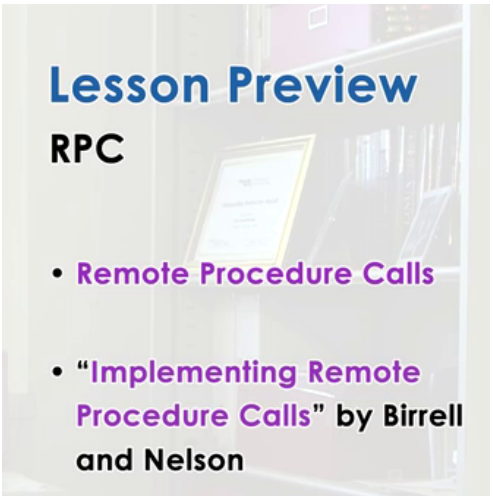
P4L1 - Remote Procedure Calls

01 - Lesson Preview

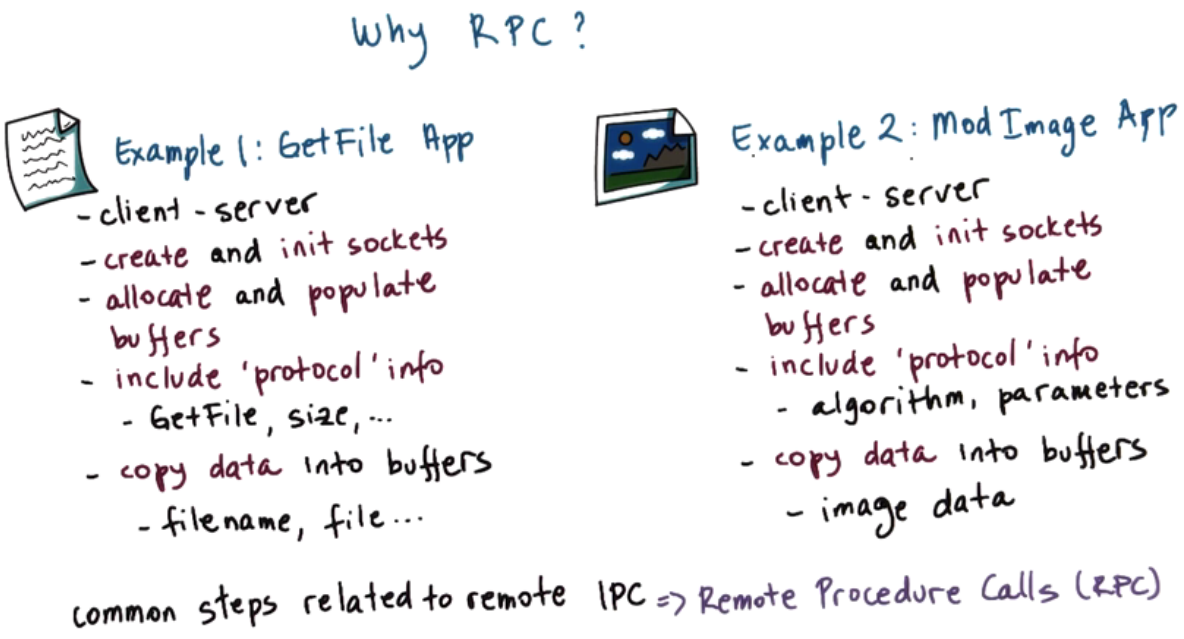


Instructor Notes

- Birrell, Andrew, and Bruce Nelson.
- "Implementing Remote Procedure Calls".
- ACM Transactions on Computer Systems, Vol. 2, No. 1, February 1984, Pages 39-59.

In the previous lessons we discussed several mechanisms for interprocess 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 discussions of RPC's, we will roughly follow the Birrell and Nelson 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 an RPC system that's in operating systems today.

02 - Why RPC

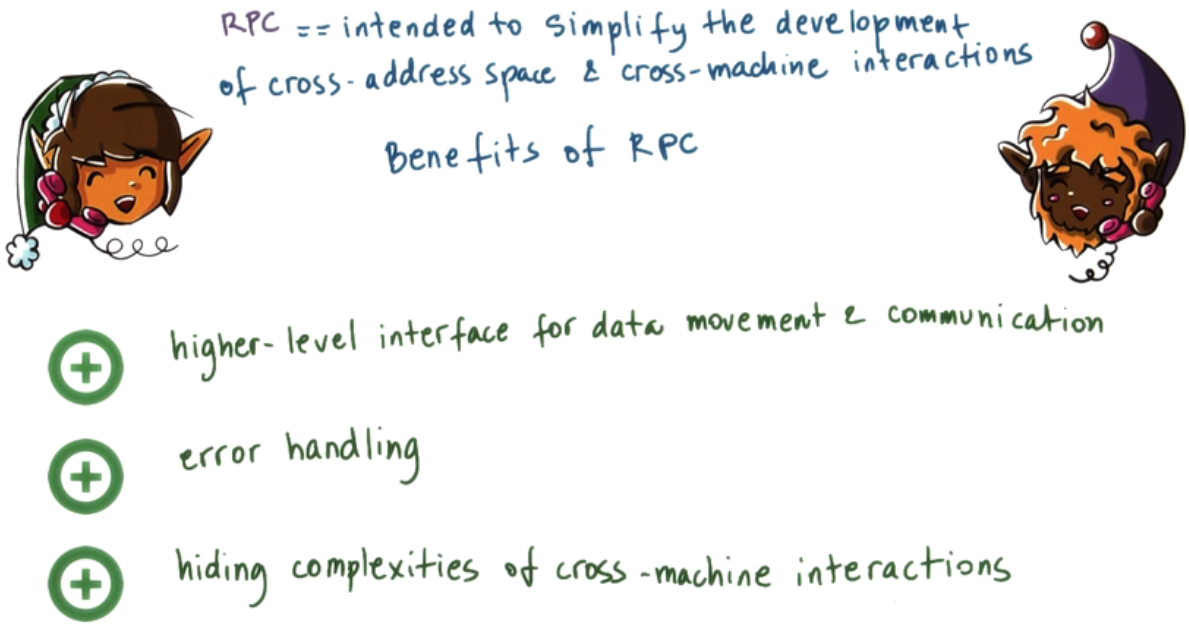


To understand why we need RPC, let's look at two example applications. The first one is an application where a client requests a file from a server and uses a simple getfile protocol that's like http request but less complex. In this location, the client and the server interact using a socket based API 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 ce this protocol will have something like getfile directives, and you have to specify the size of the buffer, and also you will have to explicitly copy the data in and out of these buffers, so copy the filename string or the actual file in and out of these buffers.

Now imagine another application that's also 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 grayscale version of an image, to creatsockets and populate them with anything that includes protocol related information, like for instance a low resolution version of an image, to apply some face detection algorithm. So it's in some sense similar to getfile but there is some additional functionality, 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 getfile 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 grayscaling or whether it's some face detection algorithm, along with any parameters that 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 the server returns that image back to the client after this particular function has been performed. That's different than the filename, the string that's being sent from the client to

the server and then the actual file that's returned in response. But a lot of the steps end up being identical in both cases.
In the 80's as networks were 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 related interprocess communications and need to be repeatedly re-implemented for 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 gave rise to Remote Procedure Calls or RPC.

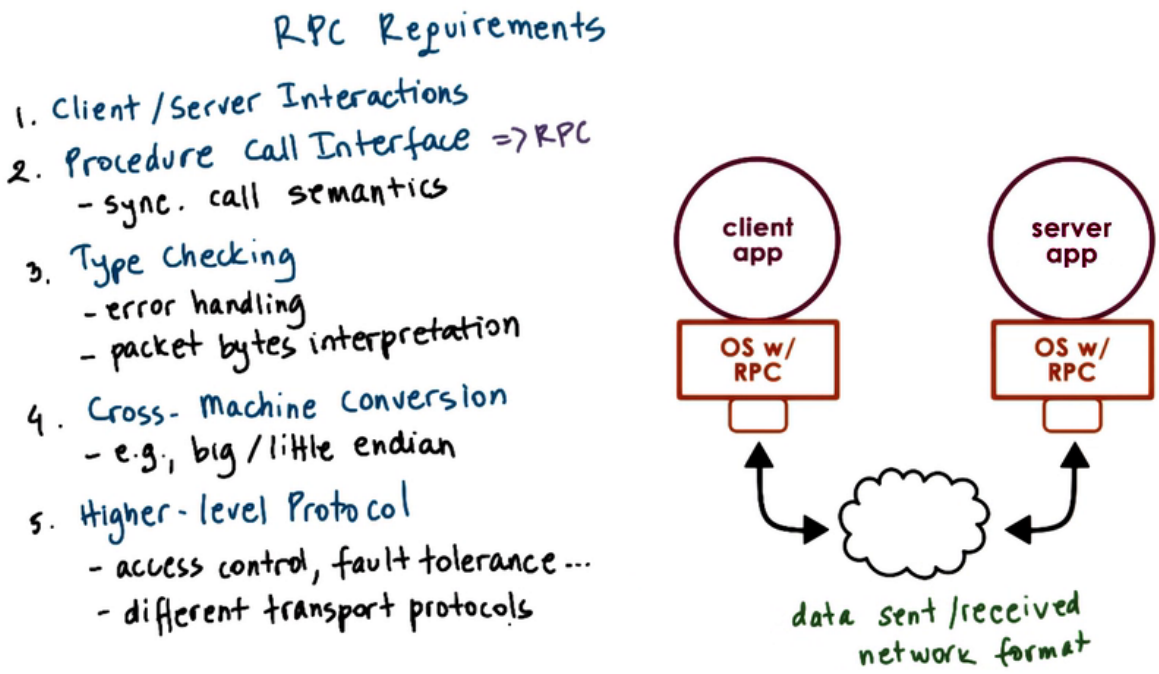
03 - Benefits of RPC



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, request, responses, acknowledgements, etc. What this also allows, it permits for RPC's to capture a lot of the error handling and automate it and the programmer doesn't really have to worry about that or at least the programmer doesn't have to explicitly re-implement the handling of all types of errors.
And finally another benefit from RPC is that it hides the complexities of cross machines interactions, 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.

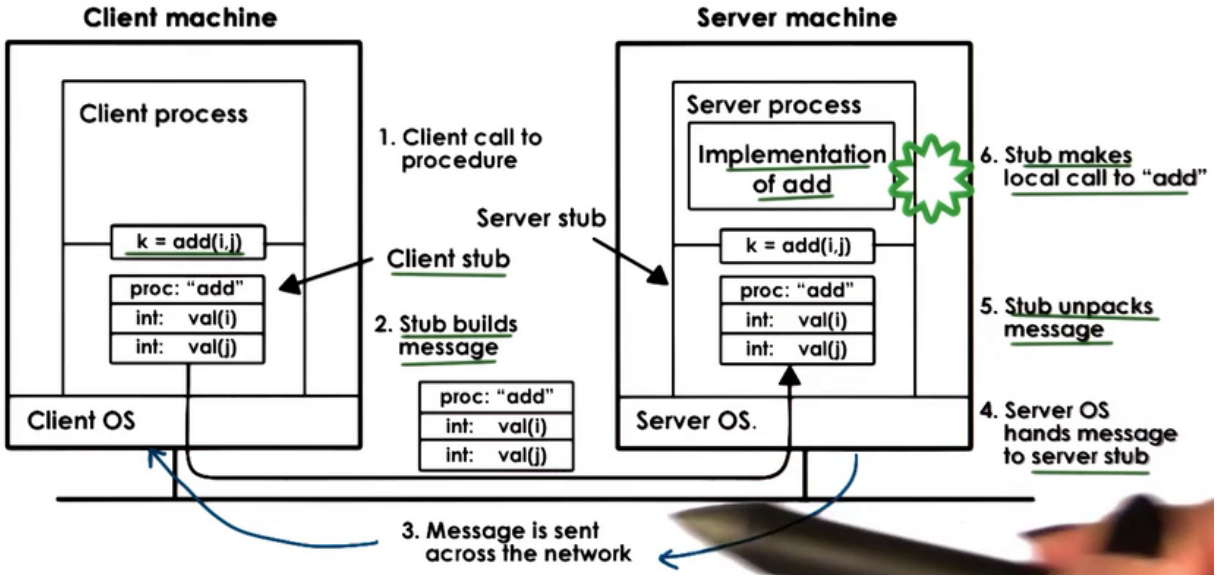
04 - RPC Requirements



Let's see what's required from the system software that provides support for RPC's. First, the model of interprocess interactions that the RPC model is intended for needs to match clients 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 were 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 call interface. This is why the term Remote Procedure Calls came to be. As a result, RPC's are intended to have similar synchronous semantics, just like regular procedure calls. What that means is that when a process makes a RPC 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 space. The execution of the thread will reach the point when the procedure call is made. At that point we jump somewhere in the address space where the procedure is implemented. The actual original thread of execution will not advance beyond that procedure call point until we get the results from the procedure and then when we move onto the next step we 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. RPC's have other useful features that are similar to regular procedure calls. One is typechecking. If you pass to a procedure an argument that's 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 runtime. 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 the those bytes can be useful when the RPC runtime is trying to interpret what do these bytes mean. Are they integers? Are they a 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 in 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 in the last position in the sequence of bytes that corresponds to the integer. 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 runtime on both endpoints to agree upon a single data representation for the data types. For instance, it can agree 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, RPC is 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 kind 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 the same request to either the same server or it can make an attempt to contact a replica of that original server that it was trying to contact.

