

# Lesson Preview

## RPC

- Remote Procedure Calls
- "Implementing Remote Procedure Calls" by Birrell and Nelson

1. In the previous lessons, we discussed several mechanisms for Inter-process Communication. But we said that these were fairly low-level mechanisms because they focused on providing the basic capability for moving data among address spaces. And didn't really specify anything about the semantics of those operations or the protocols that are involved. In this lesson we will talk about Remote Procedure Calls, or RPC. This is an IPC mechanism that specifies that the processes interact via procedure call interface. For the general discussion of RPCs, we will roughly follow Birrell and Nelson's paper, Implementing Remote Procedure Calls. This is an older paper, but it discusses very nicely the general design space of RPC. Then, we will discuss in more detail Sun RPC. It's a concrete implementation of an RPC system, that's common in operating systems today.

## Why RPC?



### Example 1: GetFile App

- client - server
- create and init sockets
- allocate and populate buffers
- include 'protocol' info
  - GetFile, size, ...
- copy data into buffers
  - filename, file...



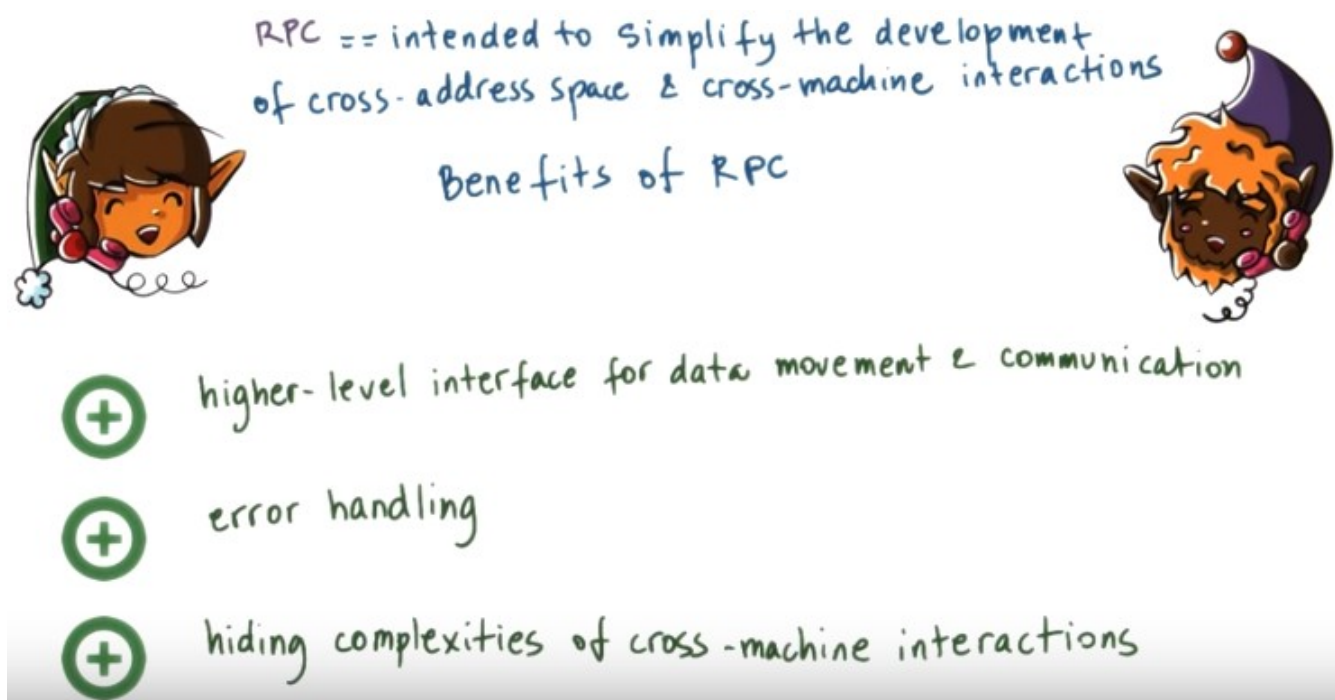
### Example 2: ModImage App

- client - server
- create and init sockets
- allocate and populate buffers
- include 'protocol' info
  - algorithm, parameters
- copy data into buffers
  - image data

common steps related to remote IPC  $\Rightarrow$  Remote Procedure Calls (RPC)

2. To understand why we need RPC, let's look at two example applications. The first one is an application, where a client requests a file from a server. And uses a simple get file protocol that's like HTTP request, but less complex. In this application the client and the server interact using a socket

based API. And as a developer you would have to explicitly create and initialize the sockets, then allocate any buffers that are going to be sent via those sockets, and populate them with anything that includes protocol related information. Like for instance, this protocol will have something like, get file directives. And you have to specify the size of the buffer. And also you'll have to explicitly copy the data in and out of these buffers. So copy the file name, string or the actual file in and out of these buffers. Now imagine another application that's also a client server application in which the client interacts with a server to upload some images and that it requests them from the server for these images to be modified. To create a gray scale version of an image, to create a low resolution version of an image, to apply some phase detection algorithm. So it's in some sense similar to get file, but there are some additional functionalities, some additional processing that needs to be performed for every image. The steps that are required from the developer of this particular application are very similar, in fact some of them are identical to the steps that are required in the get file application. One difference is that the protocol related information that would have to be included in the buffers would have to specify things like the algorithm that the client is requesting from the server to be performed. Like whether it's grace key link or whether its some face detection algorithm along with any parameters are relevant for that algorithm. And also the data that is being sent between the client and the server, we said in this case the client uploads an image to the server and then the server returns that image back to the client after this particular function has been performed. That's different than the file name, the string that's being sent from the client to the server, and the actual file that's returned in response. But a lot of the steps end up being identical in both cases. In the 80s, as networks are becoming faster, and more and more of distributed applications were being developed. It became obvious that these kinds of steps are really very common in a related interprocess communications, and need to be repeatedly reimplemented for a majority of these kinds of applications. It was obvious that we need some system solution that will simplify this process that will capture all the common steps that are related to remote interprocess communications. And this key do Remote Procedure Calls or RPC.

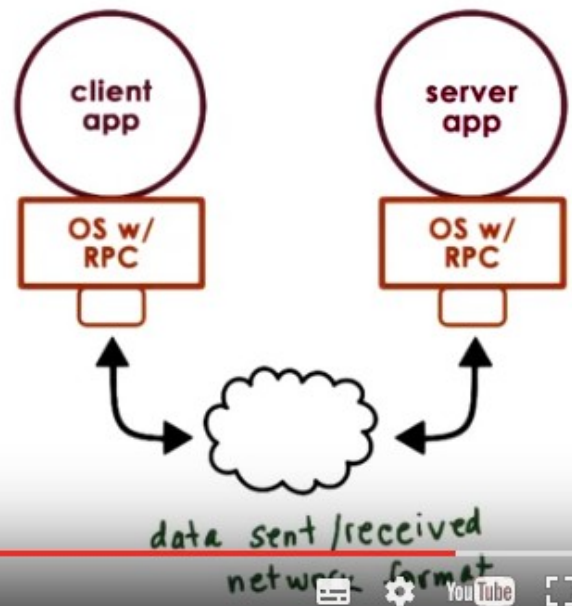


3. RPC is intended to simplify the development of cross-address space and/or cross-machine interactions. So what are the benefits? RPC offers a higher-level interface that captures all aspects of data movement and communications, including communication establishment, requests, responses,

acknowledgements, et cetera. What this also allows it permits for RPC's to capture a lot of the error handling and automated, and the programmer doesn't really have to worry about that. Or at least the programmer doesn't have to explicitly reimplement the handling of all types of errors. And finally, another benefit from RPC is that it hides the complexities of cross-machines interactions, so the fact that machines may be of different types, that the network between them may fail, that the machines themselves may fail. That will be hidden from the developer. So as a programmer, when using RPC, we don't have to worry about those differences.

## RPC Requirements

1. Client / Server Interactions
2. Procedure Call Interface  $\Rightarrow$  RPC
  - sync. call semantics
3. Type Checking
  - error handling
  - packet bytes interpretation
4. Cross-Machine Conversion
  - e.g., big / little endian
5. Higher-level Protocol
  - access control, fault tolerance...
  - different transport protocols



From wiki:

端序(endian)

在幾乎所有的機器上，多位元組對象都被存儲為連續的位元組序列。例如在 C 語言中，一個類型為 int 的變量 x 地址為 0x100，那麼其對應地址表達式 &x 的值為 0x100。且 x 的四個位元組將被存儲在存儲器的 0x100, 0x101, 0x102, 0x103 位置。

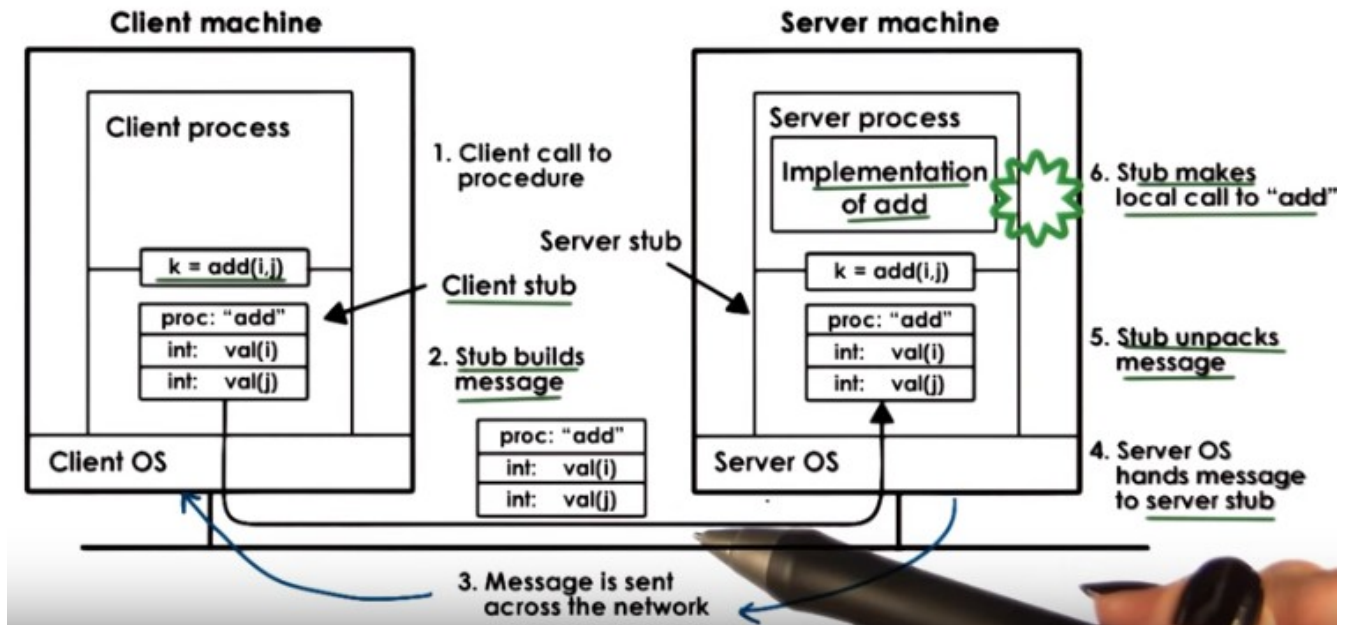
而存儲地址內的排列則有兩個通用規則。一個多位的整數將按照其存儲地址的最低或最高位元組排列。如果最低有效位在最高有效位的前面，則稱小端序；反之則稱大端序。在網絡應用中，位元組序是一個必須被考慮的因素，因為不同機器類型可能採用不同標準的位元組序，所以均按照網絡標準轉化。

