Last Frontier Application Serving

Peter Lo

Application Serving

- "The Ultimate Service"
  - Can deliver anything
  - Compare with
    - FTP - file transfer
    - SMTP - email
  - We've already seen NFS
    - File access using RPC

Application Serving (or sharing) is really the ultimate service because the general mechanism can be used to create any kind of service that we can imagine.

Compare this to very specific services like FTP (which only transfers files) or SMTP (which only handles email)

The one example we have seen of application sharing technology so far was RPC which was used by NFS.

What is being shared?

- The CPU of a remote machine
  - By causing execution on the remote machine we get work done
    - Instead of executing ourselves
    - Return value of function/procedure is result we want
  - NFS is a confusing example
    - Little actual remote execution
      - Just enough to read the file and send it back
      - Return value is content of file/directory

The general proposition is that we can get work done by asking a remote machine to do it. This lets us off the hook of having to execute everything ourselves while leaving things in terms of the program we run locally looking just like a function/procedure call with a return value. Transparently (this is the trick, the illusion) this particular call gets remotely executed.

When we look at NFS we do not immediately see that remote execution is taking place. Whilst RPC is a completely general mechanism for doing things on a remote machine NFS is a rather poor example of getting something done remotely because all that really happens is that the contents of a file get sent back.

Nevertheless, in the NetXRay lab on NFS we saw a remote procedure call being sent to the server and a return value, which in many cases was the content of a file or directory coming back to the client.

"Client-server"

- Term used to describe an architecture where processing is spread between locations
- Database access is classic example
  - Client expresses query in SQL (Structured Query Language)
  - Server does the work of searching
  - Resulting records returned to client

I have used the terms Client and Server throughout NPS but very often, in the context of application sharing they are used back to back - **Client/Server**.

What is being described here is an architecture where the work is spread between the two machines in a way that is intended to be better - from the point of view of workload, network traffic and/or ease of maintenance.

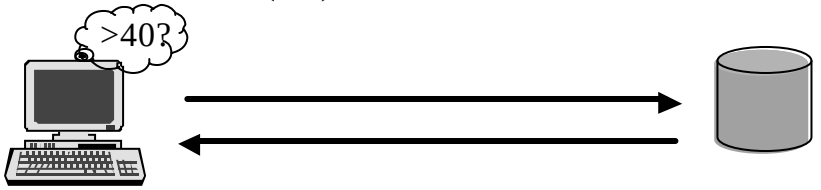
The classic example of client/server architecture is database access using SQL (Structured Query Language)

SQL is designed to take the load of searching a database away from the client and locate it at the server. The client simply sends a query in SQL and then waits for the server to send back the required records.

SQL Example

Question: "Find all clients older than 40"

Normal database (file) access



SQL database (client/server) access

The major benefit of this architecture is shown in this slide - the big picture view of this slide is to see the amount of network traffic that has been saved by the client/server architecture.

The conventional, file based model of access to a database requires the client to inspect each and every record to see if it fits a search criteria where as an SQL query can describe the criteria to the server which then does the searching on the client's behalf.

An example for us

- Remote arithmetic server
  - "Don't bother doing your own sums - get someone else to do them"
  - Trivial example
    - If the sums are easy!
  - Makes sense
    - For extremely complex calculations
      - Rendering animation video frames
      - Predicting the weather

The example that we are going to look at in more detail now and in the lab this week is the idea of a remote arithmetic server.