05 - Structure of RPC



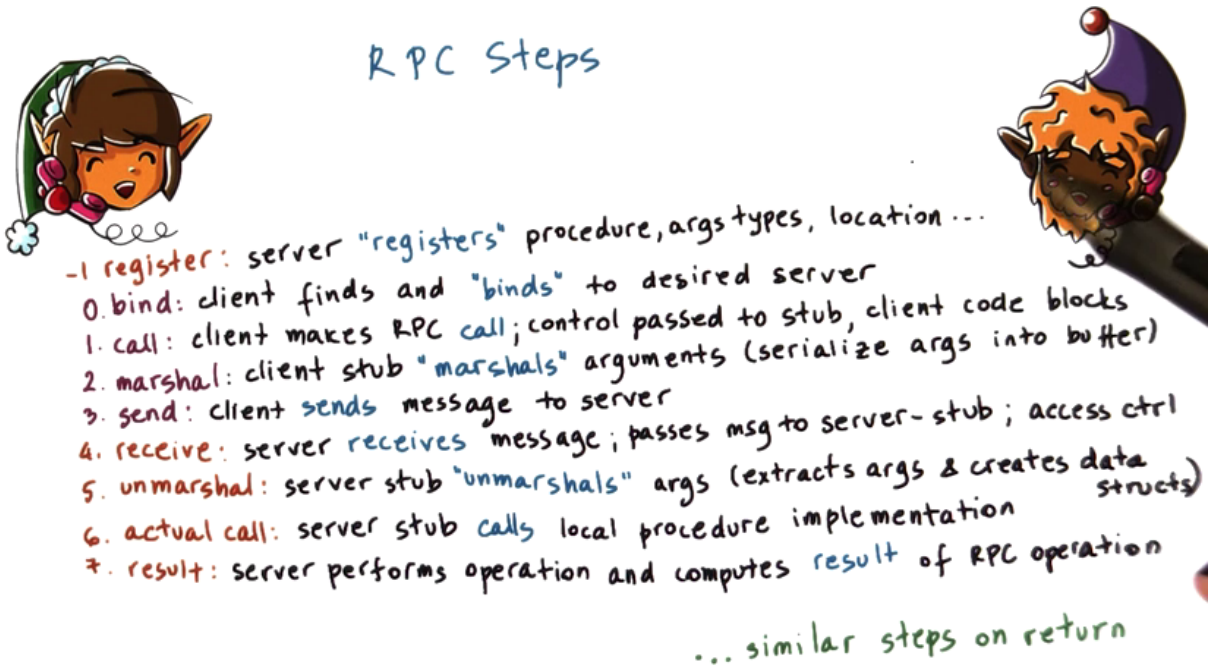
Errata

Here is a downloadable version of the Structure of RPC diagram.

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 operation, 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 of 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 details, this communication pattern will use RPC. Let's consider in this example that 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= add(i,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 the stub does is something entirely different. The responsibility of the client 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 this call "add" here..the call takes the execution of the client process into a portion of the RPC runtime, and by that we mean the system software that implements all of the RPC functionality. And the first step here is that stub implementation. After the buffer is created, the RPC runtime will send a message to the server process. This may be via TCP socket 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 the server process is running that is available to the client and that information is used by the RPC runtime to establish the connection and carry out all of the communications. On the server side, when the packets are received for this connection, they will be handed off to the server stub. This is the code that will know how to parse and interpret all of 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 server stub, once it sees that it needs to perform this add, it will know that the remaining bytes need to be interpreted like 2 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 space of the server process. Once all of this information is extracted on the server side and 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 addition 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 stub 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 runtime, the packets will be received, the result will be extracted from the 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, 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.

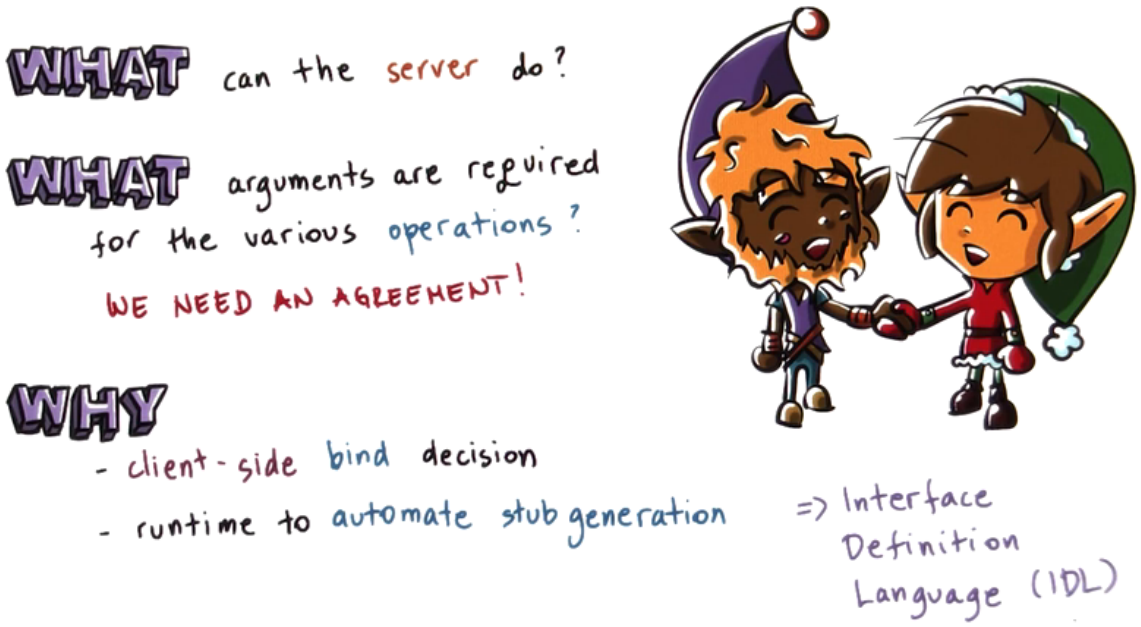
06 - Steps in RPC



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 address space, but the RPC runtime will need to send a contiguous buffer to the sockets for transmission. So the marshalling process will take care of this and will place all the arguments into a buffer that will be passed to the sockets. Once the buffer is available, the RPC runtime will send the message and 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 on 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 stub that this message needs to be passed to. In addition, it’s possible to include certain access control checks at this particular step. The server stub will unmarshall the data. Unmarshalling is clearly the reverse of marshalling, so this will take the byte stream that’s coming from the received buffers, it will extract the arguments, and it will create whatever data structures are needed to hold the values of those arguments. Once the arguments are allocated and set to the appropriate values, the actual procedure call can be made. This calls the implementation of the procedure that’s part of

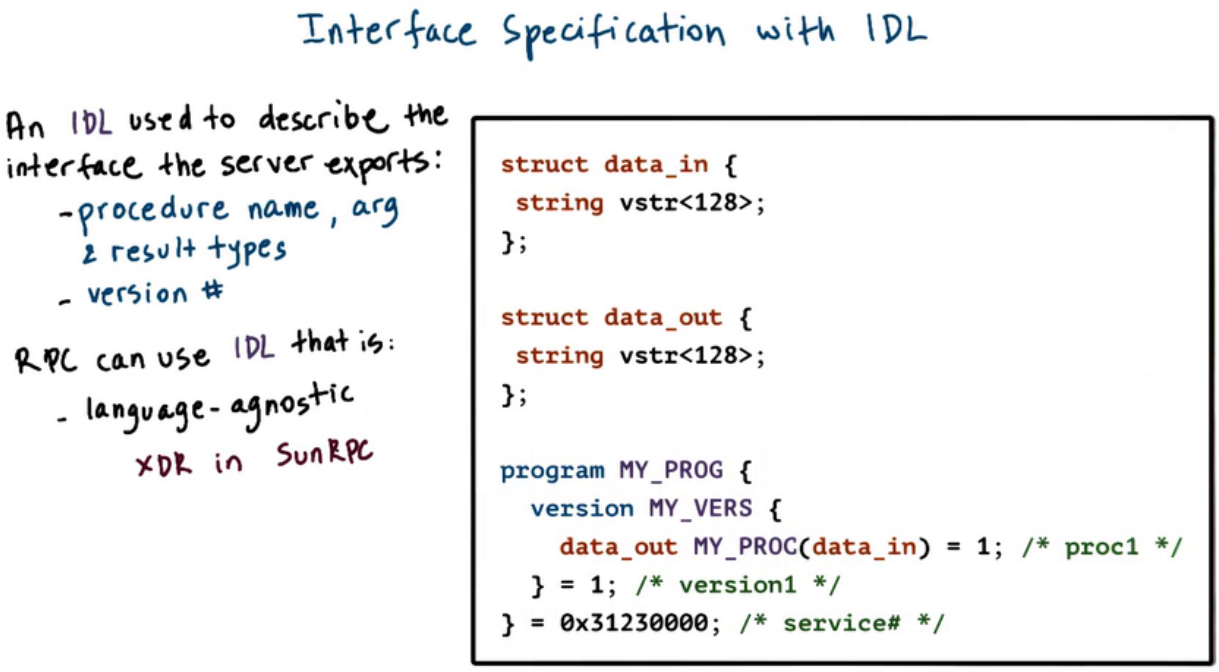
the server process. The server will compute the result of the operation or potentially it will conclude that there is some kind of error message that needs to be returned. The result will be passed to the server side stub and it will follow a similar reverse path in order to be returned back to the client. 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 bind or discover the server so that it can bind with it, but before that can happen somehow the server needs to do something so that it can be found. The server will need to announce to the rest of the world, what is 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.

07 - Interface Definition Language



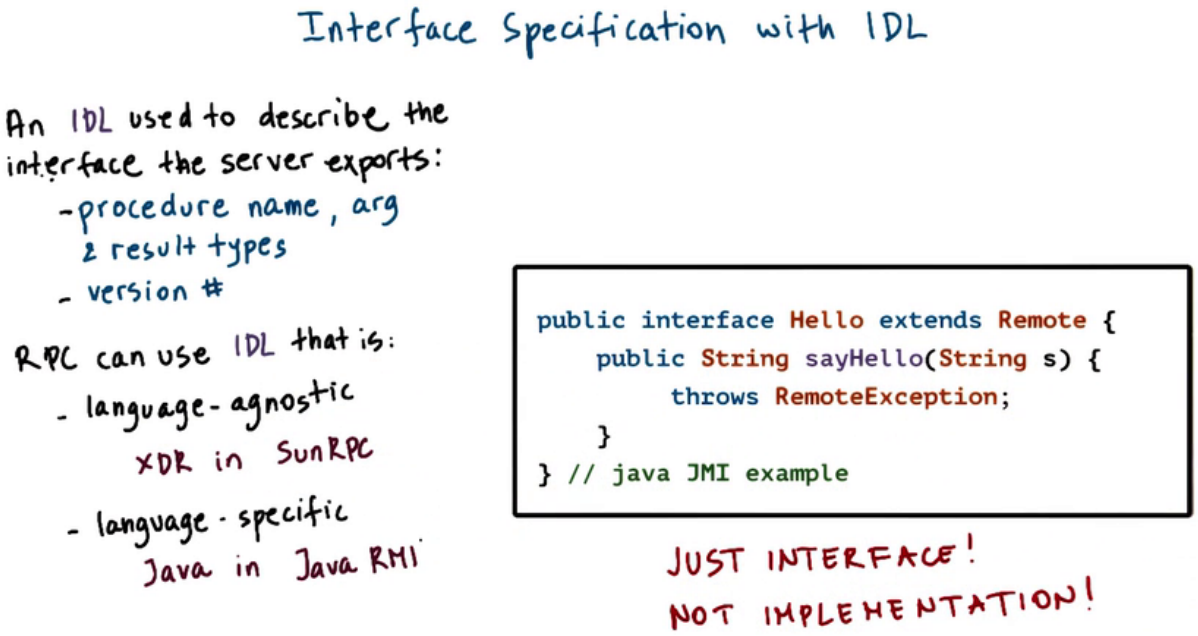
The nice thing about RPC is that the client and server don't have to developed together as part of the same application. They may be completely independent processes written by different developers, written even in completely different programming languages. But for this to work, there must be some kind of agreement, so that the server can explicitly say 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 runtime 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 IDL's. The IDL's serve as a protocol of how this agreement will be expressed.

08 - Specifying an IDL



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 a number of 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 on 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 client 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 for the interface specification that's completely agnostic to the programming languages that are

otherwise used to write the client and the server processes.