例如假設上述變量 x 類型為 int，位於地址 0x100 處，它的十六進位為 0x01234567，地址範圍為 0x100~0x103 位元組，其內部排列順序依賴於機器的類型。大端法從首位開始將是：0x100: 01, 0x101: 23,...。而小端法將是：0x100: 67, 0x101: 45,...。

4. Let's see what's required from the system software that provides support for RPCs. First, the model of inter-process interactions that the RPC model is intended for needs to manage [client/server interactions](#). A server supports some potentially complex service. Maybe it's running a complex computation but really fast, or maybe it's a file service that services a remote file content. The clients do not need to have the same capabilities or they don't have to be able to perform accesses to the same

data. They just need to be able to issue requests to the server for whatever they need. The second requirement has to do with the fact that when RPC was first developed, the state-of-the-art programming languages for procedural languages including Basic and Pascal, and Fortran and C. So this is what programmers were familiar with. One goal of the RPC systems was to simplify the development of distributed applications underneath a [procedure called interface](#). This is why the term remote procedure calls came to be. As a result, [RPCs are intended to have similar synchronous semantics. Just like regular procedure calls. What that means is that when a process makes our remote procedure calls, the calling process or at least the calling thread, will block, and wait until the procedure completes, and then returns the result. This is the exact same thing that happens when we call a procedure in a single address base.](#) The execution of the thread will reach the point when the procedure call is made. At that point, we jump somewhere in the address base where the procedure is implemented. The actual original thread of execution will not advance beyond that procedure called point until we get the results from the procedure. And then when we move on to the next step, we act, actually already have the results. [So this is what we mean by the synchronous call semantics.](#) And this is what we require from the RPC systems as well. RPCs have other useful features that are similar to regular procedure calls, and one is [type checking](#). [If you pass to a procedure an argument of the wrong type, you'll receive some kind of error. This is one useful reason why RPC systems would incorporate type checking mechanisms. In addition, type checking mechanisms are useful because they allow us, in certain ways, to optimize the implementation of the RPC run-time. When packets are being sent among two machines, it's just a bunch of bytes that reach from one endpoint to another. And some notion about the types of the data that's packaged into those bytes can be useful when the RPC run-time is trying to interpret what do these bytes mean. Are they integers or they're file. Do I need to put them together so as to create some kind of image or some kind of array. This is what the type information can be used for.](#) Since the client and the server may run on [different machines](#), there may be differences in how they represent certain data types. For instance, machines may differ in the way they use big endian or little endian format to represent integers. This determines whether the most significant byte of that integer is in the first or the last position in the sequence of bytes that corresponds to the integers. Or machines may differ in their representation of floating point numbers, may use different representations for negative numbers. [The RPC system should hide all of these differences from the programmer,](#) and should make sure that data is correctly transported, and it must perform any of the necessary conversions, any of the necessary translations among the two machines. [One way to deal with this conversion is for the RPC run-time in both endpoints to agree upon a single data representation for the data types. For instance, it can agree of that everything will be represented in the network format. Then there is no need for the two endpoints to negotiate exactly how data should be encoded, exactly how data should be represented. Finally, RPCs intended to be more than just a transport-level protocol like TCP and UDP that worries about sending packets from one endpoint to another in an ordered reliable way. RPC should support underneath different kinds of protocols, so we should be able to carry out the same types of client-server interactions, regardless of whether the two machines use UDP or TCP, or some other protocol to communicate. But RPC should also incorporate some higher level mechanisms like access control, or authentication, or fault tolerance. For instance, if a server is not responding, a client can retry and reissue their same request to either the same server or it can make an attempt to contact the replica of that original server that it was trying to contact.](#)





Stub: 殘余,存根

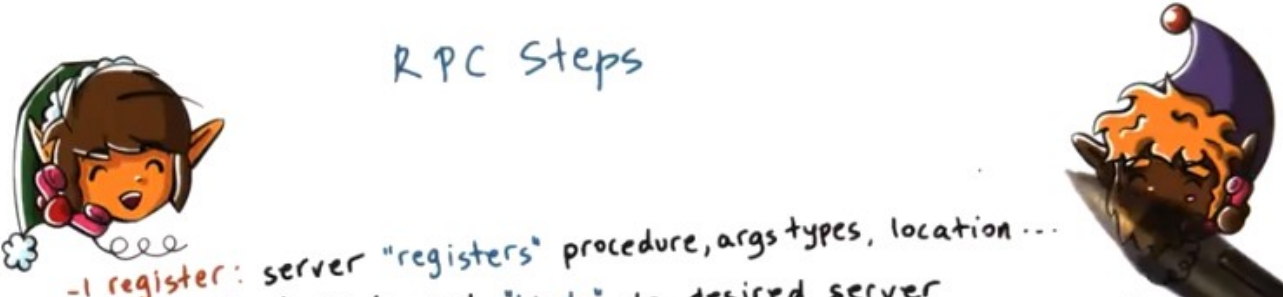
可以仔細看看上圖中 stub 的作用

本段雖長, 卻很好懂, 廢話多:

5. To illustrate the structure of the RPC system I will walk you through an example. Consider a client and server system. The client wants to perform some arithmetic operations, let's say addition, subtraction, multiplication, but doesn't know how to. The server is the calculator process, and it knows how to perform all of these operations. In this scenario, whenever the client needs to perform some arithmetic operation. It needs to send the message over to the server that specifies what is the operation it wants performed, as well as the arguments. The server is the one that has the implementation of that operation. So it will take those arguments, perform the operation, and then return the results. To simplify all the communications related aspects of the programming, like creating sockets, allocating managing the buffers, for the arguments and for the results, and all the other detail, this communication pattern will use RPC. Let's consider in this example the client wants to perform an addition. It wants to add i and j and it wants to obtain the results of this computation in k. The client doesn't have the implementation of the addition process, only the server knows how to do it. However, with RPC the client is still allowed to call something that looks just like a regular procedure k equals add of i and j. In a regular program, when a procedure call is made the execution jumps to some other point in the address space where the implementation of that procedure is actually stored. So the program counter will be set to some value in that address space that corresponds to the first instruction of the procedure. In this example, when the RPC add is called the execution of the program will also jump to another location in the address space. But it won't be where the real implementation of add is, instead it will be in a stub implementation. From the rest of the client's process it will look just like the real add, but internally what this stuff does is something entirely different. The responsibility of the client's stub is to create a buffer and populate that buffer with all of the appropriate information. In this case, it's the descriptor of the function that the client wants the server to perform, the add, as well as its arguments, the integers i and j. The stub code itself is automatically generated via some tools that are part of the RPC package so the programmer doesn't have to write this code. So when the client makes the call add() here the call takes the execution of the client process into a portion of the RPC run time, and by

that we mean the system software that implements all of the RPC functionality. In the first step here is that stub implementation. After the buffer is created, the RPC run time will send a message to the server process. This may be the TCP/IP sockets or some other transport protocol. What we're not showing in this figure is that there is some information about the server machine like the IP address and the port number where this server process is running. That is available to the client. And that information is used by the RPC run time to establish the connection, and to carry out all of the communication. On the server side when the packets are received for this connection they will be handed off to the server stub. This (server stub) is a code that will know how to parse and interpret all the received bytes in the packets that were delivered to the stub and it will also know how to determine that this is an RPC request for the procedure add with arguments i and j. The servers stub once it sees that it needs to perform this add(). It will know that the remaining bytes need to be interpreted like two integers i and j. So it will know how many bytes to copy from the packet stream, how to allocate data structures for these particular integer variables to be created in the address piece of the server process. Once all this information is extracted on the server side these local variables are created in the address space. The stub is ready to make a call in the user level server process that has the actual implementation of all of the operations, including the add. Only at that point did the actual implementation of the add procedure will be called and the results of the edition of i and j will be computed and stored in a variable in the server process address space at that point. Once the result is computed, it will take the reverse path. It will go through the server step that will first create a buffer for that result and then it will send the response back via the appropriate client connection. That will arrive on the client side into the RPC run time. The packets will be received. The result will be extracted from those packets by the client side stub, be placed somewhere in memory in the client address space, and then ultimately the procedure will return to the client process. For the entire time while this is happening, the client process will be blocked on this add operation will be suspended here. It will not be able to continue. Which is exactly what happens when a client process makes a local procedure call. The execution of the client process will continue only once the results of that procedure call are available.

## RPC Steps



- 1 register: server "registers" procedure, args types, location ...
0. bind: client finds and "binds" to desired server
1. call: client makes RPC call; control passed to stub, client code blocks
2. marshal: client stub "marshals" arguments (serialize args into buffer)
3. send: client sends message to server
4. receive: server receives message; passes msg to server-stub; access ctrl
5. unmarshal: server stub "unmarshals" args (extracts args & creates data structs)
6. actual call: server stub calls local procedure implementation
7. result: server performs operation and computes result of RPC operation

... similar steps on return

2:02 / 2:16

Marshal: 排列,集合,元帥

上圖對 marshal 的作用說得好, 即 `serialize args into buffer`. 注意 marshal 和 unmarshal 也是 stub 做的.