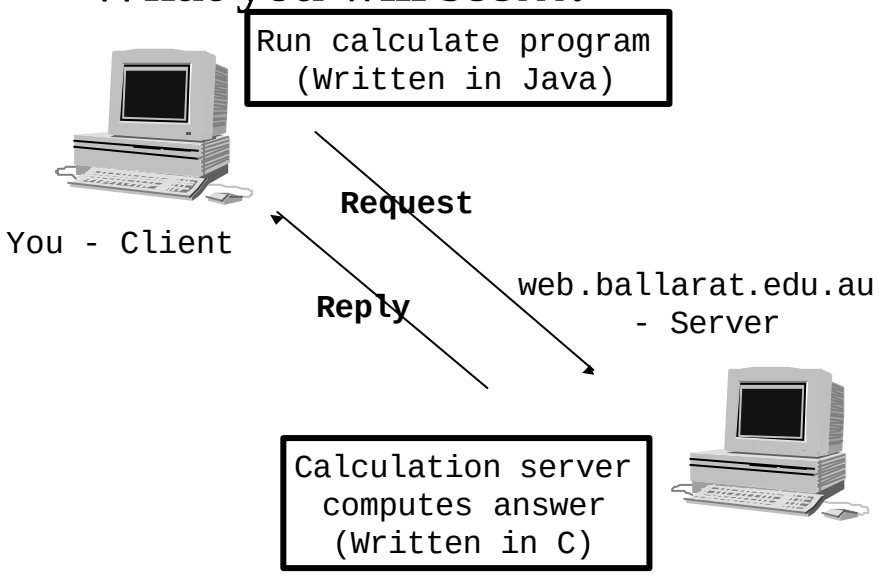
A remote arithmetic server will do the sums for you - which is a trivial thing if the sums are easy but if it were extended to much more complex calculations might be something worth while.

In the real world this technique would be applied to processes that were known to be very intense (computationally!) - the motivation being to send these enormous calculations to a machine that was specially built or configured to this kind of work and leave our humble 400 MHz Pentium to do something easier. A couple of examples of this scale of task are:

Rendering video frames (turning "Toy Story" from an computer animation into a set of frames in an hour long film - 3600 seconds * 70 - say frames per second * 10,000,000 - say dots per cinema screen = 2,520,000,000,000 dots to be calculated) is a big job and best not done on your desktop machine!

Predicting the weather - now moving out to 5/6/7 day forecasts is another hugely complex calculation.

What you will see….



In the lab you will be running a little “calculator” program written in Java and using that to send off calculation requests to a server (written in C) that will be running on the Uni web server.

You will use NetXRay to look at what passes between the two machines.

Define the interface

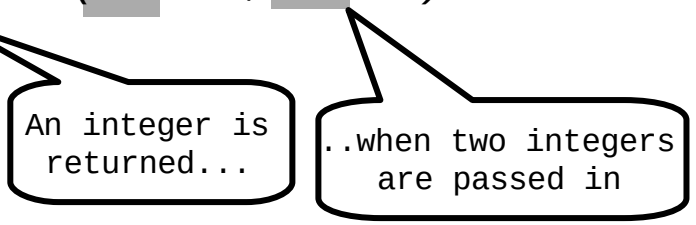
- Arithmetic server
- Has four procedures:
  - add
  - subtract
  - multiply
  - divide

The first step to distributing a task between two machines in this way to to very clearly state what the task is. This is known as defining the **interface.**

The interface to the calculation server is trivial - it can add, subtract, multiply and divide!

Say more about these

- *int add(int val1, int val2)*
- *int subtract(int val1, int val2)*
- *int multiply(int val1, int val2)*
- *int divide(int val1, int val2)*



Here (in C or Java) we see in more detail what each of these entail - all four return an integer result, and all four expect two parameters which in turn are integers.

How to write this program?

- We need to write two programs
  - A client which calls add(), subtract() etc
  - A server which **implements** these calls
- Client and server must contain everything needed to communicate with each other

The next step is to prepare ourselves to write **two** separate programs - the client program (we are going to use Java) which calls these four routines and the server program (we are going to use C) which actually **implements** these complex routines.

Clearly if these two programs are going to interact they must each contain all everything that they need in order to communicate with each other.

Three steps

- Define the interface
  - add(), subtract() etc
- Generate the stubs
  - Everything “remote” is in the stubs
  - Use a tool for this
- Write a client and a server
  - Two “local” programs

This process consists of three distinct steps:

Defining the interface

Which we have begun to do already.

Generating the stubs

The stubs are the glue of this client/server situation. All the remote communication parts of the program - both for the client end and the server end - will be in the stubs. Fortunately we will not have to write all this stuff ourselves. This part of the process has been automated with a software “tool” that writes all the stubs for us.

Write a client and a server

A bit of Java that will let us type in two values and select an operation (add, subtract….) and a bit of C that will perform an addition, a subtraction ….

