# *An Introduction to Spring*

Usenix OSDI: November 14, 1994

Thomas W. Doeppner Jr.
Department of Computer Science
Brown University
Providence, RI 02912-1910
twd@cs.brown.edu

A 1-day version of this course and a 2-day hands-on version of this course are available. Contact IAPS at 617-497-2075, send email to info@iaps.com, or write to IAPS, 955 Massachusetts Ave., Cambridge, MA 02139.

# *Overview and Architecture* *1*

## Overview and Architecture

Outline

- Why Spring?
- The Spring System



### Explanation:

Spring is a new research operating system developed over the past six years by Sun Microsystems Laboratories and SunSoft. Over time, this research technology is being incorporated into SunSoft's mainstream products like Solaris, DOE, etc. As such, Spring is a kind of early peek into upcoming versions of Solaris, etc.

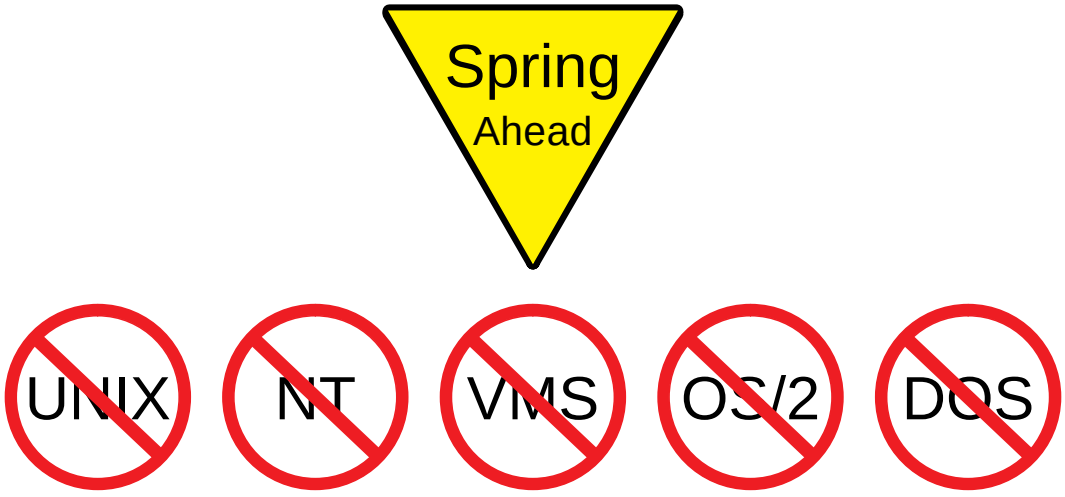
Spring is a distributed object-oriented operating system designed to provide high-performance object invocation, improved security, reliability, etc. SunSoft researchers are sharing this research technology with their colleagues in universities and research institutions world-wide in the form of the Spring Research Distribution.

This tutorial shows what Spring is all about and explains how it works. In this first module we discuss why a major computer company is working on yet another operating system and explain a bit of the general architecture.

*Notes:*

## Why Spring?

The Good, the Bad, and the Ugly ...




Spring

Overview and Architecture Slide 2 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

### Explanation:

Spring is a new operating system. It has been influenced by other systems, but it is not an attempt to support the same old style of doing things with yet another implementation.

The Spring effort has exerted a major influence on the Object-Management Group (OMG). Not too surprisingly, much of the CORBA specification from OMG resembles functionality that is supported by Spring. SunSoft's Distributed Objects Everywhere product (DOE) includes an implementation of CORBA ideas.

Spring is a research project: its ideas will find their way into DOE and Solaris (and perhaps other systems as well). As the Spring project and the DOE product evolve, there will be some convergence of the approaches and interfaces used in both.

*Notes:*

## Why Spring?

Goals

- Improve dramatically on existing systems
- Keep what is right and fix what is wrong
- Make distributed programming significantly easier



### Explanation:

Why was Spring developed? As the slide shows, the goals are impressive. Were these goals achieved? We address this issue throughout the rest of the tutorial.

*Notes:*

## Why Spring?

What's Wrong with Existing Operating Systems?

- Becoming difficult to evolve:
  - initial designs did not include distributed computing
  - initial designs did not include multithreading and multiprocessing
  - they have limits based on the machines for which they were originally designed
- Security is a problem
- They tend to be monolithic, big, and growing



### Explanation:

Listed in the slide are some of the drawbacks of existing systems. The list is certainly not exhaustive, nor do all systems have all of these problems. But these are the sorts of issues that must be dealt with by those wishing to develop a better operating system.

*Notes:*

## Why Spring?

Current Operating Systems Aren't Completely Bad ...

- A lot of applications run on them
- They are becoming relatively easy to use
- They are relatively stable
- They have some good ideas (e.g., virtual memory)
- They provide some degree of protection (e.g., memory protection)
- Many are fairly portable



### Explanation:

It would be foolish to throw out the good things about current operating systems along with the bad things. Here is a list of good things, with the same caveats as for the list of the preceding slide.

*Notes:*

## Why Spring?

Keep the Good, Fix the Bad

- Make the OS as modular, open, and extensible as possible
- Use methodologies and tools that increase quality and support modularity
- Make multithreading and multiprocessing easier
- Provide a solid foundation for security
- Interoperate with existing networked systems
- Be able to run existing applications



### Explanation:

Putting the previous two lists together, here is what we want for Spring.

*Notes:*

## Why Spring?

New Functionality

- Make it as easy as possible to write distributed applications
- Make it easy to develop sophisticated distributed system functions
- Make the system itself inherently distributed



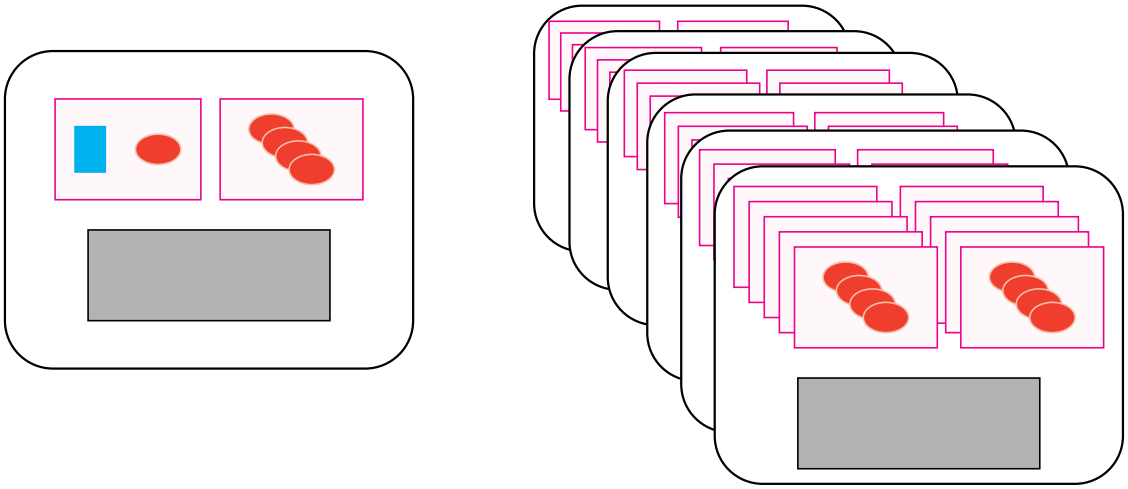
### Explanation:

Here are a few more requirements we place on Spring.

*Notes:*

## The Spring System

Anything Can Be Anywhere



Spring

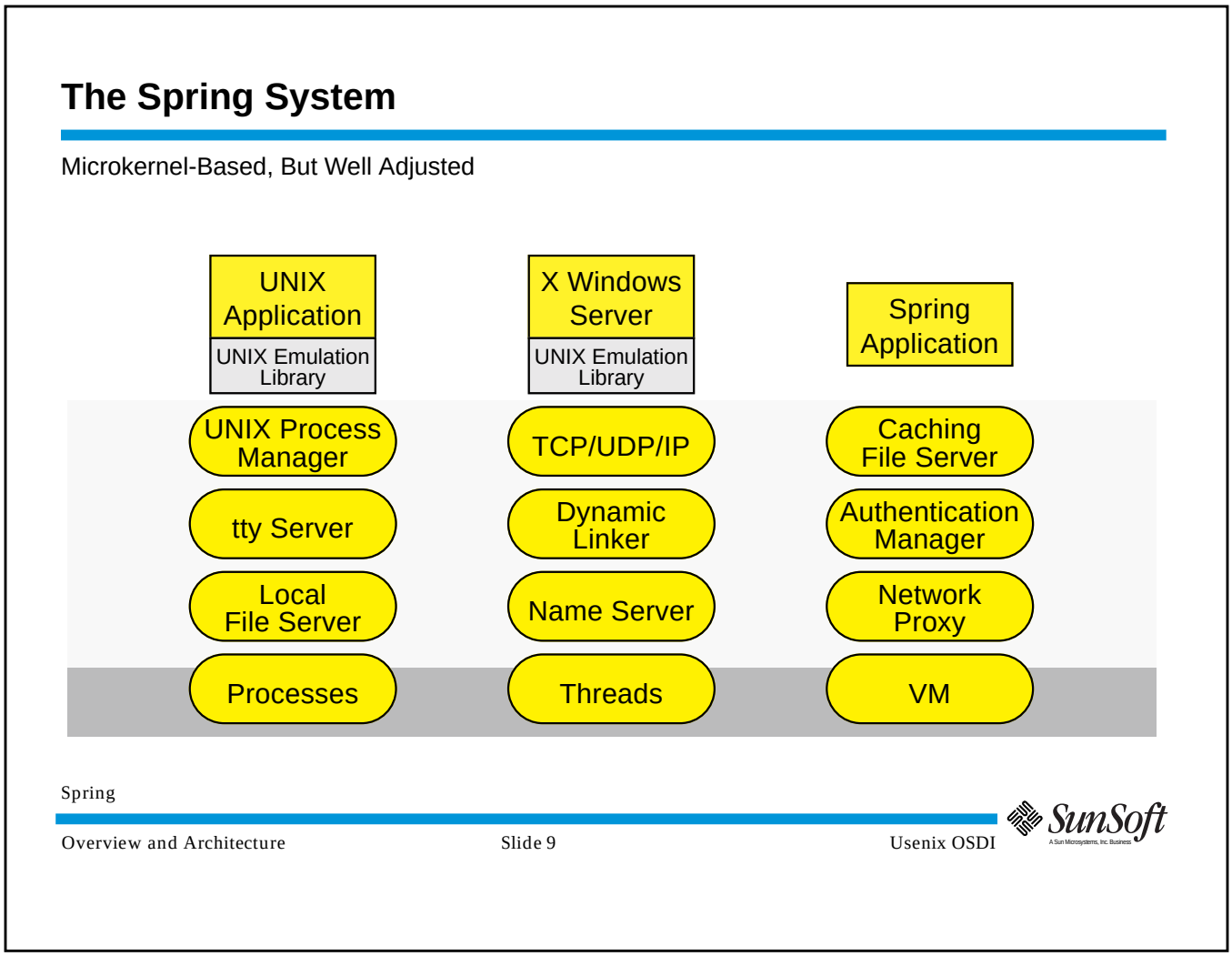
Overview and Architecture Slide 8 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

### Explanation:

One of the principles of the Spring design is that distributed programming isn't merely easy, but that there is no real distinction between distributed and non-distributed programming. In terms of the actions of the application programmer, there is no difference between accessing an object in one's address space, an object in another address on the same machine, or an object on another machine.

*Notes:*




## Explanation:

Like Chorus and Mach, Spring is built on top of a microkernel (the darker-shaded portion of the slide). Unlike Chorus and Mach, most of the effort in Spring has been devoted to components that do not run in kernel mode. The intent behind Spring is not to support binaries from existing operating systems through the use of “operating system personalities,” but to support the goals presented in the preceding slides.

