So what we're going to do today is continue our discussion of modular computer software, and specifically focus on the topic we started talking about last time called soft modularity.

And once we figure out, you know, finish our story with regard to soft modularity and understand exactly what this means, we're going to start talking about a different kind of modularity called enforced modularity.

And enforced modularity is going to actually take us through three lectures, today and the next two lectures next week.

And the topic for today in terms of enforced modularity is a particular way of obtaining modularity in computer systems called client/service organization.

Some people call it client/server computing.

And that's the plan for today.

So what we saw the last time was basically a lecture describing, for the most part, how linking worked.

And the idea in linking is that when you have a number of software modules, and your goal is to take all of them and get an executable program out of them that another program could load into memory and run.

And that task was being done by a piece of system software called a linker, and out pops an executable program.

So at the end of the lecture last time, a couple of students asked me two things.

One was why we were actually talking about this.

And the second more important question was why it wasn't in the notes.

And so let me answer the first question [SOUND OFF/THEN ON] question first.

The plan in lectures is to understand concepts of computer systems with examples.

And oftentimes we find examples that aren't actually in the notes.

And we use them, A, because if we don't use different examples, there's sort of not as much incentive to show up.

But often because some of the examples that work better in lecture don't actually work out as clearly in the notes.

So we do tend to use different examples.

To answer the first question, why we actually looked at the linker, first of all it's a common piece of software that

pretty much every program that you run today goes through a process of linking.

But what it actually allowed us to do was illustrate two main concepts in naming having to do with the way in which the main mapping algorithm worked.

And in the context of the linker, the problem was to take these symbols that you find in the program, and basically resolve them, in other words, find out where they are defined, and where the instructions corresponding to those symbols, or where the data corresponding to the symbol was actually located.

And the concepts that we saw were two different concepts.

The first was name mapping using a table lookup, where within each object file there was a symbol table that mapped between the symbols found in a module to the location where the symbols were defined further.

And another example of doing name mapping, which was the process of search through a series of contexts.

And in particular, we looked at the problem of when you are in the context of LD, which GCC uses when it finds, goes through a sequence of dot O files as well as libraries on the command line, what algorithm it actually uses to resolve its symbols.

And I realize now that what I had left out on the board, there was a couple of mistakes in the algorithm, and I asked what mistakes were.

And I realized I didn't show you what the actual algorithm is correctly.

So I thought I'd spend a minute doing that so that if you have something wrong in your notes, you can fix that up, because I actually never said what the right thing was.

So if you recall, the general problem was you have GCC running on a set of dot O files, F1 dot O all the way through Fn dot O.

And what we're trying to do is describe an algorithm where when you obtain the I'th object file and you have a set of currently defined symbols, and a set of currently undefined symbols, how you can maintain three sets: the set of object files that go into the executable, the set of defined symbols that have been seen so far, and the set of undefined symbols that have been seen so far.

And you want to keep updating that.

So one way to easily see what's going on is with a picture.

So up until now, until you've finished I minus one files, let's say you've built up a set, U, of all of the undefined symbols that have been encountered so far.

And likewise, you have another set, D, of all the symbols that have been defined so far that you've encountered and have been defined so far.

So, D and U dynamically change as you go from left to right through the sequence of files.

Now, when you are on the I'th file, there is a set of symbols that have been defined that are going to be defined in the I'th file.

And that set does not intersect with a set, D, because if it did, then you have an overlapping defined symbol and that's an error.

So the interesting case is to look at this kind of an example where you have a set, Di, of the symbols that were defined in the I'th file.

And likewise, in the I'th file, there are going to be a set of undefined symbols.

And clearly, that set doesn't overlap with a set, Di, because if it's undefined, it can't be defined in the same file.

So that said, in general, looks something like this.

It has some symbols undefined in this object file that are also undefined previously.

Some symbols that are undefined in this file that have been defined previously, and some symbols that have been undefined here that you've have really never seen before.

So you've got to update to things.

So D gets updated pretty easily.

D just becomes D union Di.