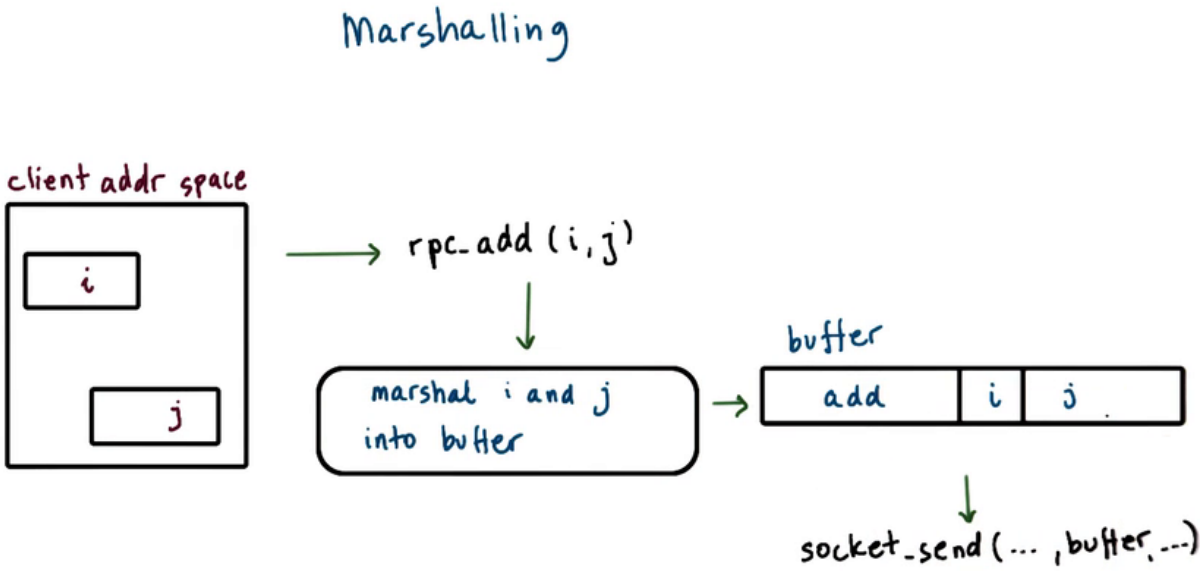


XDR (Sun RPC) Example

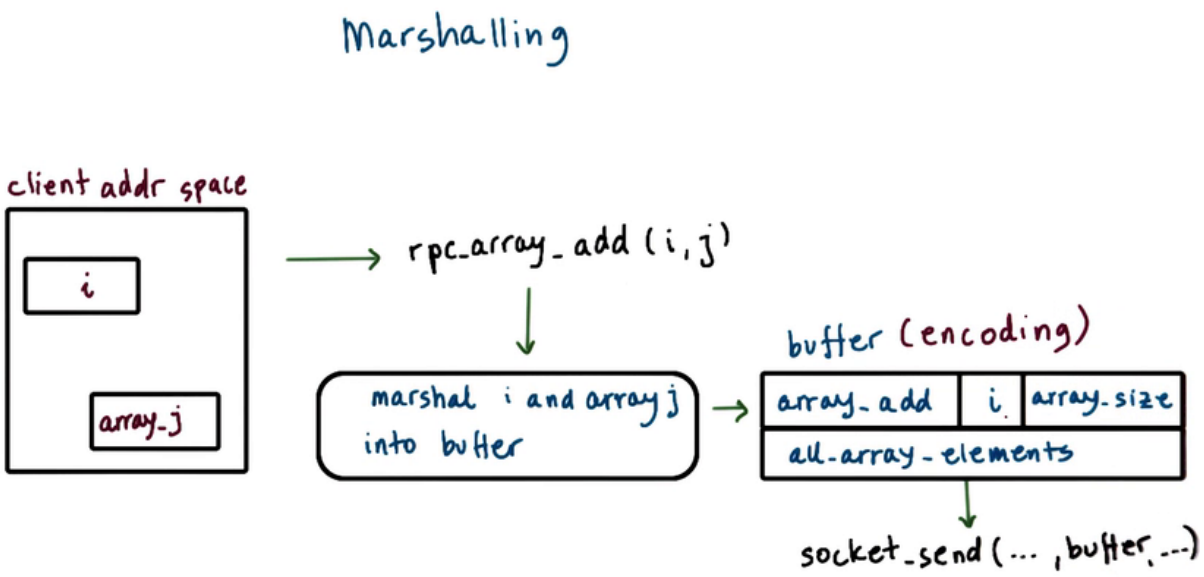
SunRPC which is an example of an RPC system that we will look at later in this lesson uses an IDL that's called XDR….External Data Representation and XDR is a completely different specification from any other programming language that's out there. We will describe XDR in more detail but here is an example 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 btw, to read ahead and examine a SunRPC example and look at XDR in more detail, there are links provided in the instructor notes. 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 the java equivalent of RPC, uses the actual, the same programming language Java to specify the interfaces that the RMI servers export. Here is an example of an interface specified for Java RMI's. 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 a specific programming language that's supported by the server for instance, then this becomes irrelevant. If they have to learn something, they might as well learn something simple and that is one of the goals that XDR has. Now let me reiterate 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 the information that's used in the service discovery process. The IDL is not actually used for the actual implementation of the service.

sca09 - Marshalling

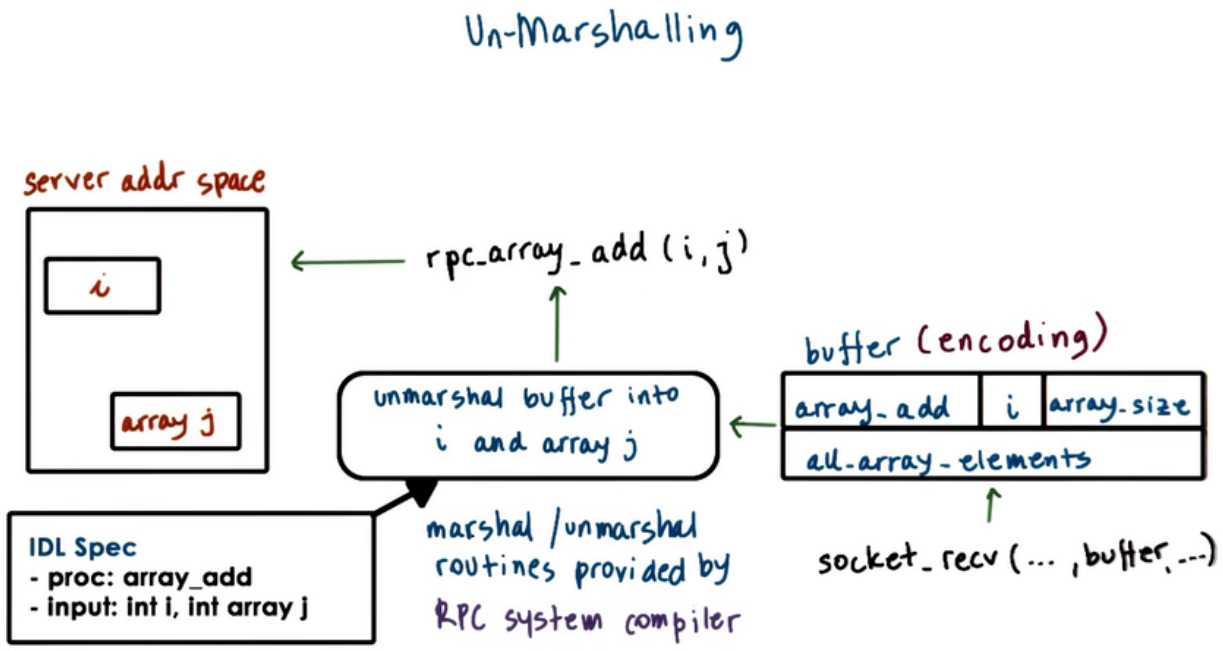


To understand marshalling, let's look at the "add" example again. The variables i and j are somewhere in the memory of the client process address space. They are two separate variables, so there's absolutely no guarantee that they will be next to one another. The client makes the call to the RPC procedure (RPC add) and passes i and j as arguments to it. At the lowest level of the RPC runtime, 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 arguments as well as some information about the actual procedure, 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.



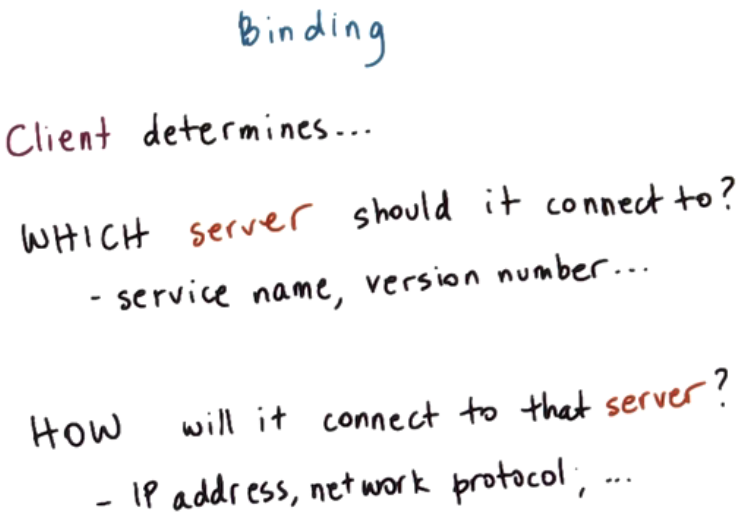
In case the previous example is too trivial, here is what would happen if we need to perform an array add procedure which takes as arguments an integer i and some array j and then adds this integer to all of the elements of the array. Then again, the marshalling code will need to serialize the arguments 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 marshalling 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 marshalling arrays is that we would just list all of the elements in 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 marshalling 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 in a byte stream so that anybody that looks at it can actually make sense of it.

10 - Unmarshalling

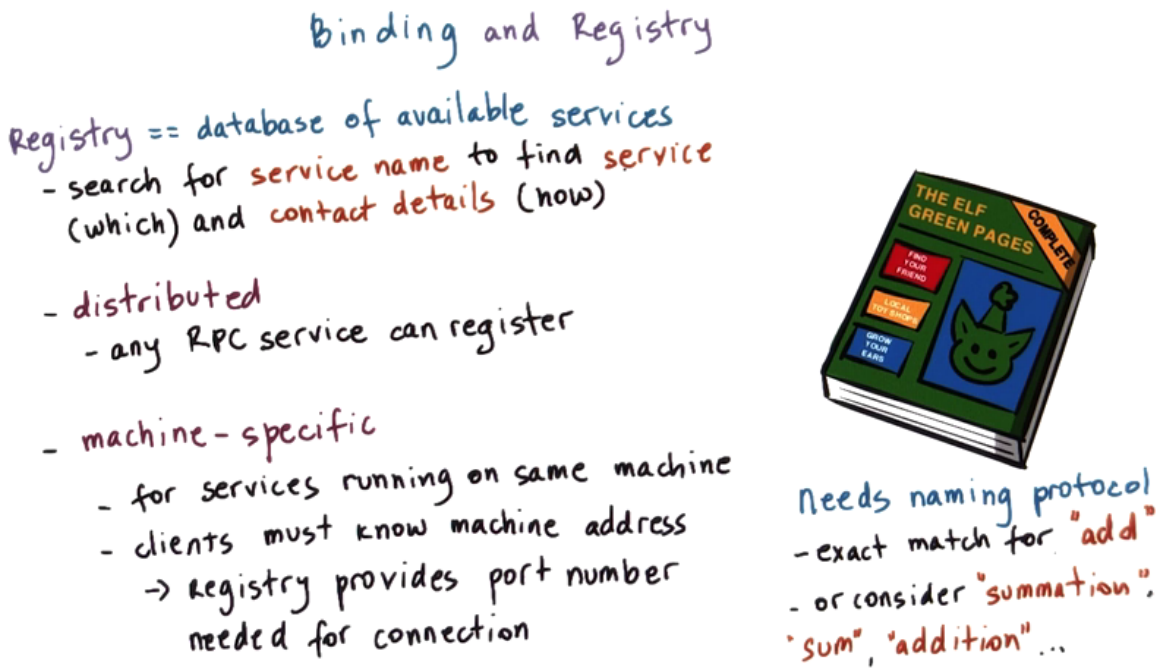


In the unmarshalling code in contrast, we take the buffer that's provided by the network protocol and then based on the procedure descriptor and the data types that we know are required for that procedure descriptor, we parse the rest of the bytestream from that buffer. We extract the correct number of bytes and we use those bytes to initialize data structures that correspond to the argument types. As a result of the unmarshalling process, these i and j will be allocated somewhere in the server address space and they will be initialized to values that correspond to whatever was placed in the message that was received by the server. Now again, the marshalling and unmarshalling 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 generates the marshalling and the unmarshalling routines that are used in the stubs 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 it's encoded in a byte stream, that's an example. What will take place in these autogenerated routines? And there are other examples of what constitutes encoding, for instance converting integers like this value i from one endian format to another endian format like from big endian to little endian depending on what's required by the server or by the client for the result, that's an example of an automated action that will be incorporated into the marshalling code. Once this IDL is compiled and all of the code is generated that provides the implementation for the marshalling and unmarshalling 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 client codes when generating executables.

11 - Binding and Registry



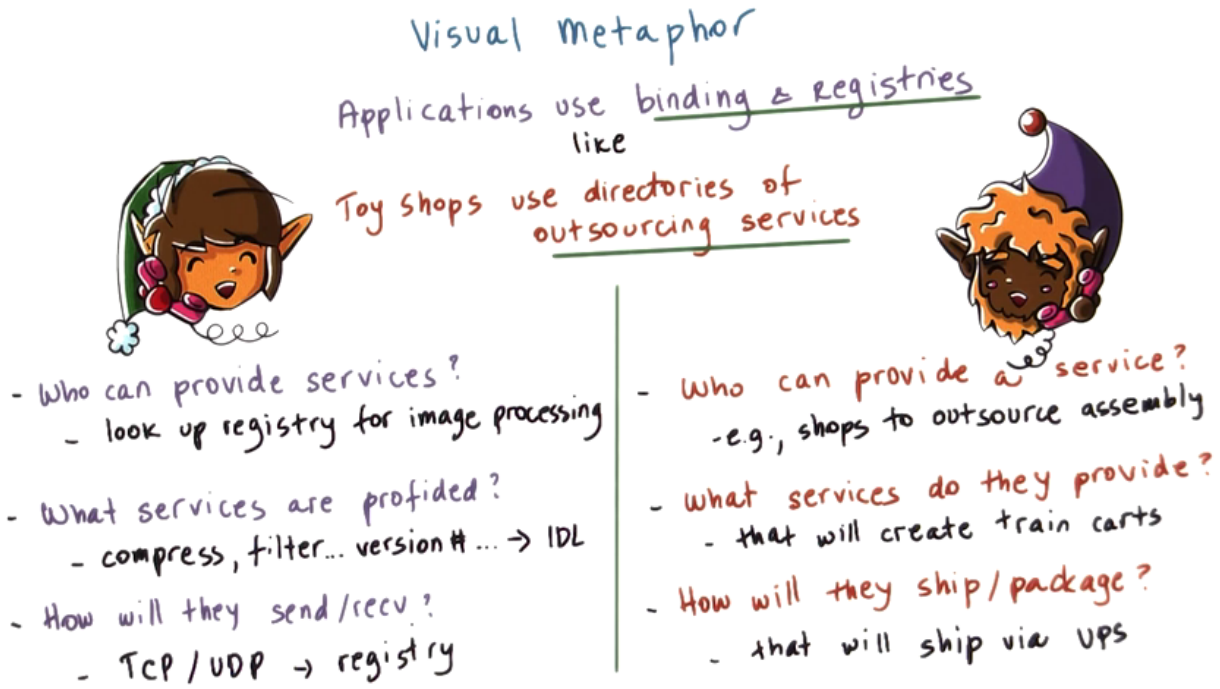
Let's talk a little bit about binding now. Binding is the mechanism that's used by the client to determine which is the server that it needs to connect to based on things like the name of the service that it needs performed, the version number of that service, and also it's used to determine how to connect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



To do this, to support binding, the system software needs to support some form of database of all of the available services and this is often called a registry. You can think of the registry as the yellow pages that you will need to look up the service name and then find the best match

based on the protocol, the version number, the proximity, or 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, maybe called something like RPCregistry.com that any RPC server can register with and the clients then have a well known contact point on 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 some 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 is 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 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 these function names or service names is a fair candidate to be considered when trying to find the best match. Allowing this type of reasoning will require support for things like oncologist? or other cognitive for learning methods and we will not discuss this in this course.

