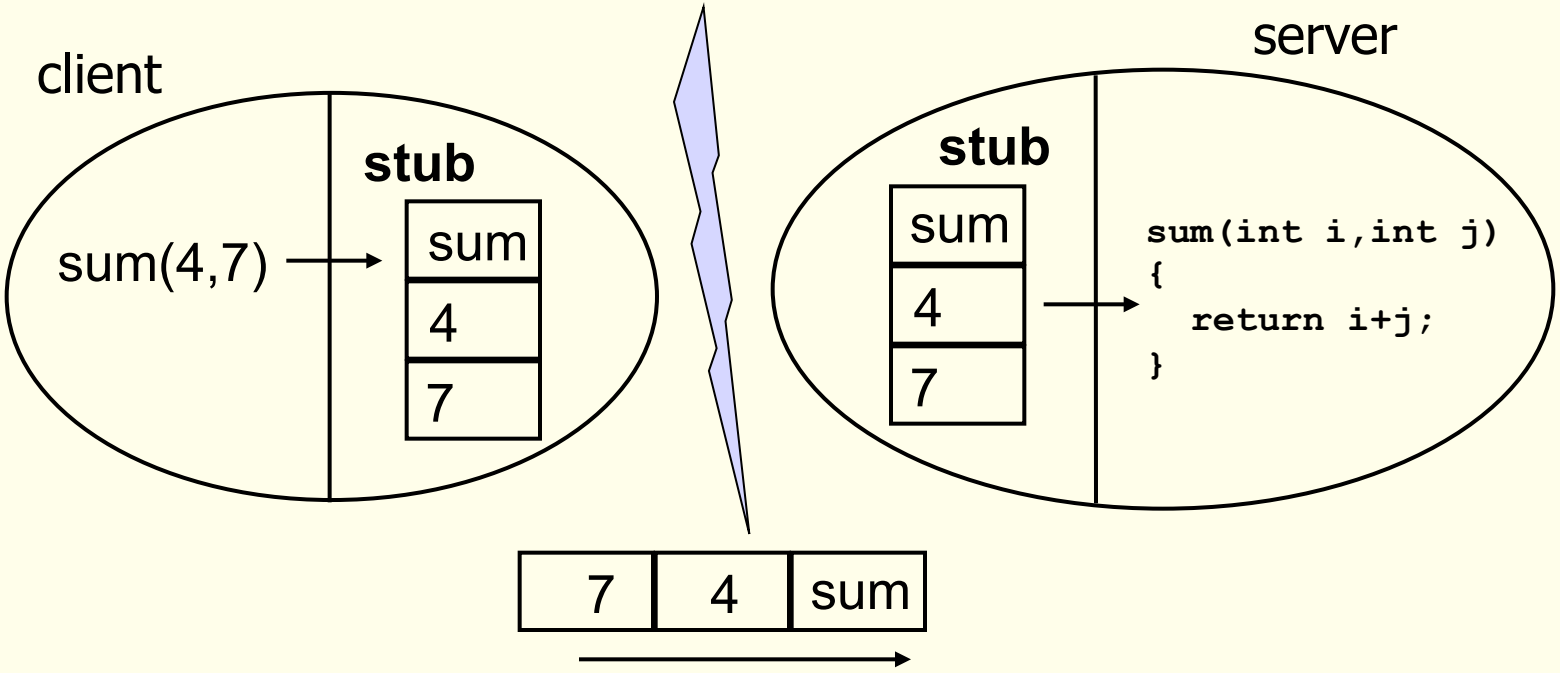
- and now for the diagram ...

RPC Operation



marshalling

- the client and server stubs are required to pack and unpack parameters to and from messages.
- this task is known as parameter marshalling.
- marshalling is a relatively complex activity.
- consider an example:

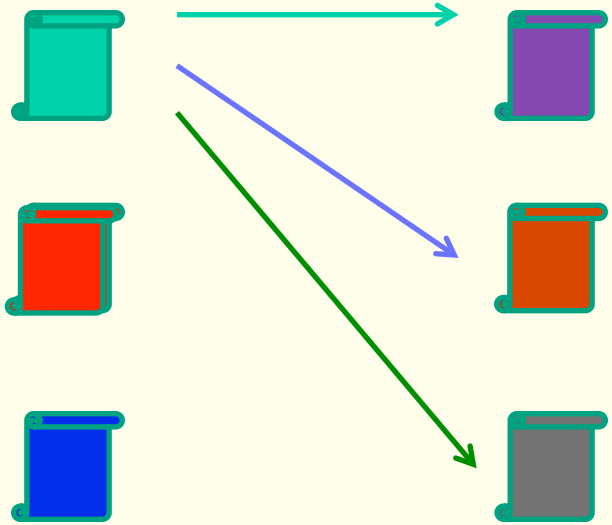


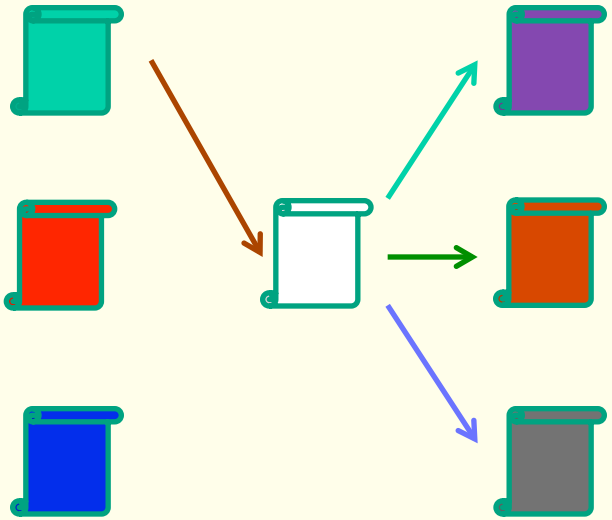
marshalling: *problems (1)*

- the server may implement multiple procedures and it may be necessary for the client to specify which procedure is to be invoked.
- different machines may use different character representations.
  - For example, ASCII vs EBCDIC vs Unicode.
- machines may represent integers differently:
  - Sun SPARC: Big-endian
  - DEC Alpha: Little-endian

marshalling: *problems (2)*

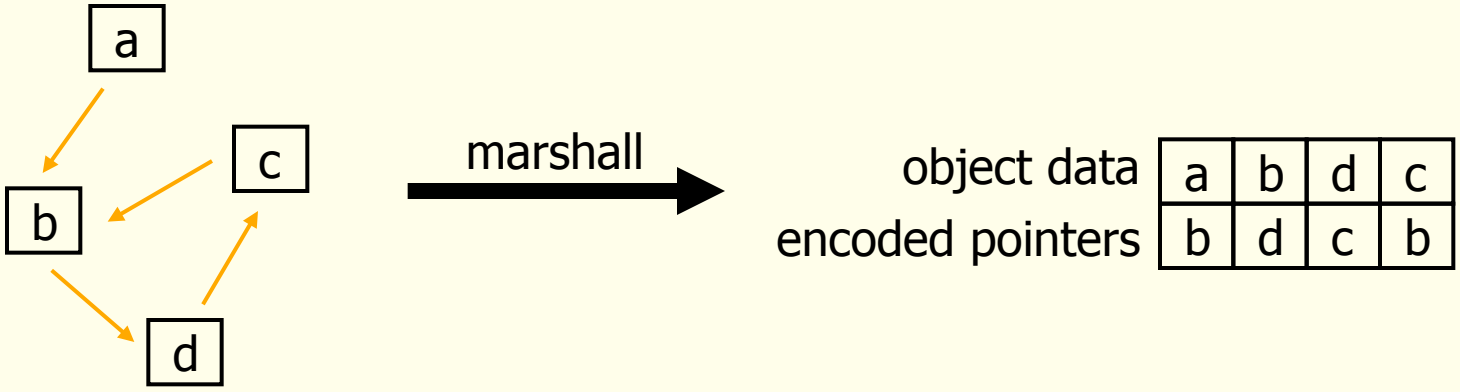
- floating point numbers may also be different.
- we must therefore send data from one machine to another in some *standard form*.
  - for example, the Sun XDR standard.





marshalling: problems (3)

- worst of all, any pointer-based data structures cannot be passed because client-side pointers will not be valid on the server.
- there are two main solutions to this problem:
  - forbid pointer-based data structures.
  - encode the data.
- when encoding the data, we need to be careful to avoid losing referential integrity:



RPC: *tools (1)*

- fortunately, we don't have to write the client and server stub code ourselves.
- instead we use tools that do most of the work for us.
- these tools are known as *interface generators* or *stub generators*.
  - the common one is *RPCGEN*.
  - also, Common Object Request Broker Architecture *(CORBA)*

RPC: *tools (2)*

- stub generators use an interface definition language (IDL) to describe the client and server code.
  - that is, what procedures may be called, what parameters do they take, and what is returned?