And if D intersection Di is not null, then you know there's an error.

And now you need to update U, and there are many ways to do it that are more efficient than what we're going to write out.

But it's pretty easy to see that U needs to get updated by unioning the current set with Ui.

So that kind of gives you this together with that.

But now, you have to subtract out everything that's been defined already.

And since we updated D first, we could do that very easily by just subtracting out the set of all defined symbols.

And you run this through until the end.

And if you find that in the end U is not null, then you know that there's an undefined symbol so that the linking doesn't actually work.

You can produce an executable out of it with static linking, OK?

OK, so when you do combined programs in this fashion to produce an executable, there are all these different modules that have been brought together to run in an interpreter.

And you have to ask, what kind of properties that program ends up having.

What kind of modularity do you get by combining these modules together in this fashion?

It will turn out that modularity is, it's a form of modularity called soft modularity.

And to understand why, you have to understand what the interfaces between the different modules look like.

Basically when you look at a C program, and you saw an example last time of how these modules hook together, the different modules track with one another through procedures.

And procedures between modules have something that I'll call a procedure contract.

And really, to understand the property of the modularity that you get from procedural contract, we need to understand a little bit about what happens underneath the callers when a caller of a procedure invokes a callee of a procedure.

And this is actually material from 004. So if you have forgotten, I'll refresh your memory a little bit about it.

So, very abstractly, if you look at a computer, it's got a processor in it that actually executes instructions, and it has a chunk of memory.

And in that memory, there is a portion of memory that corresponds to the stack which is really where procedures, the interesting stuff with procedures gets implemented.

And you also have a bunch of registers, so, inside the processor.

And you have a special variable here called the stack pointer, which keeps track of where the current head of the stack is that you can then start pulling elements off.

The general plan, the caller and the callee interact with one another by means of the stack.

So when a caller wants to invoke the callee, what it does is it takes arguments of the procedure, and pushes them on to the stack one after another.

Then the last thing it does is to tell the callee where it should return control after the procedure function has actually been executed.

And that's the last thing that pushed on top of the stack is the [NOISE OBSCURES].

So then, after it does that, the caller then jumps to a location where the callee's module is located, and then control passes to the callee.

What the callee then does is it finds out what the return address is, pops the arguments one after the other, and then goes ahead and runs the function.

And then at the end of that, it looks at the return address and passes control back.

And before it does that, it actually puts the answer.

Let's assume that it puts the answer in a special register.

And that's part of the contract as well.

And once the caller gets the control back from the callee, it can proceed as before.

Now, the important thing about this way of interacting between caller and callee with procedure stack is this contract has to obey an invariant or discipline called the stack discipline.

And the essence of the stack discipline is that the callee should leave the stack exactly the way the caller left it when it invoked the callee, which means the caller had set up a bunch of arguments on it, and it but the return address on it.

The callee should leave things as is.

And in fact, the callee is not allowed to touch anything.

It should leave everything as is this pretty much except for the register that has the answer.

And as long as this discipline is maintained, and that invariant is maintained across procedure implications, this model of using stacks is extremely powerful.

You could implement all sorts of procedures, nested procedure, I mean, recursive procedure, mutually recursive procedures and so on because each of these implications of the procedure has a certain portion of the stack that corresponds to an activation frame.

And as long as this discipline is maintained, you can do quite complicated and interesting combinations of modules.

But the problem is that that modularity depends crucially on the stack discipline being maintained.

And any violation of [SOUND OFF/THEN ON] callee can disrupt the caller and bring it down.

And there are many ways in which this discipline could be violated.

And the easiest one is that there is some error or bug in the callee, and the callee corrupts the stack.

Or the callee corrupts the stack pointer, so it actually [SOUND OFF/THEN ON] control, but the stack pointer points somewhere else and some bad instruction runs, or you have a problem.

Now, another problem is that the callee crashes.

For instance, there is a divide by zero, or there is some other violation that causes the callee to crash.

And then the caller comes crashing down as well.