6. To generalize from the example that we saw in the previous video. We will now summarize the steps that have to take place in an RPC interaction between a client and a server. The first step, a server binding occurs. Here the client finds and discovers the server that supports the desired functionality. And that it will need to connect to. For connection oriented protocols, like TCP/IP that require that a connection be established between the client and the server process, that connection will actually be established in this step. Then, the client makes the actual Remote Procedure Call. This results in a call into the user stub, and at that point the rest of the client code will block. Next, the client stub will create a data buffer, and it will populate it with the values of the arguments that are passed to the procedure call. We call this process, marshalling the arguments. The arguments may be located at arbitrary non-contiguous locations in the client under space. But the RPC runtime will need to send a contiguous buffer to the sockets for transmission. So the marshal link process will take care of this and replace all the arguments into a buffer that will be passed to the sockets. Once the buffer is available, the RPC run time will send the message in the sending will involve whatever transmission protocol that both sides have agreed upon during the binding process. This may be TCP, UDP, or even shared memory based IPC if the client and the server are in the same machine. When the data is transferred onto the server machine, it's received by the RPC runtime and all of the necessary checks are performed to determine what is the correct server step that this message needs to be passed to. And in addition, it's possible to include certain access control checks at this particular step. The server stop will unmarshal the data. Umarshalling is clearly the reverse of marshalling. So this will take the byte stream that's coming from the receive buffers. It will extract the arguments and it will create whatever data structures are needed to hold the values of those arguments. One of the arguments are allocated and set to appropriate values the actual procedure call can be made. This calls the implementation of this procedure that's part of the server process. The server will computed the result of the operation, or potentially it will conclude that there is some kind of error message that needs to be returned. The result will be passed to the server side stub, and it will follow a similar reverse path in order to be returned back to the client. One more step is needed for all of this to work. Here we have as the zero initial step. That the client will need to find or discover the server, so that it can bind with it. But before that can happen somehow the server needs to do some things so that it can be found. The server will need to announce to the rest of the world what is the procedure that it knows how to perform, what are the argument types that are required for that procedure. What is its location? The IP address, the port number, any information that's necessary for that server to be discovered and so that somebody can bind with it. What that means is that the server also executes some registration step when this operation happens.

**WHAT** can the server do?

**WHAT** arguments are required for the various operations?

WE NEED AN AGREEMENT!

**WHY**

- client-side bind decision

- runtime to automate stub generation

=> Interface  
Definition  
Language (IDL)



7. Another thing about RPC is that the client and the server don't need to be developed together as part of the same application. They may be completely independent processes written by different developers, written even in completely different programming languages. But for this to work there must be some type of agreement so that the server can explicitly see what are the procedures that it knows how to execute and what are the arguments that are required for those procedures. The reason this information is needed is so that, on the client side, the client can perform decisions, which particular server it should bind with. Standardizing how this information is represented is also important so that the RPC run time can incorporate certain tools that will automate the process of generating the stub functionality. To address these needs RPC systems rely on use of interface definition languages, or IDLs. The IDLs serve as a protocol of how this agreement will be expressed.



## Interface Specification with IDL

An IDL used to describe the interface the server exports:

- procedure name, arg  
2 result types
- version #

RPC can use IDL that is:

- language-agnostic
- XDR in SunRPC

```
struct data_in {  
    string vstr<128>;  
};  
  
struct data_out {  
    string vstr<128>;  
};  
  
program MY_PROG {  
    version MY_VERS {  
        data_out MY_PROC(data_in) = 1; /* proc1 */  
    } = 1; /* version1 */  
} = 0x31230000; /* service# */
```

agnostic: 不可知論者

XDR: External data representation

8. An interface definition language is used to describe the interface that a particular server exports. At the minimum, this will include the name of the procedure and also the type of the different arguments that are used for this procedure as well as the result type. So you see this is very similar to defining a function prototype. Another important piece of information is to include a version number. If there are multiple servers that perform the same operation, the same procedure, the version number helps a client identify which server is most current, which server has the most current implementation of that procedure. Also the use of version numbers is useful when we are trying to perform upgrades in the system. For instance, we don't have to upgrade all the clients and all the servers at the same time. Using this version number however, the clients will be able to identify the server that supports exactly the type of procedure implementation that is compatible with the rest of the client program. So this is basically useful for so-called incremental upgrades. The RPC system can use an interface definition language for the interface specification that's completely agnostic to the programming languages that are otherwise used to write the client and the server processes. SunRPC which is an example of an RPC system that we will look at later in this lesson, uses an IDL that's called XDR, External data representation. And XDR is a completely different specification from any other programming language that's out there. We will describe XDR in more detail, but here is an example of something that's described with XDR. And you can notice that the definitions of things like the string variable with these angular brackets, that's not really something that's used in other programming languages. It's very XDR specific. If you would like, by the way, to read ahead and examine a SunRPC example and look at XDR in more detail, there are links provided in the instructor notes.

## Interface Specification with IDL

An IDL used to describe the interface the server exports:

- procedure name, arg  
2 result types
- version #

RPC can use IDL that is:

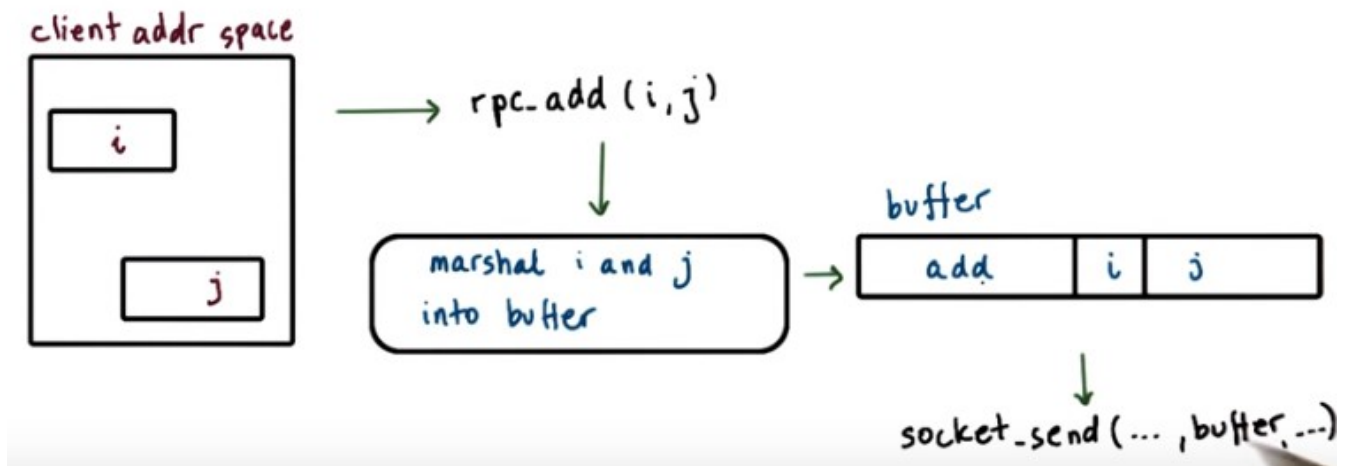
- language-agnostic  
XDR in SunRPC
- language-specific  
Java in Java RMI

```
public interface Hello extends Remote {  
    public String sayHello(String s) {  
        throws RemoteException;  
    }  
} // java JMI example
```

JUST INTERFACE!  
NOT IMPLEMENTATION!

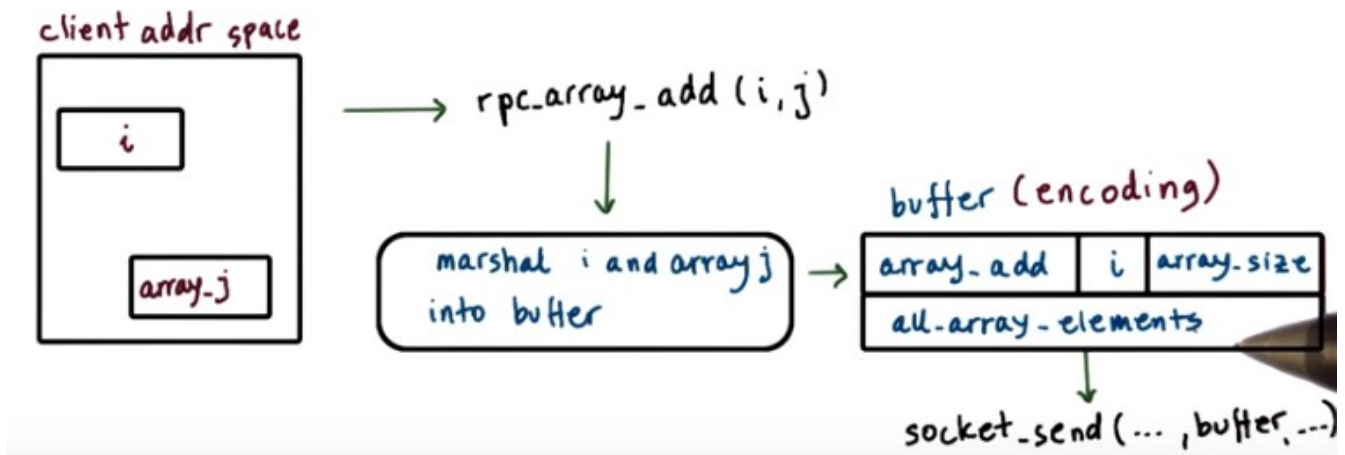
The opposite of a language-agnostic choice for an IDL is to choose a language-specific IDL to describe the interfaces. For instance, the Java RMI, which is a Java-equivalent of RPC uses the actual, the same, programming language JAVA. To specify the interfaces that the RMI server is exporting. Here is an example of an interface specified for Java RMIs. Those of you that know Java will immediately recognize that this looks just like Java. For programmers that know Java, use of a language specific IDL is great because they don't have to learn yet another set of rules to, how to define data structures or procedures in another language. For those that don't know Java that are not familiar with the specific programming language that's supported by the server for instance. Then this becomes irrelevant if you have to learn something they might as well learn something simple and that is one of the goals that XDR has. Now let me iterate one more time that whatever the choice for the IDL language, this is used only for specification of the interface that the server will export. The interface, whatever is written with this IDL language will be used by the RPC system for tasks like automating the stub generation process. Generating the marshalling procedures. And to generate information that's used in the service discovery process. The IDL is not actually used for the actual implementation of the service.

## Marshalling



9. To understand Marshalling, let's look at the add example again. The variables *i* and *j* are somewhere in the memory of the client processing address space. They're two separate variables so there's absolutely no guarantee that they will be next to one another. The client makes a call to the RPC procedure `rpc.add` and passes *i* and *j* as arguments to it. At the lowest level of the RPC run time, this will somehow need to result in a message that's stored in some buffer that needs to be sent via socket API to some remote server. This buffer needs to somehow be some contiguous location of bytes that includes the argument as well as some information about the actual procedures, some identifier for the procedure, so that on the other end, the server can make sense of what needs to be done and how the rest of the bytes in this packet need to be interpreted. And this buffer gets generated by the marshalling code. The marshalling code will take these variables *i* and *j*, and then it will copy them into this buffer. It will serialize the arguments of the procedure into a contiguous memory location in this manner.