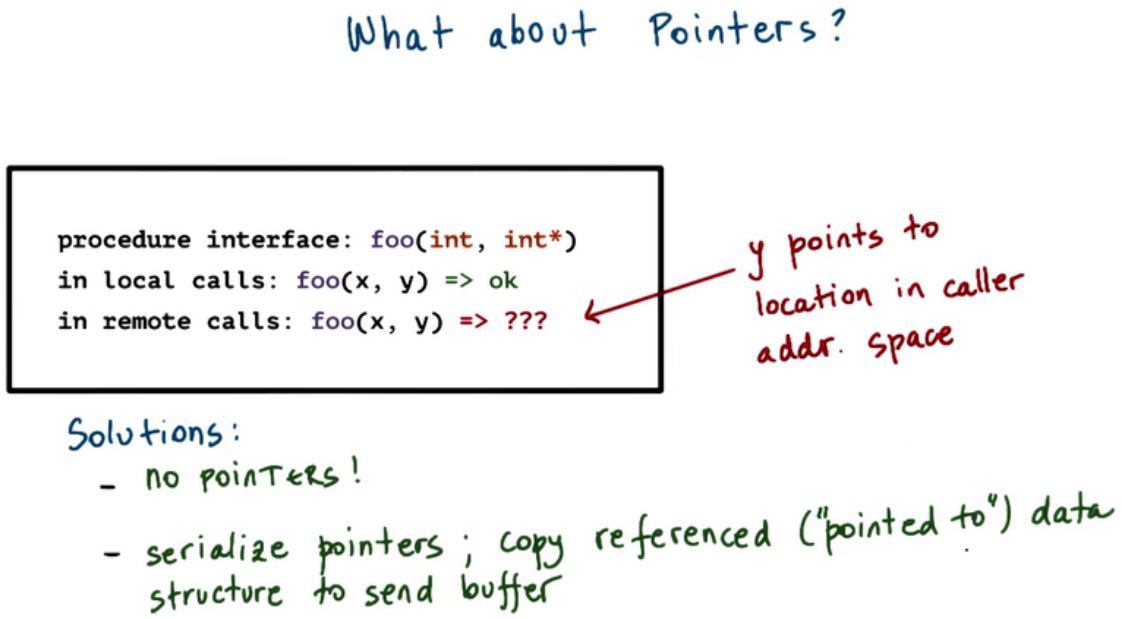
12 - Visual Metaphor



To illustrate the use of binding and registries by applications when they use RPC's 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 use some kind of outsourcing service, the manager will

want to know who out there can provide that particular service, what are the specific service details that those outsourcing companies offer, and then exactly what are the specific shipping or packaging options they provide. For instance, the toy shop manager may consider looking at the directory service to find out what are the shops where outsourcing of assembly operations can be supported, you may 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 carts 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 system and the applications use of bindings and registries in RPC. Now we can see the same types of steps are required to be performed by applications when they rely on 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 interfaces can be described in some standard way and then finally also the registry will provide information regarding the protocols that a particular server or service instance supports like TCP or UDP. The applications can take all 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.

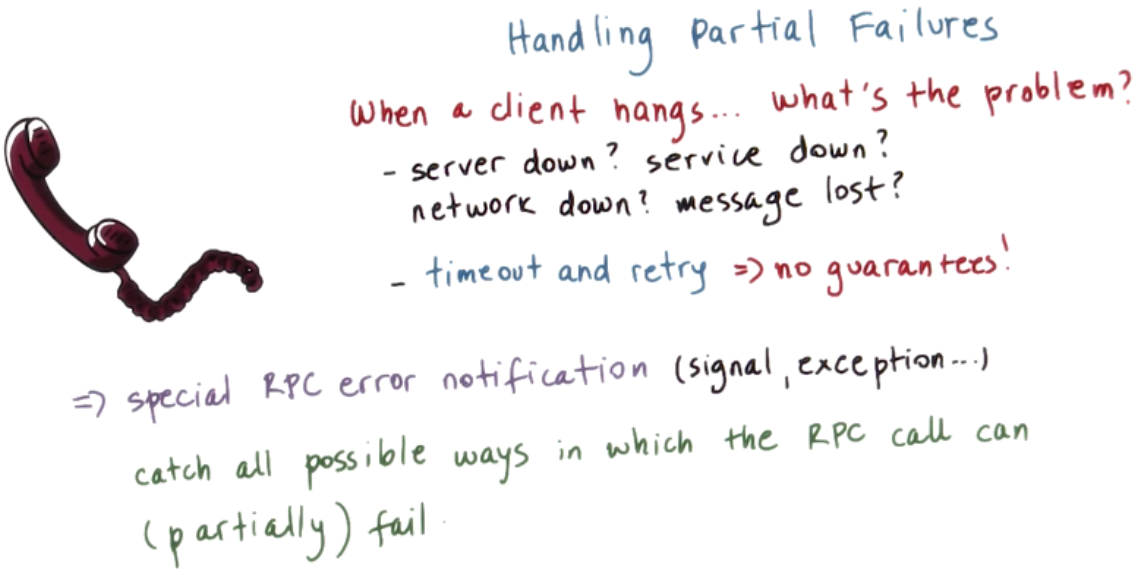
13 - Pointers in RPCs



A tricky issue when it comes to RPC's is the use 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 a second argument is a pointer to an integer or an integer array. When this procedure is called, the second argument is a pointer to some address in the address

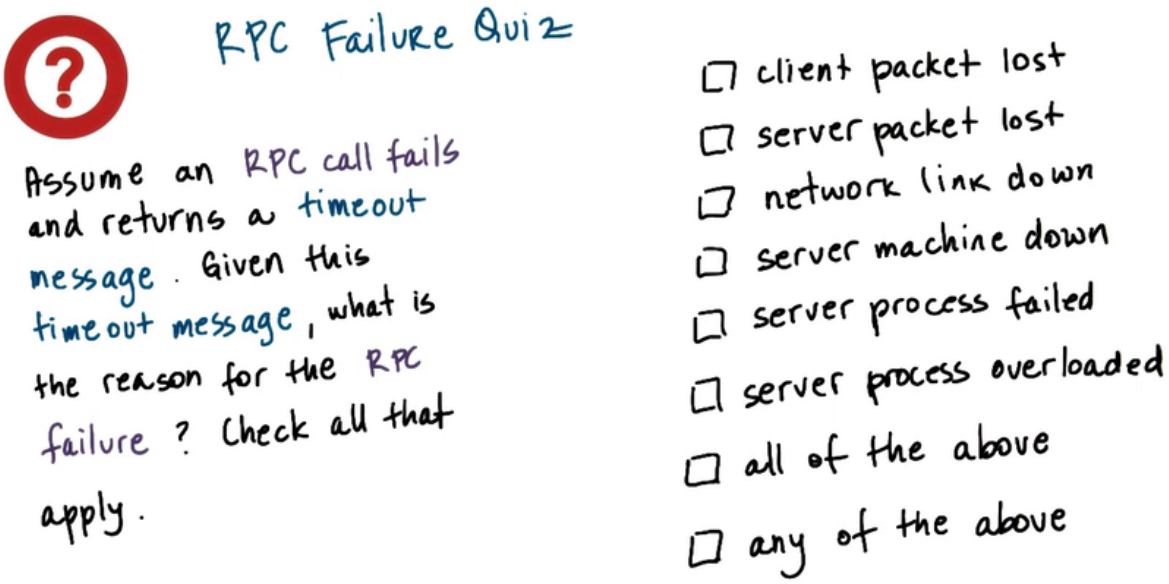
space of the calling process where the particular variable, this argument is stored. However in RPC, passing a pointer to the remote server makes no sense since this pointer points to some location in the caller's address space. The server cannot possibly get to the contents that are stored at this particular address. To solve this problem, RPC systems can make one of two decisions. The first decision is not to allow for pointers to be used as arguments of any procedure that's 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 runtime to insure that the marshalling code that gets generated understands the fact that an 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 is that it will copy the reference into the data buffer, into one serial representation. On the server side, the RPC runtime will first have to unpack all of the data to create the same data structure, then it will record the address to this data structure and that is the value that the pointer will use as an argument when it makes the call to the actual local implementation of this particular operation.

14 - Handling Partial Failures



Since we're talking about the trickiness of RPC calls, let's also talk about errors and fault handling and reporting. When the client hangs while waiting on a remote procedure call, it is often hard to take what is 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, 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 timeout whenever a client RPC call hangs and then retries them automatically, there really is no guarantee 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's possible to really understand what is the cause of the error, but in principle that is too complex, it would take a lot of overhead, and ultimately it's still unlikely that it will provide a definitive answer. For this reason, RPC 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.

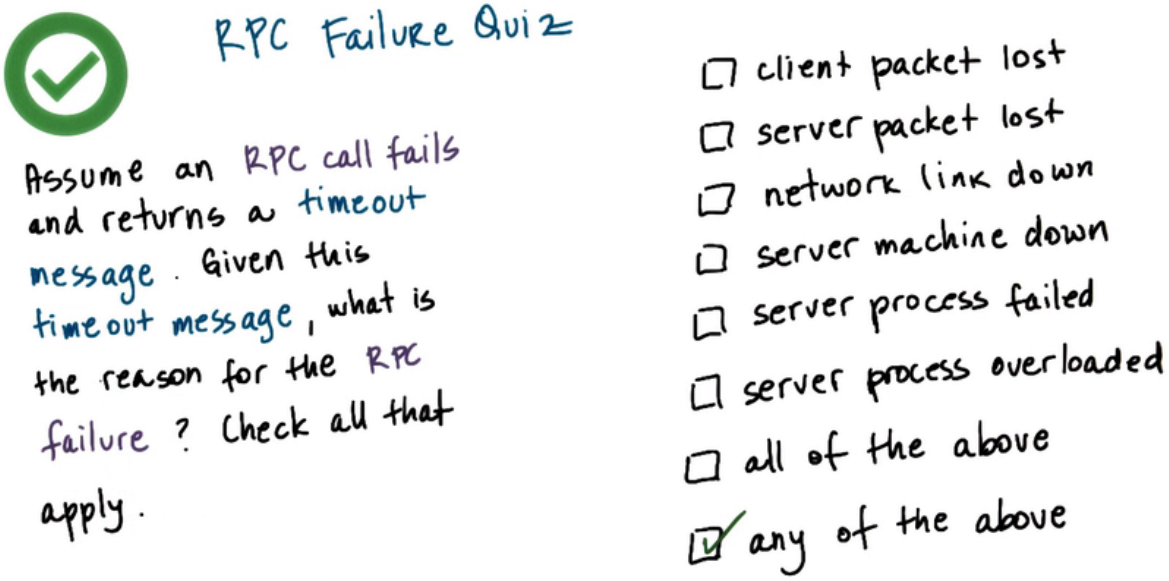
15 - RPC Failure Quiz



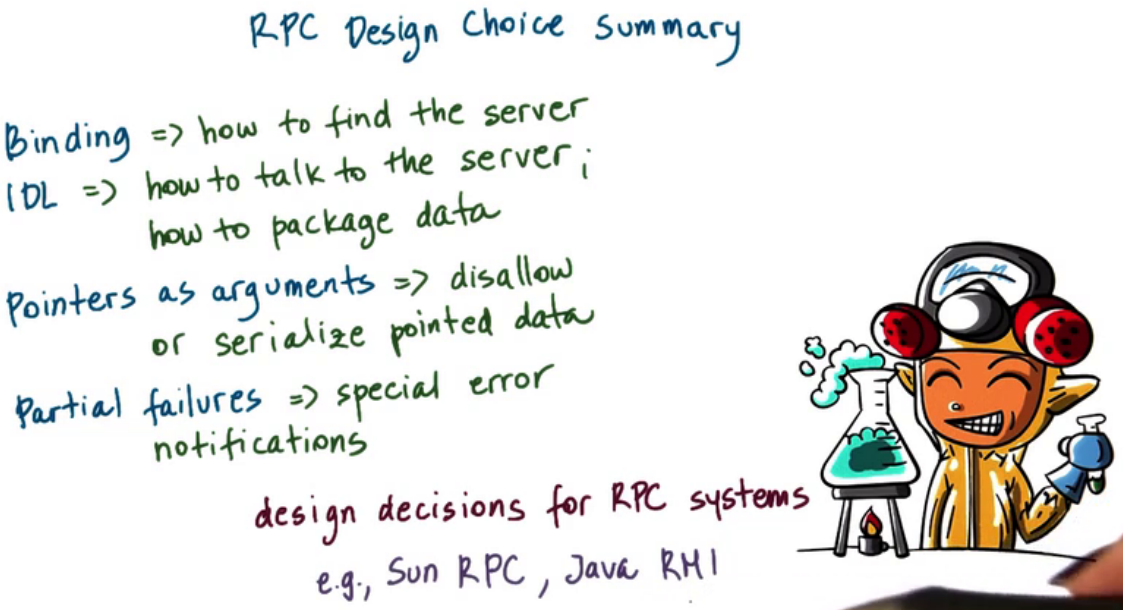
Quiz Errata

This is a multiple choice question so you may ignore the "Check all that apply" clause.