- common languages:
  - *RPCL* (Sun)
  - *IDL* (CORBA)

A Case Study: *Sun RPC*

- Sun RPC was developed by Sun Microsystems as part of its Open Network Computing (ONC) initiative.
- the other component developed at this time was the external data representation (XDR) standard.
  - this is used to overcome the differences in data formats between machines.

A Case Study: *Sun RPC*

- Sun RPC has been adopted by a number of vendors including DEC and HP.
  - thus, many Unix systems support Sun RPC and provide the appropriate tools for generating client and server stub code.
- Sun RPC is commonly used in the implementation of the Network File System (NFS).
  - the NFS protocol is actually specified in terms of Sun RPC and XDR.

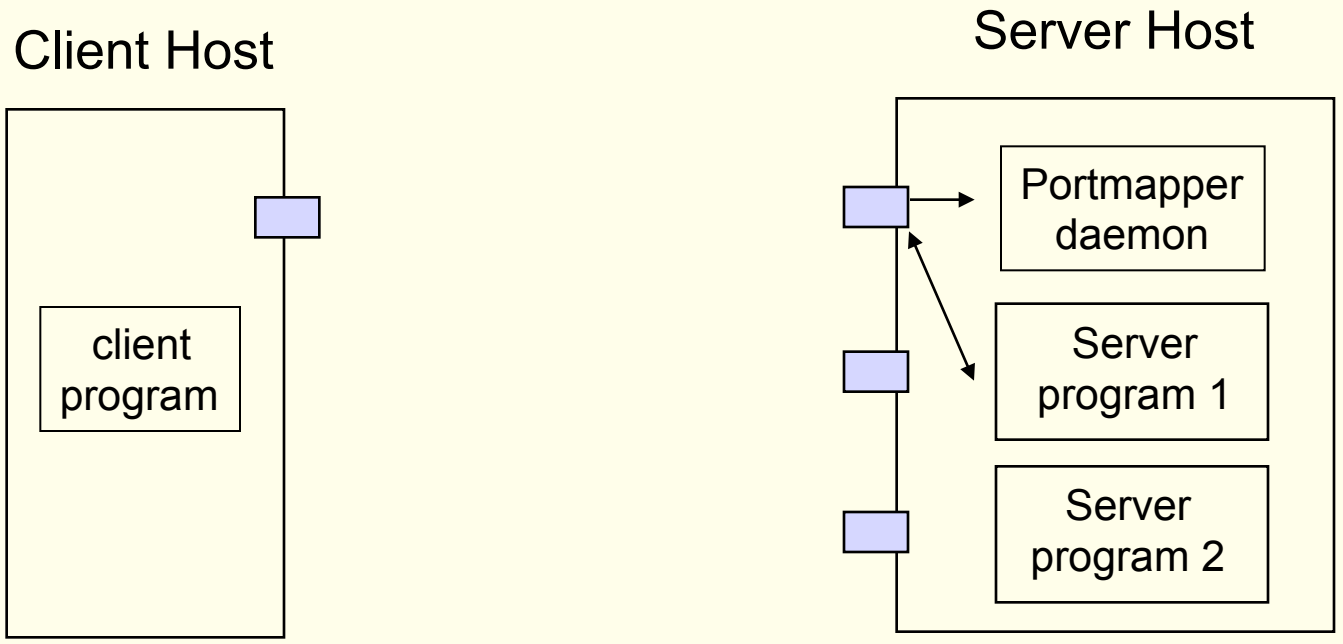
Sun RPC: *components (1)*

- RPCL
  - an interface definition language, used as input to RPCGEN.
- RPCGEN
  - a pre-compiler that takes RPCL and generates client and server code.
  - the output is in C.
- XDR
  - eXternal Data Representation: a standard for encoding transmitted data.