However, it is certainly convenient to be able to use existing software. Spring provides a “UNIX Emulation Library” that makes it possible to compile and link much existing UNIX source code; except for a few special cases, UNIX binaries are supported. These UNIX applications are needed to help bootstrap the environment and to allow Spring systems to interact easily with non-Spring systems.

*Notes:*

## The Spring System

Clean Interfaces

Object User
- What I provide
- What I receive

Object Provider
- What I receive
- What I provide

Client

Server

Spring

Overview and Architecture Slide 10 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

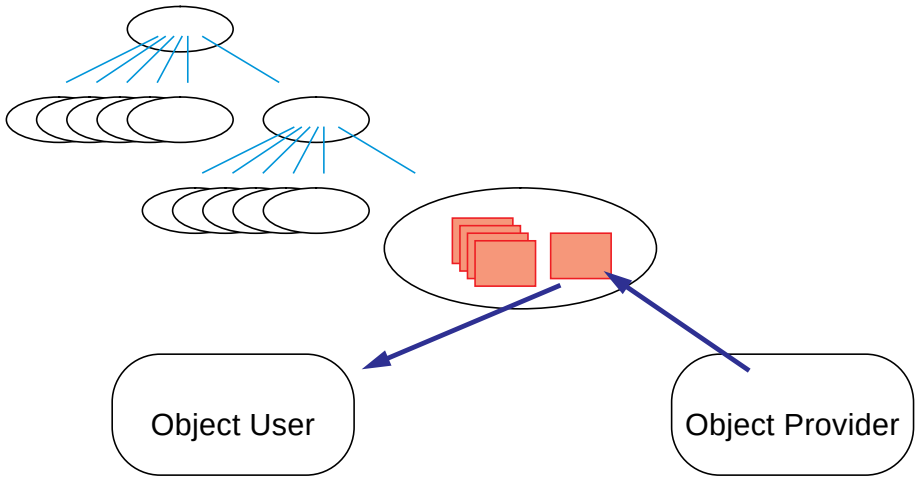
### Explanation:

One of the goals of Spring is that it be an open and highly modular system. This is made feasible by providing (and requiring) clean, tightly specified interfaces for each component. These interfaces can be thought of as a *contract* between client and server, a contract specified in *IDL*—the interface definition language.

*Notes:*

## The Spring System

Naming is Not Just for Files




Spring

Overview and Architecture Slide 11 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

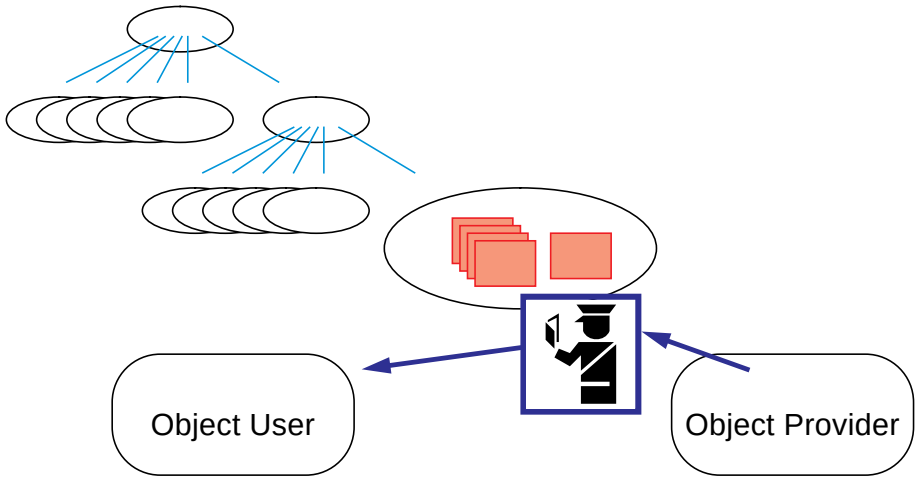
### Explanation:

A feature adding to the usability of Spring is its flexible approach to naming. Everything that can be manipulated in Spring is an object. References to any object can be placed into a name space for retrieval by potential users.

*Notes:*

## The Spring System

Secure




Spring

Overview and Architecture Slide 12 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

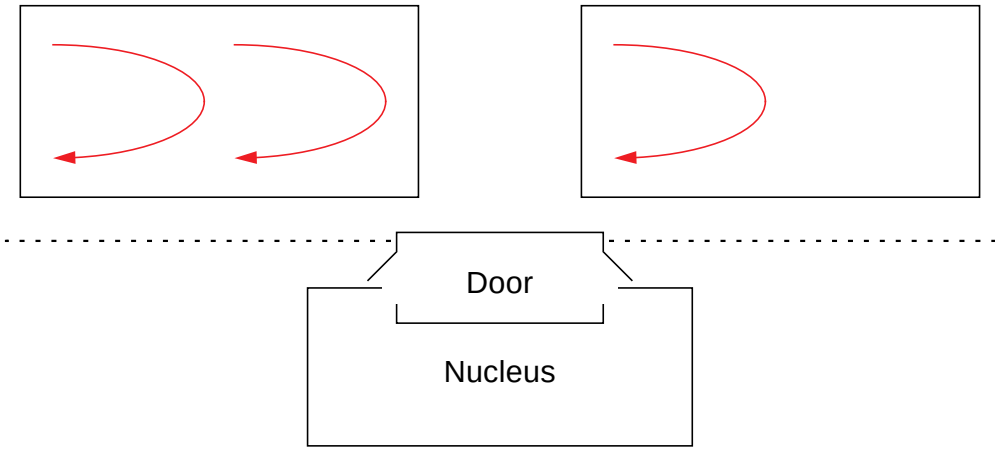
### Explanation:

Associated with naming is security. References to objects are inherently secure; authentication and authorization facilities provide a means for managing security.

*Notes:*

## The Spring System

Cross-Address-Space Invocation




Spring

Overview and Architecture Slide 13 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

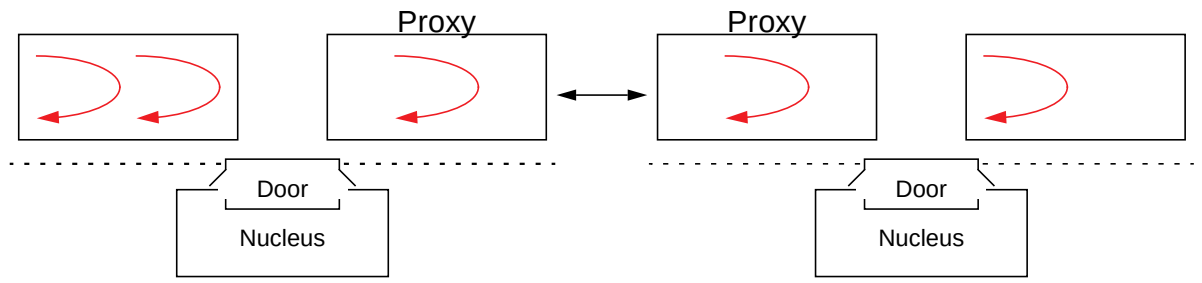
### Explanation:

Some fundamental notions in Spring are *processes*, *threads*, and *doors*. The first two are what one expects. Doors provide an efficient and secure means for cross-address-space invocation within a machine.

*Notes:*

## The Spring System

Network Proxies




Spring

Overview and Architecture Slide 14 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

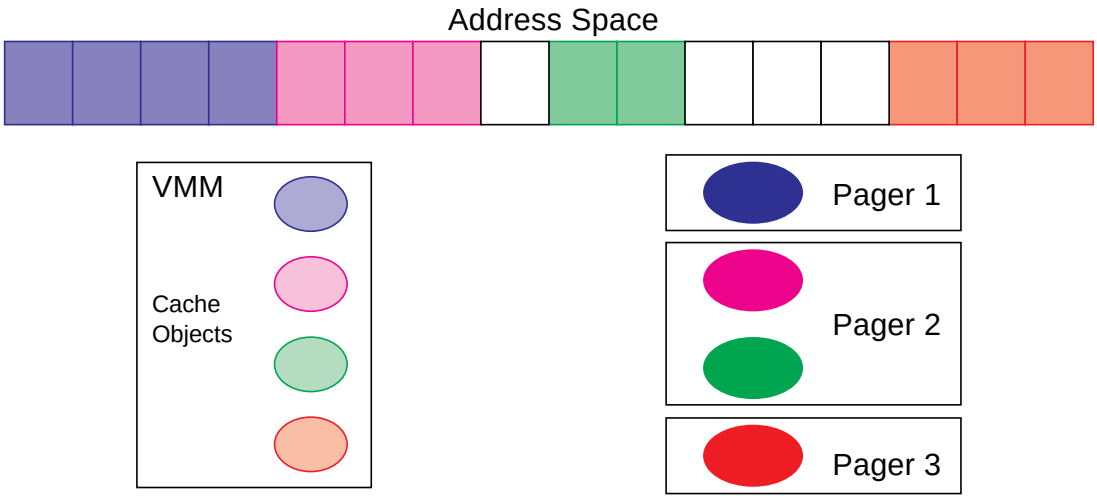
### Explanation:

Doors are a relatively simple mechanism provided in the nucleus. They are easily extended for cross-machine invocation with user-level *proxy* processes.

*Notes:*

## The Spring System

VM




Spring

Overview and Architecture Slide 15 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

### Explanation:

The virtual memory architecture of Spring is not totally dissimilar from that of Chorus and Mach: objects are mapped into an address space. These objects are provided by user-level managers called *pagers*. The virtual-memory manager in the kernel is strictly concerned about local virtual-memory issues. The management of the memory objects is the domain of the pagers.

*Notes:*

# *The Spring Object Model* 2

## The Spring Object Model

Outline

- Basics
- Name Servers
- Interfaces
- Inheritance
- Subcontracts



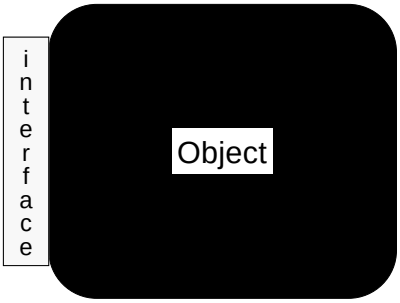
### Explanation:

In this module we introduce the Spring notion of objects and get you started programming with them.

*Notes:*

## Basics

It's the Abstraction That's Important




Spring

The Spring Object Model Slide 2 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

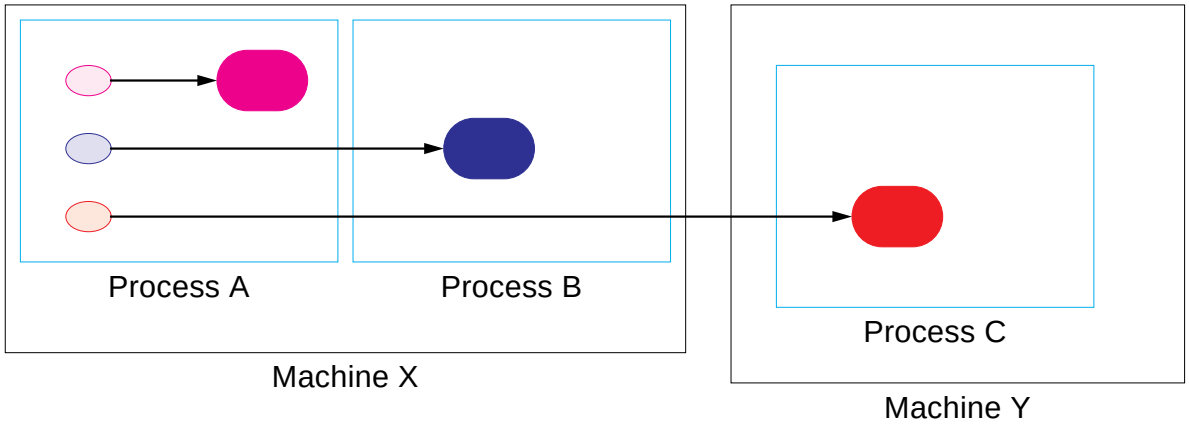
### Explanation:

Spring objects are fully *abstract*—all that a user of an object can find out about the object is its public interface; no information about its internals is made available to the outside world. This makes it possible for users of an object to be completely separated from the implementation of the object, an important concern for distributed computing. What's more, multiple implementations of an object can all share the same interface.