16 - RPC Failure Quiz



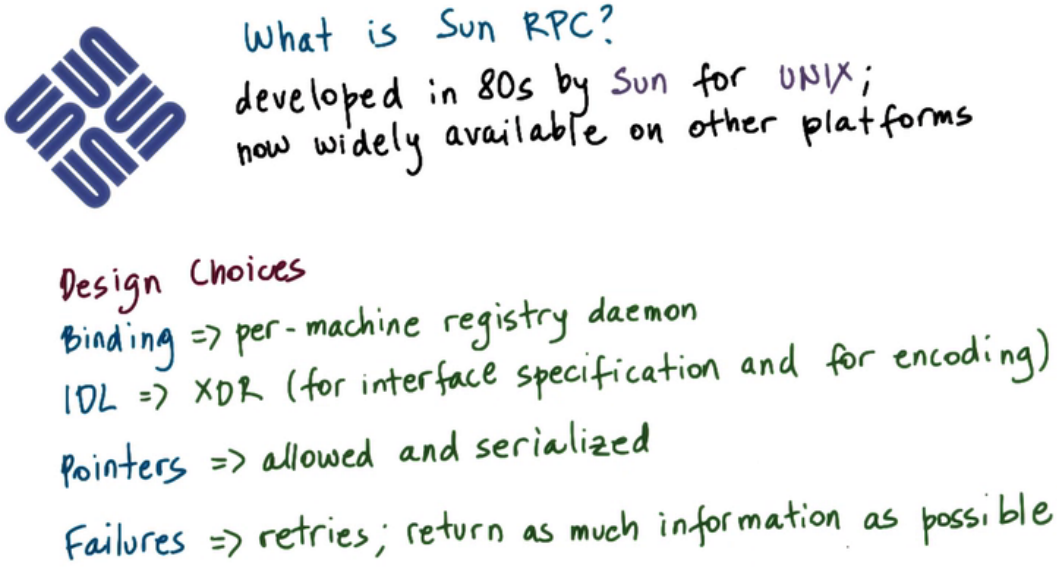
17 - RPC Design Choice Summary



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 discussed the use of interface definition languages to determine how to package arguments and results that are being exchanged among the client and the server. And in that sense, the ideals used to specify how the client and the server talk to one another, how they're able to understand each other. Next we observed the problem of dealing with pointers as

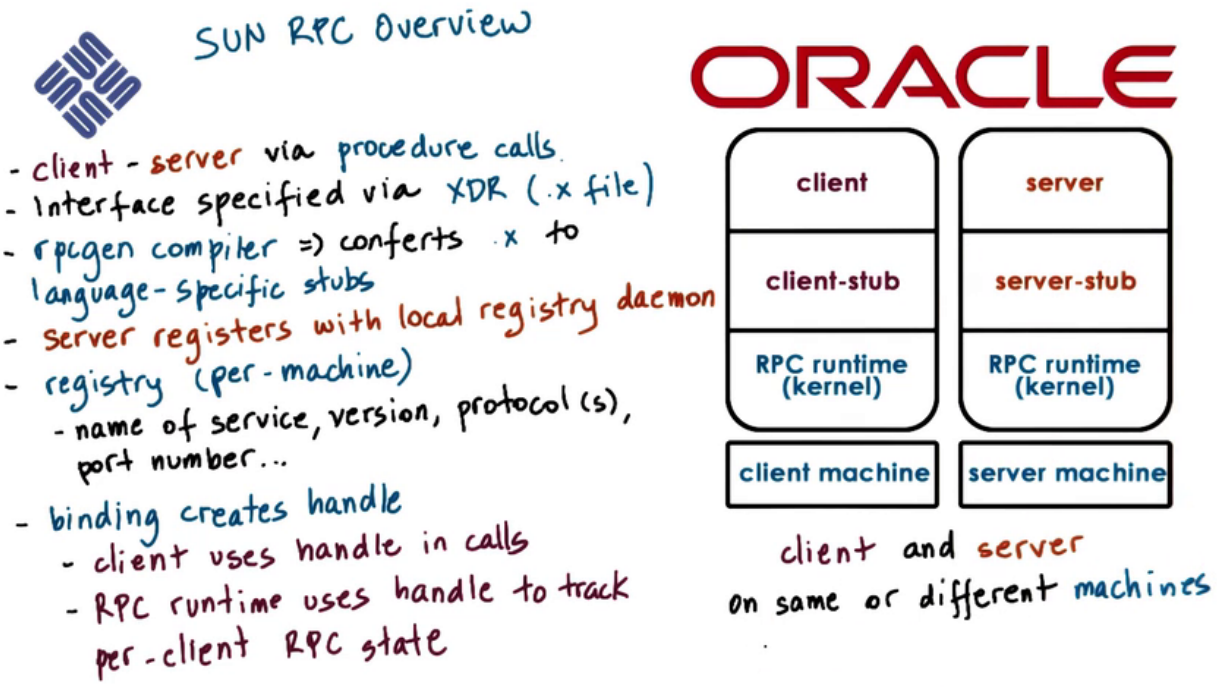
arguments in remote procedure calls and we said that the use of pointers should either be completely disallowed or that the RPC system should build in some kind of support to serialize the data that's being pointed to. Finally, we also talked about partial failures and explained how it is tricky to determine exactly what went wrong in an RPC system and then instead the RPC runtime provides some special errors and tries to, inasmuch 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 mentioned 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 will 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.

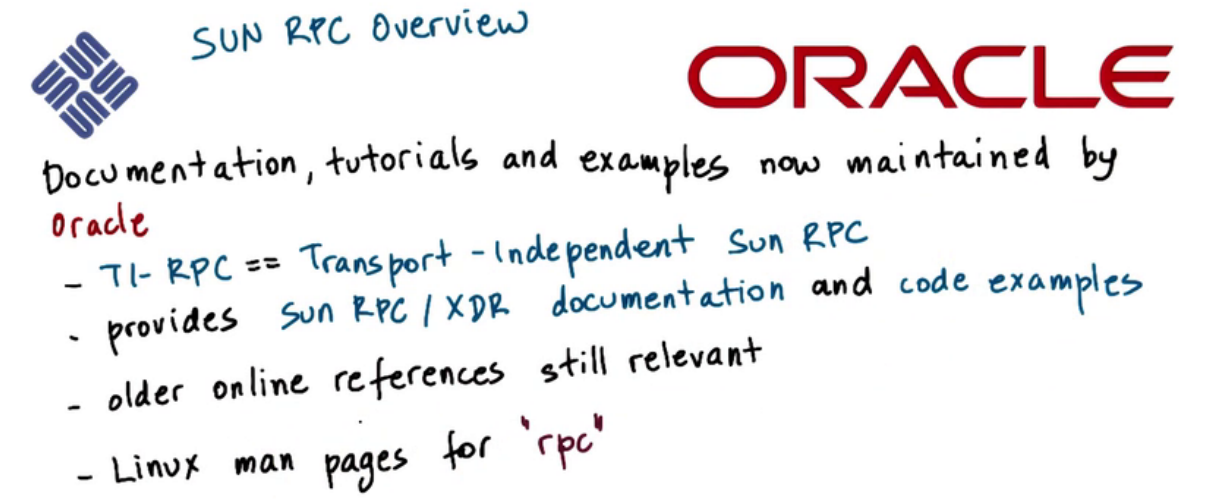
18 - What is SunRPC



Sun RPC is an RPC package originally developed by Sun in the 80's for their network file system or NFS for unix systems. But it became popular and now it's widely available on other platforms. Sun RPC makes the following design choices. In SUN RPC, the server machine is known upfront 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's used by client or by the server process. To maintain neutrality, Sun RPC relies 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. Sun 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 a 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 runtime 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 there is an unsupported protocol or version or there is simply a timeout related failure that just covers all of the other types of possible failures.

19 - SunRPC Overview



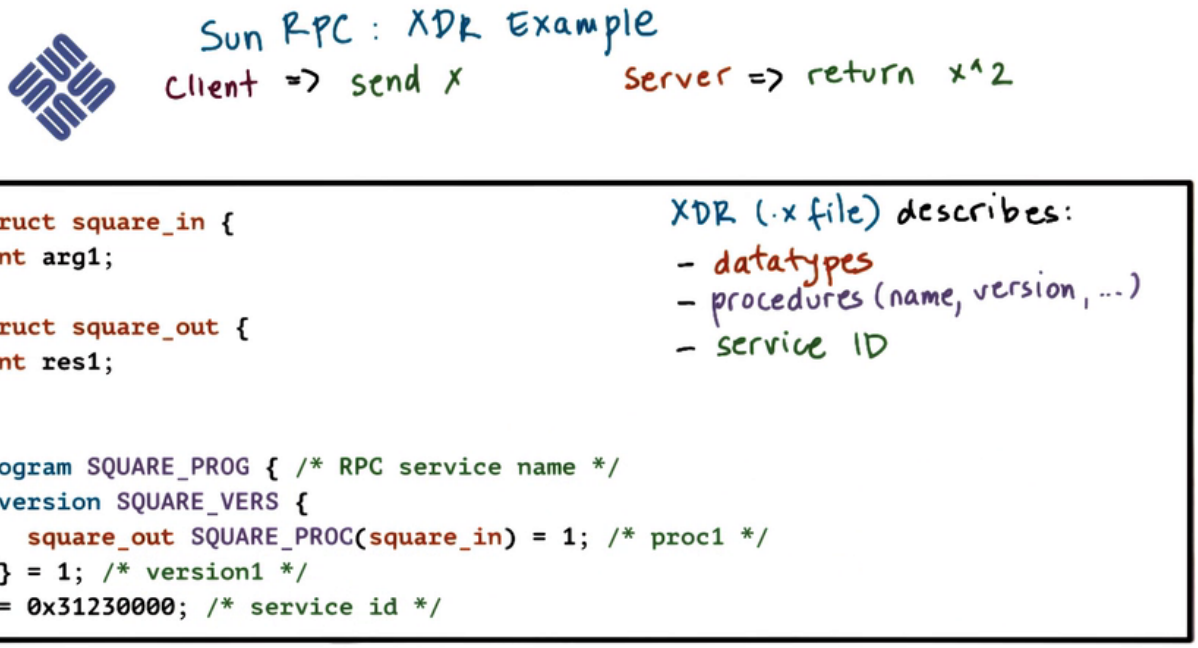


[Sun RPC Documents (maintained by Oracle)]

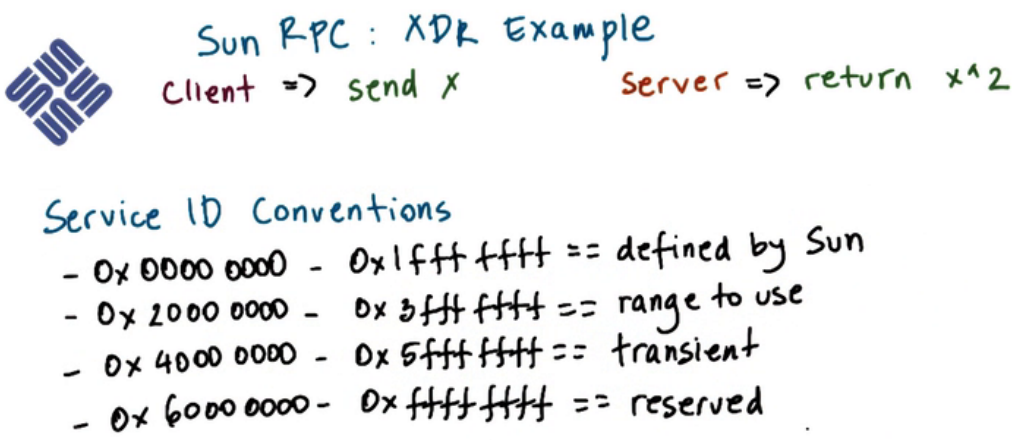
Similarly to the generic description of RPC, with Sun RPC the client and the server are allowed to interact via procedure call interface. The server specifies the interface that it supports in a .x file written in XDR. Also Sun RPC includes a compiler called rpcgen that will compile the interface specified in the .x file to a language specific stub. It will generate separate stubs for the client side and for the server stub. The server process when launched will register itself with the 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, any 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 to the server. A 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 a 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 states. I should note that with Sun RPC or any other RPC for that matter, the client and 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 call semantics which is more complex than the IPC mechanisms that we saw before. 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 the 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 Transport Independent RPC and that means that protocol that will be used for the client and server communication doesn't have to specified at compile time. It can be specified dynamically at runtime. Other than that, the documentation and the examples closely follow the original SunRPC 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 the linux supported API's.

20 - SunRPC XDR Example



We'll now take a look at the various components of Sun RPC using an example. The client again will be contacting a server than can perform calculations except this time the client will pass a single argument X for which it will want the server to compute the square value X2. Here is 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_PROG that has one argument of the type square_in and returns a result of the type square_out. The data types 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, a 32 bit integer. Also note that this notation '_in', '_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 procedures 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 is associated with it. This is 1 in this case. This number is not used by the programmer. This will be used internally by the RPC runtime when it's trying to identify which particular procedure is being called. So it's not going to pass between the client and 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 service is 1. Over time however we may choose to refine the SQUARE_PROC procedure or add additional procedures and as we're doing that we don't want to be forced to immediately and 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 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 version of the same procedure and this helps with in general the evolution of the system. We don't have to coordinate and upgrade all of the servers and all of 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 such as 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.



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.

