0:00:00 So what we're going to do today is continue our discussion of 0:00:05.714 modular computer software, and specifically focus on the 0:00:10.952 topic we started talking about last time called soft 0:00:15.809 modularity. And once we figure out, 0:00:19.047 you know, finish our story with regard to soft modularity and 0:00:24.761 understand exactly what this means, we're going to start 0:00:30 talking about a different kind of modularity called enforced 0:00:35.619 modularity. And enforced modularity is 0:00:40.182 going to actually take us through three lectures, 0:00:44.211 today in the next two lectures next week. 0:00:47.569 And the topic for today in terms of enforced modularity is 0:00:52.354 a particular way of obtaining modularity in computer systems 0:00:57.306 called client service organization. 0:01:01 Some people call it client/server computing. 0:01:04.444 And that's the plan for today. So what we saw the last time 0:01:09.09 was basically a lecture describing, for the most part, 0:01:13.335 how linking worked. And the idea in linking is that 0:01:17.341 when you have a number of software modules, 0:01:20.705 and your goal is to take all of them and get an executable 0:01:25.271 program out of them that another program could load into memory 0:01:30.237 and run. And that task was being done by 0:01:33.966 a piece of system software called linker, 0:01:36.503 and out pops an executable program. 0:01:38.66 So at the end of the lecture last time, a couple of students 0:01:42.402 asked me two things. One was why we were actually 0:01:45.447 talking about this. And the second more important 0:01:48.492 question was why it wasn't in the notes. 0:01:50.966 And so let me answer the first question last and the second 0:01:54.645 question first. The plan in lectures is to 0:01:57.246 understand concepts of computer systems with examples. 0:02:00.608 And oftentimes we find examples that aren't actually in the 0:02:04.287 notes. And we use them, 0:02:06.888 A, because if we don't use different examples, 0:02:09.388 there's sort of not as much incentive to show up. 0:02:12.055 But often because some of the examples that work better in 0:02:15.222 lecture don't actually work out as clearly in the notes. 0:02:18.277 So we do tend to use different examples. 0:02:20.444 To answer the first question, why we actually looked at the 0:02:23.666 linker, first of all it's a common piece of software that 0:02:26.777 pretty much every program that you run today goes through a 0:02:30 process of linking. But what it actually allowed us 0:02:34.066 to do was illustrate two main concepts in naming having to do 0:02:38.066 with the way in which the main mapping algorithm worked. 0:02:42 0:02:46 And in the context of the linker, the problem was to take 0:02:48.974 these symbols that you [SOUND OFF/THEN ON] in the program, 0:02:52.003 and basically resolve them, in other words, 0:02:54.234 find out where they are defined, and where the 0:02:56.625 instructions corresponding to those symbols, 0:02:58.909 or where the data corresponding to the symbol was actually 0:03:01.937 located. And the concepts that we saw 0:03:05.245 were two different concepts. The first was the name mapping 0:03:09.898 using a table lookup, where within each object file 0:03:13.909 there was a symbol table that maps between the symbols found 0:03:18.641 in a module to the location where the symbols were defined 0:03:23.213 further. And another example of doing 0:03:26.101 name mapping, which was the process of search 0:03:29.631 through a series of contexts. And in particular, 0:03:34.259 we looked at the problem of when you are in the context of 0:03:38.246 LD, which GCC uses when it finds, goes through a sequence 0:03:42.163 of dot O files as well as libraries on the command line, 0:03:46.01 what algorithm it actually uses to resolve its symbols. 0:03:49.787 And I realize now that what I had left out on the board, 0:03:53.634 there was a couple of mistakes in the algorithm, 0:03:56.922 and I asked what mistakes were. And I realized I didn't actually fix and show 0:04:01.58 you what the actual algorithm is correctly. 0:04:03.95 So I thought I'd spend a minute doing that so that you know if you have 0:04:07.392 something wrong in your notes, you can fix that up, 0:04:10.213 because I actually never said what the right thing was. 0:04:13.261 So if you recall, the general problem was you 0:04:15.744 have GCC running on a set of dot O files, F1 dot O all the way 0:04:19.186 through FN dot O. And what we're trying to do is 0:04:21.838 describe an algorithm where when you obtain the I'th object 0:04:25.28 file and you have a set of currently defined symbols, 0:04:28.215 and a set of currently undefined symbols, 0:04:30.472 how you can maintain three sets: the set of object files 0:04:33.576 that go into the executable, the set of defined symbols that 0:04:36.905 have been seen so far, and the set of undefined 0:04:39.501 symbols that have been seen so far. 0:04:43 And you want to keep updating that. 0:04:45.523 So one way to easily see what's going on is with a picture. 0:04:49.828 So up until now, until you've finished I minus 0:04:53.167 one files, let's say you've built up a set, 0:04:56.285 U, of all of the undefined symbols that have been 0:04:59.847 encountered so far. And likewise, 0:05:02.709 you have another set, D, of all the symbols that have 0:05:05.545 been defined so far that you've encountered and have been 0:05:08.6 defined so far. So, D and U dynamically change 0:05:11.054 as you go from left to right through the sequence of files. 0:05:14.218 Now, when you are on the I'th file, there is a set of symbols 0:05:17.49 that have been defined that are going to be defined in the I'th 0:05:20.872 file. And that set does not intersect 0:05:22.836 with a set, D, because if it did, 0:05:24.581 then you have an overlapping defined symbol and that's an 0:05:27.636 error. So the interesting case is to 0:05:30.876 look at this kind of an example where you have a set, 0:05:34.242 DI, of the symbols that were defined in the I'th file. 0:05:37.672 And likewise, in the I'th file, 0:05:39.613 there are going to be a set of undefined symbols. 0:05:42.72 And clearly, that set doesn't overlap with a 0:05:45.503 set, DI, because if it's undefined, it can't be defined 0:05:48.997 in the same file. So that said, 0:05:50.939 in general, looks something like this. 0:05:53.334 It has some symbols undefined in this object file that are 0:05:57.087 also undefined previously. Some symbols that are undefined 0:06:01.835 in this file that have been defined previously, 0:06:04.559 and some symbols that have been undefined here that you have 0:06:08.052 really never seen before. So you've got to update two 0:06:11.072 things. So D gets updated pretty 0:06:12.907 easily. D just becomes D union DI. 0:06:15.039 And if D intersection DI is not now, then you know there's an 0:06:18.592 error. And now you need to update U, 0:06:20.664 and there are many ways to do it that are more efficient than 0:06:24.217 what we're going to write out. But it's pretty easy to see 0:06:28.646 that U needs to get updated by unioning the current set with 0:06:32.243 UI. So that kind of gives you this 0:06:34.256 together with that. But now, you have to subtract 0:06:37.182 out everything that's been defined already. 