And there's a bunch of other reasons, but all of them have to do with the stack discipline being violated, or with the callee crashing and control never returning to the caller, which means that the caller and the callee share fate.

If something bad happens to the person who has called, sorry, the callee, then the caller struggles as well and isn't able to continue.

So colloquially this is referred to as fate sharing.

And the resulting modularity is soft because any fault or error in the callee affects the caller.

The caller, there isn't any kind of a firewall where errors in the callee are insulated from errors in the caller.

There's no shielding between the caller and callee.

And, where is this thing going?

Like, all right, so there's no insulation between caller and callee.

And the resulting modularity is not as hard as we would like it to be.

So this is the problem we'd like to solve today.

And the first solution we're going to discuss is a way of organizing callers and callees into an organization called client service organization.

And the main idea is going to actually involve a different abstraction by which callers and callees communicate with one another from the abstractions we've seen already.

We've already seen the memory abstraction where you could write values to a name, and another person could read from it.

And we've already seen the interpreter abstraction.

It could turn out we are going to use a different abstraction A location path abstraction to implement the client service organization.

And the idea is the following.

The program is going to be decomposed into clients and services.

And you might have many clients and many services, and you could have a client which is a client of one service and service in turn is a client of yet another service and complicated things like that.

But any pair-wise interaction is going to be between a client and a service.

And think of mapping that example onto here.

Think of the callee, for example, being the service and the caller being the client.

The caller wants some work done, so it's the client.

And it invokes the service, the callee, to get that work done.

And the plan is going to be that the client and the service are going to run on different computers, physically

different computers, and we are going to connect the computers up with wire.

And the idea is the moment you do that, the crash of a callee doesn't actually bring the caller coming down because it's running on a completely different processor.

The stack is not shared.

The memory is not shared.

The stack point is not shared.

There's really no problem with regard to the callee crashing, bringing the caller down.

Of course, we now need a way by which the client can communicate its arguments to the service, and the service can communicate its arguments back to the client.

Previously, we did the first one.

The arguments were communicated by putting them using memory on the stack, and the answers were coming back to us from register.

And we don't have that shared state anymore.

So, we're going to have to implement that using messages.

And we're going to take these messages, use the communication path abstraction, and send messages from the client to the service.

So imagine that time flows downwards starting from when the client invokes the service.

A message is sent from the client as a message to the service saying here's all the arguments, and here's the procedure that I want you to run.

And it takes that information up, somehow packages it up into a message, and calls send.

OK, and the assumption is that the client somehow already knows something about the name of the service or the location of the service.

That's outside of the scope of the current discussion.

Let's pretend somebody tells you that here's where the service is running that can run this function for you.

So it takes the arguments in the name of the procedure, packages it up the message, and send it across.

When the service gets the message.

It validates the message to make sure that it's the right sizes, and it's not too big, and so on.

And then, it takes this message and then does some processing on that message.

The technical term for it is going to be called un-marshalling because when we took these arguments and put into a message, that process is called marshalling.

The service is going to un-marshall this message and obtain the actual arguments and the name of the procedure, and then it's going to run it.

I don't know if it's one L or two L's.

And then it's going to run the procedure that's named here.

And it's going to find the answer.

And then when it gets the answer, it does the same thing back.

It puts it back into the message and sends it across to the client.

And now the client is waiting for this message because it sends off the message to the service to run this thing.

It gets this answer back.

It does the same thing.

It takes the message in that it recovers the answer from it, and then it runs continuously.

So this is the basic idea in client/server organization.

And the way in which we solve this problem is that these two problems are solved because the callee can't really corrupt the stack or the stack point or anything else that in this model would have affected the caller.

And because we have put them both on physically different computers and hooked them up with, let's say, a wire, the service crashing does not actually bring the client, if the service decides that it crashes, then the client actually doesn't come crashing down.

Of course, the client has to somehow have a plan by which it knows that the service is still, for example, the client

has to know whether the service has crashed or whether the service is just taking a long time to run something.

That's something we have to address, and we will in a bit.