21 - Compiling XDR

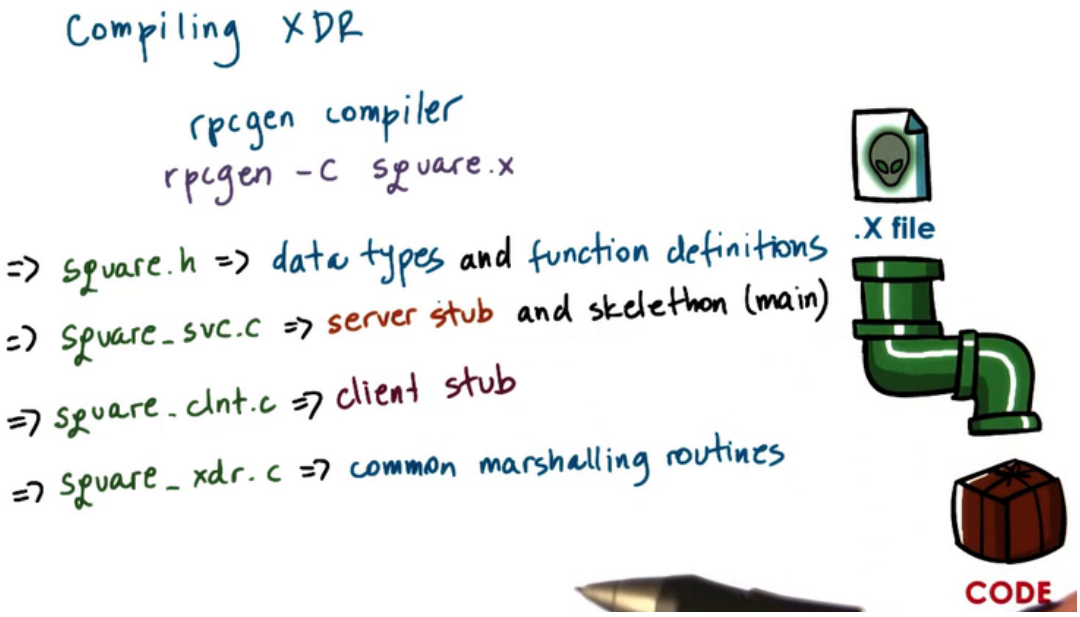
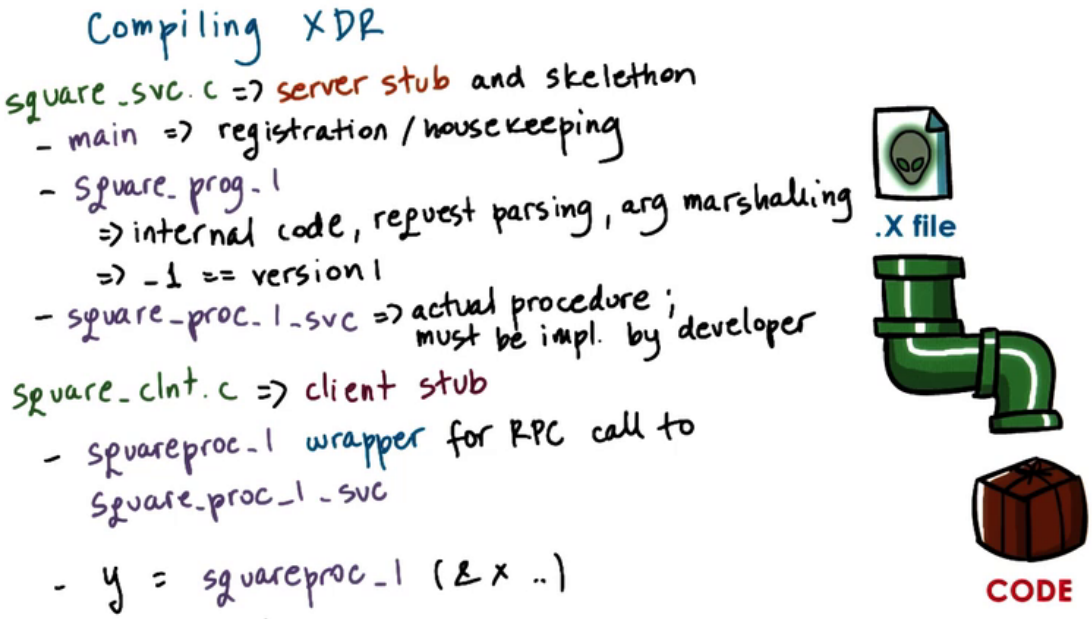


Image Errata: skelethon = skeleton

Let's show you actually compile a .x file. Assume that we're using the same Squared example as in the previous videos and 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 the full command is `rpcgen -c square.x`. The outcome of this operation will be that a number of files will be generated. First, it will generate a header file square.h that will have all of the language specific definitions of data types and

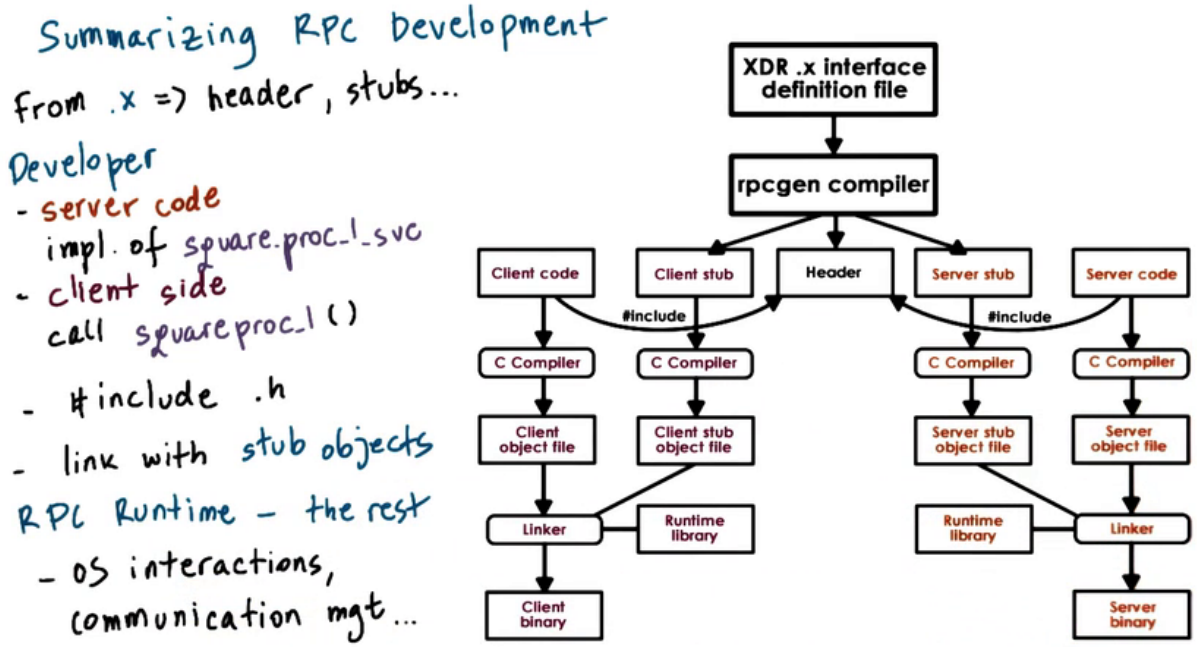
function prototypes. Next, it will generate the code for the client and the server side stubs. For the client, this is a proper stub for the server side code this actually also includes the skeleton of the actual server, so it has the main routine included. The only thing that's not available will be the actual implementation of the service of the procedure and this makes perfect sense since the compiler has no way of knowing what exactly do we want a particular procedure to do, in this case squaring a number. Finally, the compilation step will also generate a separate file square_xdr.c. And this will include some common code that's related to the marshalling and unmarshalling routines for all of the data types, the arguments and the results that are used both at the client and on the server side.



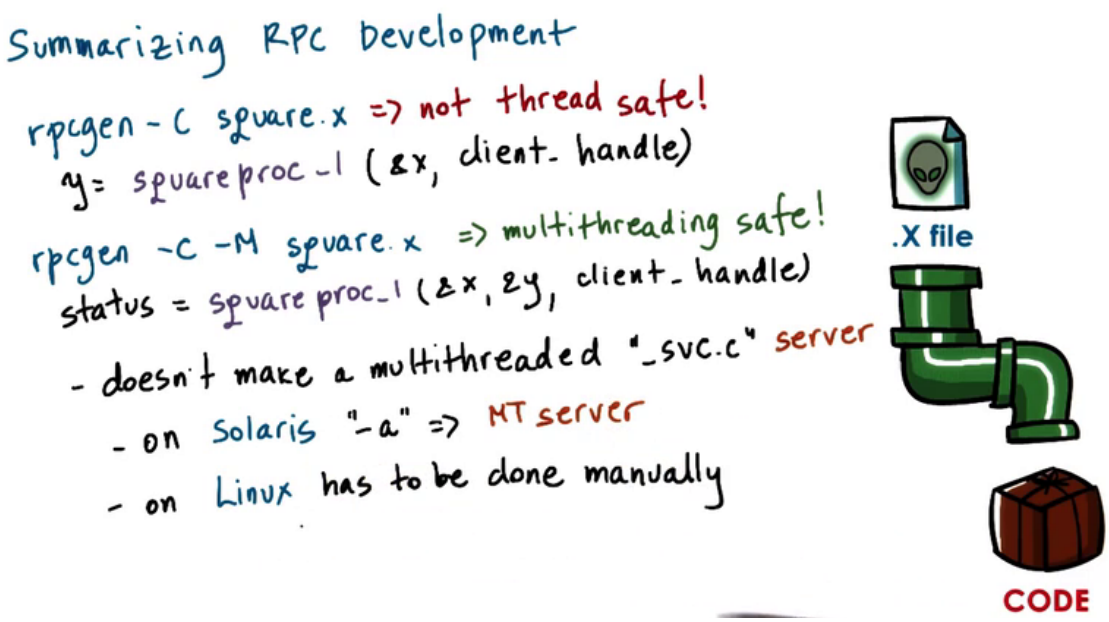
If you take a look at the file square_svc.c which stands for service, you will see that it has two parts. The first part is the main function for the server and that will include the code that does the registration step and also some additional housekeeping operations. In addition to main, the stub will 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 automatically generate 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 operations will be invoked here and other steps. In addition, in the stub 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 described, this is the procedure name and that will include also the _1 that refers to the version number and this piece of 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 square_proc_1_svc is actually called. Once we have all of this, the developer then writes the client application and makes a

call to this wrapper function that looks something like this y=squareproc_1(&x…). This very much look 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.

22 - Summarizing XDR Compilation



We will now summarize one more time the steps involved in developing RPC applications. 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 _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, 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 the .h file particularly the auto generated ones from the rpcgen compiler. And also that it links the client and server code with the stub objects. The RPC runtime that is called from the stub provides all other functionality including interactions with the operating systems, creating sockets, managing connections and everything else.

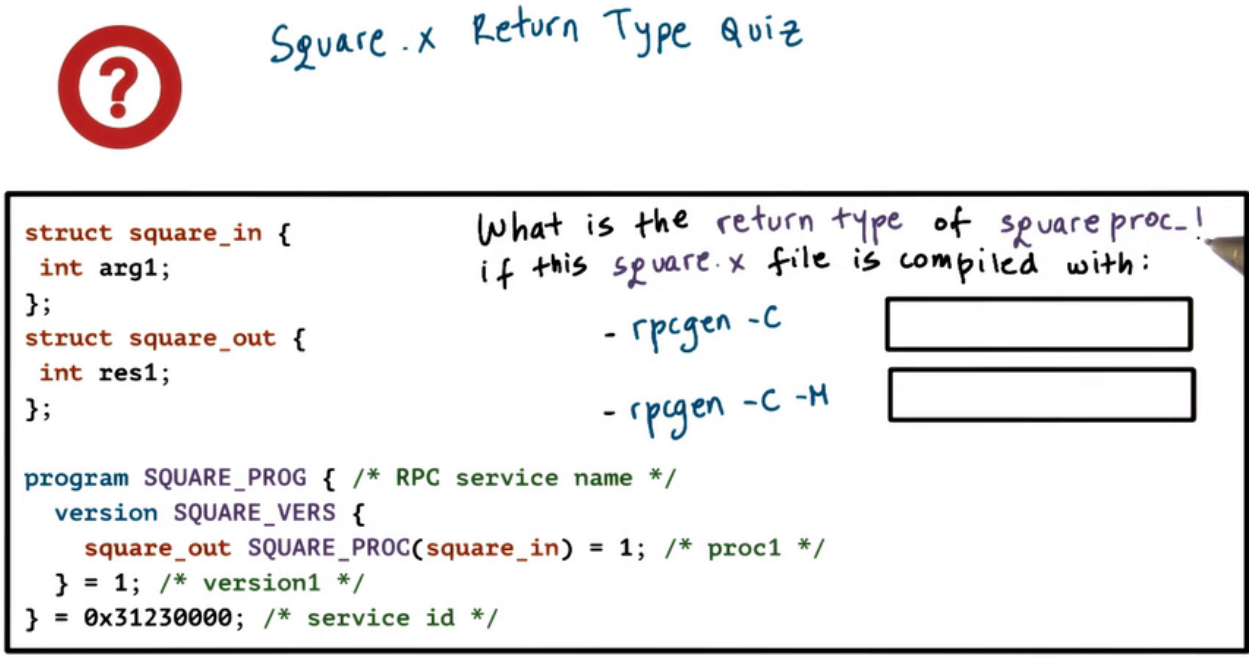


Errata

Here is a more readable version of the XDR compilation diagram.

I should point out that rpcgen, when used only with the flag -c generates code that is 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 at the runtime level, there are a number of statically allocated data structures including 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 for multithreading safe. This will also create a wrapper function square_proc_1 however it has a different signature and it’s 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 multi threaded server. The implementation that’s provided that generated in the _svc.c file.. that won’t be multithreaded. On Solaris platforms there is another option “-a”. Using this option that actually generates multi-threaded server code. But on 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 - Square.x Return Type Quiz



Instructor Notes

To complete the quiz, download the square.x file (if following along with the hints, you will need to rename it to square.x), and run rpcgen!

You can run rpcgen on the Ubuntu VM provided with this class. If you have not setup the Ubuntu VM yet, follow the Ubnutu VM Setup Instructions.