*Notes:*

## Basics

Location Transparency



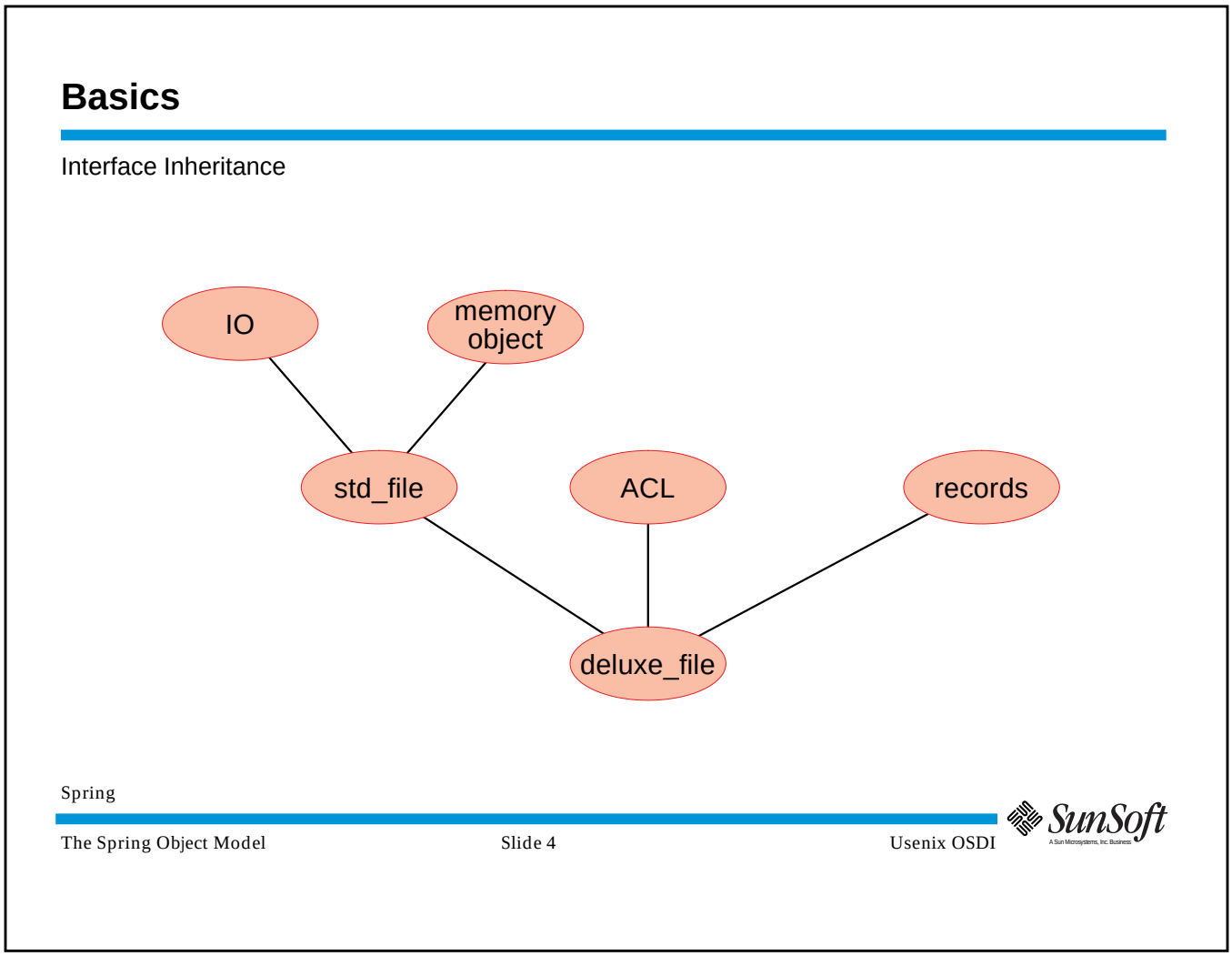



### Explanation:

An important aspect of our object model is that methods are referred to and invoked on objects in exactly the same way, no matter where the object is. In the picture, the small ovals are *object references* and the rectangles with rounded corners are objects. The user (or *client*) of an object uses the object reference as a means for invoking methods on the object. From the user's perspective, it should make no difference whether the object is in the same address space as the client, in a different address space but on the same machine as the client, or on a different machine from the client.

Clearly, there are differences in how the runtime handles the three situations shown in the picture. The invocation of a method on a local object can (and does) take advantage of optimizations that aren't available for remote objects.

*Notes:*




## Explanation:

Most approaches to object-oriented computing make use of what is known as *implementation inheritance*. For example, in C++, the declaration of a class can specify that the class inherits from other classes. The picture shows an inheritance hierarchy in which *deluxe_file* inherits from *std_file*, *ACL*, and *records*. *Std_file* inherits from *IO* and *memory object*. In C++, a declaration of the *deluxe_file* class would be based on the following:

```
class deluxe_file :
        public std_file, public ACL, public records
{
public:
    ...
private:
    ...
};
```

*Notes:*

Thus all of the methods of *std_file*, *ACL*, and *records* are available in objects of type *deluxe_file*. Moreover, the implementation of these methods are inherited in *deluxe_file*: the code for *deluxe_file* includes the code of the classes it inherits. This is a useful concept for C++. One can save a lot of time by the reuse of code that this notion of implementation inheritance makes easy.

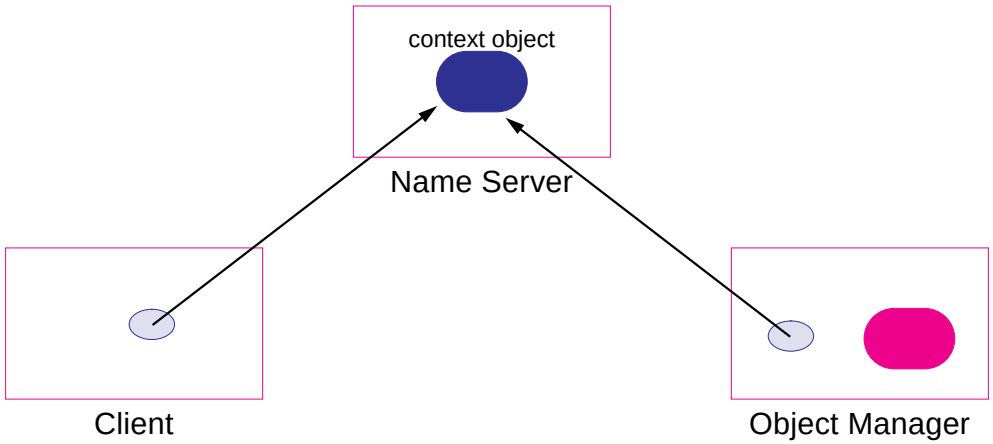
However, implementation inheritance is not ideal in all situations. In the example in the slide, *ACL* is a “class” that provides an interface for manipulating access-control lists. Thus, since *deluxe_file* inherits from *ACL*, *deluxe_file* objects can be treated as access control lists—one can invoke ACL-specific methods on files as if they were purely access-control lists. But, even though we have a standard interface for operating on ACLs, we do not want to require that all ACLs be implemented in the same way. Different file systems might use different implementations. ACLs might be used in a database whose implementation of ACLs might bear little resemblance to the ACL implementations of file systems.

The ability to specify interfaces that are common to a number of different classes is very important. But it would be a serious problem if we were then forced to use the same implementation for all uses of the interface.

Note that implementation inheritance is not ruled out: Spring’s notion of interface inheritance permits implementation inheritance but does not require it.

## Name Servers

Name Servers




Spring

The Spring Object Model Slide 5 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

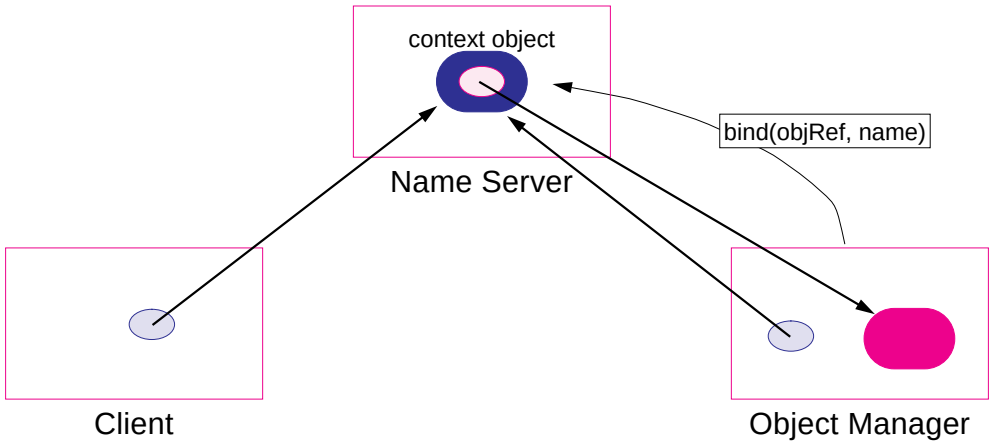
### Explanation:

A *Name Server* is an agent that maps names to object references. It provides *contexts*, which are analogous to the directories of file systems.

*Notes:*

## Name Servers

Binding





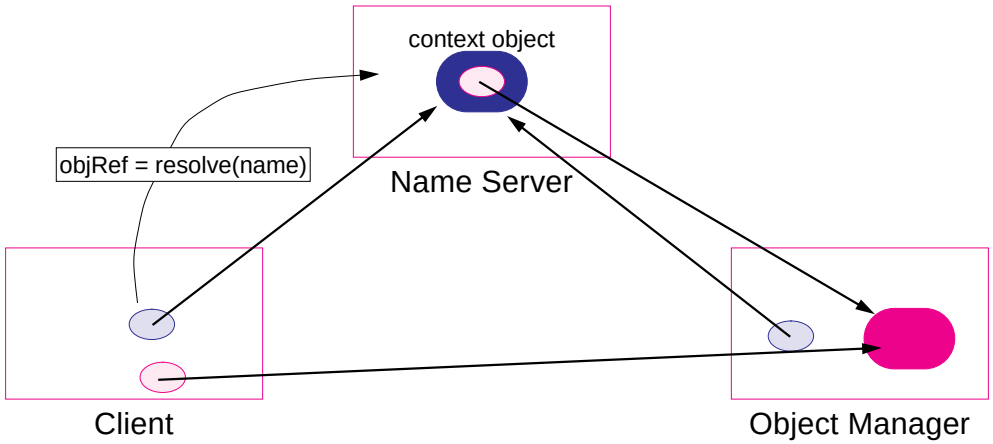

### Explanation:

Here the object manager *binds* a reference to its object to a name and puts this binding into the context object, by invoking the context object's *bind* method.

*Notes:*

## Name Servers

Resolving





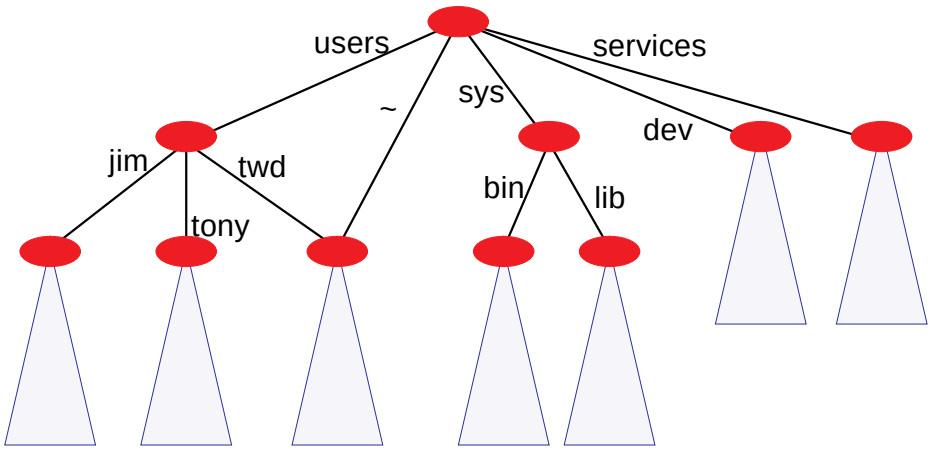

### Explanation:

The client now obtains a copy of the object reference by *resolving* the name from the context object (for which it has a reference).