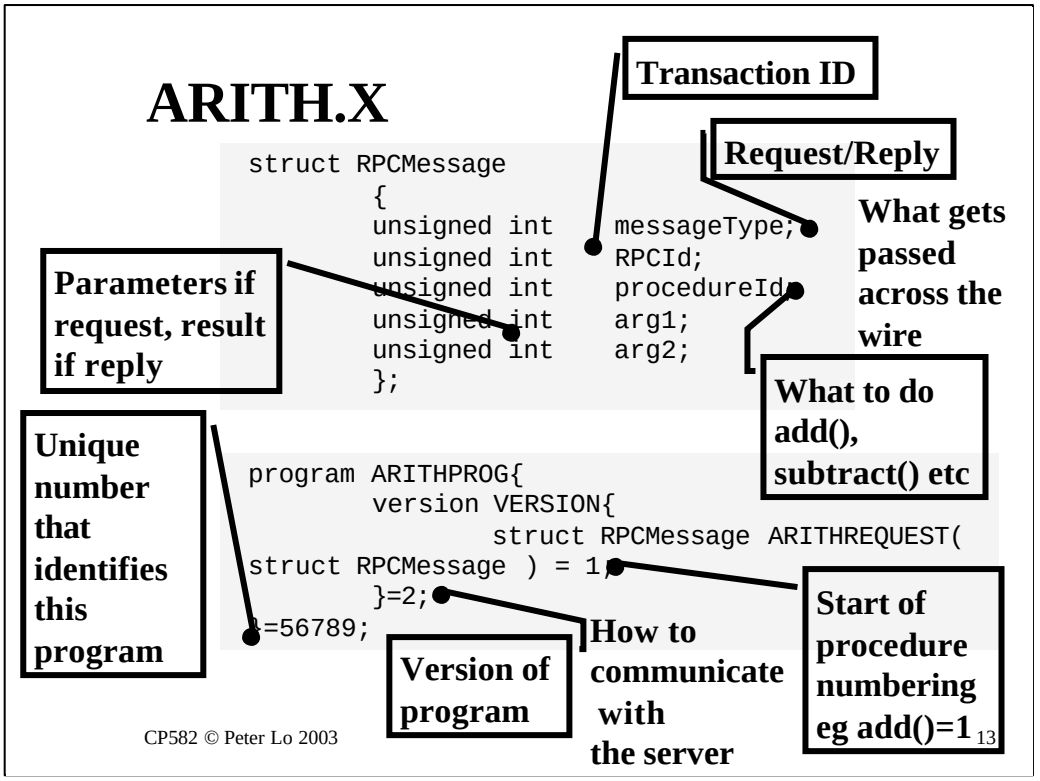
Both these programs can be written almost as if they were local programs because all the remote stuff is taken care of in the stubs.

Define the interface

- Use IDL (Interface Definition Language) to write a .X file
  - This defines what will be passed in and out of the remote procedure

Because we are about to use a tool to automate the generation of the stubs we have to express the interface in a standardised way that the tool can understand.

The standard way to define an interface is **Interface Definition Language (IDL)** and the task of IDL is simply to state formally what will be passed to and from the remote procedure.



This slide shows the IDL definition of the arithmetic interface.

There are two major parts to this:

Defining the communication packet

Just exactly will pass across the wire for each request - and reply?

• We need to pass across two integers with each request and get back an integer answer.

• We need to say whether to add or subtract etc

• For safety - in case messages get muddled - we need to say whether we are sending a request or a reply. You already saw request/reply flagged in the RPC header of NFS

• Another safety feature would be to count our sums - then if a packet gets lost we do not end up confusing the answer to one sum with the next.

All of these aspects are covered by the "struct RPCMessage" shown above - basically this the proposed layout for the messages that will be sent back and forth and you will be able to see this in NetXRay.

Define the server

Remember in NFS that the RPC "program ID" was 100003? Well this had to have been defined somewhere and here we are defining the remote program ID for the arithmetic server. I have picked "56789" - hopefully nobody else has!