Sun RPC: *components (2)*

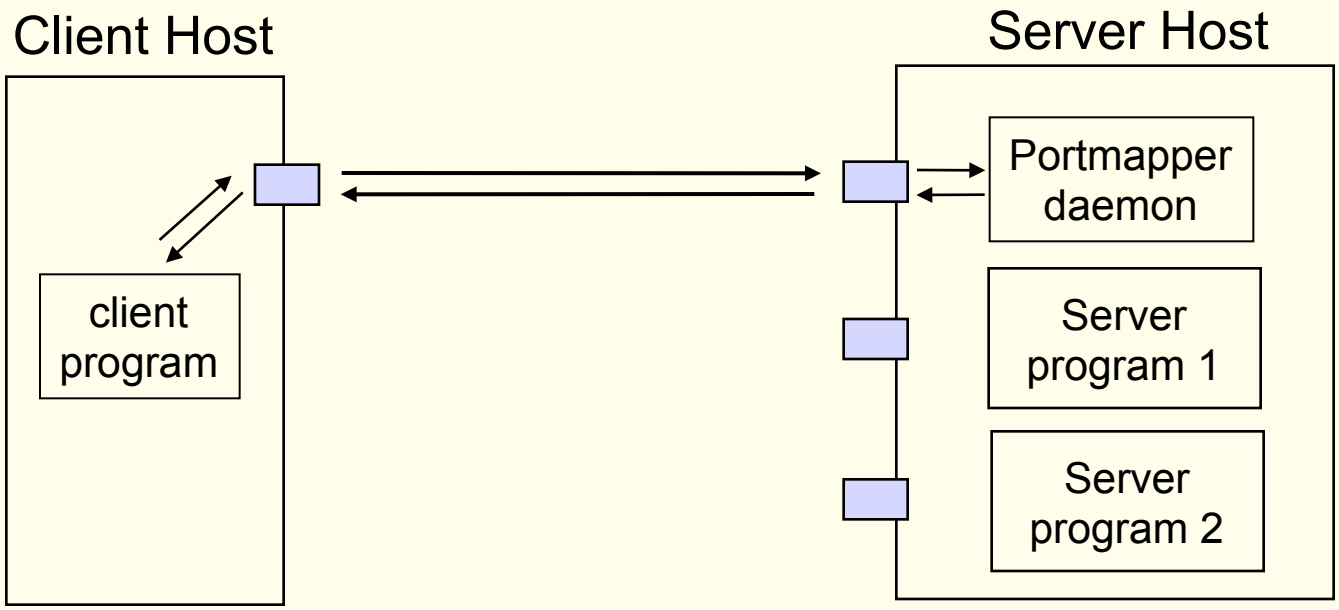
- RPC library
  - a library for handling all of the communication involved in building the RPC mechanism.
  - this uses sockets to effect communication.
- Portmapper daemon
  - an instance of the *portmapper* program exists on each machine.
  - it acts as a local binding agent (name space server) for making dynamic associations between clients and servers.

typical interaction using Sun RPC (1)



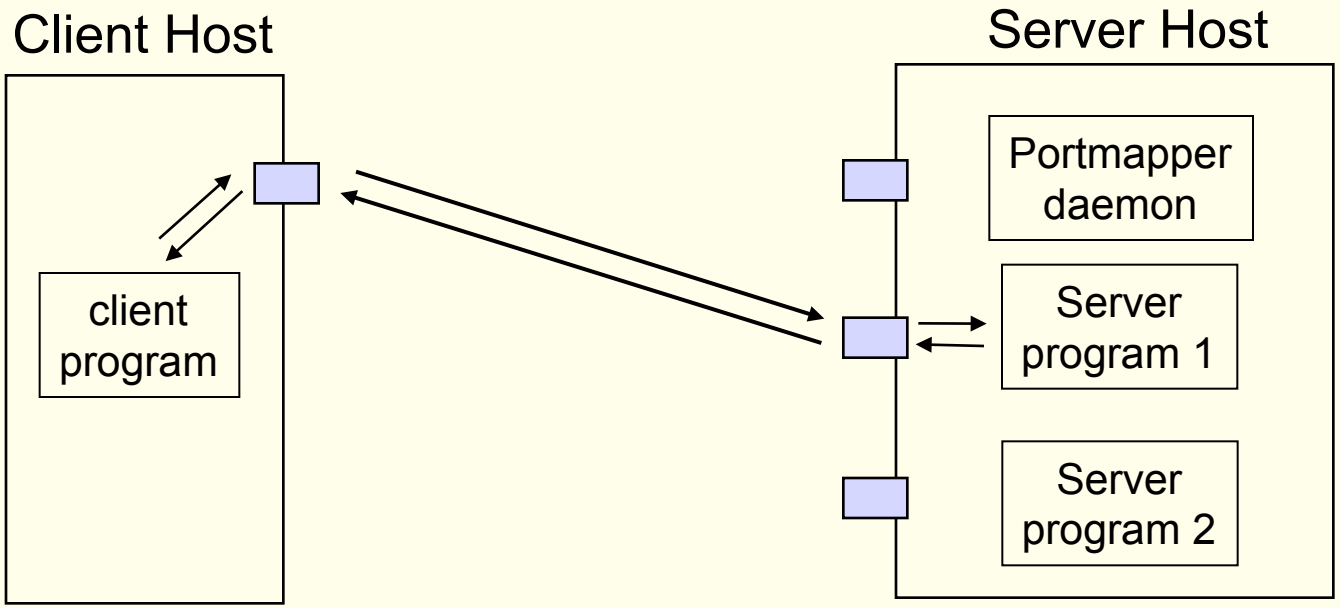
1. server program registers with the Portmapper daemon using a call to pmap_set().