*Notes:*

## Name Servers

Process Name Space




Spring

The Spring Object Model Slide 8 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

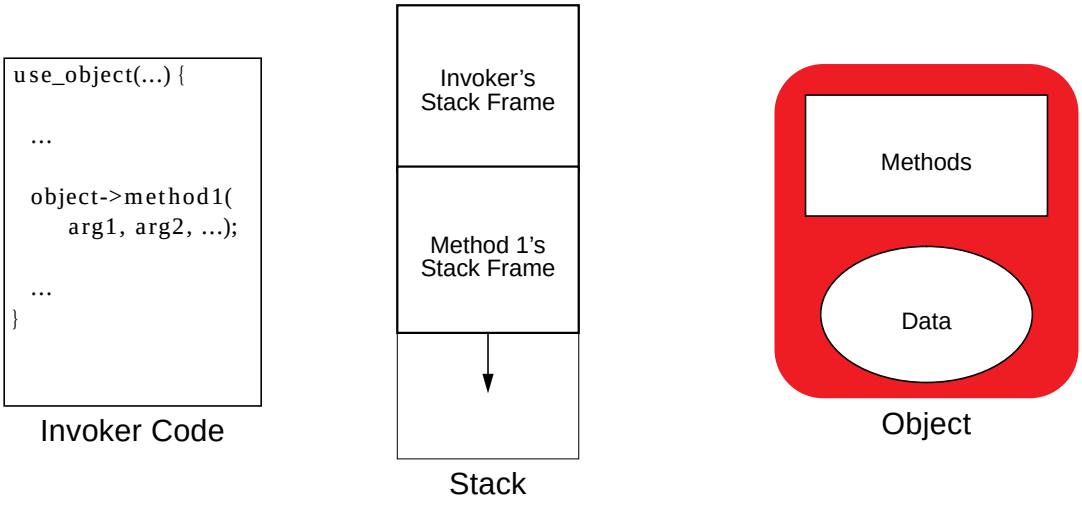
### Explanation:

A number of name servers provide the complete Spring name space, each managing a portion of it. Each process has its own name which consists of both private and global portions.

*Notes:*

## Interfaces

Local Objects




Spring

The Spring Object Model Slide 9 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

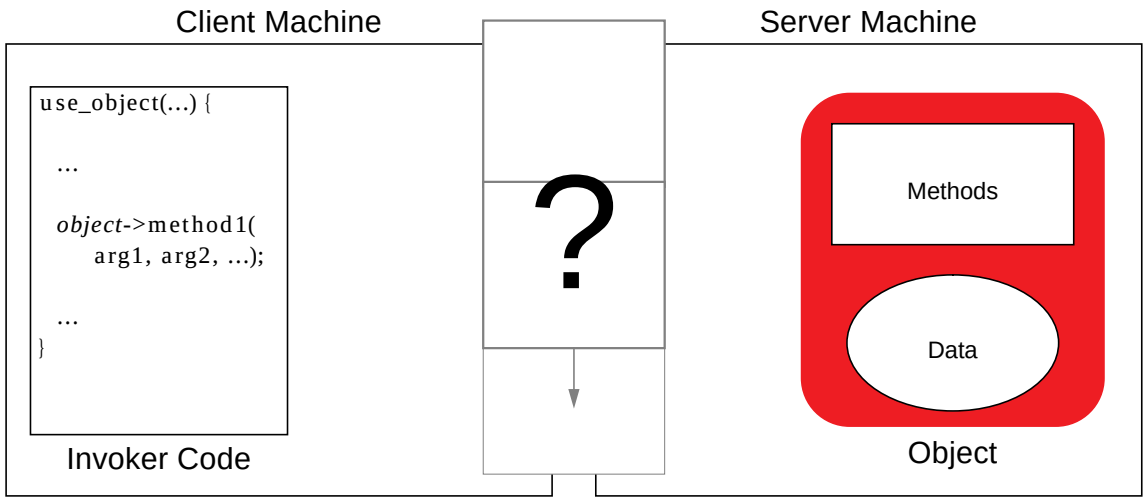
### Explanation:

Let's consider what happens when one invokes a method on a local object. Associated with the current thread is a stack. The caller has a stack frame in which it puts, among other things, the arguments for the method invocation. As part of the method invocation, a new stack frame is pushed on the stack. The code for the method reaches up into the caller's stack frame to find its arguments.

*Notes:*

## Interfaces

Remote Objects




Spring

The Spring Object Model Slide 10 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

### Explanation:

Now consider the case of invoking a method on a remote object. If we try to set up the stack as in the local case, we run into a problem: the invoker might put the arguments in its stack frame, but how does the remote code of the method access these arguments?

*Notes:*

## Interfaces

Concerns

- Separation of *interface* and *implementation*
  - *Contract* (*IDL* specification)
- Managing the interaction between invoker and object
  - Subcontract



### Explanation:

What's needed are some intermediaries, known as *stubs*, to handle the communication of control and data. The client-side stub presents an interface that looks like that of the actual object, i.e., the client-side stub appears to be the actual object. Instead, however, its methods contain code to contact the server. On the server is a server-side stub routine that receives requests from the client-side stub and converts these requests into an invocation of the actual object. On return from the object's method, control goes back to the server-side stub, which communicates the results back to the client-side stub, which finally returns the results back to the original invoker.

There are two issues we need to deal with:

- how do we produce the stubs?
- what do the stubs actually do?

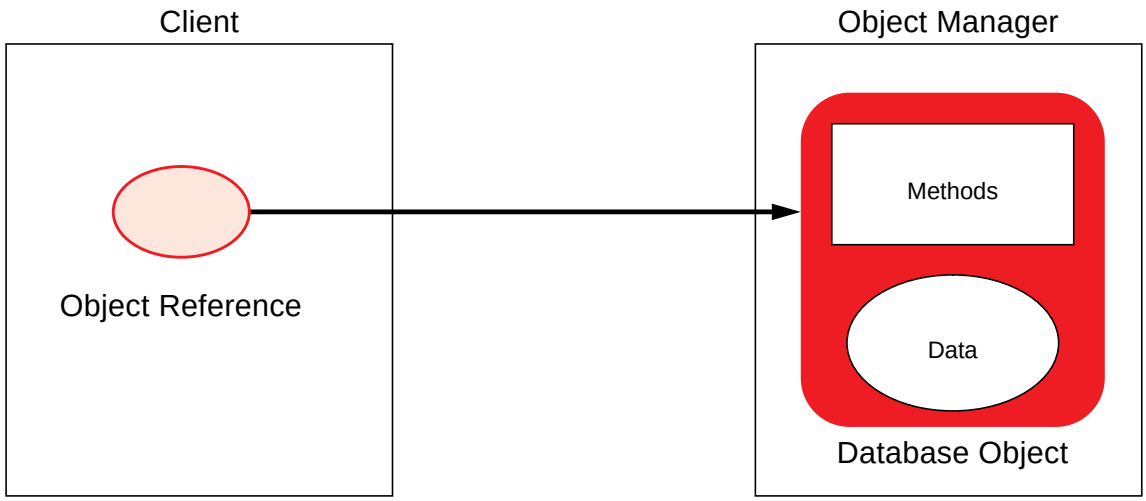
*Notes:*

The answer to the first question is that we make stub generation as automatic as possible. To produce a stub, all that is required is a specification of the interface provided by the object. This specification, or *contract*, is written in *IDL*—interface definition language. Given such a specification, the necessary stubs and associated declarations and definitions are produced by the IDL compiler. This approach has the additional benefit of keeping separate the description of an object's interface from the object's implementation. Thus the implementation can change without requiring any changes in the clients of the object.

The other issue is what is going on inside the stub. Here too we split the problem into two pieces. What is actually generated by the IDL compiler is code that makes use of another layer, the *subcontract*, to do the actual work. The subcontract handles communication, the marshaling and unmarshaling of objects, and anything else that might be required to handle the interaction between client and server. A number of subcontracts are supplied with Spring; they all provide the same interface to the IDL-compiler-generated code, so substitution is easy.

## Interfaces

An Example






### Explanation:

As an example, we look at an object manager that handles a simple database. The client has a reference to this object, as shown in the slide. In the framework of Spring objects this reference is (at least conceptually) a pointer. However, when we map this onto a programming language such as C++, it is something a bit more complicated.

*Notes:*

## Interfaces

Interface vs. Declaration

```
// Contract (in IDL)

interface NVcollection {
  string query(copy string name);
  boolean add(copy string name,
      copy string value);
  boolean remove(copy string name);
}
```

```
// Declaration (in C++)

class NVcollection {
public:
  string query(string name);
  bool add(string name, string value);
  bool remove(string name);

  ...
private:
  // this is really none of your business
  type1 func1(type2 arg1, type3 arg2);
  type4 func2(type5 arg1);
  ...
  type146 root;
}
```

Spring

The Spring Object Model Slide 13 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

### Explanation:

A Spring object presents an implementation as specified in IDL. Here we give the *NVcollection* interface, which supplies three methods. Note that, besides identifying the types of the parameters to the methods, we also specify the parameter-passing *modes*. We discuss these in more detail later, but here we must supply instructions for how the parameters should be transferred. Here we are using *copy*, which specifies that a copy of parameter is to be passed from the client to the server. The implicit mode for the return value of these methods is *produce*, which means that the return value is transferred (without copying) from the server back to the client.

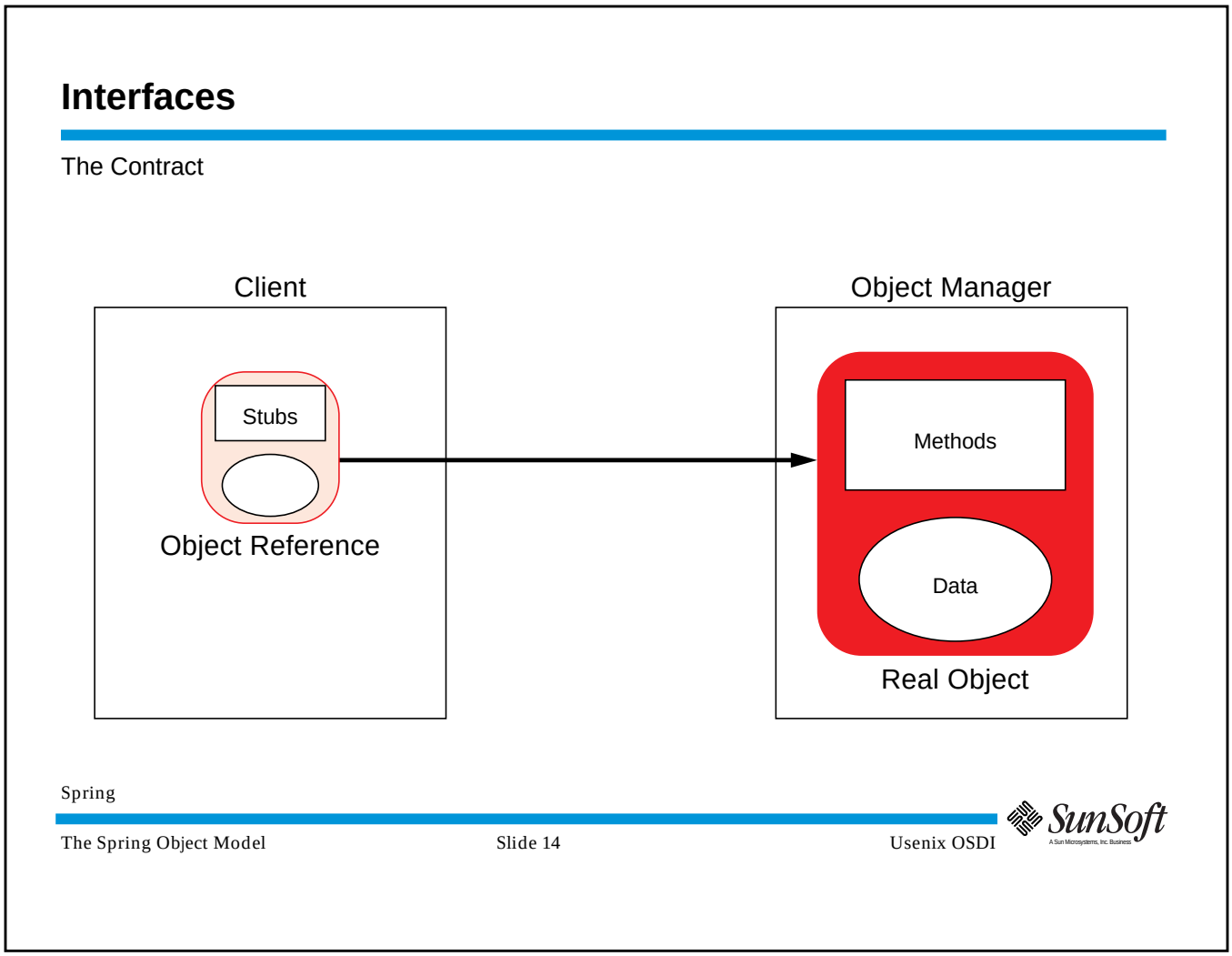
We must, of course, represent our Spring objects in terms of some programming language. There is no architecturally preferred language; in principle any programming language could be used, whether object-oriented or not. In practice