Generate the Stubs

- What's a STUB?
  - Reality is that a program executes locally
    - Client needs something local that will send a remote request
    - Server needs something
      - to listen for requests
      - convert them into a local (i.e. on server) call
      - return the result
  - Data is "flat" when it crosses the wire
    - Complex data must be flattened at client
      - and unflattened at server

The **stub** that was referred to before has several jobs to do:

On the client:

• Send the request across the network to the server and wait for the reply

On the server:

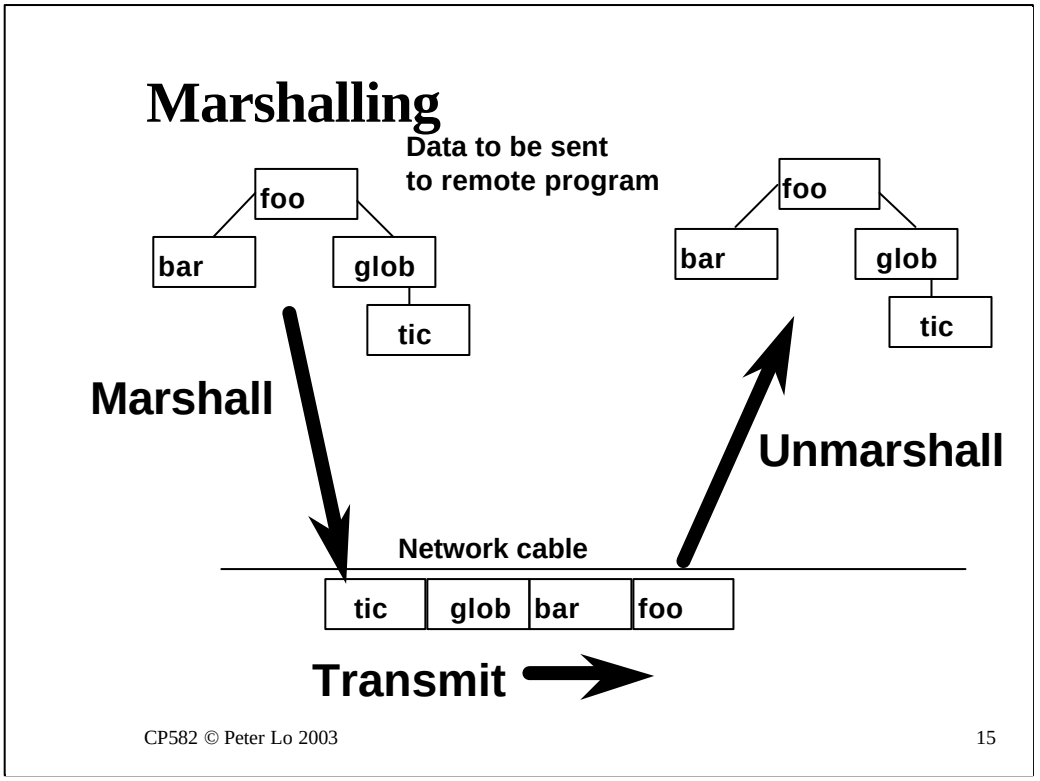
• Listen for request from clients

• Do the sum

• Send the reply across the network to the client

Both ends:

• Deal with complex data structures if they are part of what is being passed - clearly not the case in the arithmetic server but things could be more complex as the next slide will show.

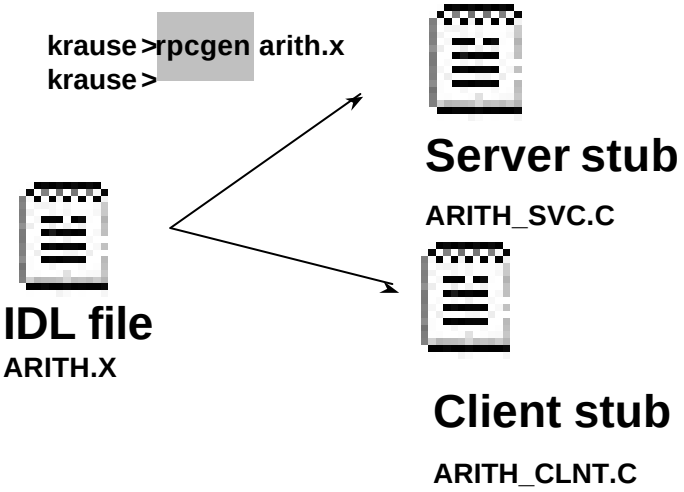


This animation shows a fundamental issue in remote execution - the **marshalling** of data.

When data is passing across the wire in packets it is a “flat” series of bytes yet within the memory of the client or server the data could well have some structure as shown.