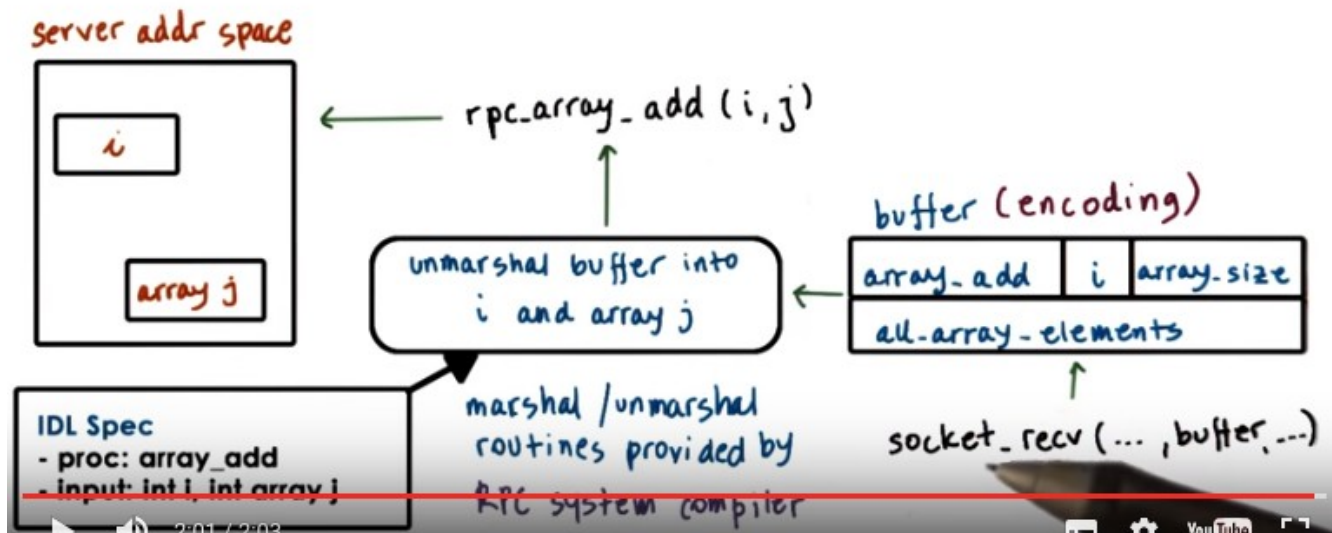
## Marshalling



In case the previous example is too trivial, here is what would happen if we need to perform a array add procedure, which takes as arguments and integer i. And some array, j and then adds this integer to all of the elements of the array. Then again, the marshalling code will need to serialize the arguments i and j. **Serializing the array j can be done in different ways.** For instance, the agreement can be that arrays are serialized in a way that we first place the size of the array. And then we add all of the elements of the array. So then the total buffer that's produced, as a result of the marshaling process, will include both the specification of the procedure, in this case, it's a different procedure. Array add. The first element, i, the first argument and then the second argument of the procedure, j, that happens to be an array. And in this particular process, the agreement is that the array includes the array size and then the elements. Another type of agreement that can make sense for a marshal in arrays is that we would just list all of the elements of the array. And then we would include some special character to denote the end of array. That's, for instance, what's typically used for strings, and then the null character is used to denote the end of array. Either way, what this means is that the marshaling process needs to encode the data into some agreed upon format. So that it can be correctly interpreted on the receiving side. The encoding specifies the data layout when it's serialized to the byte stream so that anybody that looks at it can actually make sense of it.



## Un-Marshalling




10. In the un-marshalling code in contrast, we take the buffer that's provided by the network protocol. And then based on the procedure descriptor and the data types that we know are required for that procedure descriptor, we parse the rest of the byte stream from that buffer. We extract correct number of bytes and we use those bytes to initialize data structures that correspond to the argument types. As a result of the un-marshall link process, these I and J will be allocated somewhere in the server address space, and they will be initialized to values that corresponds to whatever was placed in the message that was received by the server. Now again, the marshal link and un-marshal link routines aren't something that the developer will explicitly have to write, instead the RPC systems typically include a special compiler that takes an IDL specification, a specification that describes the procedure prototype and the data types for the arguments. And from that it generates the marshal link and the unmarshal link routines that are used in the steps to perform these translations. These routines are also responsible to generate the appropriate encoding related actions. So exactly how will an array be represented when its encoded in a byte stream. That's an example what will take place in these auto generated routines and there are other examples of what constitutes encoding. For instance, converting integers like this value i from one NDN format another NDN format, like from big NDN to little NDN depending on what's required by the server or by the client for the results. That's an example of a automated action that would be incorporated into the marshal encode. Once this IDL is compiled and all of the code is generated that provides the implementation for the marshal link and un-marshal link routines, all the developer needs to do is to take that code and just to make sure that it links it. With the program files for the server, or the client codes when generating executables.

## Binding

Client determines...

WHICH **server** should it connect to?  
- service name, version number...

HOW will it connect to that **server**?  
- IP address, network protocol, ... 

11. Let's talk a little bit about binding now. Binding is the mechanism that's used by the client to determine which is the server that it needs to connect to. Based on things like the name of the service that it needs perform, the version number of that service. And also it's used to determine how to connect to that particular server to basically discover the IP address or the network protocol that need to be used for that connection to be established.

## Binding and Registry

Registry == database of available services

- search for service name to find service (which) and contact details (how)

- distributed

- any RPC service can register

- machine-specific

- for services running on same machine

- clients must know machine address

- registry provides port number

~~needed for connection~~



Needs naming protocol

- exact match for "add"

- or consider "summation".

~~"sum", "addition"~~

To do this, to support binding, the system software needs to support some form of database of all the available services. And this is often called a registry. You can think of the registry as the Yellow Pages that you need to look up based on the service name that you require and then find the best match based on the protocol, the version number, the proximity. Some other information. That match will then provide you with the contact details for that particular service instance, so the address, the port number, the protocol that needs to be used. At one extreme this registry can be some distributed online service may be called something like rpcregistry.com, that any RPC server can register with. And the clients then have a well-known contact point, how they can find information regarding the services they need. At the other extreme, the registry can be a dedicated process that runs on every single server machine and knows only about those services that run on that particular machine. That means that the clients have to know the machine address, when they need to request some particular service. And the registry still provides useful information. It will tell the clients what is the port number that they need to use when they try to connect with the particular server. Regardless of how the registry's implemented, it will require some sort of naming protocol, some sort of naming conventions. For instance, the simplest approach could require that a client has to specify the exact name and version number of the service that it requires. Or a more sophisticated naming scheme could consider the fact that words like summation and sum and addition are likely equivalent to the use of the word add. And so any service that uses any one of these function names or service names is a fair candidate to be considered when trying to find the best match. Allowing this kind of reasoning for required supports for things like oncologists or other cognitive for learning methods, and we will not discuss this in this course.

## Visual metaphor

Applications use binding & registries  
like

Toy shops use directories of outsourcing services



- Who can provide services?
  - look up registry for image processing

What services are provided?

- compress, filter... version # ... → IDL

- How will they send/recv?
  - TCP/UDP → registry



- Who can provide a service?
  - e.g., shops to outsource assembly
- What services do they provide?
  - that will create train carts
- How will they ship/package?
  - that will ship via UPS

Outsourcing: 外包

12. To illustrate the use of binding and registries by applications when they use RPCs, we will draw an analogy with how toy shops rely on directories of outsourcing services. For instance, in a toy shop when considering whether or not to use some kind of outsourcing service, the manager will want to know who out there can provide that particular service. What are the specifics service details that those outsourcing companies offer? And exactly, what are the shipping or packaging options they provide. For instance, the toy shop manager may consider looking at the directories service to find out what are the shops where outsourcing of assembly operations can be supported. He will look up what are the exact services that each of these shops provide. And for instance he's trying to find the service where the assembly of train cars can be provided. And then the manager may be interested in exactly what are the shipping options that they offer. For instance, whether they ship with UPS. To give an analogous example in the context of operating systems and the applications use of binding and registries in RPC. Now we can see that the same types of steps are required to be performed by applications when they rely on the RPC to execute some service. For instance, they have to look up the registry to find out who can provide a particular service. They can look up a registry with a service name that requires specify somehow some image processing. The registry provides some detail regarding the various services that are provided by each server, the version number. All of this relies on the use of some interface definition language so that the interface can be describe in some standard way. And then finally, also the registry will provide information regarding the protocols that a particular server or services support like TCP or UDP. The applications can take all of this information into consideration when determining which particular process to bind with, which particular server to bind with. And similarly in the toy shop, the toy shop manager can consider the answers to all of these questions when determining how to outsource a service.



## What about Pointers?

```
procedure interface: foo(int, int*)  
in local calls: foo(x, y) => ok  
in remote calls: foo(x, y) => ???
```

y points to  
location in caller  
addr. space

Solutions:

- NO POINTERS!
- serialize pointers; copy referenced ("pointed to") data structure to send buffer

13. A tricky issue when it comes to RPC's is the use of pointers as arguments to procedures. In regular procedures it makes perfect sense to have procedures like this foo that takes two arguments, an integer, and the second argument is a pointer to an integer or even an integer array. When this procedure is called. The second argument, is a pointer to some address, in the address base of the calling process, where the particular, the area about this argument is stored. However, in RPC, passing a pointer to the remote server makes no sense. Since this pointer points to some location in the caller address space, the server cannot possibly get to the contents that are stored at this particular address. To solve this problem, RPC systems can make one of two decisions. The first decision is not to allow for pointers to be use this argument of any procedure that an RPC procedure that will be exported and can be called in remotely. The second solution is to allow pointers to be used but in the RPC run time to ensure that the marshalling code that gets generated understands the fact that the argument is a pointer. And that, instead of just taking that argument and copying it into the send buffer, that it actually serializes the pointer. What that means that it will copy the referenced, the "pointed to" data structure, into the data buffer into one serial representation. On the server side, the RPC runtime will first have to unpack all the data to create the same data structure. Then it will record the address to this data structure and that is the value that's the pointer that it will use as an argument when it makes the call to the actual local implementation of this particular operation. 意思可能是指, 在 server 程序中繼續使用這個 pointer, 但此時 pointer 是指向 server 中的地址了



## Handling Partial Failures

When a client hangs... what's the problem?

- server down? service down?
- network down? message lost?

- timeout and retry => no guarantees!