*Notes:*

what has been used is strictly C++. Though it is object-oriented, the C++ notion of an object is not the same as that of Spring. We therefore must make use of various conventions for representing Spring objects and object references in C++.

The slide shows part of the representation of a Spring object in C++. The important thing to note here is that the declaration of the object in IDL makes no mention of any internal details—such information is not just hidden, it's not there at all. The C++ version of the declaration (which is produced with the aid of the IDL compiler) must provide full detail. This detail is essential on the server but irrelevant on the client, which only needs the level of detail provided by the IDL specification.

This separation of interface from implementation, even to the extent that the implementation is described in a C++ declaration, is of crucial importance in Spring. It allows the complete separation of client from server; any change whatsoever can be made to the implementation of an object, as long as the implementation remains faithful to the specification of its interface.




## Explanation:

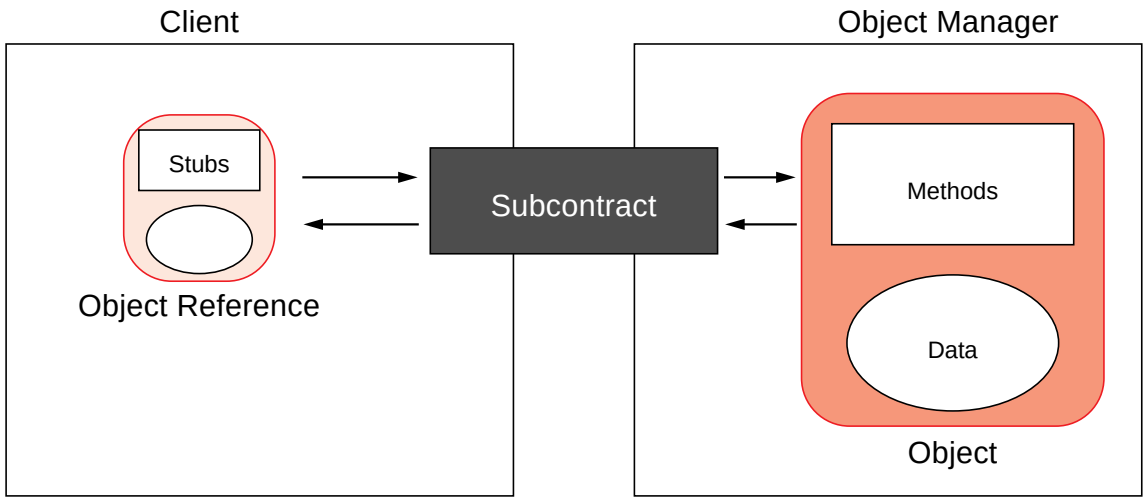
A reference to a remote object can't be simply a pointer to the object—the client and object are in different address spaces. What must happen when a client invokes a method on a remote object via the object reference is that client-side stub code is invoked, which in turn marshals (or causes to be marshaled) the outgoing data, places the call, receives the response, unmarshals the incoming data, and finally returns to the caller.

Thus, while in concept an object reference is merely a pointer, in practice it is a C++ object itself (or something equivalent if another programming language is being used) that provides public methods corresponding to the methods of the Spring object, but whose implementation forms the stub.

*Notes:*

## Interfaces

The Subcontract




Spring

The Spring Object Model Slide 15 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

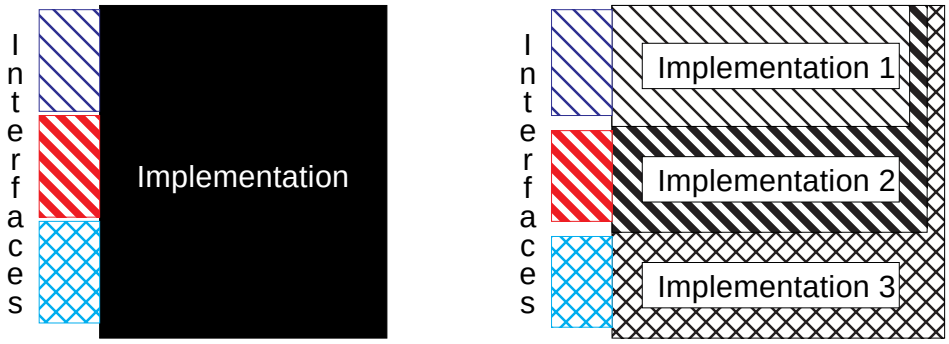
### Explanation:

The subcontract, i.e., the code that handles the communication with the object manager and that marshals object references is a layer in itself. We discuss subcontracts in more detail in later modules; however, any issues involving efficiency of data transfer, replication, caching, etc., are dealt with at this layer and are independent of the specification and implementation of the object and object reference.

*Notes:*

## Inheritance

Interface Inheritance vs. Implementation Inheritance






### Explanation:

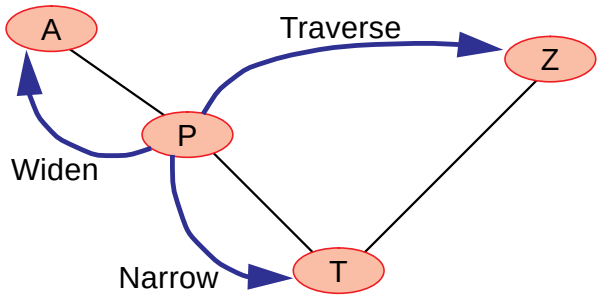
With the usual notion of implementation inheritance, the implementation of a subclass contains the implementations of the base classes. With interface inheritance, all that is required is that the subclass provide an implementation for all of its interfaces. For example, the *context* class requires an interface that includes *bind* and *resolve*. The *fs_context* class, which is derived from *context*, adds a new method, *create_file* (*fs_context* is used in file systems for directories). If we have a routine that expects to operate on a reference to a *context*, we can safely give it a reference to an *fs_context*, even though the implementations of the two have nothing in common other than the *context* interface. Thus, stating that *fs_context* inherits from *context* means not only that objects of type *fs_context* respond to all the methods to which objects of type *context* respond, but also that objects of type *fs_context* can be used wherever objects of type *context* are used.

*Notes:*

We say that an instance of a subclass can be *widened* to be an instance of an ancestor class. Turning this around, an object that purports to be an instance of a class might also be an instance of a subclass. If it is, we can *narrow* the object so as to treat it as a member of the subclass.

## Inheritance

Fun With Inheritance




Spring

The Spring Object Model Slide 17 Usenix OSDI

SunSoft

### Explanation:

This is a pictorial depiction of narrowing and widening: we have been given a reference to an object whose type we *perceive* to be *P*. Since the object at least can be treated as being of type *P*, and we know that *P* is derived from *A*, we can always *widen* the object to *A*. This requires no runtime type checking, since *P* is a subclass of *A*.

We may have received the reference to the object of type *P* as the result of an invocation of a method that was constrained to return a reference to *P* to us. However, the sender of the object may have had to widen an object of type *T*, derived from *P*, in order to meet the constraints. Thus we *perceive* the object to be of type *P* though its *true type* is *T*. In this case we can safely *narrow* our object so as to treat it as type *T*. Whether or not we can narrow an object depends upon information known only at runtime—some form of dynamic type checking is required.

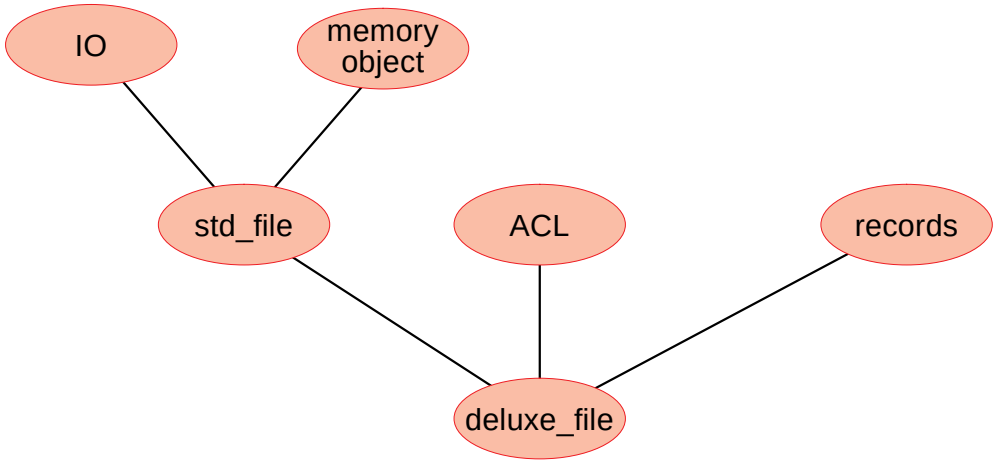
*Notes:*

Taking this a step further, if our object is really of type *T*, then (according to the picture), since *T* is also derived from *Z*, we should be able to treat our object as of type *Z* as well. This sort of transformation is called a *traverse*, and clearly also requires the use of dynamic type checking.

We can perform widen, narrow, and traverse operations, where meaningful, on any Spring object. If dynamic type checking determines that the operation cannot be done, an exception is thrown. Since the operations might not be supported by the underlying programming language (narrowing and traversing are certainly not supported by C++), they are handled by the runtime support of Spring.