Both clients and servers need to be able to **marshall** (flatten) and **un-marshall** (reconstitute) data that may be the return value or a parameter for a remote procedure call.

Generating Server & Client Stubs In C on krause

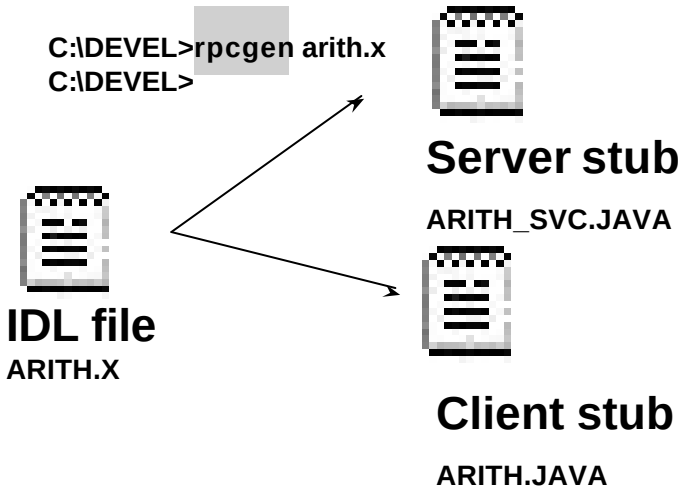


The tool that converts IDL files into communicating, marshalling stubs is called an RPC compiler. The **RPC compiler** on Unix is called **rpcgen**.

The input to **rpcgen** is an IDL file such as we saw earlier and the output is two automatically written C programs - one that does the work of a server stub and the other that can be used as a client stub. The names of these stubs are derived from the name of IDL file.

Since we have decided to use Java for the client we can throw away ARITH_CLNT.C. No big deal - we did not have to write it ourselves.

Generating Server & Client Stubs In Java on my PC



To create a client stub in Java we take the **same IDL file** and use a Java version or **rpcgen**. This time we will discard the server stub.

The really important thing that is going on here is that the same IDL file is being used. The IDL file uniquely defines the way that the component parts of our program will communicate - once this has been tied down it becomes irrelevant what language is used for the two halves of our program.