But as long as the client is able to do that, a service going away or crashing is not really going to bring the caller or the client down with it.

So, some properties of this organization are, first of all, that it's modular.

It has essentially the same modularity as we had with procedures because if you had enough computers, you could move all of the different procedures on different computers, all of the caller/callee relationships, and you can preserve, essentially, the same modularity that you had before.

Moreover, this modularity has a different adjective in front of it different from soft.

This modularity is enforced.

What that means is not only is it a modular organization, it's one where errors of, for example, things where one module fails or crashes does not bring the other one come crashing down.

So the modularity is more enforced than when they were both running on the same computer using the procedure interface.

And the third property of this kind of modularity is something already mentioned.

It lies on the message abstraction, actually the communication path abstraction.

The client and service communicate with each other through messages.

And these aren't actually messages.

You can't just sort of take a random string of bytes and send it to the service.

It's actually messages that correspond to a particular format.

And these are really known in advance.

So, that way the service isn't surprised and the client isn't surprised when messages that don't conform to that pattern arrive.

I mean, you know exactly what kind of message is going to arrive.

And anything that doesn't conform to what is agreed upon in advance is rejected.

But you basically make it so that you explicitly declare the nature of these messages much like you did the nature of procedural interface.

But now, because you've physically separated it, you have a more enforced modularity than in the previous model.

Of course, nothing comes for free.

It's not like you got this enforced modularity, and you have all of these nice properties that you did before.

Here, you had a nice property that the callee could, has the property that either when it runs and it returns to you, you know exactly what happened.

And if it doesn't return, you know that you usually come crashing down.

If the callee doesn't return, it means that control never comes back to the caller.

So it's not like the caller is left wondering what really happened because the caller doesn't get control again.

Here, you have a problem.

If the callee, the service doesn't return back to the client, the client actually doesn't know what's going on.

When you have two machines, two computers connected over a wire or over a network, it's extremely hard to distinguish between a service running really, really slow, and a service that's gone away.

And we'll revisit this a few different times in the course.

But it's going to be impossible for us to tell for sure, although we're going to try very hard.

It's going to be really hard for us to tell for sure whether something exactly happened.

And if the service didn't return, we don't really know for sure without much more machinery whether it was just that the service is still running, or whether it's crashed, and we are just waiting.

And so, this organization requires a timer at the client, and there are many names given to this timer.

I mean, people call them keep-alives, or people call them with various names.

I'm going to call it a watchdog timer, where the client has to keep track using some kind of a timer of the service.

And if the service doesn't return within a certain period of time, the client has to time out and say, well, the service didn't return, and I'm not quite sure what happened.

It might be that the procedure I wanted to execute had ran, but I didn't get the answer.

Or it might be that the procedure didn't get executed at all, and I have to deal with it.

And you might be able to, by retrying the procedure, or you might contact another service which provides the same functionality, but the client has to deal with all of that.

So fundamental to this client's organization is the notion of a time out.

And we didn't have that here because here, if the callee decides that it's just going to continue on and not return, the caller never gets a message back.

So, it doesn't have this decision to make.

There is no such notion of a watchdog that we have to worry about in this other organization.

So another nice property of this client service organization is that so far we've presented it in the context of the client and the service being modularized from each other, and we've enforced modularity to the client and the service.

But in fact, there is another nice property to it, which is that a client service organization allows us to design modules and design systems where clients get modularized from each other.

We can achieve soft modularity by protecting clients from each other.

The idea here is that if you have many clients all of which want to use a given service, for example, there's a service that's, let's say, implemented by a bank and what it does is it's the service that deals with managing your accounts, and you can move money between accounts, and it will tell you your account balance and so on.

You can implement that as one service, and many, many clients can share the same service.

Now, all of the clients trust the service because, I mean, if you are a customer of a bank, and you are using your browser to look for your account balance, that means you sort of trust the bank.

And all the clients trust the service.

But the clients sure don't trust each other.

And the nice thing about this organization is that you can use the service in the form of an intermediary that allows the clients to be separated from each other, each of which can use the service.