## Inheritance

Example




Spring

The Spring Object Model Slide 18 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

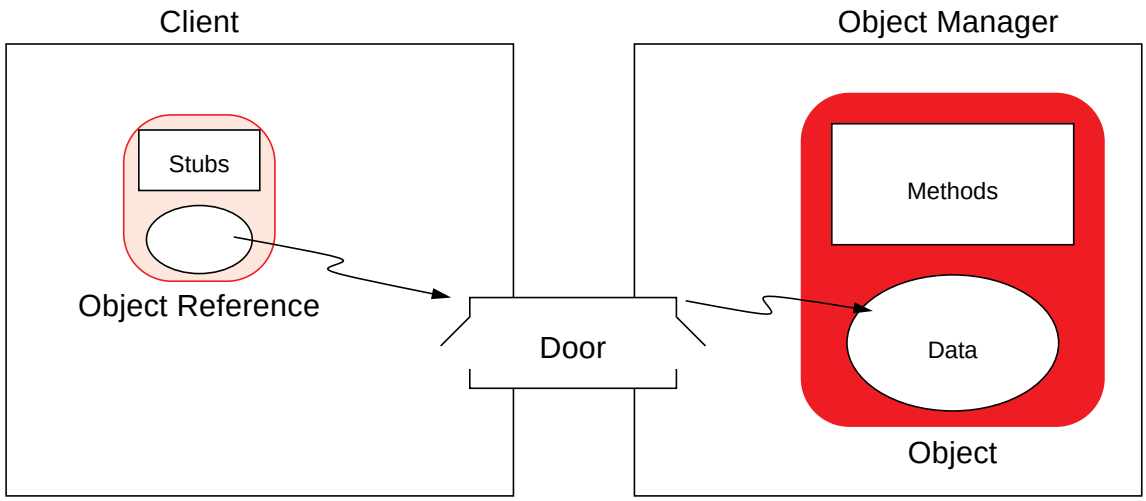
### Explanation:

Here is an example of the use of narrowing, widening, and traversing. Suppose that the usual sort of file supported by our system is of type *std_file*, which is derived from *IO* and *memory_object*. Thus, via widening, whenever we have a file we can pass it to any method that expects objects of type *IO* or *memory_object*. Suppose that we have decided to support, in addition to *std_file*, a more sophisticated sort of file of type *deluxe_file*. This new type is derived from not only *std_file*, but also *ACL* and records. We might call some routine to obtain a reference to a file for us. This routine was written before *deluxe_file* was invented, so all it knows about is *std_file*. However, on obtaining the file, we can traverse the reference to be an *ACL* so that we can perform ACL operations on it, and then we can traverse the reference to be of type *records* so that we can perform record operations on it. Without such dynamic type manipulation, if the object we are given is said to be of type *std_file*, we could not safely coerce it to be a *deluxe_file*, an *ACL*, or a *records*.

*Notes:*

## Subcontracts

The *Singleton* Subcontract




Spring

The Spring Object Model Slide 19 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

### Explanation:

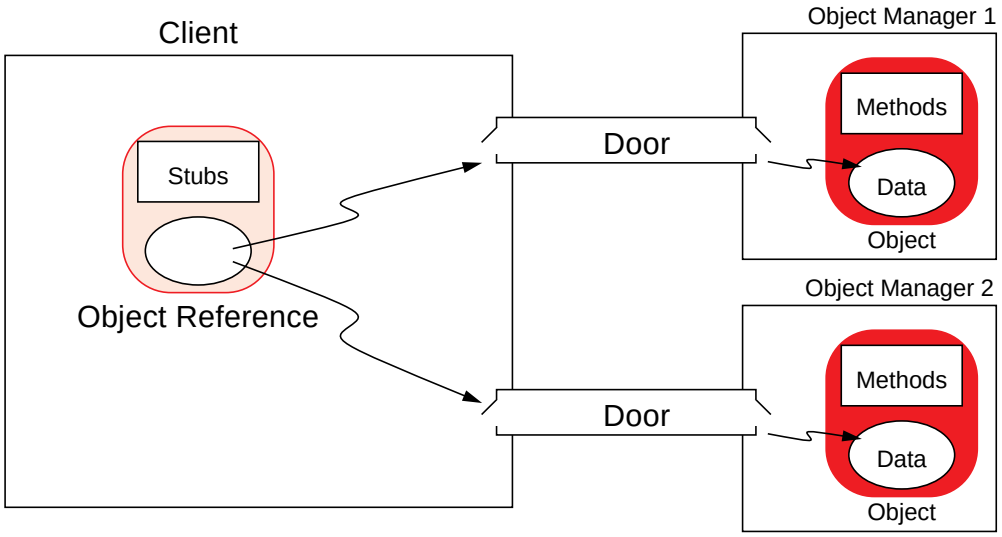
Subcontracts define how object references really refer to objects. The standard subcontract, illustrated in the slide, is the *singleton* subcontract. It makes use of a kernel-supported construct, the *door*, as it means for referring to remote objects. Doors, which we discuss in more detail later, provide a secure means for referencing objects; they are, in effect, *capabilities*. For each door it has access to, a client has a nonforgeable door identifier. Methods on the remote object are invoked by placing a call *through* the door, as discussed starting on page 118. If an object reference must be marshaled so that it can be passed as a parameter, the marshaled form is the door identifier.

For references to local objects, i.e., objects in the same process (address space) as the object reference, the singleton subcontract simply uses pointers and places direct calls.

*Notes:*

## Subcontracts

The *Replicon* Subcontract





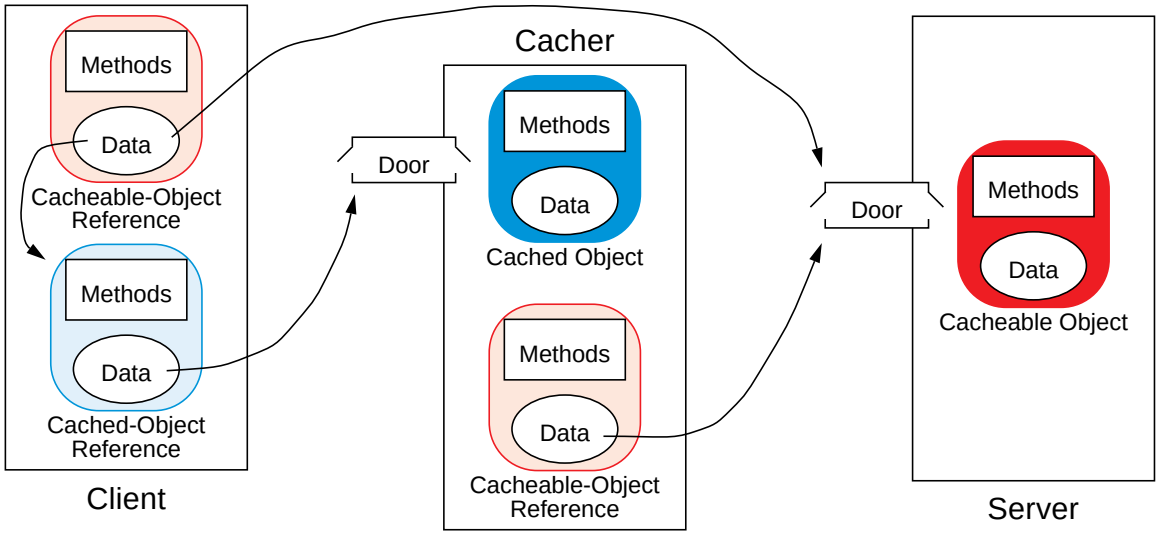

### Explanation:

An alternative subcontract is *replicon*. While singleton communicates with a single object manager using a single door, replicon communicates with multiple but identical object managers using multiple doors. Note that whether singleton or replicon is used independent of the IDL description of the interface.

*Notes:*

## Subcontracts

The Cacher Subcontract




Spring

The Spring Object Model Slide 21 Usenix OSDI

SunSoft A Sun Microsystems, Inc. Business

### Explanation:

Our final example of a subcontract is *cacher*. This is more complicated than the others; it provides a cache between the object reference (or collection of object references) and the object. When an object that employs the cacher subcontract is unmarshaled, it is found to refer (via a name in the name space) to a “cacher” that probably is managed in a separate process on the client’s machine. The reference, on the client, to the original object is set to refer to a reference, still on the client, to a cached object, which is in the cacher process. Thus the cached object is interposed between the client and the cachable object and provides the caching functionality.

Starting on page 170 we discuss how this cacher subcontract is used to provide caching for file systems.

*Notes:*

# *The Name Service* 3

## The Name Service

Outline

- The Naming Model
- Security
- Persistence



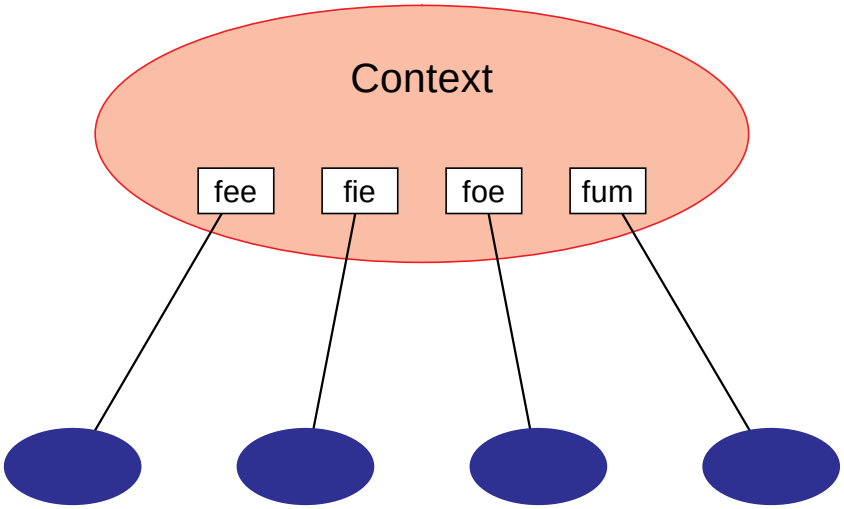
### Explanation:

In this module we describe naming in Spring. We start by discussing the basic model. Naming and security are intimately related in Spring, so we go over security next. Finally, we discuss how persistence is handled in Spring.

*Notes:*

## The Naming Model

Naming Contexts




Spring

The Name Service Slide 2 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

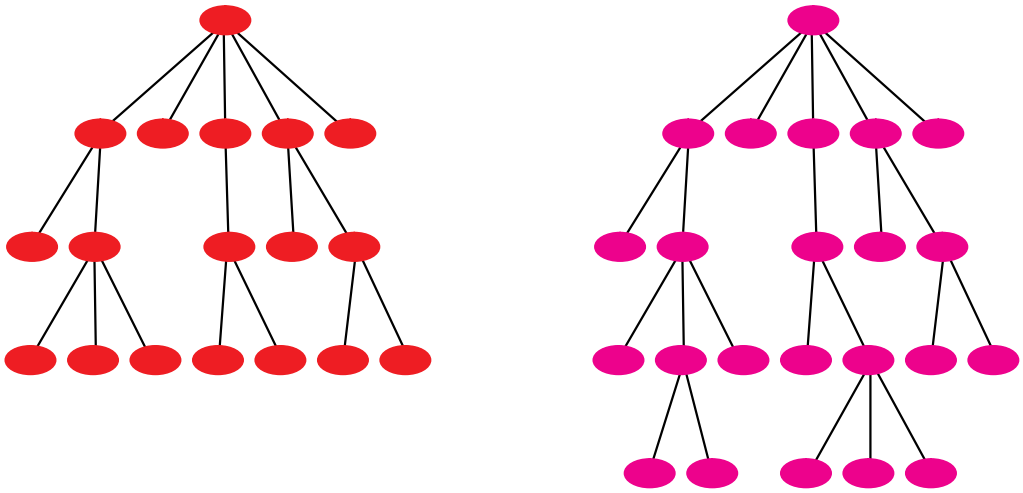
### Explanation:

Spring provides an interface known as *context* for associating names with objects. A context is what in a file system is called a directory—it provides a scope for names. Two of the methods provided by the *context* interface are *bind* and *resolve*: the former is used to associate an object reference with a name; the latter is used to get the object reference associated with a name.