=> special RPC error notification (signal, exception...)  
catch all possible ways in which the RPC call can  
(partially) fail.

client hangs: 即 client 停住不動了

14. Since we're talking about the trickiness of RPC calls, let's also talk about errors in fault handling and reporting. When the client hangs while waiting on a remote procedure call, it is often hard to take what exactly is the problem. The server can be overloaded, the client request may be lost, the response may be lost, the server machine may have crashed, or the server process may have crashed, or some element in the network, some switch or router may be down. Even if the RPC runtime incorporates some mechanisms that time out whenever a client RPC call hangs, and then retries them automatically. They're really no guarantees that the problem will be resolved or that the RPC runtime will be able to provide some better understanding of what's going on. And potentially, for some cases, it is possible to really understand what is the cause of the error, but in principle that is too complex. It would have involve a lot of overhead, and ultimately, it's still unlikely that it will provide a definitive answer. For this reason IPC systems typically try to introduce a new type of error notification or a new type of signal or exception that tries to capture what went wrong with an RPC request without claiming to provide the exact detail. This serves as a catch all for all types of errors, all types of failures that can potentially happen during an RPC call. And it also can potentially indicate a partial failure, so maybe the call really didn't quite fail, it's just that the client doesn't know what succeeded and what failed.

15. Consider the following scenario. An RPC call fails and returns a timeout message. Given this timeout message, what is the reason for the RPC failure, that can be concluded by the RPC run time? Here are the options that are available. The client packet was lost. The server packet was lost. The network link was down. The server machine was down. The server process failed. The server process was overloaded. All of the above. Or any of the above. Check all that apply.



## RPC Failure Quiz

Assume an RPC call fails and returns a timeout message. Given this timeout message, what is the reason for the RPC failure? Check all that apply.

- ☐ client packet lost
- ☐ server packet lost
- ☐ network link down
- ☐ server machine down
- ☐ server process failed
- ☐ server process overloaded
- ☐ all of the above
- ☒ any of the above

16. The only answer that the RPC run time can definitely be confident that is the correct answer is any of the above. As we explained in the previous morsel, any one of these things can be a possible cause of failure. Also, hypothetically, though perhaps not very likely, it is possible that every one of these things happened at the same time, and that's why even "all of the above" is one of the possible answers. So, "any of the above" is the only correct answer. That is the only thing that the RPC runtime can know for sure when it sees that request time back.

## RPC Design Choice Summary

Binding => how to find the server  
IDL => how to talk to the server;  
how to package data

Pointers as arguments => disallow  
or serialize pointed data

Partial failures => special error  
notifications

design decisions for RPC systems

e.g., Sun RPC, Java RMI



17. In the last few videos, we described some issues with remote communication and the RPC mechanisms that solve them. This included the binding mechanism that's used so that the clients can figure out how to find the server and what is the server that they need to talk to in the first place. We discuss the use of interface definitional languages, to determine how to package arguments and results that are being exchanged among the client and the server. And in that sense, the IDL is used to specify how the client and the server talk to one another. How they are able to understand each other. Next, we observe the problem of dealing with pointers as arguments in remote procedure calls. And we said that, the use of pointers should either be completely disallowed, or that the RPC system should build in some kind of support to serialize the data that's being pointed. Finally, we'll also talk about partial failures, and explained how it is tricky to determine exactly what went wrong in an RPC system. And that instead, the RPC run-time provides some special errors and tries to, in as much as possible, determine what exactly was the cause of the error without making any kind of guarantees that it will be able to provide a precise answer. For all of these, we mention that there are multiple choices that can be made in the concrete implementation of an RPC system. For instance, for binding, we can choose to have a distributed or a per machine registry. Or we can choose to use a language agnostic or language specific interface definition language. In summary, these issues define the design space for an RPC system in different RPC or RPC like solutions, we'll make different choices in this space. And we will also very briefly contrast this with the RPC like support in Java called remote method invocations or Java RMI





What is Sun RPC?

developed in 80s by Sun for UNIX;  
now widely available on other platforms

### Design Choices

Binding => per-machine registry daemon

IDL => XDR (for interface specification and for encoding)

Pointers => allowed and serialized

Failures => retries; return as much information as possible

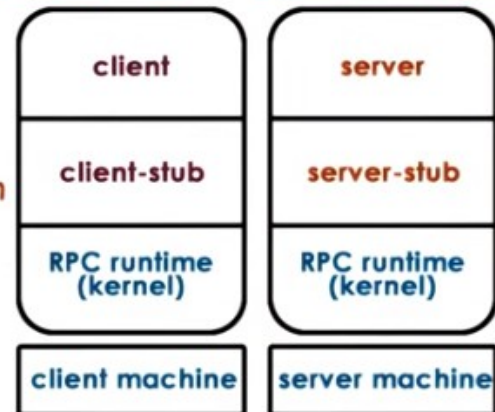
18. Sun RPC is an RPC package originally developed by Sun in the 80s for their network file system NFS for UNIX systems but it became popular and now it's widely available in other platforms. Sun RPC makes the following design choices. In Sun RPC it's assumed that the server machine is known up front and therefore the registry design choice is such that there is a registry daemon per-machine. When a client wants to talk to a particular service, it needs to first talk to the registry on that particular machine to find out how to contact the exact service that it requires. Sun RPC makes no assumption regarding the programming language that used by the client or by the server process. To maintain neutrality Center PC relays on a language agnostic interface definition language, XDR. And this is used both for the specification of the interface of the RPC service, as well as for the specification of the encoding. How data types will be encoded when they're being transmitted amongst machines? Some RPC does allow the use of pointers and data structures that are pointed by these pointers will be serialized. And finally, Sun RPC incorporates some mechanisms for dealing with errors. First, it has internally retry mechanism to retry contacting a server when a connection times out. This will be done for a specific number of times. Second, as much as possible, the RPC run time will try to return meaningful errors. So that a caller can at least distinguish between things like the server is not available, or there is a mismatch, or unsupported protocol or version. Or there is simply a time out related failure that just covers all of the other types of possible failures.



## SUN RPC Overview

- client - server via procedure calls.
- Interface specified via XDR (.x file)
- rpcgen compiler => converts .x to language-specific stubs
- server registers with local registry daemon
- registry (per-machine)
  - name of service, version, protocol(s), port number...
- binding creates handle
  - client uses handle in calls
  - RPC runtime uses handle to track per-client RPC state

## ORACLE



client and server  
on same or different machines

handle 即 client call server function 時的 `y = square_proc_1(&x, client_handle)` 中的 `client_handle`  
`client_handle` 是通過以下 binding 得到的:

```
CLIENT* client_handle = clnt_creat(rpc_host_name, SQUARE_PROG, SQUARE_VERS, "tcp");
```

19. Similarly to the generic description of RPC, like some other PC, the client and the server are allowed to interact via a procedure called interface. The server specifies the interface that it supports in a .x file written in XDR. Also Sun RPC includes a compiler called rpcgen that will compile the interface specified in the .x file to language specific stub. It will generate separate stubs for the client side and for the server side stuff. The server process when launched will register itself with their registry daemon that's available on the local machine. The per machine registry will keep track of information that includes the name of the service, the version, many of the protocols that are supported with the service, and also the port number that needs to be contacted when the client side RPC sends a request through the server. The client must explicitly contact the registry on the target machine in order to obtain information about the server process. When the binding happens, the client creates an RPC handle, and this handle is used whenever the client makes any RPC calls. And in this way, the runtime is able to track all of the per-client RPC-related state. I should note that with Sun RPC, or any other RPC, for that matter, the client and the server process that are communicating amongst each other may be on different machines. Or they may be on the same machine, just two processes running on the same physical node. So, the RPC in that case works like other forms of IPC, except it has a much higher level semantics. It has procedure called semantics, which is more complex than the IPC mechanisms that we saw before.



## SUN RPC Overview

ORACLE

Documentation, tutorials and examples now maintained by  
Oracle

- TI-RPC == Transport - Independent Sun RPC
- provides Sun RPC / XDR documentation and code examples
- older online references still relevant
- Linux man pages for 'rpc'

Before we look at the key components of Sun RPC, if you would like to view a more complete reference. Then, take a look at these Sun RPC tutorial and examples that are now maintained by Oracle. Oracle purchased Sun in 2010. The link to this is provided in the instructor notes. At that link, you will find references to TI-RPC as opposed to Sun RPC. TI stands for transfer independent RPC. And that means that the protocol that will be used for the client and server communication doesn't have to be specified at compile time. It can be specified dynamically at run time. Other than that and a few smaller issues the documentation and the examples closely follow the original Sun RPC specification as well as the XDR interface definition language. Also, a number of older online references are still valid reference points. And you can, finally, look at the Linux man pages by looking for man rpc. This will give you all of the Linux supported APIs.



## Sun RPC : XDR Example

Client => send x

Server => return  $x^2$

```
struct square_in {
    int arg1;
};
struct square_out {
    int res1;
};

program SQUARE_PROG { /* RPC service name */
    version SQUARE_VERS {
        square_out SQUARE_PROC(square_in) = 1; /* proc1 */
    } = 1; /* version1 */
} = 0x31230000; /* service id */
```

XDR (.x file) describes:

- datatypes
- procedures (name, version, ...)
- service ID

PROG 應該就是 program 的簡寫, PROC 應該就是 procedure 的簡寫

20. We'll now take a look at the various components of Sun RPC using an example. The client again will be contacting a server that can perform calculations except this time the client will pass a single argument x for which it will warn the server to compute the squared value,  $x^2$ . Here's the .x file for this example with which the server specifies its interface. In the .x file, the server specifies all the data types that are needed for the arguments, or the results of the procedures that it supports. In this case, the server supports one procedure, square proc (不是 square prog). That has one argument of the type square in. And the returns are resolved of the type square out. The data type square in, and square out, are both defined in the .x file. If we take a look at them, it turns out that both of them have a single element and that's an int. And in XDR an int, is an integer just like the integers in C. So it's a 32-bit integer. Also note that this notation under square in, square out is not any part of the required syntax for specifying the input and the output data types in XDR. Other than the data types, the .x file describes the actual RPC service and all of the procedure it supports. First there is the name of the RPC service. In our case that's square prog. And this is the name that will be used by clients when they're trying to find an appropriate service to bind with. A single RPC server can support one or more procedures. For instance, a calculator server can support all sorts of arithmetic operations. In our case, the square prog service supports exactly one procedure and that's square proc, procedure. There is an ID number that's associated with it (square proc). This is one in this case. This number is not used by the programmer. This will be used internally by the RPC run time. When it's trying to identify which particular procedure is being called. So it's not going to pass between the client and the server in the packets. The name SQUARE\_PROC, instead it will use this value 1 as a reference. In addition to this ID number and the input and output data types, each procedure is also identified by a version. And in fact the version may apply to an entire collection of procedures. We see that in this case, the version number for a service is 1. Over time, however, we may choose to refine that SQUARE PROC procedure or add additional procedures. And as we're doing that, we don't want to be forced to immediately go ahead and update all of the clients with this perhaps semantically different or syntactically different square proc procedure. In that case, what makes sense is that whenever clients



and servers interact, they reference the version number of the procedure that they're requesting. When a client contacts a server that does not support a procedure with the appropriate version number, then the communication can be rejected. What this also illustrates is that it's possible for a single server to support multiple versions of the same procedure, and this helps with, in general, the evolution of the system. We don't have to coordinate an upgrade of all the servers and all the clients at the exact same time. Finally, the .x file also specifies service ID. This id is a number that's used by the RPC runtime to differentiate among the different services. So the client will use things like service name, and procedure name, and the version number, whereas the RPC runtime will refer to the service id, the procedure id, and again, it has to know the version id.