But the clients don't have to trust each other, and clients don't really have to know about each other's information.

And this idea of using a service to modularize clients from each other is called a trusted intermediary.

There is many examples of trusted intermediaries that we'll see in this course.

In fact, tomorrow's recitation on the X Windows system has a system where your computer screen is going to be managed by a service called by the X Windows system.

And there are many clients that are going to use that service.

And the clients don't actually trust each other.

They want to get modularized away from each other.

And the X Windows system as a trusted intermediary deals with that.

It managed this resource -- your display, and it arranges for the clients to be designed each independent from the other.

And in general, we are going to see in the next few lectures, many examples of the operating system being a trusted intermediary, arranging for many different clients to use some resource on your computer like the processor or the memory or the disk.

And our architecture for the operating is going to end up being in the form of these trusted intermediaries.

So, so far we've seen what client service organization means.

It means you have a client and the service, and they communicate with messages using the communication path abstraction of send and receive.

And we've seen some properties of client service organization.

But I haven't actually told you how to implement any of this stuff.

And so that's what we're going to do the rest of today.

And in fact, we are going to continue with different ways of implementing various forms of client service many

times in the course.

So there are many ways to implement client service organization.

And all of them have to do with, all of them involved different ways in which messages are sent between client and service.

A common way, and a pretty standard way, of implementing it is something called a remote procedure call.

There are many examples of remote procedure called systems.

I mean, one of the most common ones is something called the Sun Remote Procedure Call system, or Sun RPC.

That's one example.

There are many other examples as well.

A more modern example which some of you may have heard of is a relatively new system, about five years old, called XML RPC.

So if you've heard of buzzwords like Web services in business-to-business interactions, or business-to-business applications, these things use something called XML RPC.

And, there is a lot of different three letter acronyms and four letter acronyms.

This has led to something called SOAP, which stands for the Simple Object Access Protocol.

So there are many different ways of implementing RPC systems.

And until last year or a couple of years ago, we used to talk about Sun RPC as an example in this class.

But I decided that's so 20th century.

So we're going to talk about XML RPC today.

It has a property that's much more inefficient, but that's sort of keeping with the fact that computers have become faster.

We don't have to worry in many cases about efficiency.

So we are going to talk a little bit about how XML RPC works.

So let me first show you what a client written what this kind of RPC looks like.

It's going to show you a code snippet.

All right, I had to work hard to make sure it would fit on this.

OK, all right, the way this thing works is actually very, very simple.

This uses something called XML RPC Library for Java that was written by the Apache people.

And once you incorporate that library, your program becomes completely easy.

So let me just walk you through this.

The high level idea here is that it's transferring money from one account to the other through a service that's run by the bank.

So, the first line of this thing here creates an XML RPC object, an XML RPC client object.

And what you give it is actually the name of the service.

So somebody has to tell you the name of the service.

And I don't want to get into the details of everything here, but the basic idea is your backname.com colon 8080 is the name, the DNS name at which the service runs and the port.

The thing about XML RPC is that it runs over HTTP which is what you use to transfer objects on the Web.

And underneath, we talk about how it's implemented; underneath this is implemented using a standard method in HTTP called a POST which allows, normally HTTP has a GET where you retrieve But it also has a POST that many of you are familiar with where the client can push some stuff to the server.

And it just uses POST method.

What's going on is very easy?

You create a vector of parameters, and you fill that vector in with, in this case, your account number.

And let's say here the idea is this sort of thing is very popular in big companies like Ford or Cisco, which have thousands of suppliers.

And, they never actually maintain a lot of inventory of their own.

They're always trying to figure out the latest cost of any of their supplies.

And they are using this Web service like interface.

In fact, this is also called a Web service interface to communicate with their suppliers to always have the latest info whether their suppliers have any given item in stock, and how much it costs and so on.

So, let's pretend you have done that in your company, and you are trying to pay off, you are taking your suppliers' account number and pay some money to him, OK?

So in this case, whatever, dollars is that argument.

So you create parameters.