*Notes:*

## The Naming Model

Name Spaces



Spring

The Name Service Slide 3 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

### Explanation:

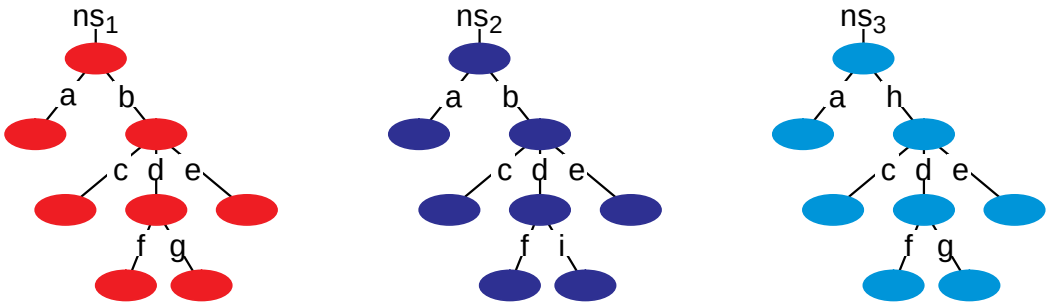
As one certainly expects, one context (an object itself) can be named in another context, i.e., a reference to one context is bound to a name in the other context. If we distinguish some context as being the *root*, we can put together a file-system-like naming tree. The fully qualified name of an object reference is then the usual concatenation of the names encountered when following a path to the object reference from the root.

However, unlike the case for most file systems, it is not necessary for a context to have a name that begins with the root. Contexts are certainly objects in Spring; they are “first-class objects” (there is no business- or coach-class in Spring): references to them can be passed around and manipulated just like references to any other object. Thus one can set up a private name space and pass it around, and one can link together otherwise unrelated name spaces by binding a reference to the root of one into a context of the other.

*Notes:*

## The Naming Model

Ordered Merges (1)




Spring

The Name Service Slide 4 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

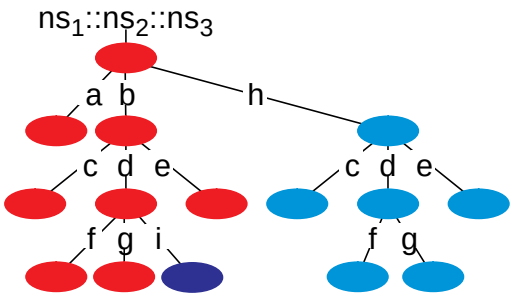
### Explanation:

In a number of situations (such as, for example, setting up a search tree for finding commands) it is convenient to merge together a number of name spaces in a orderly fashion. The slide shows three disjoint name spaces that have naming conflicts, i.e., they share certain path names. We would like to combine these name spaces so that when we look up a name in the combined name space, it has the effect of first looking up the name in $ns_1$, then, if not found, looking it up in $ns_2$, and, if still not found, looking it up in $ns_3$.

*Notes:*

## The Naming Model

Ordered Merges (2)




Spring

The Name Service Slide 5 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

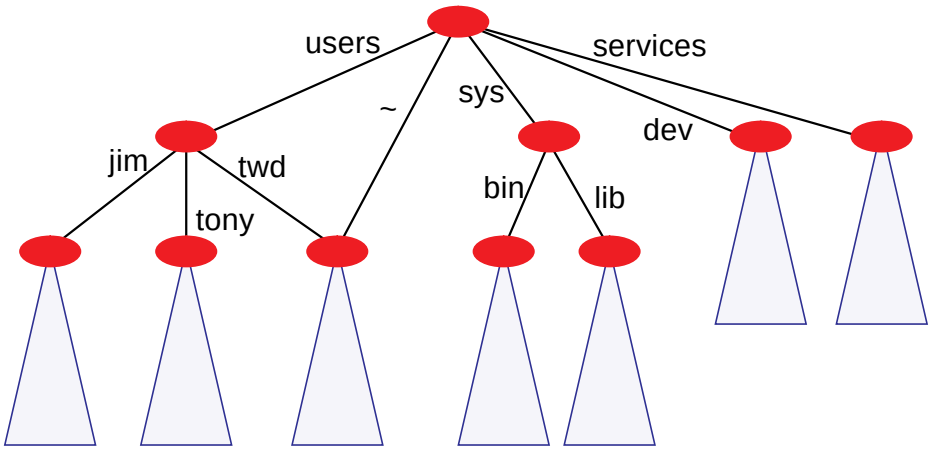
### Explanation:

This operation is known as an *ordered merge* in Spring. The result of an ordered merge of the three name spaces is a new context object that behaves as the root of a new name space with the semantics discussed with the previous slide.

*Notes:*

## The Naming Model

Process Name Space




Spring

The Name Service Slide 6 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

### Explanation:

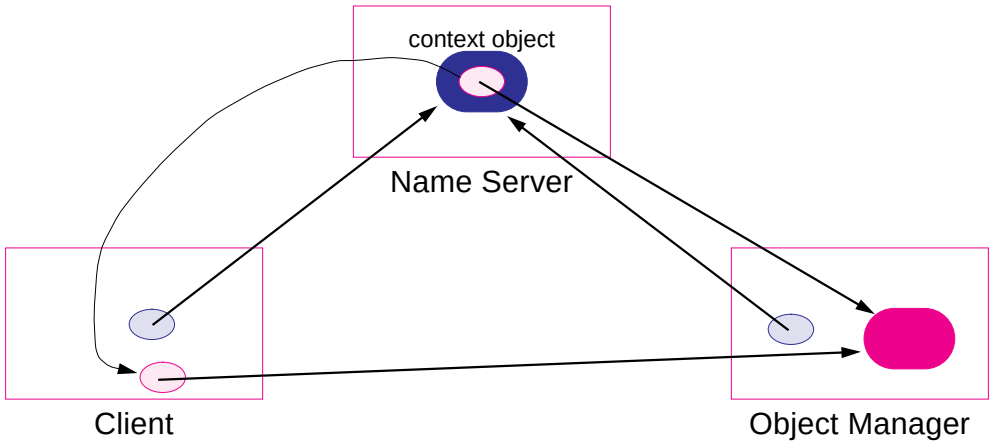
Each Spring process is given its own name space, which is the result of a collection of ordered merges. Portions of this name space are private, other portions are shared with others.

This notion of process name spaces, combined with the generality of naming in Spring, provides a uniform treatment for all forms of naming. For example, UNIX's nicely integrated file system can name almost everything but not everything everything. Each process has an "environment" (consisting of environment variables and values) which is set up and accessed differently from how files are set up and accessed. Simple databases such as "terminfo" and "printcap" are represented as files, but effectively define separate name spaces that are dealt with differently from the file-system name space. In Spring, the environment notion and the simple databases can be easily represented as objects collected into contexts. The resulting name subspaces can be either private to a process (such as for the environment) or shared with others.

*Notes:*

## Security

Capabilities




Spring

The Name Service Slide 7 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

### Explanation:

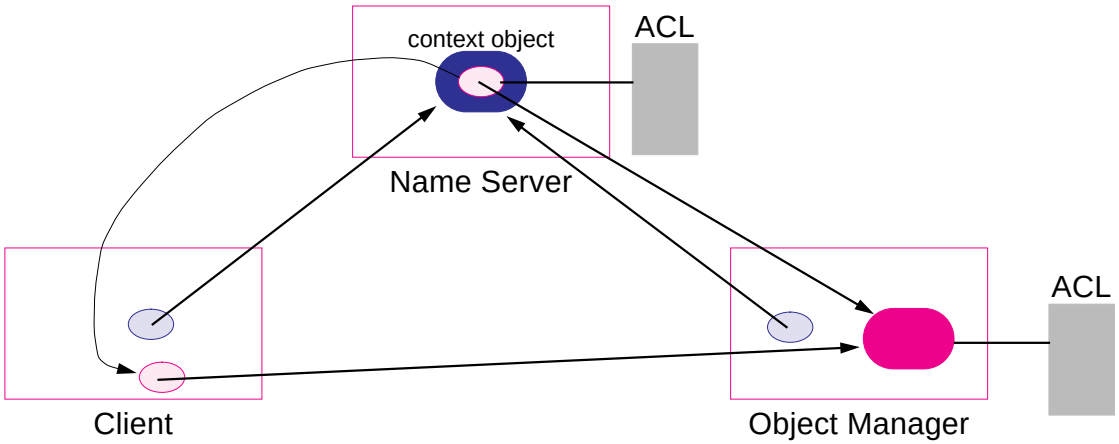
An object reference is a *capability*—if one posseses an object reference, then one has full access to the object. Thus when you pass an object reference to another process, that process gets full access to the object.

At the least, this means that one should be very careful to whom one passes object references. Thus when one binds an object to a name within a context object managed by some name server, one is either offering complete access to the object to the world, or trusting the name server to be very choosy about whom it allows to resolve the name.

*Notes:*

## Security

Access Controls




Spring

The Name Service Slide 8 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

### Explanation:

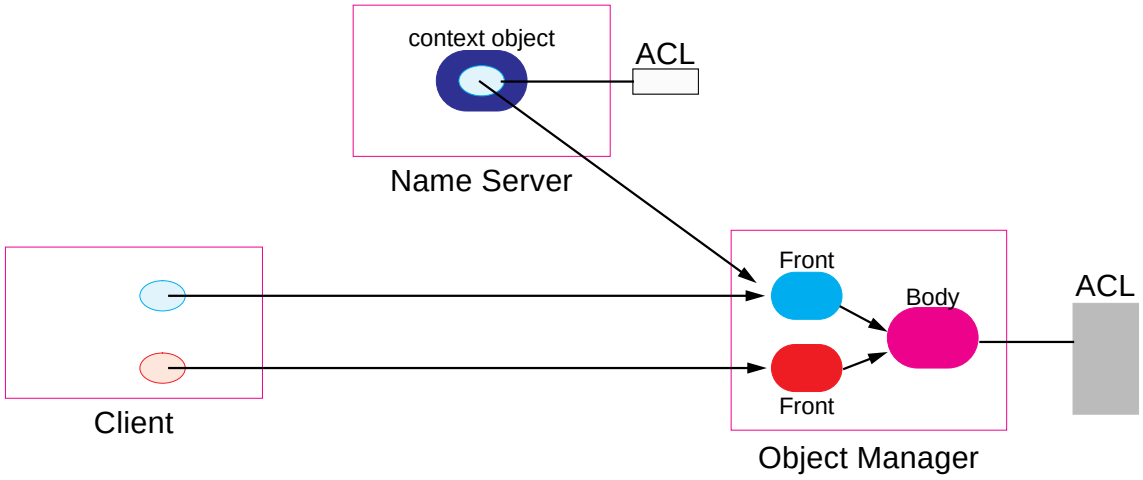
A way to control access to an object is to use *access control lists* (*ACL*s) to indicate who is allowed access to an object. For example, ACLs can be attached to context objects to control who is allowed to invoke the bind method and who is allowed to invoke the resolve method. An ACL can also be attached to the bindings themselves (i.e., to the name / object-reference combinations) to control who is allowed to resolve a particular name and how they are allowed to use the resulting object reference.

Note that there is a problem here. Since object references are capabilities, how can one restrict the use of a capability? For example, if a name server gives a process a capability for a file object, and if capabilities provide full access to the referred-to object, how can one restrict this capability so that it can only be used for read access?

*Notes:*

## Security

Fronts




Spring

The Name Service Slide 9 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

### Explanation:

An approach currently implemented in Spring (it's not necessarily the only possible approach) is for an object manager to bind, not a reference to the "real" object (now called the *body*), but a reference to a *front* object. The binding has an ACL associated with it that grants "default" rights to the front/body, which are typically no rights at all.