0:06:39.743 And since we updated D first, we could do that very easily by 0:06:43.402 just subtracting out the set of all defined symbols. 0:06:46.512 And you run this through until the end. 0:06:48.829 And if you find that in the end U is not null, 0:06:51.573 then you know that there's an undefined symbol so that the 0:06:55.048 linking doesn't actually work. You can produce an executable 0:06:58.646 out of it with static linking, OK? 0:07:02 OK, so when you do combined programs in this fashion to 0:07:05.54 produce an executable, there are all these different 0:07:08.885 modules that have been brought together to run in an 0:07:12.229 interpreter. And you have to ask, 0:07:14.327 what kind of properties that program ends up having. 0:07:17.672 What kind of modularity do you get by combining these modules 0:07:21.606 together in this fashion? It will turn out that 0:07:24.885 modularity is, it's a form of modularity 0:07:27.442 called soft modularity. And to understand why, 0:07:31.42 you have to understand what the interfaces between the different 0:07:35.488 modules look like. Basically when you look at a C 0:07:38.587 program, and you saw an example last time of how these modules 0:07:42.526 hook together, the different modules track 0:07:45.173 with one another through procedures. 0:07:47.433 And procedures between modules have something that I'll call a 0:07:51.372 procedure contract. And really, to understand the 0:07:54.471 property of the modularity that you get from procedural 0:07:57.958 contract, we need to understand a little bit about what happens 0:08:01.961 underneath the covers when a caller of a procedure invokes a 0:08:05.836 callee (sic) of a procedure. And this is actually material 0:08:11.12 from 004. So if you have forgotten, 0:08:13.606 I'll refresh your memory a little bit about it. 0:08:16.969 So, very abstractly, if you look at a computer, 0:08:20.332 it's got a processor in it that actually executes instructions, 0:08:24.865 and it has a chunk of memory. And in that memory, 0:08:28.375 there is a portion of memory that corresponds to the stack 0:08:32.542 which is really where procedures, the interesting 0:08:36.051 stuff with procedures gets implemented. 0:08:40 And you also have a bunch of registers, so, 0:08:42.991 inside the processor. And you have a special variable 0:08:46.694 here called the stack pointer, which keeps track of where the 0:08:50.967 current head of the stack is that you can then start pulling 0:08:55.169 elements off. The general plan, 0:08:57.305 the caller and the callee interact with one another by 0:09:01.08 means of [SOUND OFF/THEN ON]. So when a caller wants to 0:09:05.626 invoke the callee, what it does is it takes 0:09:08.357 arguments of the procedure, and pushes them on to the stack 0:09:12.13 one after another. Then the last thing it does is 0:09:15.252 to tell the callee where it should return control after the 0:09:19.024 procedure function has actually been executed. 0:09:21.951 And that's the last thing that's pushed on top of the stack is 0:09:25.788 the [UNINTELLIGIBLE]. So then, after it does that, 0:09:29.831 the caller then jumps to a location where the callee's 0:09:33.299 modular is located, and then control passes to the 0:09:36.504 callee. What the callee then does is it 0:09:38.99 finds out what the return address is, pops the arguments 0:09:42.588 one after the other, and then goes ahead and runs 0:09:45.728 the function. And then at the end of that, 0:09:48.411 it looks at the return address and passes control back. 0:09:51.943 And before it does that, it actually puts the answer. 0:09:55.345 Let's assume that it puts the answer in a special register. 0:09:59.14 And that's part of the contract as well. 0:10:03 And once the caller gets the control back from the callee, 0:10:07.416 it can proceed as before. Now, the important thing about 0:10:11.678 this way of interacting between caller and callee with procedure 0:10:16.56 stack is this contract has to obey an invariant or a discipline 0:10:21.21 called the stack discipline. 0:10:24 0:10:31 And the essence of the stack discipline is that the callee 0:10:33.974 should leave the stack exactly the way the caller left it when 0:10:37.158 it invoked the callee, which means the caller had set 0:10:39.872 up a bunch of arguments on it, and it but the return address 0:10:42.951 on it. The callee should leave things 0:10:44.829 as is. And in fact, 0:10:45.769 the callee is not allowed to touch anything. 0:10:48.013 It should leave everything as is this pretty much except for 0:10:51.092 the register that has the answer. 0:10:52.762 And as long as this discipline is maintained, 0:10:55.058 and that invariant is maintained across procedure 0:10:57.563 implications, this model of using stacks is 0:10:59.755 extremely powerful. You could implement all sorts 0:11:03.954 of procedures, nested procedure, 0:11:06.043 I mean, recursive procedure mutual recursive procedures, and so on so on 0:11:09.547 because each of these implications of the procedure 0:11:12.917 has a certain portion of the stack that corresponds to an 0:11:16.691 activation frame. And as long as this discipline 0:11:19.858 is maintained, you can do quite complicated 0:11:22.689 and interesting combinations of modules. 0:11:25.317 But the problem is that that modularity depends crucially on 0:11:29.293 the stack discipline being maintained. 0:11:33 And any violation of it, then the callee callee can disrupt 0:11:36.541 the caller and bring it down. There are many ways in 0:11:39.9 which this discipline could be violated. 0:11:42.282 And the easiest one is that there is some error or bug in 0:11:45.702 the callee, and the callee corrupts the stack. 0:11:49 0:11:54 Or the callee corrupts the stack pointer, 0:11:57.171 so it actually [SOUND OFF/THEN ON] control, but the stack 0:12:01.611 pointer points somewhere else and some bad instruction runs, 0:12:06.289 or you have a problem. Now, another problem is that 0:12:10.253 the callee crashes. For instance, 0:12:12.79 there is a divide by zero, or there is some other 0:12:16.595 violation that causes of the callee to crash. 0:12:20.084 And then the caller comes crashing down as well. 0:12:25 0:12:29 And there's a bunch of other reasons, but all of them have to 0:12:32.913 do with the stack discipline being violated, 0:12:35.717 or with the callee crashing and control never returning to the 0:12:39.695 caller, which means that the caller and the callee share a 0:12:43.282 fate. If something bad happens to the 0:12:45.63 person who has called, sorry, the callee, 0:12:48.239 then the caller struggles as well and isn't able to continue. 