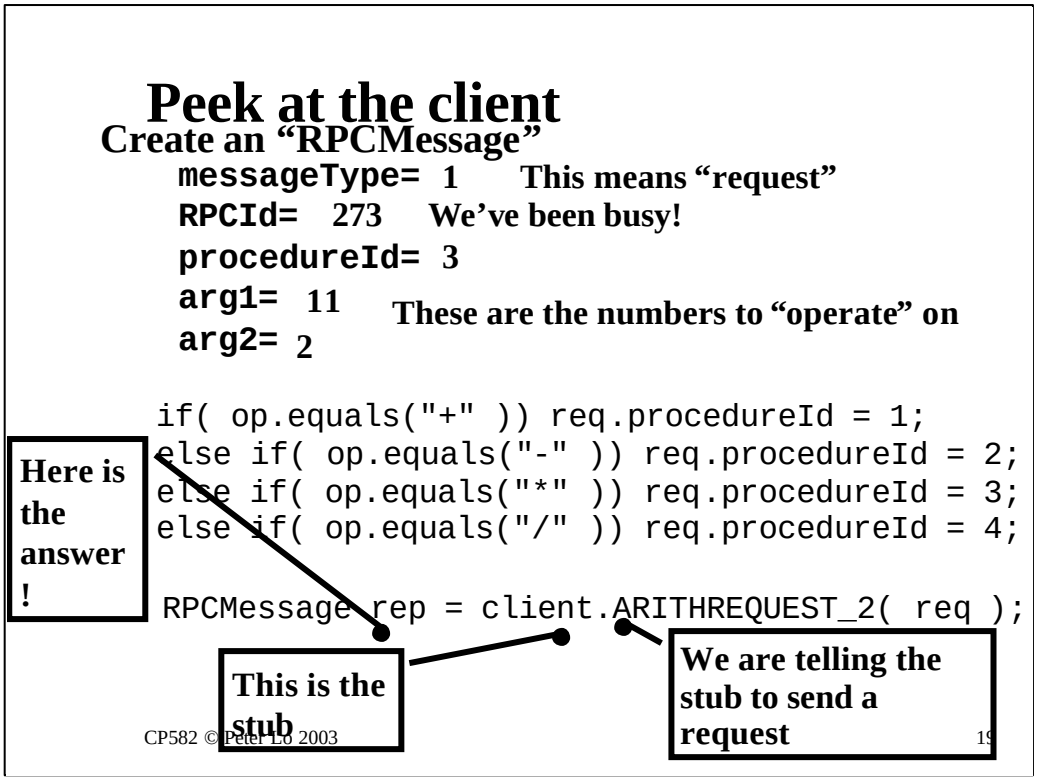
Write the program

- Client (JAVA) needs to call appropriate routine in client stub
- Server (C) needs to do what is required for each remote procedure

Now that the stubs have been generated the actual programs can be written.

The client needs a User Interface and that calls the routines in the client stub.

The server needs to implement these routines.



These next two slides are a peek into the working of the client and the server.

Both programs are focussed around the creation and decoding of the RPCMessage format that was defined in the IDL file. When you look at the traffic between the server and the client in NetXRay you too will be looking at these packets.

The client's job is to fill out one of these messages, requesting an action (+, -, etc) by setting the procedureID to an agreed value, and passing the values that are to be operated on. In NetXRay you will see these values being passed as "big-endian" four byte values. (eg 2 = 00 00 00 02, 11 = 00 00 00 0B)

To accomplish remote execution the client calls a routine that has been automatically generated in the stub - client.ARITHREQUEST_2 - passing in the filled out message.

What is returned by this routine (after all the remote communication and execution has taken place) is a new message - this time filled out with the answer!

Peek at the server

```
if( procedureID == 1 )
      result = arg1 + arg2;
else if(procedureID == 2 )
      result = arg1 - arg2;
else if(procedureID == 3 )
      result = arg1 * arg2;
else if(procedureID == 4 )
      result = arg1 / arg2;
```


The server's main job is to decide what it is being asked to do by inspecting the procedure ID in the incoming message.

The result is put into a new message and the server stub returns this to the client.

What's wrong with all this?

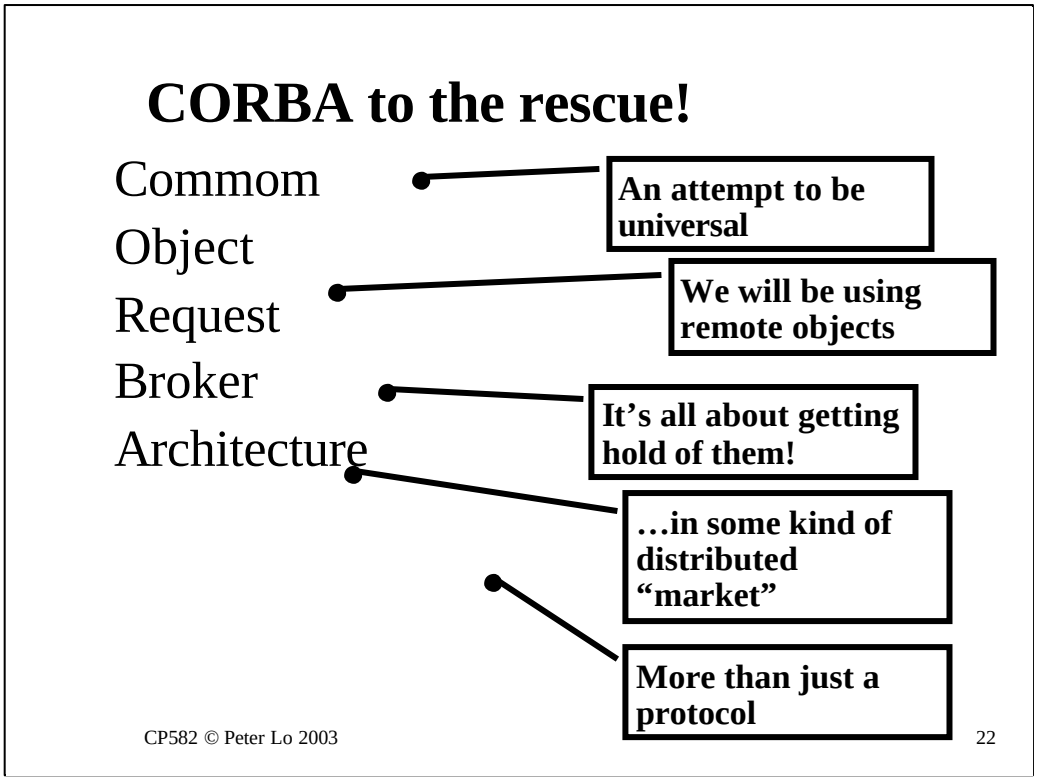
- Too many "special" numbers
  - "56789" is our program
    - "2" is "subtract"
  - "100003" is NFS
    - "4" is "ReadDir"
- Finding a server
  - "web" has been assumed
- What about more complex data?
  - Objects (as in Smalltalk or Java)