### Sun RPC : XDR Example

Client => send  $x$

Server => return  $x^2$

### Service ID Conventions

- $0x\ 0000\ 0000 - 0x\ 1fff\ ffff ==$  defined by Sun
- $0x\ 2000\ 0000 - 0x\ 3fff\ ffff ==$  range to use
- $0x\ 4000\ 0000 - 0x\ 5fff\ ffff ==$  transient
- $0x\ 6000\ 0000 - 0x\ 7fff\ ffff ==$  reserved

For the service ID, you're allowed to specify a value in this range. The remaining values for service ID's either have some predefined values like for instance, for the network file system, or they're reserved for future use.

## Compiling XDR

rpcgen compiler  
rpcgen -C square.x

- => square.h => data types and function definitions
- => square\_svc.c => server stub and skeleton (main)
- => square\_clnt.c => client stub
- => square\_xdr.c => common marshalling routines



實踐表明, `rpcgen -C` 中的 `C` 是大寫, 若用小寫, 則不能生成文件.

21. Let's show how you actually compile a .x file. Assume that we're using the same squared example as in the previous videos. In the file, the .x file for that example is square.x. You'll see that by using this .x file, we will automatically generate a bunch of the code that's used for the client and the server-side processing. To do this, Sun RPC relies on a compiler, rpcgen. And to generate C code, rpcgen is used with the option -c, so that full command is `rpcgen -c`, and then `square.x`. That's the .x XDR file. The outcome of this operation will be that a number of files will be generated. First, they will generate a header file, `square.h`, that will have all of the language-specific definitions of data types and function prototypes. Next, they will generate the code for the client and the server-side stubs. For the client, this is a proper stub, for the server side code, this actually also includes the skeleton of the actual servers. It has the main retaining. The only thing that's not available will be the actual implementation of the service, of the procedure, and this makes perfect sense since the compiler has no way of knowing what exactly what do we want a particular procedure to do. In this case, squaring a number. Finally, the compilation stub will also generate a separate files, `square_xdr.c`. And this will include some common code that's related to the marshalling and unmarshalling routines for all of all of the data types, the arguments and the results, that are used both at the client and on the server-side.

## Compiling XDR

`square_svc.c` => **server stub** and skeleton

- `main` => registration / housekeeping

- `square_prog_1`

  - => internal code, request parsing, arg marshaling

  - => `_1` == version 1

- `square_proc_1_svc` => actual procedure ;  
must be impl. by developer

`square_clnt.c` => **client stub**

- `squareproc_1` wrapper for RPC call to  
`square_proc_1_svc`

- `y = squareproc_1 (&x ..)`



.X file



CODE

If you take a look at the file `square_svc`, which stands for service, you will see that it has two parts. The first part is the `main` function for the server and that will include the code that, that does the registration step and also some additional housekeeping operations. In addition to `main`, the stubble contain all of the code that's related to the particular RPC service. So in our squared case, this is the `square_prog` service. And, it is the first version of that particular service so for all of the procedures in that particular service, the file will include automatic regenerate code in order to parse the request. So as to determine which particular procedure to be called to generate the arguments, all of the argument marshalling corporations will be invoked here, and other steps. In addition, in the step file, the auto-generated code will include the prototype for the actual procedure that's invoked in the server process. For the `square_proc` procedure that we describe, this is the procedure name. And that will include also the `_1`, that refers to the version number. And this piece if code has to be implemented by the developer, this is not automatically generated. The client stub will include a procedure that's automatically generated, `squareproc_1`. And this will represent a wrapper for the actual RPC call that the client makes to the server-side process where the implementation of the service, this `squareproc_1_svc` is actually called. Once we have all of this, the developer then writes the client application and makes call to this wrapper function that looks something like this, `y = squareproc_1 (&x)`. This very much looks like a regular procedure call. There is no need to create sockets, create buffers, copy data into the buffers, and this is what makes RPC appealing.

## Summarizing RPC development

From .x => header, stubs...

Developer

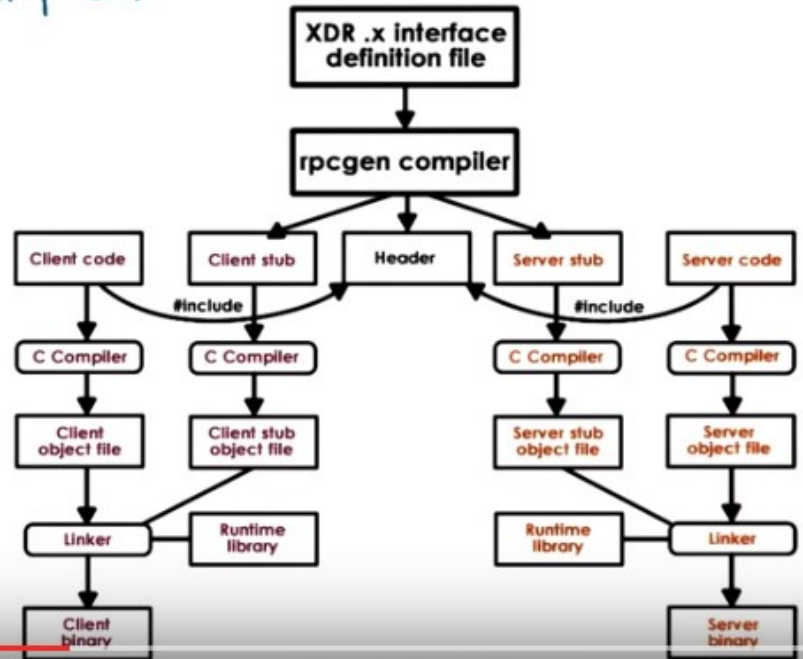
- server code  
impl. of square.proc.1\_svc
- client side  
call squareproc.1()

- #include .h

- link with stub objects

RPC Runtime - the rest

- OS interactions,  
communication mgt...



22. We will now summarize one more time the steps involved in developing RPC applications. And this figure here will serve as an illustration. We have to write the .x file in XDR and pass it through the rpcgen compiler. That will generate a number of files. The header file, the stubs. It will generate even the skeleton for the server. And it will also generate an underscore XDR file that has a number of helpful marshalling routines. For the server-side application, the developer has to provide the implementation of the actual service procedure. The square.proc.1\_svc, for the first version, \_svc. This is the naming convention. On the client side, the developer has to develop the client application and whenever necessary, call the wrapper procedure squareproc.1(). This is what will actually invoke all of the communication with the server process and the execution of this particular service implementation. The developer has to make sure that he includes all of .h file, particularly the auto-generated ones from the rpcgen compiler. And also that it links the client and the server code with the stub object. The RPC runtime that is called from the stub things, provides all other functionality, including interactions with the operating systems, creating sockets, managing connections, and everything else.

## Summarizing RPC development

`rpcgen -C square.x` => not thread safe!

`y = squareproc_1(&x, client-handle)`

`rpcgen -C -M square.x` => multithreading safe!

`status = squareproc_1(&x, &y, client-handle)`

- doesn't make a multithreaded `"_svc.c"` server
- on Solaris `"-a"` => MT server
- on Linux has to be done manually



.X file



CODE

I should point out that, that `rpcgen`, when used only with the flag `-C` generates code that's not thread safe. The output of the compilation results in a function that will need to be called with something like this. And the problem with this function is that internally, the implementation of this operation, as well as at the runtime level, there are a number of statically allocated data structures included for the result. And this leads to race conditions when multiple threads are trying to make RPC calls to this routine, concurrently. To generate thread safe code, the code must be compiled with the `-M` option, and `M` stands here for multithreading safe. This will also create a wrapper function `squareproc_1`, however, it has a different signature and its implementation differs, for instance it will dynamically allocate memory for the results of this operation. So some of the issues that are coming up with the previous implementation will not come up in this case. >> Using the `-M` flag doesn't actually create a multithreaded server, the implementation that's provided, that generated in the `_svc` file. That won't be multithreaded. On Solaris platforms there's another option, `-a`, using this option, that actually generates multithreaded server code. But in Linux, this option is not supported and any multithreaded server has to be created manually. Of course, with using the multithreaded safe routines as a starting point.

23. Soon enough you will be writing your own XDR files and implementing RPC. But right now let's take a look at what would happen if we were compiling this `square.x` file that's used in the examples that we talked about in the previous videos. So here is a short quiz. For this `square.x` file, that's also provided in the instructor's notes, what is the return type of the `square proc_1` procedure, when the `square.x` file is compiled with `rpcgen -C` or `rpcgen -C -N`? Write your answers in the text boxes.