0:12:53 0:12:57 So colloquially this is referred to as fate sharing. 0:13:00.849 And the resulting modularity is soft because any fault or error 0:13:05.528 [SOUND OFF/THEN ON] affects the caller. 0:13:08.396 The caller, there isn't any kind of a firewall where errors 0:13:12.773 in the callee are insulated from errors in the caller. 0:13:16.773 There's no shielding between the caller and callee. 0:13:20.547 And, where is this thing going? Like, all right, 0:13:24.094 so there's no insulation between caller and callee. 0:13:29 And the resulting modularity is not as hard as we would like it 0:13:35.784 to be. So this is the problem we'd 0:13:39.396 like to solve today. And the first solution we're 0:13:44.649 going to discuss is a way of organizing callers and callees 0:13:50.996 into an organization called client service organization. 0:13:58 0:14:04

And the main idea is going to actually involve a different 0:14:07.504 abstraction by which callers and callees communicate with one 0:14:11.192 another from the abstractions we've seen already. 0:14:14.143 We've already seen the memory abstraction where you could 0:14:17.586 write values to a name, and another person could read 0:14:20.782 from it. And we've already seen the 0:14:22.872 interpreter abstraction. It could turn out we are going 0:14:26.192 to use a different abstraction [SOUND OFF/THEN ON] path 0:14:29.512 abstraction to implement the client service organization. 0:14:34 And the idea is the following. The program is going to be 0:14:37.432 decomposed into clients and services. 0:14:39.639 And you might have many clients and many services, 0:14:42.642 and you could have a client which is a client of one service 0:14:46.259 and [SOUND OFF/THEN ON] is a client of yet another [SOUND 0:14:49.691 OFF/THEN ON] things like that. But any pair-wise 0:14:52.756 interaction is going to be between a client and a service. 0:14:56.25 And think of mapping that example onto here. 0:14:58.886 Think of the callee, for example, 0:15:00.847 being the server and the caller being the client. 0:15:04.586 The caller wants some work done, so it's the client. 0:15:09 And it invokes the service, the callee, to get that work 0:15:11.986 done. And the plan is going to be 0:15:13.723 that the client and the service are going to run on different 0:15:16.981 computers, physically different computers, and we are going to 0:15:20.294 connect the computers up with wire. 0:15:22.14 And the idea is the moment you do that, the crash of a callee 0:15:25.398 doesn't actually bring the caller coming down because it's 0:15:28.493 running on a completely different processor. 0:15:30.828 The stack is not shared. The memory is not shared. 0:15:34.602 The stack point is not shared. There's really no problem with 0:15:38.448 regard to the callee crashing, bringing the caller down. 0:15:41.974 Of course, we now need a way by which the client can communicate 0:15:46.012 its arguments to the service, and the service can communicate 0:15:49.858 its arguments back to the client. 0:15:51.91 Previously, we did the first one. 0:15:53.961 The arguments were communicated by putting them using memory on 0:15:57.935 the stack, and the answers were coming back to us from register. 0:16:03 And we don't have that shared state anymore. 0:16:05.804 So, we're going to have to implement that using messages. 0:16:09.456 And we're going to take these messages, use the communication 0:16:13.369 path abstraction, and send messages from the 0:16:16.173 client to the service. So imagine that time flows 0:16:19.304 downwards starting from when the client invokes the service. 0:16:23.152 A message is sent from the client to the service 0:16:26.804 saying here's all the arguments, and here's the procedure that I 0:16:30.913 want you to run. And it takes that information 0:16:34.823 up, somehow packages it up into a message, and calls send. 0:16:38.407 OK, and the assumption is that the client somehow already knows 0:16:42.306 something about the name of the service or the location of the 0:16:46.141 service. That's outside of the scope of 0:16:48.531 the current discussion. Let's pretend somebody tells 0:16:51.738 you that here's where the service is running that can run 0:16:55.259 this function for you. So it takes the arguments in 0:16:58.403 the name of the procedure, packages it up the message, 0:17:01.736 and send it across. Well, the service gets the 0:17:05.523 message. It validates the message to 0:17:07.574 make sure that it's the right sizes, and it's not too big, 0:17:10.914 and so on. And then, it takes this message 0:17:13.316 and then does some processing on that message. 0:17:15.953 The technical term for it is going to be called un-marshaling 0:17:19.468 because when we took these arguments and put into a 0:17:22.398 message, that process is called marshaling. 0:17:24.859 The service is going to un-marshal this message and 0:17:27.789 obtain the actual arguments and the name of the procedure, 0:17:31.128 and then it's going to run it. 0:17:34 0:17:40 I don't know if it's one L or two L's. 0:17:42.17 And then it's going to run the procedure that's named here. 0:17:45.573 And it's going to find the answer. 0:17:47.509 And then when it gets the answer, it does the same thing 0:17:50.736 back. It puts it back into the 0:17:52.437 message and sends it across the client. 0:17:54.666 And now the client is waiting for this message because it 0:17:57.952 sends off the message to the service to run this thing. 0:18:02 It gets this answer back. It does the same thing. 0:18:05.461 It takes the message in that it recovers the answer from it, 0:18:09.716 and then it runs continuously. So this is the basic idea in 0:18:13.899 client/server organization. And the way in which we solve 0:18:17.937 this problem is that these two problems are solved because the 0:18:22.336 callee can't really corrupt the stack or the stack point or 0:18:26.519 anything else that in this model would have affected the caller. 0:18:32 And because we have put them [SOUND OFF/THEN ON] physically 0:18:34.889 different computers and hooked them up with, 0:18:37.032 let's say, a wire, the service crashing does not 0:18:39.373 actually bring the client, if the service decides that it 0:18:42.163 crashes, then the client actually doesn't come crashing 0:18:44.854 down. Of course, the client has to 0:18:46.498 somehow have a plan by which it knows that the service is still, 0:18:49.637 for example, the client has to know whether 0:18:51.729 the service has crashed or whether the service is just 0:18:54.37 taking a long time to run something. 