And all you do at the end is you use these XML RPC client objects, and invoke a method it presents to you called execute.

And you give it two arguments.

The second argument is the parameters.

And the first argument is the name of the procedure that you wish to run on the service.

OK, that's in this case called MoneyTransfer.

So, corresponding to this, somewhat longer is a piece of code running on the service which implements the server side of it, which basically obtains [SOUND OFF/THEN ON] calls an object that will un-marshall the arguments, and then it will execute money transfer, both of which [SOUND There is very little that actually has to run on the service.

It's just a little bit longer than this.

Now, this line here is important.

That's the line on which you get your result.

And this gives the result as a string.

If I tell you that I finished transferring X dollars from this account to that account, or if I tell you I couldn't transfer, and there was some kind of an error, or you might actually not get any answer, in which case the underlying

library that implements this procedure call would throw an exception your code has to deal with.

And how you deal with it is a little tricky because you don't, and you'll see this in a moment, you don't quite know what happened when you didn't get an answer back from the You don't know if this actually got your transfer request and crashed after that.

And, it got the request.

It actually implemented the transfer and then crashed.

It just couldn't send your response back.

So, you're not quite sure whether you need to retry the request or not.

You will actually see how to deal with this in a few minutes.

Now, there's one thing that's really important about this line of code.

This xmlrpc.execute("MoneyTransfer") block, that line of code actually is not something that runs on the service.

OK, it's a local procedure.

xmlprc.execute() is a local procedure.

OK, and that procedure is an example of something called a stub because what it is, is a stub that to this caller fakes out the fact that there's a service somewhere else.

I mean, it prevents the caller from having to deal with 15 arguments and putting it in the message and sending it to the other side.

The caller just calls it over the procedure, giving it suitable arguments that pinch for this function now to do the work of sending a message across the network.

But this is a local call.

So, what happens underneath?

Underneath in the library, once you call xmlrpc.execute() in this example, somebody does work.

That library does the work of taking the different arguments that have been presented to it and converting them into a message, marshalling all of the stuff into a message, and then shipping that message off to the server.

That has to be going to some kind of a format on the Fly, right?

Ultimately on this wire connecting the client to the service, there's format.

And I already mentioned that this runs on top of HTTP.

And so we can actually look at what that looks like.

OK, so POST here is the method that you use in HTTP.

The /RPC2 is actually the same thing that was used in, if you remember in the previous screenshot, when we did the new call to get a new XML RPC client, we gave it a server name and a port number.

But we also gave it something called RPC2. Now, that's just like a file on the other side.

It says there are many different RPC programs running on your service.

And I want RPC2 to run.

I mean, I could have named it anything I wanted.

And that's a lot like giving a file name on a URL.

And then, you go on.

You give it the host name.

This is a lot like an HTTP header.

The interesting new stuff here is in the XML arguments.

But those were not familiar with XML.

It's just a method, a way of sending things that have attributes and values associated with them.

So, every method has the format that it's attributes and values, and they can be nested within each other.

The only interesting thing that's here, this particular XML RPC system supports a few different data formats.

You can do integers, and characters, and strings, and doubles, and floats, and a few different things like that.

And I thought it just means that the 32 bit integer, and you can take numbers that sum up your account number, your supplier's account number, and the amount of money that you want to transfer.

The good thing is that all this stuff has gone underneath the covers, and you don't actually have to deal with it if you are writing the client or you are writing the service.

All this work happens underneath.

So, it greatly simplifies your ability to take, implement client service programs with clients and services being separated from one another.

OK.

So, so far, we've made it look a lot except for this little timer that you have to maintain. We've made it look a lot like a remote procedure call, it's like a procedure call.

In fact, the code here, the only difference is you replace what was previously what would have been one big transfer with arguments we replace with a stub call, taking the name of the procedure we want to run on the service [NOISE OBSCURES].

So, it looks a lot like a procedure call.

That actually is a pretty deceptive thing.

And in fact, a hint at that is you can get at the bottom of this thing up there, there is a light that says you have to deal with XML RPC exception.