## Square.x Return Type Quiz

```

struct square_in {
    int arg1;
};
struct square_out {
    int res1;
};

program SQUARE_PROG { /* RPC service name */
    version SQUARE_VERS {
        square_out SQUARE_PROC(square_in) = 1; /* proc1 */
    } = 1; /* version1 */
} = 0x31230000: /* service id */
        
```

What is the return type of squareproc\_1 if this square.x file is compiled with:

- rpcgen -C
- rpcgen -C -H

square\_out\*

enum clnt\_stat

24. After compiling the file you should get the following answers. Our compiled files have been included in the instructor notes. Note that the thread safe and the non-thread safe versions of this function have a different prototype and they resolve in a different return values



## Sun RPC : Registry

RPC daemon == portmapper

- /sbin/portmap (need sudo privileges)

Query with rpcinfo -p

- /usr/sbin/rpcinfo -p
- program id, version, protocol (tcp, udp), socket port number, service name...
- portmapper runs with tcp and udp on port 111



25. Let's talk briefly about the Sun RPC registry. Remember we said already that the actual code that the server needs to register with the registry's auto generated in the RPC general process, and it's part of the main function. In Sun RPC the registry process or the registry daemon is a process that runs on every single machine and it's called portmapper. To start this process in Linux you have to have

administrative permissions or sudo access privileges and then you can launch it with the following command, `./sbin/portmap`. This is the process that has to be contacted both by the servers when they need to register a service, and also by the clients when they need to find what is the specific contact information for a particular service they are looking for. Now given that the client already got to talk to this RPC daemon, it clearly knows what is the IP address of the machine that it will need to interact with. So the information that the client can extract from the port mapper includes things like what is the port number that the client needs to use to talk to a server, or whether the particular version and protocol are supported for the server that the client requires. Once the RPC daemon is running we can explicitly check what are the services that are registered with it using our `rpcinfo -p`. You may need to explicitly type in the full path for this command but once you run it you will see that it returns information like what is the program ID, the service name, the version of every single service that's registered on that particular machine. Also for every service it will incorporate the contact information. So what is the protocol that that service speaks so to say. And what is the socket port number that needs to be contacted by the client side RPC runtime when it wants to initiate communications with a service. When you run this service, you will also probably notice that the port mapper service is registered with tcp and udp on the same port number, 111. This means that there are two different sockets that this server is listening to. One is a tcp socket, and the other one is a udp socket, and they both happen to use the exact same port number, 111. This means that this service, the port mapper, will be able to talk to both the tcp, as well as udp clients.



## Sun RPC : Binding

```
// in client
CLIENT* clnt_create(char* host, unsigned long prog,
                    unsigned long vers, char* proto);
```

```
// for square example
CLIENT* clnt_handle;
clnt_handle = clnt_create(rpc_host_name, SQUARE_PROG, SQUARE_VERS, "tcp");
```

CLIENT type

- client handle
- status, error, authentication ...

26. And the last part of Sun RPC that I wanted us to talk about is the binding process. The binding process is initiated by the client using the following operation. So, `clnt_create` with a number of parameters. For the specific squaring example that we talked about, this operation will look like this. We will specify the host name of the server, as well as the protocol that we want to use when communicating with the server. And we will specify the name of the RPC service as well as the version number. These two arguments of the `clnt_create` operations are auto-generated in the RPC generation process from the `.x` file. And will be included in the header file in the `.h` file as hash defined values. What this means is that if the client needs to now support a different version number, it will need to be recompiled, given that this is essentially a static piece of information. However, none of the

other portions of the client code have to be modified. Also note that the return from this operation is a variable clnt handle that's of data type CLIENT. This is the clnt handle that the client will include in every single RPC operation that it requests. And this handle will be used to track certain information, such as what is the status of the current RPC operation, any error messages, or it can even be used to capture certain authentication-related conformation.

## XDR Data Types

### Default Types

- char, byte, int, float...

### Additional XDR types

- const (#define)
- hyper (64-bit integer)
- quadruple (128-bit float)
- opaque (~ C byte)
- uninterpreted binary data



27. In the basic square RPC example, we said that all of the data types for the input and output arguments must be described in the .x file. All of these types and data structures must be XDR supported data types. Some of the default XDR data types are those that are commonly available in programming languages like C for things like character and byte and integer and float. But XDR supports many other data types. For instance, if you specify that something is a const, it will be translated after compilation into a constant, which is C, will be a #define value. Data types like hyper or quadruple are used to refer to a 64-bit integer or a 120-bit float, respectively. And XDR also supports a so-called opaque type, which really corresponds to data type that's uninterpreted binary data. So, similar to the C byte type. So for instance, if you want to transfer an image, that image will be represented as an array of opaque elements.

## XDR Data Types

Fixed-length array

- e.g., `int data[80]`

Variable-length array

- e.g., `int data <80>`  $\Rightarrow$  translates into a data structure with "len" and "val" fields

except for strings

- `string line <80>`  $\Rightarrow$  C pointer to char

- stored in memory as a normal null-terminated string

- encoded (for transmission) as a pair of length and data

Variable-length array 是在 .x 文件被編譯後，自動在 .h 中生成一個 struct，其中包含 actual length 和 data pointer。所有程序中都是用的這個 struct。

Let's talk more specifically about arrays because in XDR, you can specify two types of arrays. The first is a fixed-length array that's described as follows. And here, the exact number of elements in the array is specified. The RPC runtime will allocate the corresponding amount of memory whenever arguments of this data type are sent or received. And it will also know exactly how many bytes from the incoming packet stream it should read out in order to populate a variable that's of this data type, this type of array. There are also variable-length arrays, where the length is specified in angular brackets. And this doesn't denote the actual length, rather the maximum expected length. When compiled, this will translate into a data structure that has two fields. An integer, len, that corresponds to the actual size of this array. And a pointer, val, that is the address of where the data in this array is actually stored. When the data is sent, the sender has to specify len, the size of the array, and then set val to point to the memory location where the data is stored. On the receiving end, the server will know that it's expecting data structure that's a variable-length. So it will know to read the first 4 bytes to determine what is the length, what is the size of the array. And then to allocate the appropriate amount of memory, and then to read the remaining portions of the incoming byte stream and to populate that memory with those values. The only exception to this are the strings. A variable-length string is defined as follows, and this line is really just the C pointer to character. In memory, the string will be stored just like a normal null-terminated string, so it will be an array of characters with the null character at the end. Operations like string copy and string length need that particular representation in order to be able to determine where is the end of the string (注意此時沒有 len 這麼個數據，長度要通過 null 位置求出來)。However, when that variable-length string is encoded for transmission, it will be encoded as a pair of length and data. So from that perspective, that will be similar, actually identical to what we see for other variable-length data structures.

28. Let's look at the use of XDR in a quiz. Let's assume that an RPC routine uses a variable length integer array of a maximum size 5. Now if the array is full, how many bytes are needed in order to represent this data structure in a client in C on a 32-bit machine? You should provide your answer in

bytes.



## XDR Data Types Quiz

A RPC routine uses the following XDR data type  
`int data[5];`

Assume the array is full. How many bytes are needed to represent this 5 element array in a C client on a 32-bit machine?

`int len` → 4B

`int *val` → 4B

`int * 5elem` → 4B \* 5

**28** bytes

MO: 由於是 32 位機, 故一個數據(int, int \*等)佔的大小為 32 bits, 即 4B.

### Instructor Notes Quiz Help

Number of bytes:

`data_structure_bytes + data_bytes`

From online:

Most computers made in the 1990s and early 200s were 32-bit machines. A 32-bit system can access  $2^{32}$  (4,294,967,296) memory addresses. Practically speaking, a 32-bit computer is limited to accessing 4,294,967,296 bytes (4 GB) of RAM.

While a 32-bit processor can access  $2^{32}$  memory addresses, a 64-bit processor can access  $2^{64}$  memory addresses. This is not twice as much as a 32-bit processor, but rather  $2^{32}$  (4,294,967,296) times more. This means a 64-bit machine could theoretically access 18,446,744,073,709,551,616 memory addresses or more than 18 billion GB of RAM. However, as of 2013, there are few computers that can physically support more than 64 GB of RAM.

29. Now since this is a variable length array it will be compiled in C, the length of the array `len`, and the pointer, the address where the actual data structure is stored, `val`. `len` is an integer. So that is four bytes. And `val` is an address. And given that this is a 32 bit machine `val` will also be four bytes. To add to that, the memory that's required for five integers, that's four bytes each. The total amount of memory is 28 bytes.



## XDR Routines

Marshalling/unmarshalling  
- found in square\_xdr.c

### Clean-up

- xdr\_free()
- user-defined freeresult procedure
- e.g., square\_prog\_1\_freeresult
- called after results returned



30. XDR provides the RPC runtime with some helpful routines. For instance, after we compile a .h XDR file, the compiler will generate a number of routines that are used for marshalling or unmarshalling, the various data types in the RPC operations. In the example that we talked about, the square rpc example, these will all be found in the square\_xdr.c file. In addition, the compiler will generate certain clean up operations like xdr\_free(), that are used to deallocate to free up memory regions that are used for the data structures, the arguments in the RPC operations. These routines will typically be called within a procedure that the name of the procedure typically ends with freeresult. For instance, in our square program it will be square\_prog\_1\_freeresult. And this is yet another user defined procedure, where the user can specify what are all of the different data structures, or pieces of state, that need to be freed up. And the allocated, after the runtime is done servicing RPC request and returning results. So the RPC runtime will automatically call this procedure after it's done computing the result.

What goes on the wire?



Transport header  
- e.g., TCP, UDP

RPC header

- service procedure ID, version number, request ID...

Actual data

- arguments or results

- encoded into a byte stream depending on data type



31. One thing that we didn't explain is what actually ends up in the buffers that are being passed for transmission among the client and the server. For instance, the server can support multiple procedures. It is important to not just pass the arguments but actually to include an RPC header that will uniquely identify what is the procedure that's being called, the version number, something about the requests so that we can detect repeated requests on retries. Similar type of information will be sent from the server back to the client, again, as part of the RPC header. So this is one component of what actually goes on the wire in the packets that are being transmitted. Then, clearly we have to put the actual data, the actual arguments or results on the wire as well. However, as opposed to just directly copying from memory into the packets, we have to first encode all of the different data types for the arguments in the results into a byte stream to serialize them in a way that depends on the actual data type. It is important to have a specific agreement on how this encoding is done so that the server has the ability to interpret the byte stream and recreate the appropriate data structure in the server address space. In order for the server to actually call the procedure that implements the service, it needs to have the arguments present in the server memory. That's why this step is necessary. Similar kind of requirement, we have also on the return when the result is passed to the client. The client needs to be able to look at this byte stream and figure out how it needs to take that information in populated data structure in the client memory. In some cases there may be a one to one mapping between the in memory representation and how the data is encoded in the packet. But in other cases that may not be the case. And finally when all this information is placed in a packet that needs to be preceded with the transport header with the networking header that will specify the protocol, the destination address and will make sure that on the client and on the server, all of the protocol specific operations take place.