0:18:56.113 That's something we have to address, and we will in a bit. 0:19:00 But as long as the client is able to do that, 0:19:02.882 a service going away or crashing is not really going to 0:19:06.42 bring the caller or the client down with it. 0:19:09.236 So, some properties of this organization are, 0:19:12.119 first of all, that it's modular. 0:19:14.15 It has essentially the same modularity as we had with 0:19:17.556 procedures because if you had enough computers, 0:19:20.57 you could put the different procedures on 0:19:24.042 different computers, all of the caller/callee 0:19:26.924 relationships, and you can preserve, 0:19:29.217 essentially, the same modularity that you 0:19:31.838 had before. Moreover, this modularity has a 0:19:36.737 different adjective in front of it different from soft. 0:19:41.505 This modularity is enforced. What that means is not only is 0:19:46.627 it a modular organization, it's one where errors of, 0:19:51.131 for example, things where one module fails or 0:19:54.928 crashes does not bring the other one come crashing down. 0:19:59.785 So, modularity is more enforced than when they were both running 0:20:05.348 on the same computer using the procedure interface. 0:20:11 And the third property of this kind of modularity is something 0:20:15.587 already mentioned. It lies on the message 0:20:18.596 abstraction, actually the communication path abstraction. 0:20:22.807 The client and service communicate with each other 0:20:26.493 through messages. And these aren't 0:20:29.501 arbitrary messages. You can't just sort of take a 0:20:33.337 random string of bytes and send it to the service. 0:20:38 It's actually messages that correspond to a particular 0:20:40.853 format. And these are really [SOUND 0:20:42.684

OFF/THEN ON]. So, that way the service isn't 0:20:45 surprised and the client isn't surprised when messages that 0:20:48.123 don't conform to that pattern arrive. 0:20:50.061 I mean, you know exactly what kind of message is going to 0:20:53.076 arrive. And anything that doesn't 0:20:54.799 conform to what is agreed upon in advance is rejected. 0:20:57.653 But you basically make it so that you explicitly declare the 0:21:00.83 nature of these messages much like you did the nature of 0:21:03.792 procedural interface. But now, because you've 0:21:07.747 physically separated it, you have a more enforced 0:21:11.394 modularity than in the previous model. 0:21:14.205 Of course, nothing comes for free. 0:21:16.713 It's not like you got this enforced modularity, 0:21:20.208 and you have all of these nice properties that you did before. 0:21:24.843 Here, you had a nice property that the callee could, 0:21:28.718 has the property that either when it runs and it returns to 0:21:33.125 you, you know exactly what happened. 0:21:37 And if it doesn't return, you know that you usually come 0:21:40.522 crashing down. If the callee doesn't return, 0:21:43.275 it means that control never comes back to the caller. 0:21:46.605 So it's not like the caller is left wondering what really 0:21:50.192 happened because the caller doesn't get control again. 0:21:53.586 Here, you have a problem. If the callee, 0:21:56.083 the service doesn't return back to the client, 0:21:58.965 the client actually doesn't know what's going on. 0:22:03 When you have two machines, two computers connected over a 0:22:06.538 wire or over a network, it's extremely hard to 0:22:09.332 distinguish between a service running really, 0:22:12.064 really slow, and a service that's gone away. 0:22:14.733 And we'll revisit this a few different times in this course. 0:22:18.334 But it's going to be impossible for us to tell for sure, 0:22:21.749 although [SOUND OFF/THEN ON] very hard. 0:22:24.108 It's going to be really hard for us to tell for sure whether 0:22:27.771 there is something exactly happened. 0:22:31 And if the service didn't return, we don't really know for 0:22:34.521 sure without much more machinery whether it was just that the 0:22:38.227 service is still running, or whether it's crashed, 0:22:41.254 and we are just waiting. And so, this organization 0:22:44.281 requires a timer at the client, and there are many names given 0:22:48.05 to this timer. I mean, people call them 0:22:50.397 [Unintelligible] eyes, or people call them with 0:22:52.806 various names. I'm going to call it a watchdog 0:22:55.586 timer, where the client has to keep track using some kind of a 0:22:59.355 timer [SOUND OFF/THEN ON] of the service. 0:23:03 And if the service doesn't return within a certain period 0:23:06.406 of time, the client has to time out and say, well, 0:23:09.387 the service didn't return, and I'm not quite sure what 0:23:12.612 happened. It might be that the procedure 0:23:14.984 I wanted to execute had ran, but I didn't get the answer. 0:23:18.391 Or it might be that the procedure didn't get executed at 0:23:21.737 all, and I have to deal with it. And you might be able to, 0:23:25.205 by retrying the procedure, or you might contact another 0:23:28.49 service which provides the same functionality, 0:23:31.228 but the client has to deal with all of that. 0:23:35 So fundamental to this client's organization is the notion of a 0:23:39.021 time out. And we didn't have that here 0:23:41.421 because here, if the callee decides that it's 0:23:44.275 just going to continue on and not return, the caller never 0:23:47.972 gets [SOUND OFF/THEN ON]. So, it doesn't have this 0:23:51.151 decision to make. There is no such notion of a 0:23:54.07 watchdog that we have to worry about in this other 0:23:57.248 organization. 0:23:59 0:24:13 So another nice property of this client service organization 0:24:16.896 is that so far we've presented it in the context of the client 0:24:20.924 and the service being modularized from each other, 0:24:24.16 and we've enforced modularity to the client and the service. 0:24:28.056 But in fact, there is another nice property 0:24:30.83 to it, which is that a client service organization allows us 0:24:34.726 to design modules and design systems where clients get 0:24:38.226 modularized from each other. We can achieve soft modularity 0:24:42.923 by protecting clients from each other. 0:24:46 0:24:56 The idea here is that if you have many clients all of which 0:24:59.504 want to use a given service, for example, 0:25:01.921 there's a service that's, let's say, implemented by a 0:25:05.063 bank and what it does is it's the service that deals with 0:25:08.447 managing your accounts, and you can move money between 0:25:11.649 accounts, and it will tell you your account balance and so on. 0:25:16 You can implement that as one service, and many, 0:25:18.713 many clients can share the same service. 0:25:20.965 Now, all of the clients trust the service because, 0:25:23.795 I mean, if you are a customer of a bank, and you are using 0:25:27.086 your [SOUND OFF/THEN ON] to look for your account balance, 0:25:30.377 that means you sort of trust the bank. 0:25:32.514 And all the clients trust the service. 