This all sounds incredibly neat and can potentially make it possible to write any imaginable program to be distributed amongst any number of separate computers but there are some drawbacks:

There are a lot of "special numbers" that have been used - the program ID ("56789" for the arithmetic server, "100003" for NFS) and the procedure IDs (2 for subtract, 4 for ReadDir in NFS) and unless clients get these right they will not be able to use the service.

Locating a machine that provides the service we want is another issue. If I had not told you where to go for arithmetic services how could you have found out?

There are also problems if we begin to look for more complex options in our interfaces such as Objects in Smalltalk or Java.



The answer to these problems is a more sophisticated approach to remote program execution named CORBA.

CORBA (which is an **architecture**) defines the way that clients and servers can get together in kind of online marketplace - hence the "Broker".

How is CORBA different?

- Too many “special” numbers
  - CORBA uses names
    - For programs
    - For procedures
- Finding a server
  - CORBA defines a naming service
- What about more complex data?
  - CORBA passes object references back and forth
  - CORBA objects inherit from each other

The Java client for the arithmetic service can ask for the calculation via RPC or via CORBA.

CORBA answers the problems that we set out for RPC:

Too many special numbers

CORBA uses names - for servers and for remote procedures. You will be able to see this in NetXRay when you look at the packets in a CORBA request.

Finding a server

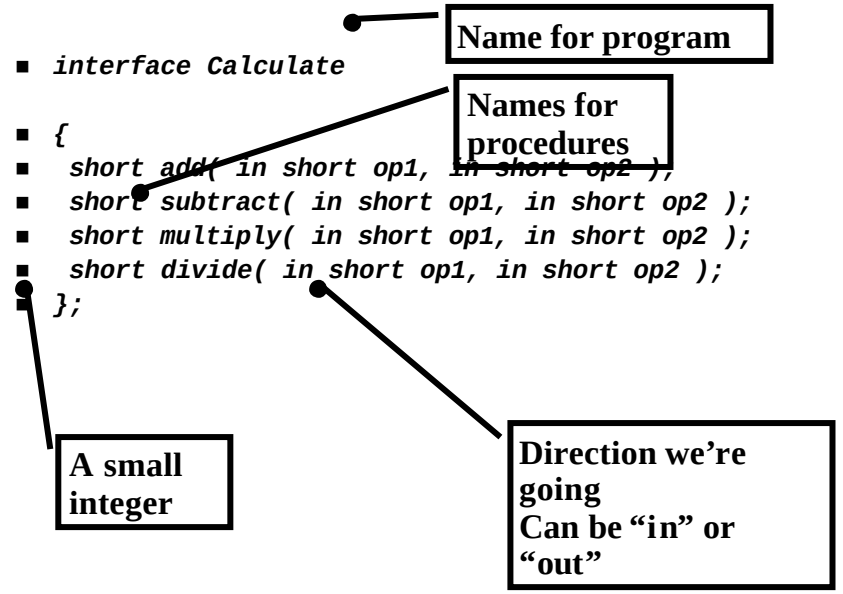
The client you are using has the CORBA server name welded into it but there is a CORBA naming service that would let a client simply say “ I want to calculate” and to receive back the address of a calculate server.

Complex data

CORBA recognises terms like “objects” and “methods”. What the client receives in the first instance (when they connect to the server) is a “remote object reference” on which they can then call the methods that are available.

If you have time in the lab you can look at the CORBA communication between two NetSim clients and see some of this in operation.