And that's the kind of exception you get when the underlying RPC library decides that it hasn't heard an answer from the service in a while, it throws an exception.

And your code has to deal with it.

You are never going to get that kind of exception from a regular procedure call.

I didn't hear back from the caller or from the callee to do something.

See, that's a new exception mode that you didn't previously have to deal with.

I mean, you have to deal with other kinds of exceptions but not this one.

So, an RPC is not the same as a procedure call.

And in fact, it's a little unfortunate that, for several reasons, we are stuck a little with a name procedure call.

In fact, increasingly more and more RPC systems don't look like procedure calls at all in terms of the semantics.

But we were so stuck with the name that people continue to have various kinds of procedure calls, and they used the same name for it.

And the first main difference arises from the fact that there is no fate sharing between client and service.

If the service crashes, the client doesn't crash.

Already you have a big difference between a regular procedure call.

And previously I presented this as an advantage because if you have fate sharing, then this caller is always at the mercy of the callee.

But because you have no fate sharing, you have other problems to deal with.

And in particular, all of these stem from the fact that it's extremely hard to distinguish between a failure versus extremely slow.

And it will turn out that we revisit this over and over again.

We'll talk about networks and reliable concentration over networks and deal with it, and then we're going to talk about fall tolerance, and we're going to talk about an idea for [NOISE OBSCURES], and then we're going to talk about something called transactions.

And they're all going to deal with this problem that it's going to be very hard for us to tell, when you ask somebody to do a piece of work, whether they did it fully or did nothing.

OK, and this is going to be a repeated theme, a theme that is going to repeat in the course.

But to complicate why this is hard, let me say three possible things that could happen when you have a client talk to service and what kind of semantics you want from a client They all stem from the fact that this is extremely hard to determine.

The first semantics that you might want is the idea semantic.

The client talks to the service, and either the service answers with a response or it doesn't.

OK, and that's something called exactly-once semantics.

So in this example here, underneath the library, it may time out.

And it may wish to retransmit.

Or it may wish to throw an exception at the caller.

The client may want to send this again.

But in an ideal case, you want exactly once this amount of money to be moved from bank account one to bank account two, right?

You certainly don't want your amount of money to be moved twice to your supplier.

The term of this is ideal, and it's going to be pretty difficult, and extremely hard, to achieve exactly-once semantics.

And, we're going to talk about different methods in the course.

This is not an easy problem at all.

It stems from the fact that it is very hard to know.

So, you might give up a little bit and say, OK, I can't really get exactly-once semantics very easily.

But let me try for at least once.

So what this means is the client will keep retrying or the the library will keep retrying.

And you can decompose it in either way until it is sure that this call succeeded at least once.

Now, it might have succeeded more than once because the service is very slow.

And then it times out.

You try it again, the service says, oh, OK, I see it; I must not do it again.

But, it's at least once, OK?

Now, at least once is not nice semantics that you're using at least one semantics in your code here, right, because your supplier might end up with $7 million instead of $1 million.

But at least once is OK if the service has some kind of semantic called idem-potent semantics.

What that means is that when you do an operation more than once, the answers are the same as at-least-once. A simple example of this was the transfer of money, but you checking your bank account It doesn't really matter that you do it a hundred times.

As long as one of them succeeds and you get your bank balance, you are fine.

I mean, doing it seven times does not really change anything; it does not move extra money anywhere else.

So that's an example of an idem-potent action, which works out well with at-least-once semantics.

And at-least-once semantics is much easier to obtain than exactly-once semantics.

And the third kind of semantics is something converse: most-once semantics.

That means zero or more times.

And here, the challenge is really figuring out if it really worked at least once around. I mean, I pushed once or not.

So, if you don't get a response back, then you time out.

And it might be that it didn't succeed at all.

And you say that's fine.

I'll deal with it separately.

And if you get a response back, then you know that it worked.

It turns out, even that is going to be a little bit tricky to implement.

But these kinds of semantics you could expect from your RPC system.