0:25:34.65 But the clients sure don't trust each other. 0:25:38 And the nice thing about this organization is that you can use 0:25:42.26 the service in the form of an intermediary that allows the 0:25:46.241 clients to be separated from each other, each of which can 0:25:50.222 use the service. But the clients don't have to 0:25:53.365 trust each other, and clients don't really have 0:25:56.577 to know about each other's information. 0:26:00 And this idea of using a service to modularize clients 0:26:03.627 from each other is called a trusted intermediary. 0:26:06.912 There is many examples of trusted intermediaries that 0:26:10.471 we'll see in this course. In fact, tomorrow's recitation 0:26:14.235 on the X Windows system has a system where your computer 0:26:18.205 [SOUND OFF/THEN ON] is going to be managed by a service called 0:26:22.38 by the X Windows system. And there are many clients that 0:26:26.144 are going to use that service. And the clients don't actually 0:26:30.25 trust each other. They want to get modularized 0:26:33.33 away from each other. And the X Windows system as a 0:26:38.037 trusted intermediary deals with that. 0:26:40.566 [SOUND OFF/THEN ON] display, and it arranges for the clients 0:26:44.711 to be designed each independent from the other. 0:26:47.943 And in general, we are going to see in the next 0:26:51.175 few lectures, many examples of the operating 0:26:54.196 system being a trusted intermediary, 0:26:56.655 arranging for many different clients to use some resource on 0:27:00.8 your computer like the processor or the memory of the disk. 0:27:06 And our architecture for the operating is going to end up 0:27:09.629 being in the form of these trusted intermediaries. 0:27:13 0:27:30 So, so far we've seen what client service organization 0:27:32.95 means. It means you have a client and 0:27:34.953 the service, and they communicate with messages using 0:27:37.848 the communication path abstraction of send and receive. 0:27:40.854 And we've seen some properties of client service organization. 0:27:44.249 But I haven't actually told you how to implement any of this 0:27:47.533 stuff. And so that's what we're going 0:27:49.537 to do the rest of today. And in fact, 0:27:51.541 we are going to continue with different ways of implementing 0:27:54.825 various forms of client service many times in the course. 0:27:59 0:28:15 So there are many ways to implement client service 0:28:18.344 organization. And all of them have to do 0:28:21.006 with, all of them involved different ways in which messages 0:28:24.965 are sent between client and service. 0:28:27.354 A common way, and a pretty standard way, 0:28:30.017 of implementing it is something called a remote procedure call. 0:28:35 0:28:43 There are many examples of remote procedure called systems. 0:28:47.626 I mean, one of the most common ones is something called the 0:28:52.253 Sun Remote Procedure Call system, or Sun RPC. 0:28:56.002 That's one example. There are many other examples 0:28:59.831 as well. A more modern example which 0:29:02.623 some of you may have heard of is a relatively new system, 0:29:07.091 about five years old, called XML RPC. 0:29:11 So if you've heard of buzzwords like Web services in 0:29:14.891 business-to-business interactions, 0:29:17.409 or business-to-business applications, 0:29:20.156 these things use something called XML RPC. 0:29:23.285 And, there is a lot of different three letter acronyms 0:29:27.329 and four letter acronyms. This has led to something 0:29:31.793 called SOAP, which stands for the Simple Object Access 0:29:35.595 Protocol. So there are many different 0:29:38.178 ways of implementing RPC systems. 0:29:40.473 And until last year or a couple of years ago, 0:29:43.63 we used to talk about Sun RPC as an example in this class. 0:29:47.719 But I decided that so 20th century. 0:29:50.158 So we're going to talk about XML RPC today. 0:29:53.171 It has a property that's much more inefficient, 0:29:56.471 but that's [SOUND OFF/THEN ON] the fact that computers have 0:30:00.632 become faster. We don't have to worry in many 0:30:05.967 cases about efficiency. So we are going to talk a 0:30:10.714 little bit about how XML RPC works. 0:30:14.076 So let me first show you [UNINTELLIGIBLE] what a client 0:30:19.417 written what this kind of RPC looks like. 0:30:23.373 [SOUND OFF/THEN ON] going to show you [SOUND OFF/THEN ON]. 0:30:30 0:30:38 All right, I had to [SOUND OFF/THEN ON] to make sure it 0:30:43.033 would fit on this. OK, all right, 0:30:46.016 the way this thing works is actually very, 0:30:49.838 very simple. This uses something called XML 0:30:53.754 RPC Library for Java that was written by the Apache people. 0:31:00 And once you incorporate that library, your program becomes completely 0:31:03.432 easy. So let me just walk you through 0:31:05.458 this. The high level idea here is 0:31:07.258 that it's transferring [UNINTELLIGIBLE] account to the 0:31:10.241 other [SOUND OFF/THEN ON] that's run by the bank. 0:31:12.942 So, the first line of this thing here creates an XML RPC 0:31:16.036 object, an XML RPC client object. 0:31:17.837 And what you give it is actually the name of the 0:31:20.482 service. So somebody has to tell you the 0:31:22.676 name of the service. And I don't want to get into 0:31:25.377 the details of everything here, but the basic idea is your 0:31:28.585 backname.com colon 8080 is the name, the DNS name at which the 0:31:32.017 service runs and [the port?]. The thing about XML RPC is that 0:31:36.762 it runs [SOUND OFF/THEN ON] which is what you use to 0:31:39.661 transfer objects on the Web. And underneath, 0:31:42.105 we talk about how it's implemented; underneath this is 0:31:45.117 implemented using a standard [SOUND OFF/THEN ON] in HTTP 0:31:48.244 called a post which allows, normally HTTP has a [get?] 0:31:51.256 where you retrieve stuff. But it also has a post that 0:31:54.212 many of you are familiar with where the client can push some 0:31:57.566 stuff to the server. And it just uses post [SOUND 0:32:00.294 OFF/THEN ON]. What's going on [SOUND OFF/THEN 0:32:03.815 ON]? You create a vector of 0:32:05.338 parameters, and you fill that vector in with, 0:32:07.916 in this case, your account number. 0:32:09.849 And let's say here the idea is this sort of thing is very 0:32:13.129 popular in big companies like [Fold?] or Cisco, 0:32:15.824 which have thousands of suppliers. 0:32:17.757 And, they never actually maintain a lot of inventory of 0:32:20.92 their own. They're always trying to figure 0:32:23.322 out the latest cost of any of their supplies. 0:32:25.899 And they are using this Web service like interface. 0:32:30 In fact, this is also called a Web service interface to 0:32:33.127 communicate with their suppliers to always have [SOUND OFF/THEN 0:32:36.718 ON] whether their suppliers have any given item in stock, 0:32:39.962 and how much it costs [SOUND OFF/THEN ON]. 