Quiz Errata

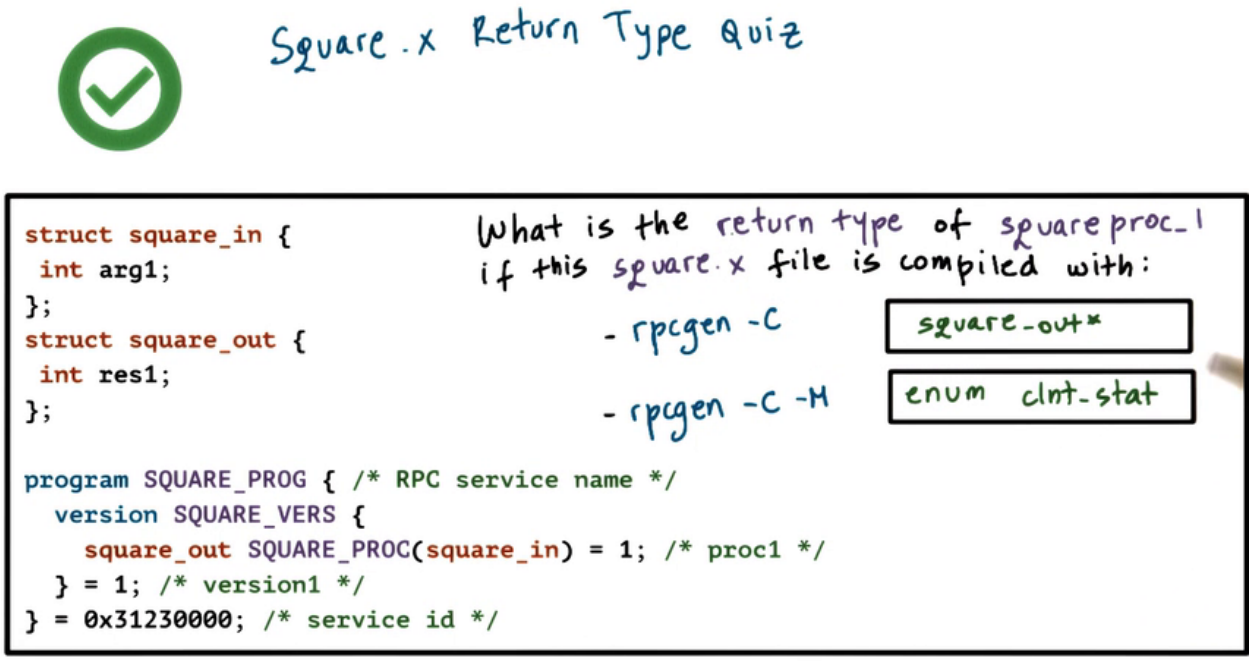
There are two issues with the rpcgen commands. First, you should use the following format:

- rpcgen square.x [flags]

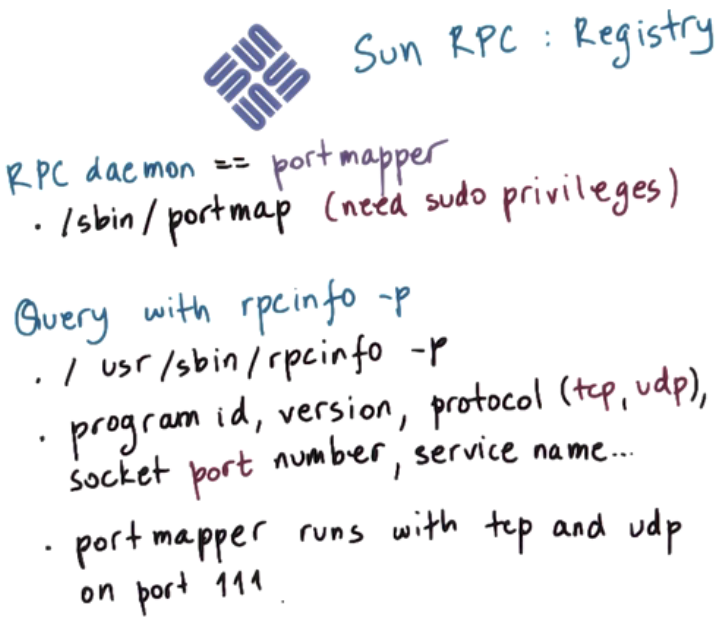
And, because the flags may be difficult to read, here are the full commands:

- rpcgen square.x -C
- rpcgen square.x -C -M

24 - Squarex Return Type Quiz



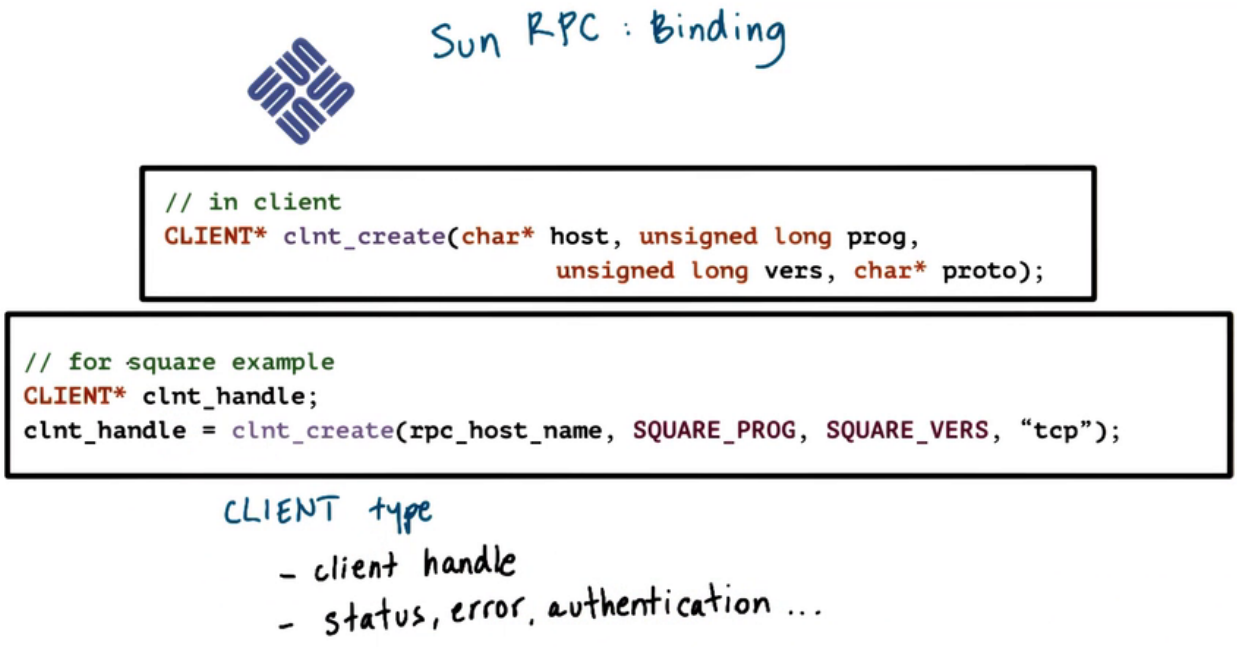
25 - SunRPC Registry



Let's look briefly about the SunRPC registry. Remember we said already that the actual code that the server needs to register with the registry is auto generated in the rpcgen process and is part of the main function. In SunRPC, 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 on linux, you have to have administrative permission 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 by both 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. 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 portmapper includes things like what is the port number that the client needs to use to talk to the 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 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 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 the service, you will also probably notice that the portmapper service is registered with TCP and UDP on the same port number 111. This means that there are two different sockets that the server is listening to. One is the TCP socket and other one is the UDP

socket and they both happen to use the exact same port number 111. This means that this service, the portmapper will be able to talk to both TCP clients as well as UDP clients.

26 - SunRPC Binding



In the last part of SunRPC that I wanted us to talk about is the binding process. The binding process is initiated by the client using the following operations. 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 hostname 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 client create operations are autogenerated in the rpcgen 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 client 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 information.

27 - XDR Data Types

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

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, byte, and int, 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 you see will be a #define value. Data types like hyper or quadruple are used to refer to a 64bit integer or a 128 bit float respectively. And XDR also supports a so called opaque type which really corresponds to a 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]

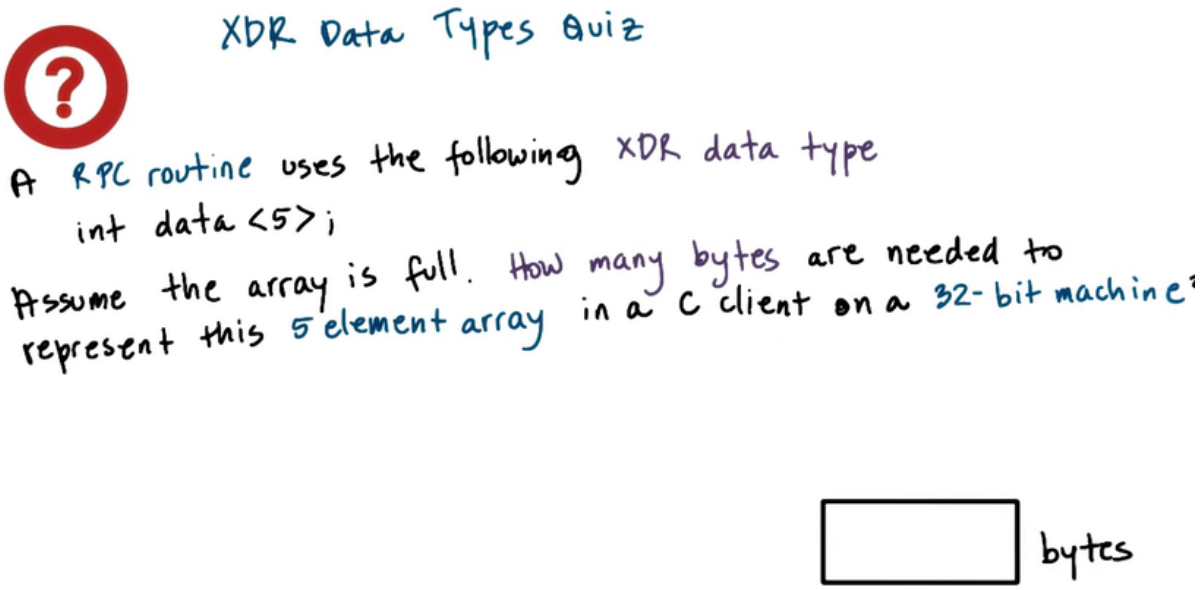
Variable-length array
- e.g., int data <80> => translates into a data structure with "len" and "val" fields

except for strings
- string line <80> => c pointer to char
- stored in memory as a normal null-terminated string
- encoded (for transmission) as a pair of length and data

RFC 4506 (Defines XDR Data Types)

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 a len, the size of the array and then send val to point to the memory location of where the data is stored. On the receiving end the server will know that it's expecting data structure that's of variable len, so it will know to read the first 4 bytes to determine what is the len, 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. In this line is really just a C pointer to character. In memory, this string will be stored just like a normal null terminated string, so it will be array of characters with the null character at the end. Operations like strcpy and strlen need that particular representation in order to be able to determine where is the end of the string. 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, or actually identical, to what we see for other variable length data structures.

28 - XDR Data Types Quiz



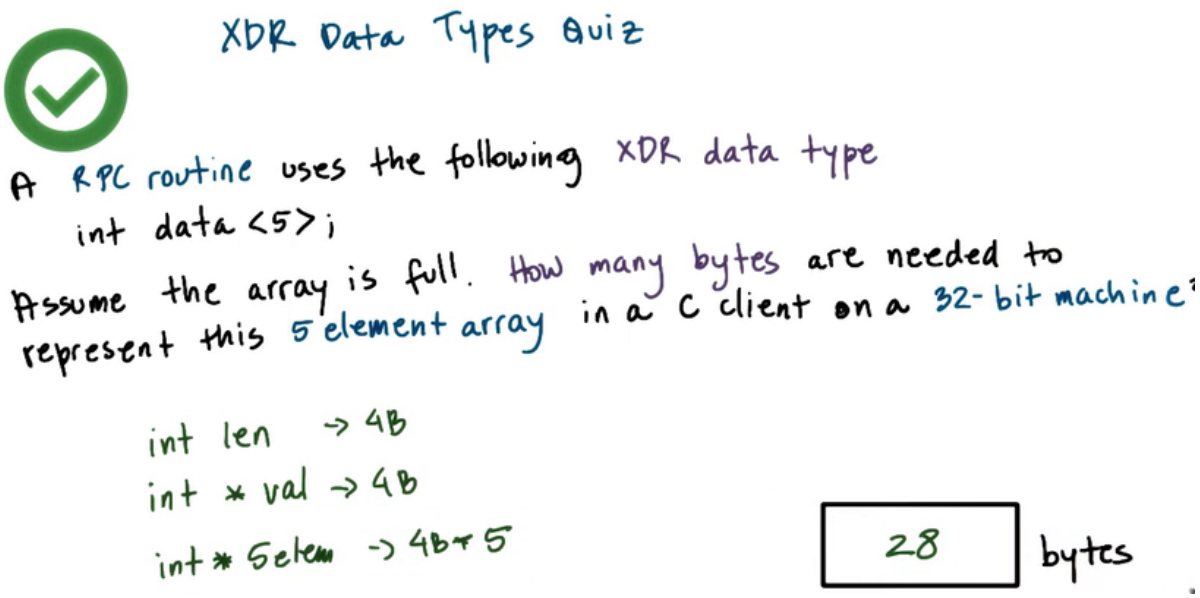
Quiz Help

Number of bytes:

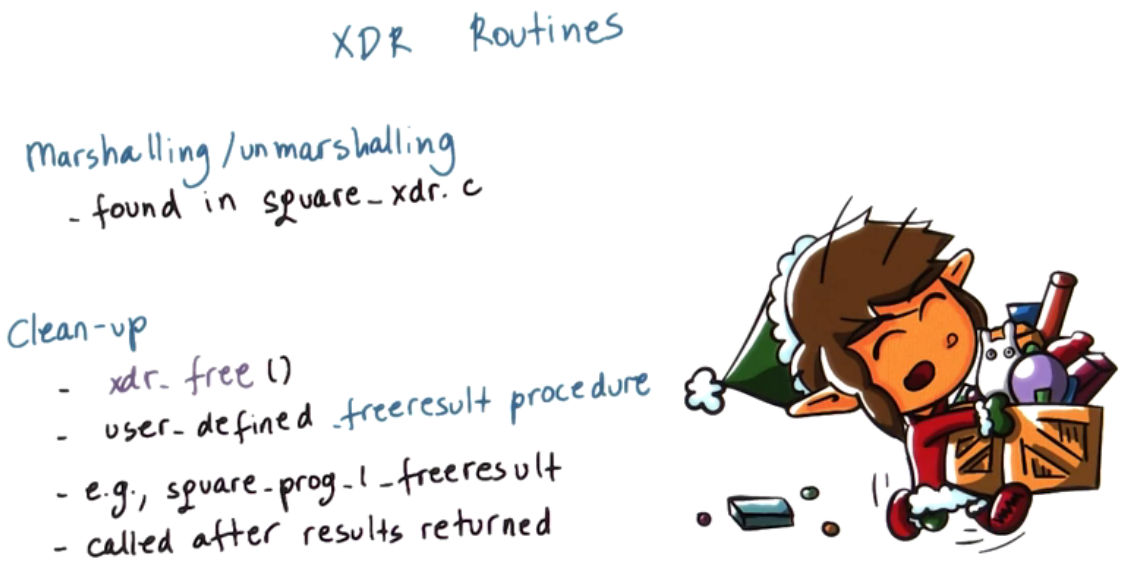
- data_structure_bytes + data_bytes

You do not have to include units in your answers.