Now, actually the first R in most RPC systems like XML RPC don't really deal with any of this in a particular, I mean, they don't really provide any well-defined semantics.

It's usually for the client sitting on top, and the service is still, they don't know what kind of semantics they need, and implement that at a higher layer at least with most of these standard protocols.

But many of them are well equipped to deal with at-least-once semantics.

OK, and through the course, there are different ways in which we'll accomplish these different goals, these different semantics that we want from our different RPC systems.

Now, there's another difference between regular procedure calls and remote procedure calls, or more generally from client service.

Remember in the second lecture, I told you that you can always get more bandwidth, and you can always get more processing past Moore's law, but one thing you can't actually change is the legacy between two computers connected by a wire connected by a network.

The speed of light doesn't change.

What that means is that with time, the number of constructions that you can run, when you have two computers separated by a wire, there is a certain delay between that no matter what you do.

So there is a certain delay that you do a procedure call that's a remote procedure call.

It takes a certain delay to send a message and to get a response back.

Even as computers get faster and faster and faster, that isn't changing at all.

But the problem is that the number of instructions you can run but within that duration is increasing with time because it's a fixed amount of time, and the number of instructions you could run locally on the computer increases with time.

So that has led people to be more aggressive about what they do in a remote procedure call.

What people said is, wait, it doesn't make sense for a client to issue a procedure call, relocate to a service, and then just sit and wait like in a regular procedure call, for the answer.

I mean, I'm just sitting there twiddling my thumbs, and this thing is that way, and I'm waiting for it to respond.

I could be doing work.

So, that's led people to changing the synchronous model of RPC, of a procedure call interface to do something called asynchronous procedure call interface.

And all the things like XML RPC and its follow-on, and the difference between SOAP and XML RPC is that I can understand this, and this builds on XML RPC.

It seems like very few people understand SOAP.

There is a lot of people who have given up on trying to understand the specification.

But XML RPC turns out to be a really simple seven or ten page document that's very easy to understand.

A lot of people use it.

But anyway, all these systems support asynchronous RPC.

The idea here is that the client sends a procedure call indicating a remote request to the service.

And then it goes about doing its work.

When the service responds with an answer, it responds with not just the answer, but also something that tells the client which service request, which procedure caller request it's responding to.

And when the response comes back, the client can handle it.

Now, the way in which you have to implement this, or the way you have to design this is that associated with every procedure call request, you also have to associate what's called a handler because you're going to issue this request to the service and then go about doing your work.

When the answer comes back, the RPC library underneath, the communication library has to know: who gets this answer?

Because you might have to a sheet of many such service requests.

So who gets this answer?

So you associate with every request a callback, a handler, which then is called back.

And that handler runs, dealing with the answer that comes back.

And so, that actually allows you to be a little more decoupled than we were with the remote procedure call; even more decoupled.

The client services are even more decoupled because the client is no longer waiting for the service to return an answer to.

So that's the first way which people have extended it.

The second way in which people have extended is using this intermediary idea that we talked about before where the services cross the intermediary.

To use this intermediary idea to actually make design remote message based communication systems where the client and service don't actually have to be up and running at the same time.

So actually, the client could send a request out to the service.

But the service is actually not up and running.

So, the idea is there is an intermediary that acts as a broker on behalf of the service, and buffers the message.

And then, when the service comes out, the service knows to pull the message from this intermediary.

That's buffered messages, correct?

And then it passes the message, and then pushes the answer back to the intermediary, which then stores the message to some other intermediary perhaps.

And then the client knows to get the answer from that intermediary.

So now, we can actually have clients and services that interact with each other without actually having to be up and running at the same time.

There are many examples of this, and the notes talk about various examples of intermediary communication.

So the general summary of what we've talked about so far is that we looked at different ways of attaining modularity, talked about soft modularity last time, and today about a particular way of enforcing modularity using client service organization.

And the next few lectures, we are going to solve a big weakness of the current system which is that you need many different computers.

So we are going to take these ideas, and implement them all on one computer.

See you next week Tuesday.