0:32:42.337 So, let's pretend you have done that in your company, 0:32:45.349 and you are trying to pay off, you are taking your suppliers' 0:32:48.824 account number and pay some money to him, 0:32:51.141 OK? So in this case, 0:32:52.241 whatever, dollars is that argument. 0:32:54.211 So you create parameters. And all you do at the end is 0:32:57.281 you use these XML RPC client objects, and [SOUND OFF/THEN ON] 0:33:00.756 it presents to you called execute. 0:33:04 And you give it two arguments. The second argument is the 0:33:07.918 parameters. And the first argument is the 0:33:10.718 name of the procedure that you wish to run on the service. 0:33:14.707 OK, that's in this case called money transfer. 0:33:17.856 So, corresponding to this, somewhat longer is a piece of 0:33:21.705 code running on the service which implements the service 0:33:25.554 side of it, which basically obtains [SOUND OFF/THEN ON] 0:33:29.333 calls an object that un-marshaled arguments, 0:33:32.342 and then it will execute money transfer, both of which [SOUND 0:33:36.541 OFF/THEN ON] little that actually has to run on the 0:33:40.04 service. It's just a little bit longer 0:33:43.896 than this. Now, this line here is 0:33:45.989 important. That's the line on which you 0:33:48.474 get your result. And this gives the result as a 0:33:51.483 string. If I tell you that I finished 0:33:53.837 transferring X dollars from this account to that account, 0:33:57.5 or if I tell you I couldn't transfer, and there was some 0:34:01.097 kind of an error, or you might actually not get 0:34:04.105 any answer, in which case the underlying library that 0:34:07.506 implements this [UNINTELLIGIBLE PHRASE] your code has to deal 0:34:11.43 with. And how you deal with it is a 0:34:14.8 little tricky because you don't, and you'll see this in a 0:34:18.275 moment, you don't quite know what happened when you didn't 0:34:21.813 get an answer back from the service. 0:34:23.986 You don't know if this actually got your transfer request and 0:34:27.71 crashed after that. And, it got the request. 0:34:31 It actually implemented the transfer and then crashed. 0:34:34.566 It just couldn't send your response back. 0:34:37.257 So, you're not quite sure whether you need to retry the 0:34:40.89 request or not. You will actually see how to 0:34:43.784 deal with [UNINTELLIGIBLE] in a few minutes. 0:34:46.677 Now, there's one thing that's really important about this line 0:34:50.781 of code. This XML RPC dot execute money 0:34:53.338 transfer block, that line of code actually is 0:34:56.299 not something that runs on the service. 0:35:00 OK, it's a local procedure. XML RPC dot execute is a local 0:35:03.277 procedure [UNINTELLIGIBLE]. OK, and that procedure is an 0:35:06.439 example of something called a stop because what it is, 0:35:09.486 is a stop that to this caller [fakes out?] the fact that 0:35:12.648 there's a service somewhere else. 0:35:14.487 I mean, it prevents the caller from having to deal with [15?] 0:35:17.937 arguments and putting it in the message and sending it to the 0:35:21.386 other side. The caller just calls it 0:35:23.398 [over?] the procedure, giving it suitable arguments 0:35:26.273 that [UNINTELLIGIBLE] for this function now to do the work of 0:35:29.722 sending a message across the network. 0:35:33 But this is a local call. So, what happens underneath? 0:35:35.961 Underneath in the library, once you call XML RPC dot 0:35:38.659 execute in this example, somebody does work. 0:35:40.933 That library does the work of taking the different arguments 0:35:44.054 that have been presented to it and converting them into a 0:35:47.016 message, marshaling all of the stuff into a message, 0:35:49.714 and then shipping that message off to the server. 0:35:52.781 That has to be going to some kind of a format on the 0:35:55.479 [UNINTELLIGIBLE], right? 0:35:56.695 Ultimately on this wire connecting the client to the 0:35:59.552 service, there's [forming?]. And I already mentioned that 0:36:02.514 this runs on top of HTTP. And so we can actually look at 0:36:07.98 what that looks like. OK, so post here is the method 0:36:13.048 that you use in HTTP. The slash RPC2 is actually the 0:36:18.115 same thing that was used in, if you remember in the previous 0:36:23.977 screenshot, when we [did the new caller?] to get a new XML RPC 0:36:30.038 client, we gave it a server name and a port number. 0:36:36 But we also gave it something called RPC2. 0:36:38.673 Now, that's just like a file on the other side. 0:36:41.673 It says there are many different RPC programs running 0:36:45.065 on your service. And I want RPC2 to run. 0:36:47.608 I mean, I could have named it anything I wanted. 0:36:50.673 And that's a lot like giving a file name on a URL. 0:36:53.869 And then, you go on. You give it the host name. 0:36:56.869 This is a lot like an HTTP header. 0:37:00 The interesting new stuff here is in the XML arguments. 0:37:03.109 But those were not [familiar?] to XML. 0:37:05.239 It's just a method, a way of sending things that 0:37:07.946 have attributes and values associated with them. 0:37:10.652 So, every method has the format that its attributes and values, 0:37:14.222 and they can be nested within each other. 0:37:16.525 The only interesting thing that's here, this particular XML 0:37:19.865 RPC system supports a few different data formats. 0:37:22.629 You can do integers, and characters, 0:37:24.644 and strings, and doubles, 0:37:26.026 and floats, and a few different things like that. 0:37:30 And I thought it just means [that the?] 32 bit integer, 0:37:32.882 and you can take numbers that sum up your account number, 0:37:35.871 your supplier's account number, and the amount of money that 0:37:39.021 you want to transfer. The good thing is that all this 0:37:41.797 stuff has gone underneath the covers, and you don't actually 0:37:45 have to deal with it if you are [writing?] the client or you are 0:37:48.362 writing [the sets?]. All this work happens 0:37:50.551 underneath. So, it greatly simplifies your 0:37:52.74 ability to take, implement [fine?] service 0:37:54.928 programs with [UNINTELLIGIBLE] and services being separated 0:37:58.024 from one another. 0:38:00 0:38:09 OK. So, so far, we've made it 0:38:10.541 look a lot except for this little timer that you 0:38:13.79 have to maintain. We've made it look a lot like a 0:38:16.433 remote procedure call, it's like a procedure call. 0:38:19.407 In fact, the code here, the only difference is you 0:38:22.105 replace what was previously what would have been [UNINTELLIGIBLE] 0:38:25.629 transfer with arguments we replace with a [UNINTELLIGIBLE], 0:38:28.823 taking the name of the procedure we want to run on the 0:38:31.742 service [UNINTELLIGIBLE]. So, it looks a lot like a 0:38:35.447 procedure [calling?]