29 - XDR Data Types Quiz

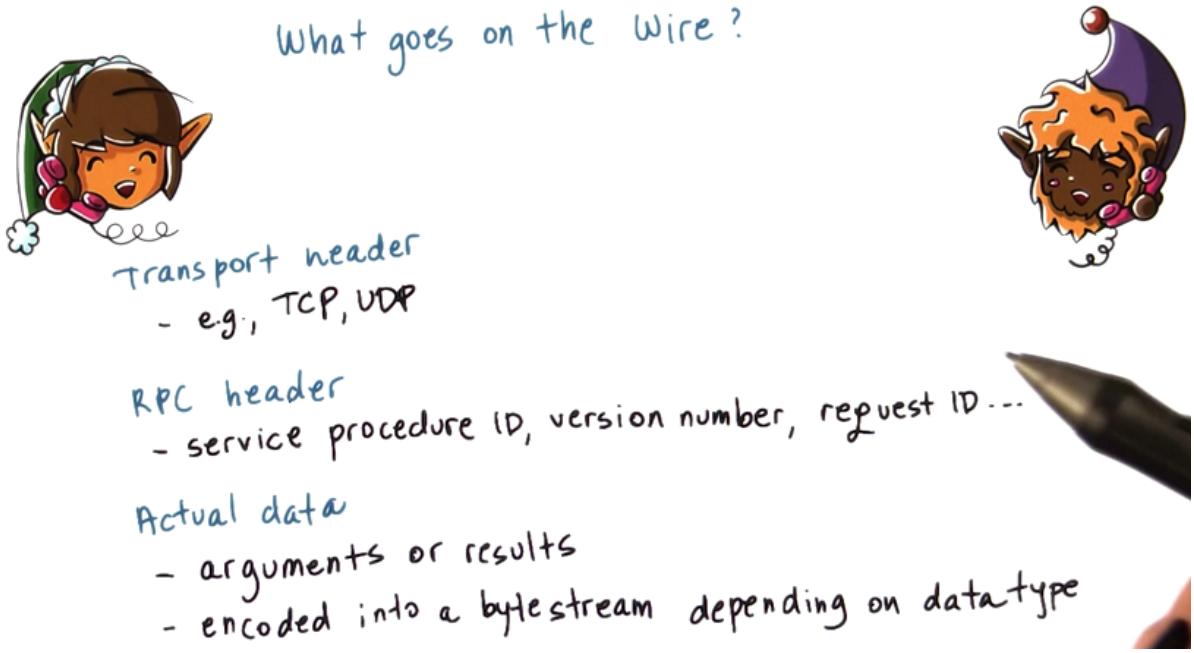


30 - XDR Routines



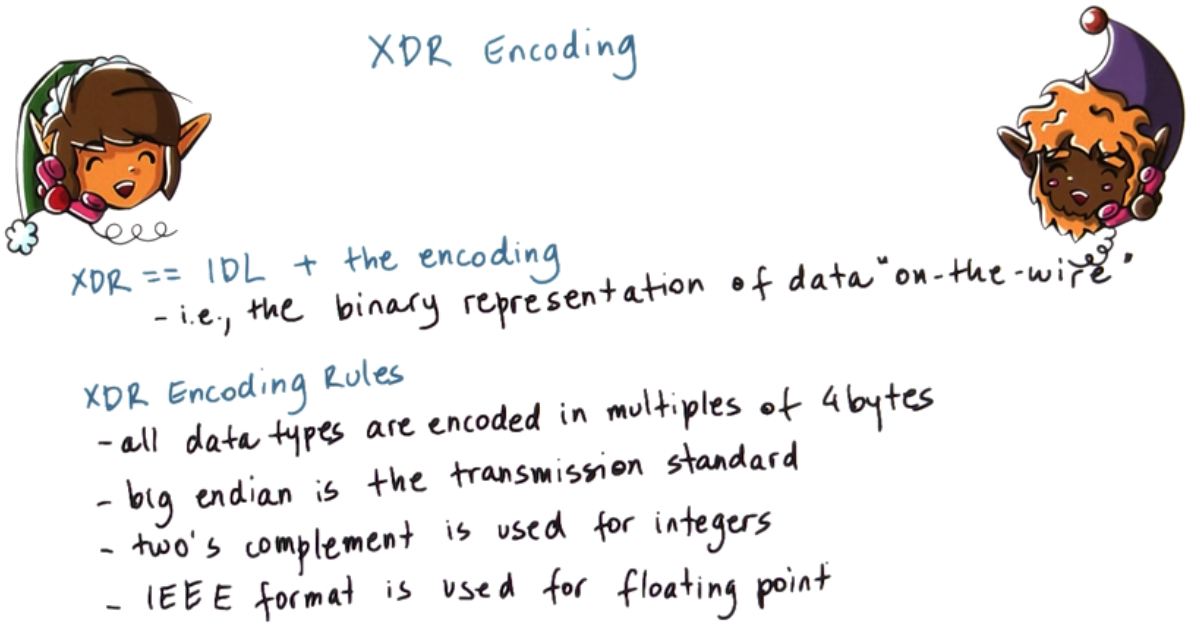
XDR provides the RPC runtime with some helpful routines. For instance, after we compile a .h XDR file, the compiler will generate a number of routines that are used for marshalling or unmarshalling the various data types in the RPC operations. In the example that we talked about, the square RPC example, these will all be found in the square_xdr.c file. In addition the compiler will generate some cleanup 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 deallocated after the runtime is done servicing RPC requests and returning the results. So the RPC runtime will automatically call this procedure after it's done computing a result.

31 - Encoding

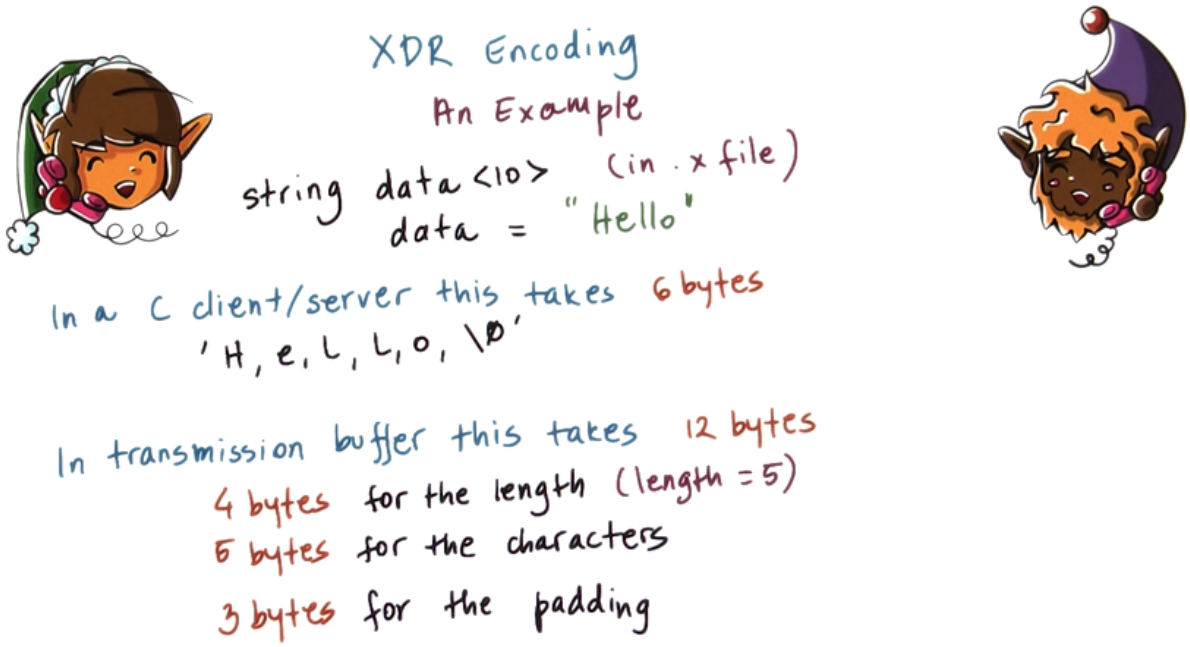


One thing that we didn't explain is what exactly ends up in the buffers that are being passed for transmission among the client and the server. First, since the server can support multiple procedures, it is important to not just pass the arguments but actually to include an RPC header that will uniquely identify the procedure that's being called, the version number, something about the request 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 and 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 and populate a data structure in the client memory. In some cases, there may be a 1 to 1 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 proceeded 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.

32 - XDR Encoding



As we hinted already with the discussion of the syntax data type, XDR specifies both the syntax, the interface definition language, so how are data types described and also it specifies the encoding so what is the binary representation of data when it's on the wire. As we hinted 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 are 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 in the network card. Big endian is used as a transmission standard. What this means is that regardless of the endian type of the client or the server machine, every type of communication will require that the data is first translated into big endian representation and then if necessary translated into the appropriate endianness of the target machine. In some cases, this may be pure overhead just because the client and server 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 complement is used to represent integers. And the IEEE format is used for floating point numbers or other rules.



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 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 6th byte for the null terminating character. However, for transmission this variable will be encoded to take twelve 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… hello. Notice we're not going to be transmitting the null terminating character. And then at the very end, we will have 3 characters used for padding because XDR specifies that everything needs to be on 4 byte boundaries.

33 - XDR Encoding Quiz

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!)

[ ] bytes

34 - XDR Encoding Quiz

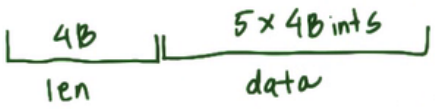
XDR Encoding Quiz

A RPC routine uses the following XDR data type:

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

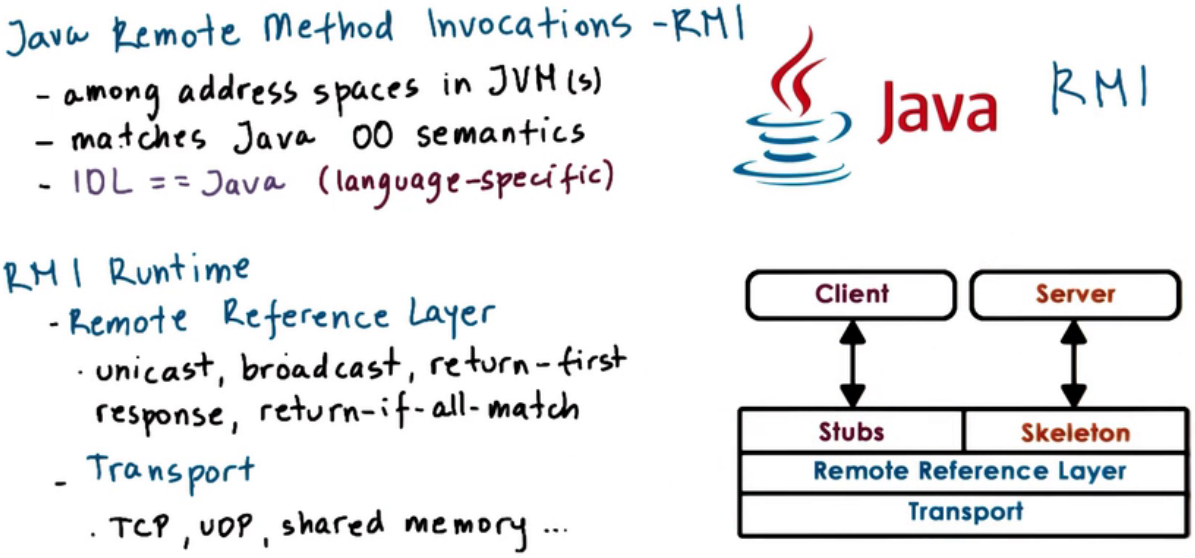
Assume the array is full. How many bytes are needed to encode this 5 element array to be sent from client to server (32-bit)?

(Do not include bytes for headers/protocol!)



[24] bytes

35 - Java RMI

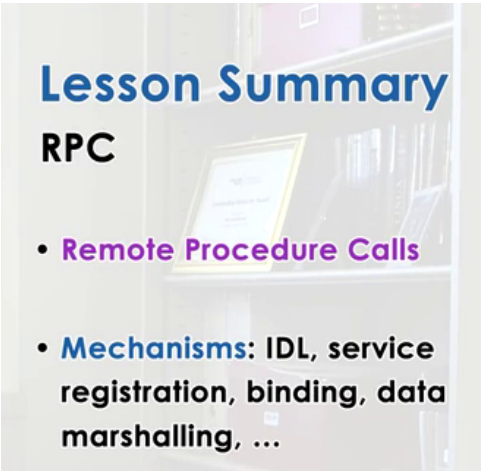


[Java RMI Tutorials]

Another popular type of RPC like system is Java RMI (Jave Remote 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 and not via procedure calls. For this reason, this interprocess communication mechanism matches the Java object oriented semantics as in the form of remote method invocations. 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's 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 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 can contact multiple servers and then the reference semantics can be such that it will return only one 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 are mentioning in this lesson Java RMI's 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 - Lesson Summary



In this lesson, we looked at remote procedure calls.. RPC's. 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 use of an interface definition language (IDL) in order for us to describe the remote service and the mechanisms such as registries, and binging, and marshalling in order to enable to 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.