typical interaction using Sun RPC (2)



2. Client program gets server's port from the remote Portmapper daemon using a call to pmap_getport().

typical interaction using Sun RPC (3)



3. client program calls server program using callrpc().
4. server program returns data using svc_sendreply().

- these steps are all carried out by the automatically generated client and server stubs.

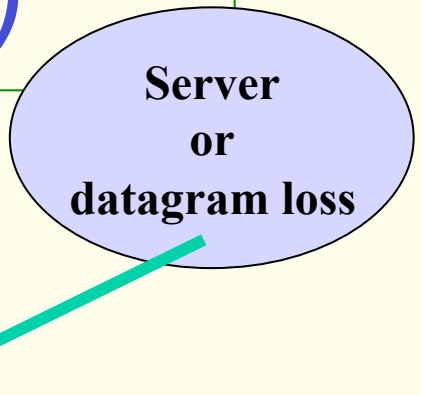
Sun RPC: *evaluation (1)*

- Sun RPC supports four different types of call, these are:
- Blocking:
  - This models the RPC paradigm described by Birrell and Nelson.
- Batching RPC:
  - This allows a series of calls to be made, the last of which must be non blocking to flush the queue. It was implemented to yield higher concurrency and less system call overhead

Sun RPC: *evaluation (2)*

- Broadcast
  - used when a client expects replies from multiple servers.
- Call back RPC
  - some applications need a server to become a client and call back on the original client. This is extremely dangerous since deadlock can easily occur if this is not done with extreme caution.

Transport: *Problems (1)*

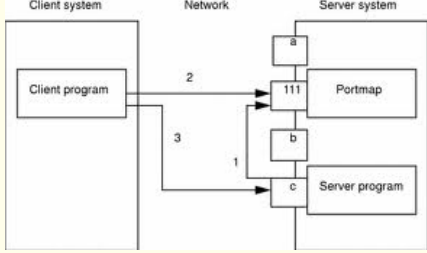


- pure RPC has *exactly-once* semantics.
- however, Sun RPC is implemented using TCP or UDP. In failure,
  - can recover using *at-most-once* semantics.
  - can recover using *at-least-once* semantics.
  - can mark remote procedure as *idempotent*

Transport: *Problems (2)*

- thus, Sun RPC depends on the underlying transport for its semantics which is not ideal.
  - Sun RPC therefore pollutes the pureness of the general RPC abstraction in that calls may not behave correctly in the event of a failure of either the server or the client.
- the semantics of RPC should be specified (and guaranteed) in a manner that is independent from the underlying transport mechanisms.

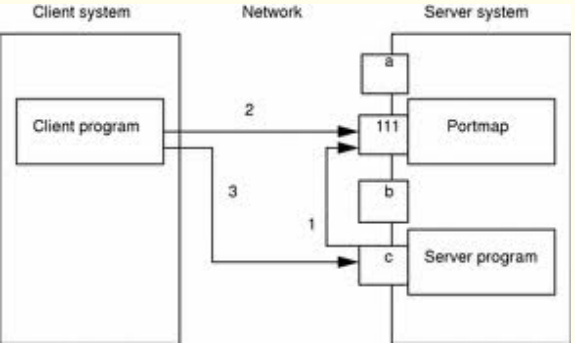
Portmapper: *Problems (1)*



- the portmapper does not detect when a registered process has died.
  - therefore the portmapper may return misleading information to a client.
  - this results in clients calling non-existent servers and a time-out occurring.
  - the portmapper could check the liveness of servers before sending out this information (it is a question of design as to whether or not it should).

Portmapper: *Problems (2)*

- when a server registers with the portmapper, the portmapper de-registers any service with the same name.
  - therefore, if an old service still exists it will receive no new connections from clients.
  - this is a serious security loophole which could be overcome by the portmapper implementing some form of ownership.



Portmapper: *Problems (3)*

- Portmappers implement no form of persistence or fault-tolerance.
  - when the portmapper crashes all registration data is lost.
  - this could be solved by replication.
  - therefore, the servers must re-register with the portmapper.