. That actually is a pretty 0:38:38.111 deceptive thing. And in fact, 0:38:39.733 a hint at that is you can get at the bottom of this thing 0:38:43.15 up there, there is a light that says you have to deal with XML 0:38:46.683 RPC exception. And that's the kind of 0:38:48.768 exception you get when online RPC library, the size that it 0:38:52.127 hasn't heard an answer from the service in a while, 0:38:55.023 it [does?] an exception. And your code has to deal with 0:38:58.15 it. You are never going to get that 0:39:00.119 kind of exception from a regular procedure call. 0:39:04 I didn't hear back from the caller or from the callee to do 0:39:08.046 something. See, that's a new exception 0:39:10.627 that you didn't previously have to deal with. 0:39:13.697 I mean, you have to deal with other kinds of exceptions but 0:39:17.744 not this one. So, an RPC is not the same as a 0:39:20.813 procedure call. And in fact, 0:39:22.697 it's a little unfortunate that, [UNINTELLIGIBLE] reasons, 0:39:26.604 we are stuck a little with a [SOUND OFF/THEN ON] procedure 0:39:30.581 call. In fact, increasingly more and 0:39:33.023 more RPC systems don't look like procedure calls at all in terms 0:39:37.418 of the semantics. But we were so stuck with the 0:39:42.013 name that people continue to have various kinds of procedure 0:39:46.111 calls, and they used the same name for it. 0:39:48.958 And the first main difference arises from the fact that there 0:39:53.125 is no fate sharing between client and service. 0:39:56.25 If the service crashes, the client doesn't crash. 0:39:59.583 Already you have a big difference between a regular 0:40:03.055 procedure call. And previously I presented this 0:40:07.072 as an advantage because if you have fate sharing, 0:40:10.281 then this caller is always at the mercy of the callee. 0:40:13.825 But because you have no fate sharing, you have other problems 0:40:17.837 to deal with. And in particular, 0:40:19.909 all of these stem from the fact that it's extremely hard to 0:40:23.787 distinguish between a failure versus extremely slow. 0:40:28 0:40:34 And it will turn out that we revisit this over and over 0:40:36.731 again. We'll talk about networks and 0:40:38.501 reliable [concentration?] over networks and deal with it, 0:40:41.334 and then we're going to talk about [fall?] tolerance, 0:40:43.964 and we're going to talk about an idea for [UNINTELLIGIBLE], 0:40:46.898 and then we're going to talk about something called 0:40:49.428 transactions. And they're all going to deal 0:40:51.754 with this problem that it's going to be very hard for us to 0:40:54.688 tell, when you ask somebody to do a piece of work, 0:40:57.167 whether they did fully or did nothing. 0:41:00 OK, and this is going to be a repeated theme, 0:41:02.865 a team that is going to repeat in the course. 0:41:05.731 But to complicate why this is hard, let me say three possible 0:41:09.639 things that could happen when you have [UNINTELLIGIBLE PHRASE] 0:41:13.613 service and what kind of semantics you want from a client 0:41:17.26 service interaction. They all stem from the fact 0:41:20.321 that this is extremely hard to determine. 0:41:22.927 The first semantics that you might want is the idea semantic. 0:41:26.835 The client talks to the service, and either the service 0:41:30.352 answers with a response or it doesn't. 0:41:34 OK, and that's something called exactly-once semantics. 0:41:37.754 So in this example here, underneath the library, 0:41:41.021 it may time out. And it may wish to retransmit. 0:41:44.219 Or it may wish to throw an exception at the caller. 0:41:47.695 The client may want to send [this?] again. 0:41:50.545

But in an ideal case, you want exactly once this 0:41:53.812 amount of money to be moved from bank account one to bank account 0:41:58.262 two, right? You certainly don't want your 0:42:01.991 amount of money to be moved twice to your supplier. 0:42:05.426 The [term?] of this is ideal, and it's going to be pretty 0:42:09.272 difficult, and extremely hard, to achieve exactly-once 0:42:12.913 semantics. And, we're going to talk about 0:42:15.66 different methods in the course. This is not an easy problem at 0:42:19.919 all. It stems from the fact that it 0:42:22.254 is very hard to know. So, you might give up a little 0:42:25.757 bit and say, OK, I can't really get exactly-once 0:42:28.985 semantics very easily. But let me try for at least 0:42:32.351 once. So what this means is the 0:42:35.44 client will keep retrying or the [UNINTELLIGIBLE] will keep 0:42:38.783 retrying. And you can decompose it in 0:42:40.857 either way until it is sure that this [UNINTELLIGIBLE PHRASE] at 0:42:44.488 least once. Now, it might have succeeded 0:42:46.735 more than once because the service is very slow. 0:42:49.444 And then it times out. You try it again, 0:42:51.691 the service says, oh, OK, I see it; 0:42:53.65 I must not do it again. But, it's at least once, 0:42:56.359 OK? Now, at least once is not nice 0:42:58.261 semantics that you're using at least one semantics in your code 0:43:01.833 here, right, because your supplier might end up with $7 0:43:04.945 million instead of $1 million. But at least once is OK if the 0:43:10.416 service has some kind of semantic called idem-potent 0:43:14.523 semantics. What that means is that when 0:43:17.583 you do an operation more than once, the answers are the same 0:43:22.335 as at-least-once. A simple example of this was 0:43:25.959 the transfer of money, but you checking your bank 0:43:29.825 account balance. It doesn't really matter that 0:43:33.987 you do it a hundred times. As long as one of them succeeds 0:43:37.895 and you get your bank balance, you are fine. 0:43:40.842 I mean, doing it seven times does not really change anything; 0:43:44.955 it does not move extra money anywhere else. 0:43:47.834 So that's an example of an idem-potent action, 0:43:50.919 which works out well with at-least-once semantics. 0:43:54.278 And at-least-once semantics is much easier to obtain than 0:43:58.116 exactly-once semantics. And the third kind of semantics 0:44:01.818 is something converses: most-once semantics. 0:44:06 That means zero or more times. And here, the challenge is 0:44:09.632 really figuring out if it really worked at least once around. 0:44:13.524 I mean, I [pushed?] once around. 0:44:15.535 So, if you don't get a response back, then you time out. 0:44:19.102 And it might be that it didn't succeed at all. 0:44:22.021 And you say that's fine. I'll deal with it separately. 0:44:25.459 And if you get a response back, then you know that worked. 0:44:30 It turns out, even that is going to be a 0:44:32.035 little bit tricky to implement. But these kinds of semantics 0:44:35.114 you could expect from your RPC system. 