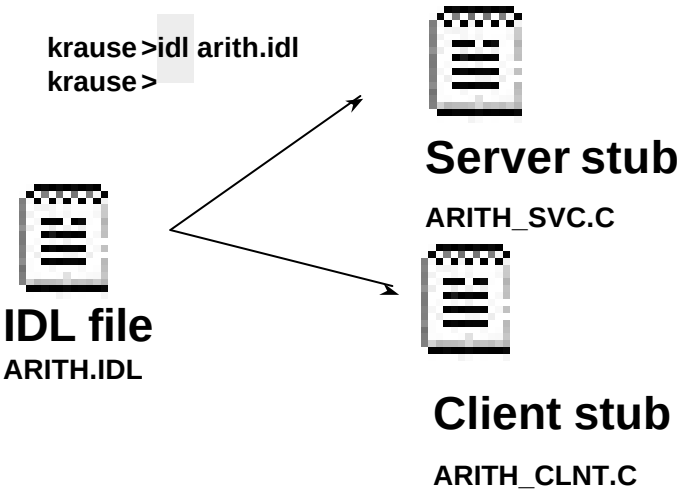
ARITH.IDL



In CORBA the IDL file looks different - but it still saying the same basic thing.

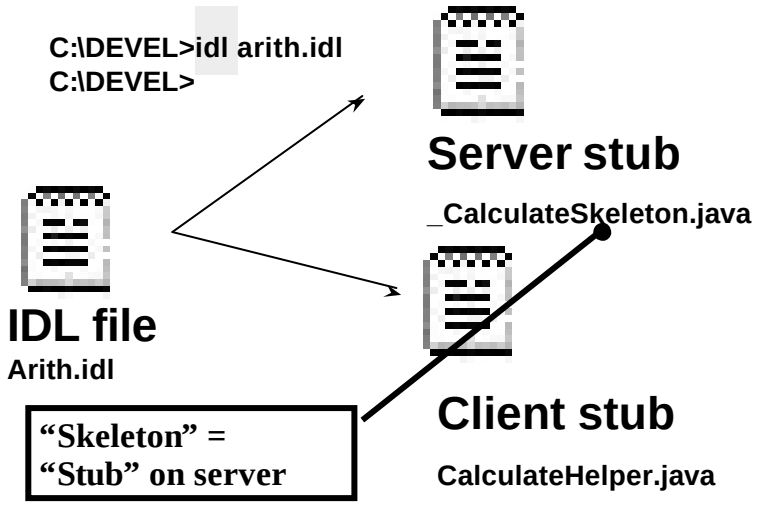
What is being defined is the **interface** of a remote service and the sub-sections are the methods that will be implemented in that interface.

We still need stubs!
In C on krause



Generation of stubs is a similar process - except that we use a CORBA IDL compiler - the name of this program is **idl (!)**

…or
In Java on my PC



In fact the server for the CORBA option is written in Java as well so both "stubs" generated here have been used. Note that CORBA uses different terminology: for the server stub - **skeleton,** for the client stub - **helper**

CORBA Client

```
if( op.equals("+" ))
     result = calc.add(val1, val2 );
else if( op.equals("-" ))
     result = calc.subtract( val1, val2 );
else if( op.equals("*" ))
     result = calc.multiply( val1, val2 );
else if( op.equals("/" ))
     result = calc.divide( val1, val2 );
```


Code looks cleaner - all the special numbers have gone.

What we write at the client is now almost indistinguishable from a purely local program.

Application sharing - So what?

- "The mother of all services"
  - Using CORBA, or RPC we can build anything
    - Clothing evaluation service
    - Remote banking…
- Foretaste of Distributed Systems units
  - Will be writing distributed applications using:
    - CORBA, RPC, DCOM, RMI…..
    - Java, C, Smalltalk, VB…..

The application serving topic in NPS is really the end of NPS in terms of looking at more and more services. This is a service that can be used to build other services (as we already saw with NFS) and in this sense we are not just at the end of NPS - we are at the beginning of something else.

The something else is the two Distributed Systems units which will investigate the writing of distributed programs using a variety of architectures for distribution.

CORBA and RPC have been talked about here - DCOM (Distributed Common Object Model) is Microsoft's way to do it, RMI (Remote Method Invocation) is a pure Java solution.

It should also be possible to use a variety of languages in these units - other than Java, which will be a pre-requisite you should be able to create distributed applications using a language of your choice.