## XDR Encoding

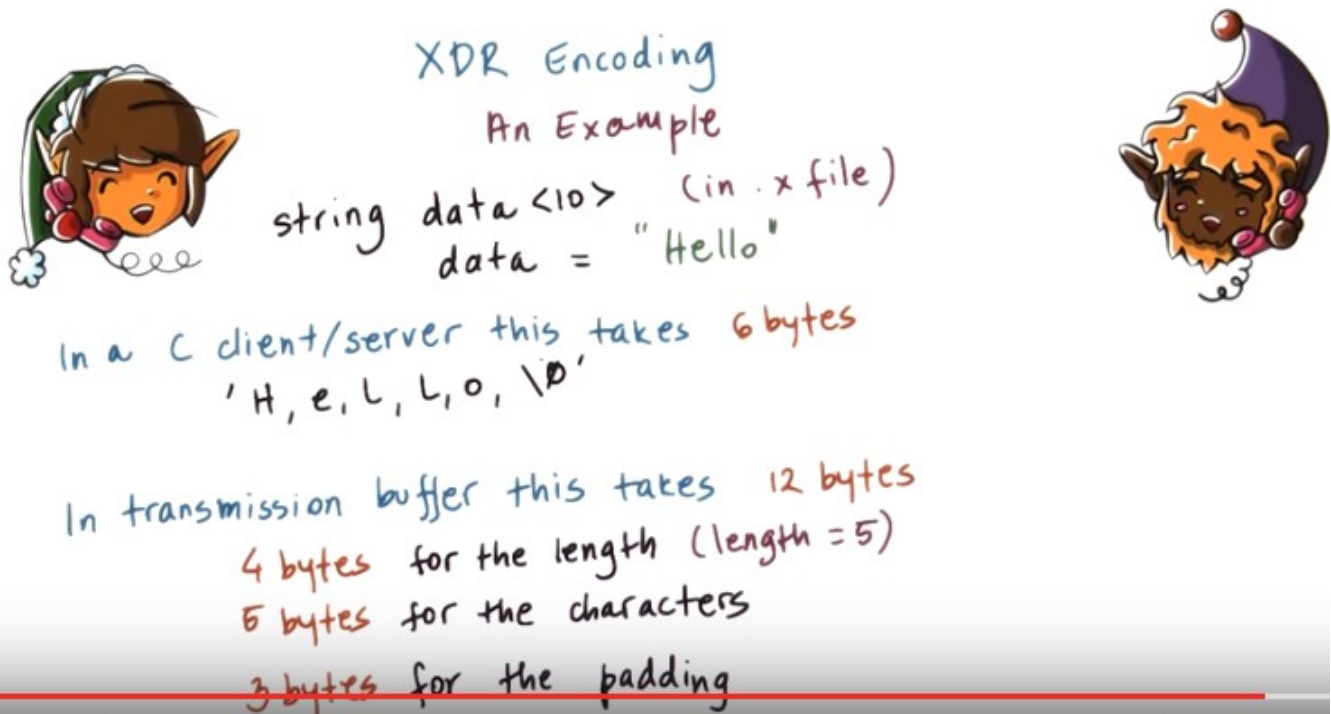


XDR == IDL + the encoding  
- i.e., the binary representation of data "on-the-wire"

### XDR Encoding Rules

- all data types are encoded in multiples of 4 bytes
- big endian is the transmission standard
- two's complement is used for integers
- IEEE format is used for floating point

32. As we hinted already with the discussion of the syntax data type, XDR specifies both the syntax, the interface definition language: so how our data type's described, and also it specifies the encoding: so what is the binary representation of data when it's on the wire. As we handled already in the discussion of the string data type, XDR that corresponds both to the interface definition language, essentially that's the syntax. How we are describing data types. And also it specifies the encoding. That's the binary representation of how is data represented when it's being transmitted between the client and the server on the wire. Here's some encoding rules. All data types are encoded in multiples of 4 bytes so transmitting a single byte argument would include a single byte for the data and 3 bytes of padding in order to make that up to 4 bytes. This is to help with alignment when moving data to and from memory and the network packets and the network card. Big endian is used as the transmission standard. What this means is that regardless of the endian in this type, of the client or the server machine, every type of communication will require the data is first translated into big endian representation and then if necessary, translated into the appropriate endian in this for the target machine. In some cases, this may be pure overhead just because the client and the server machine are both, let's say little endian machines. But, in principle, it's easier to have this type of standard agreement so that there's never any kind of ambiguity, what is the encoding that's being used on-the-wire. And how to interpret the bytes that are coming into the network packets. Other rules include things like, two's compliment is used to represent integers. And the IEEE is used for floating point numbers, so other roles.



上圖倒數第二行是 5 bytes for the characters, 不是 6 bytes for the characters

查網上可知, C 語言中, 一個 int 佔 4 bytes, 一個 char 佔 1 byte

Let's explain this a little bit better using an example. So, let's say that in the .x file, we have a definition of a data type that is a variable length array of a size up to 10. And let's say we have an argument, hello, that needs to be passed from the client to the server. In the client or the server address space, if these are C processes, this variable will take 6 bytes. 5 bytes for each of the characters and then the last 6 bytes for the null terminating character. However, for transmission, this variable will be encoded to take 12 bytes. The first 4 bytes will be used for the length. In this particular case, the length is 5, it's 5 characters. The next 5 bytes will be used for those characters, H-E-L-L-O. Notice we're not going to be transmitting the null terminated character. And then, at the very end, we will have 3 characters use the padding, because XDR specifies that everything needs to be on 4 by boundaries.

33. Here's a quiz that's very similar to the one that you recently took on XDR data types except this time, we're concerned with the data being transmitted over the network and the encoding cost of such transmissions. Let's again assume that an RPC routine uses a variable-length array of integers and that the array is full. For that situation, answer the following. How many bytes are needed to encode this five element array so that it can be sent from a client to a server where both the client and the server are 32-bit machines? In your answer, please do not include any bytes for the headers or the protocol related information. And provide your final answer concerning this encoding of this data structure in bytes.



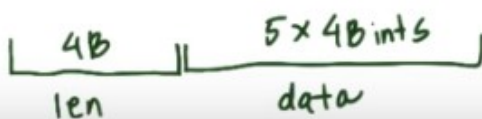
## XDR Encoding Quiz

A RPC routine uses the following XDR data type:

```
int data <5>;
```

Assume the array is full. How many bytes are needed to encode this 5 element array to be sent from client to server (32-bit)?

(Do not include bytes for headers/protocol!)



24 bytes

34. Variable length arrays will be encoded so that the first four bytes will correspond to the integer value of the array size, len, and then the rest of the bytes will correspond to the actual array elements. In this case the array has five elements where each is a four byte integer. No additional padding needs to be performed. So the total length of the encoded representation of this data structure is 24 bytes. In reality, you would need to account also for any of the RPC specific headers as well as the protocols, but we're not concerned with that in this particular answer.

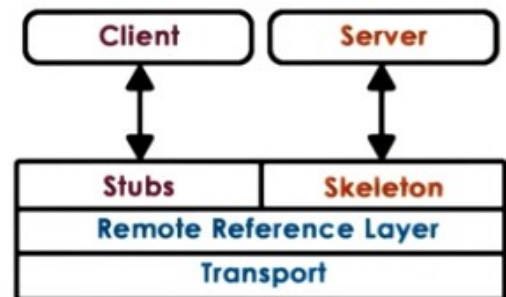
## Java Remote Method Invocations - RMI

- among address spaces in JVM(s)
- matches Java OO semantics
- IDL == Java (language-specific)



### RMI Runtime

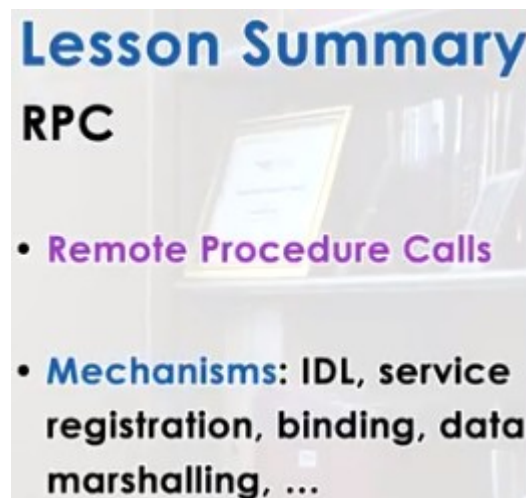
- Remote Reference Layer
  - unicast, broadcast, return-first response, return-if-all-match
- Transport
  - TCP, UDP, shared memory ...



35. Another popular type of RPC-like system is Java RMI, Java Remote Method Invocations. It's also pioneered by Sun as a form of client-server communication method, among address spaces in the Java virtual machine. Java is an object-oriented language where objects interact via method invocations(調用) and not via procedure calls. For this reason, this inter-process communication mechanism matches the Java object-oriented semantics as in the form of remote method invocations (RMI). Its architecture



is similar to what we saw with the remote procedure calls. Client and server processes have client-side stubs and server-side stubs. The server-side stub is referred to as a skeleton. In the Java virtual machines, all of the processes, all clients and all servers, are written in the Java programming language. For that reason, the interface definition language for the Java RMI is also Java. It doesn't make sense to adopt a different interface definition language, like in the case of XDR for RPC, where in this case, everything will be written in Java in the first place. So RMI uses a language-specific interface definition language choice. And in this case, that's Java. The runtime layer is separated into two components, the remote reference layer and the transport layer. This bottom layer (transport layer) implements all of the transport protocol related functionality. This can be TCP, UDP, shared memory based communications if the two processes are running on the same machine. Above that is the remote reference layer. This component captures all of the common code that's needed to provide different reference semantics. For instance, it can support unicast, where a client interacts with a single server, like what we had in the previous examples. But RMI can be used for other types of server reference semantics. For instance, with broadcast, the client will contact multiple servers. And then the reference semantics can be such that it will return only once the first response arrives, or only when all of the responses arrive and the responses match. It also makes sense to design other types of reference semantics. These are not the exclusive list. Regardless of the underlying transport protocol, this type of functionality will be implemented in a similar way. So RMI separates it and captures it in a separate component, this remote reference layer. As a developer, you can either just specify the reference semantics you want from the RMI interactions and the system will take care of the rest. Or if you want something exotic, you can implement just this component and the rest of the system can remain the same. We're mentioning in this lesson Java RMI just for completeness. We're not going to discuss it in any detail. If you would like to know more, visit the resources that are linked in from the instructor's notes.



36. In this lesson, we looked at remote procedure calls to RPCs. This is a popular interprocess communication mechanism that's used to support client-server types of interactions. We said that an RPC system requires the user of an interface definition language, or IDL, in order for us to describe the remote service, and the mechanisms such as registries and binding and marshalling in order to enable the remote data exchanges. We described in more detail Sun RPC, and with the examples that we looked at, you should have enough information to start using and implementing Sun RPC.

37. As the final quiz, please tell us what you learned in this lesson. Also, we would love to hear your feedback on how we might improve this lesson in the future.