0:44:37.045 Now, actually the first R in most RPC systems like XML RPC 0:44:40.02 don't really deal with any of this in a particular, 0:44:42.629 I mean, they don't really provide any well-defined 0:44:45.186 semantics. It's usually for the client 0:44:47.117 sitting on top, and the service is still, 0:44:49.205 they don't know what kind of semantics they need, 0:44:51.71 and implement that at a higher [layer?] at least with most of 0:44:54.841 these standard protocols. But many of them are well 0:44:57.451 equipped to deal with at-least-once semantics. 0:45:01 OK, and through the course, there are different ways in 0:45:06.168 which we'll accomplish these different goals, 0:45:10.379 these different semantics that we want from our different RPC 0:45:16.122 systems. Now, there's another difference 0:45:19.854 between regular procedure calls and remote procedure calls, 0:45:25.405 or more generally from client service. 0:45:30 Remember in the second lecture, I told you that you can always 0:45:33.003 get more bandwidth, and you can always get more 0:45:35.267 processing past [UNINTELLIGIBLE], 0:45:36.843 but one thing you can't actually change is the legacy 0:45:39.403 between two computers connected by a wire connected by a 0:45:42.11 network. The speed of light doesn't 0:45:43.784 change. What that means is that with 0:45:45.507 time, the number of constructions that you can run, 0:45:47.969 when you have two computers separated by a wire, 0:45:50.283 there is a certain delay between that no matter what you 0:45:52.99 do. So there is a certain delay 0:45:54.467 that you do a procedure call that's a remote procedure call. 0:45:57.372 It takes a certain delay to send a message and to get a 0:46:00.03 response back. Even as computers get faster 0:46:03.542 and faster and faster, that isn't changing at all. 0:46:06.242 But the problem is that the number of instructions you can 0:46:09.382 run within that duration is increasing with time because 0:46:12.632 it's a fixed amount of time, and the number of constructions 0:46:15.883 you could run locally on the computer increases with time. 0:46:19.023 So that has led people to be more aggressive about what they 0:46:22.273 do in a remote procedure call. What people said it is, 0:46:25.193 wait, it doesn't make sense for a client to issue a procedure 0:46:28.499 call, relocate to a service, and then just sit and wait like 0:46:31.749 in a regular procedure call, the answer. 0:46:35 I mean, I'm just sitting there twiddling my thumbs, 0:46:38.556 and this thing is that way, and I'm waiting for it to 0:46:42.256 respond. I could be doing work. 0:46:44.39 So, that's led people to changing the synchronous model 0:46:48.231 of RPC, of a procedure call interface to do something called 0:46:52.428 asynchronous procedure call interface. 0:46:55.06 And all the things like XML RPC and its follow-on, 0:46:58.76 and the difference between SOAP and XML RPC is that I can 0:47:02.743 understand this, and builds on XML RPC. 0:47:05.447 It seems like very few people understand SOAP. 0:47:10 There is a lot of people who have given up on trying to 0:47:13.384 understand the specification. But XML RPC turns out to be a 0:47:17.019 really simple seven or ten page document that's very easy to 0:47:20.716 understand. A lot of people use it. 0:47:22.847 But anyway, all these systems support asynchronous RPC. 0:47:26.231 The idea here is that the client sends a procedure 0:47:29.302 [UNINTELLIGIBLE] remote request to the service. 0:47:33 And then it goes about doing its work. 0:47:35.154 When the service responds with an answer, it responds with not 0:47:38.705 just the answer, but also something that tells 0:47:41.325 the client which service request, which procedure caller 0:47:44.527 request it's responding to. And when the response comes 0:47:47.671 back, the client can't handle it. 0:47:49.534 Now, the way in which you have to implement this, 0:47:52.328 or the way you have to design this is that associated with 0:47:55.647 every procedure [called?] request, you also have to 0:47:58.558 associated what's called a handler because you're going to 0:48:01.876 issue this request to the service and then go about doing 0:48:05.136 your work. When the answer comes back, 0:48:08.571 the RPC library, the communication library has 0:48:11.189 to know: who gets this answer? Because you might have to 0:48:14.39 shield many such service requests. 0:48:16.31 So who gets this answer? So you associate with every 0:48:19.278 request a callback, a handler, which then is called 0:48:22.187 back. And that handler runs, 0:48:23.758 dealing with the answer that comes back. 0:48:26.028 And so, that actually allows you to be a little more 0:48:28.995 decoupled than we were with the [UNINTELLIGIBLE] even more 0:48:32.312 decoupled. The private services are even 0:48:36.017 more decoupled because [UNINTELLIGIBLE] waiting for the 0:48:39.775 service to turn an answer to. So that's the first way which 0:48:43.811 people have extended. The second way in which people 0:48:47.359 have extended is using this intermediary idea that we talked 0:48:51.465 about before where the services [cross the?] intermediary. 0:48:55.431 To use this intermediary idea to actually make design remote 0:48:59.536 message based communication systems where the client and 0:49:03.363 service don't actually have to be up and running at the same 0:49:07.469 time. So actually, 0:49:09.599 the client could send a request out to the service. 0:49:12.099 But the service is actually not up and running. 0:49:14.4 So, the idea is there is an intermediary that acts as a 0:49:17.099 broker on behalf of the service, and buffers the message. 0:49:19.9 And then, when the service comes out, the service knows to 0:49:22.75 pull the message from this intermediary. 0:49:24.699 That's buffered messages for it. 0:49:26.75 And then it passes the message, and then pushes the answer back 0:49:29.849 to the intermediary, which then stores the message 0:49:32.3 to some other intermediary perhaps. 0:49:35 And then the client knows to get the answer from that 0:49:37.945 intermediary. So now, we can actually have 0:49:40.267 clients and services that interact with each other without 0:49:43.495 actually having to be up and running at the same time. 0:49:46.497 There are many examples of this, and the notes talk about 0:49:49.669 various examples of intermediary communication. 0:49:52.274 So just [UNINTELLIGIBLE] what we've talked about so far is 0:49:55.502 that we looked at different ways of attaining modularity, 0:49:58.674 talked about soft modularity last time, and today about a 0:50:01.846 particular way of enforcing modularity using client service 0:50:05.13 organization. And the next few lectures, 0:50:08.474 we are going to solve a big weakness of the current system 0:50:11.706 which is that you need many different computers. 0:50:14.372 So we are going to take these ideas, and implement them all on 0:50:17.831 one computer. See you next week Tuesday.