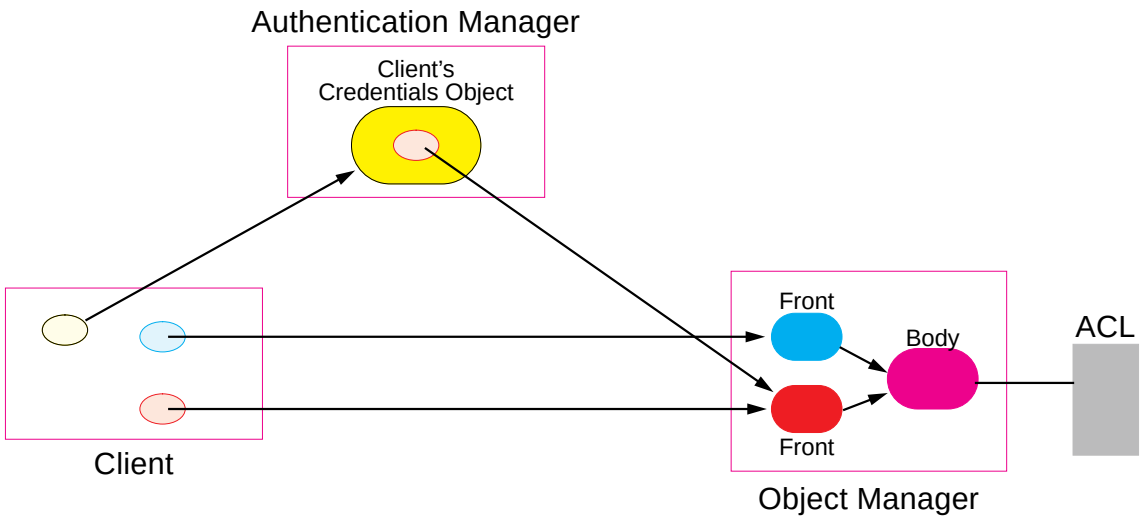
For a client actually to access the body object, it must have its reference to the front object "upgraded" so as to allow the desired object. The upgrade results in a new front object (and object reference) through which the client invokes methods on the body. The new front object filters the invocations of the client, only allowing the client to use methods for which the ACL (on the server) allows.

But we now have a new problem: how does the front object determine who the client is?

*Notes:*

## Security

Authentication




Spring

The Name Service Slide 10 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

### Explanation:

Users logging into Spring must authenticate themselves by supplying a password. This authentication interaction is with an *authentication manager* which manages *credentials objects*. The result of the initial interaction with the authentication manager is that you gets a reference to your own credentials object.

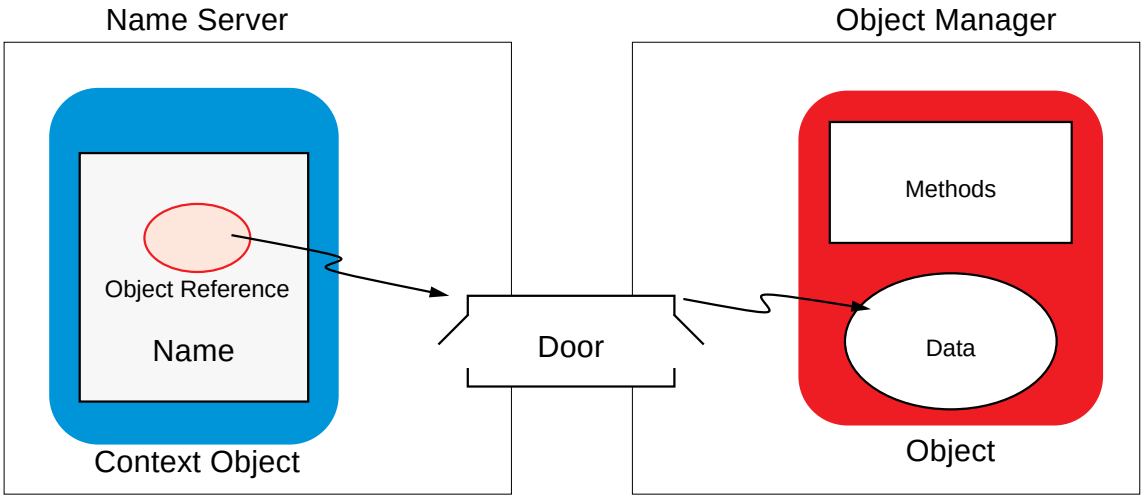
When you wish to use an object manager that insists on proof of your identity, you request via an *authentication interface* that your reference to the front object be upgraded. You supply your identity (without proof) to the object manager, which then, if the ACL permits it, binds a reference to the new front object to a name in your credentials object (which is behaving, for this purpose, as a context) and then returns the name to you. You can resolve this name and obtain the reference to the new, upgraded front object only if you have a reference to your credentials object, which will (or should) be the case only if you are who you claim to be.

*Notes:*

Two-way authentication is possible using a variation of this scheme, but is not currently implemented.

## Persistence

Implementing *Bind*




Spring

The Name Service Slide 11 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

### Explanation:

Our next issue is *persistence*. We take persistence to mean the quality of existing for an indefinitely long period of time. In particular, persistent things survive crashes: a file is (or should be) persistent. An address space typically is not. Spring has three items of interest that one might want to be persistent:

1. An object reference
2. A binding (i.e., a name / object-reference combination)
3. An object

That an object reference is persistent means, for example, that if the object manager handling the referred-to object crashes and restarts, the object reference will work with the new incarnation of the object just as it did with the old incarnation. Such a facility might be provided for special cases, but, for a number of reasons (e.g. performance), no general facility is provided for this sort of persistence.

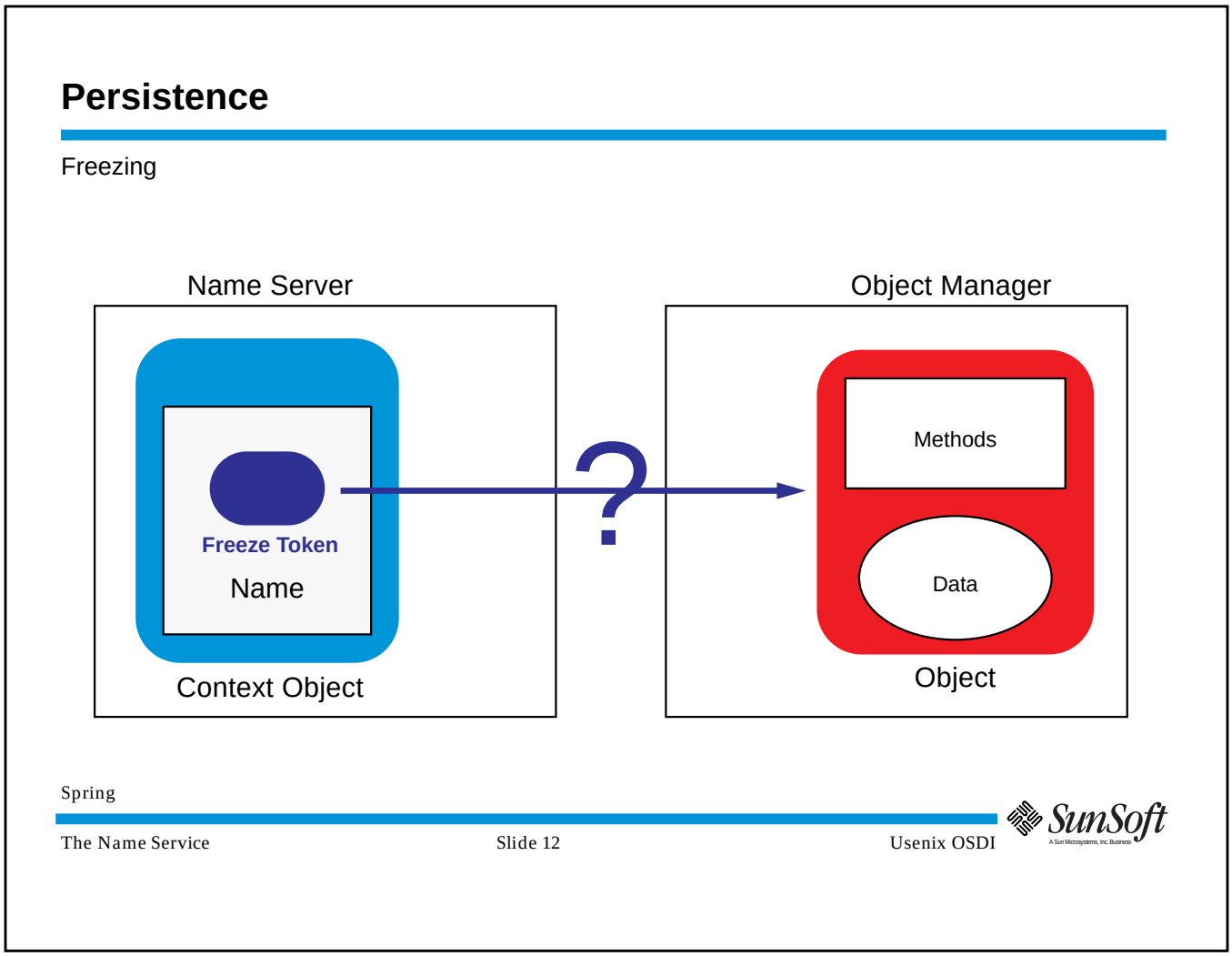
*Notes:*

Persistence of bindings and of objects, however, is extremely important. Imagine a file system that does not have the persistence property. Persistence in file systems is taken for granted—it is well understood how to obtain persistence in such systems.

How to provide persistence of bindings and of objects in the general case is not quite so obvious. What is done in Spring is to provide a general framework for persistence, but also to require the object implementor to help out.

The slide shows an object reference and an object using the singleton subcontract. The object is actively maintained by the object manager; it resides in the object manager's address space. If the object manager (or its computer) crashes, the contents of this address space are probably lost. For the object to be persistent, its state must be saved elsewhere, so that it can be recreated after a crash.

Similarly, the object reference is not persistent—its validity depends on the continuing existence of the door. If the server end of the door goes away, then so does the object reference.




## Explanation:

The term for converting an object reference into something that persists is *freezing*. One freezes an object reference (usually from a binding) and produces a *freeze token*. What exactly does a freeze token consist of? It must be something that can be converted back into an object reference, i.e., *melted*. Consider the analog of an object reference in the UNIX file system: the file descriptor. Like object references, file descriptors are not persistent. The directory entry contains the analog of a freeze token: an inode number. Given the inode number and the cooperation of the kernel and the file system, one can reopen the file and obtain a new file descriptor.

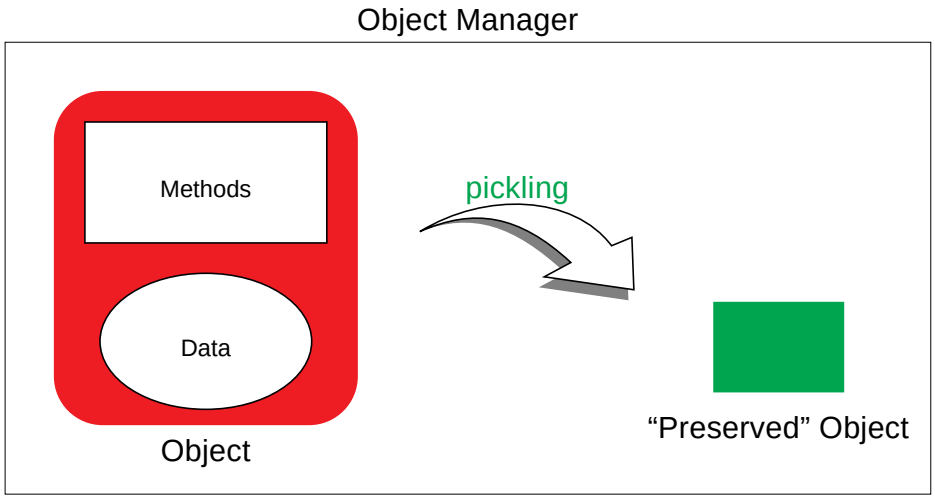
So, freezing requires the cooperation of the object manager. Something is produced that, when presented to the kernel, can be melted to recreate the object reference. Furthermore, the freeze token continues to have meaning, even if the object manager or name server crashes and is restarted.

*Notes:*

Note that an object reference may have a door associated with it; thus we can rely on the kernel to guarantee that the reference is secure. But a freeze token is merely a piece of data. Any desired security must be explicitly provided for.

## Persistence

Persistent Objects






### Explanation:

We are also interested in the persistence of objects. The term for converting an object into a persistent form is *pickling* (presumably because one is "preserving" the object). For a concrete example, we return to the UNIX file system. The analog of the object is the incore data structures representing an open file, including the contents of any buffers holding pieces of the file. The pickled form of the object is the state of the file on disk, with no buffers containing any information that is not on disk.

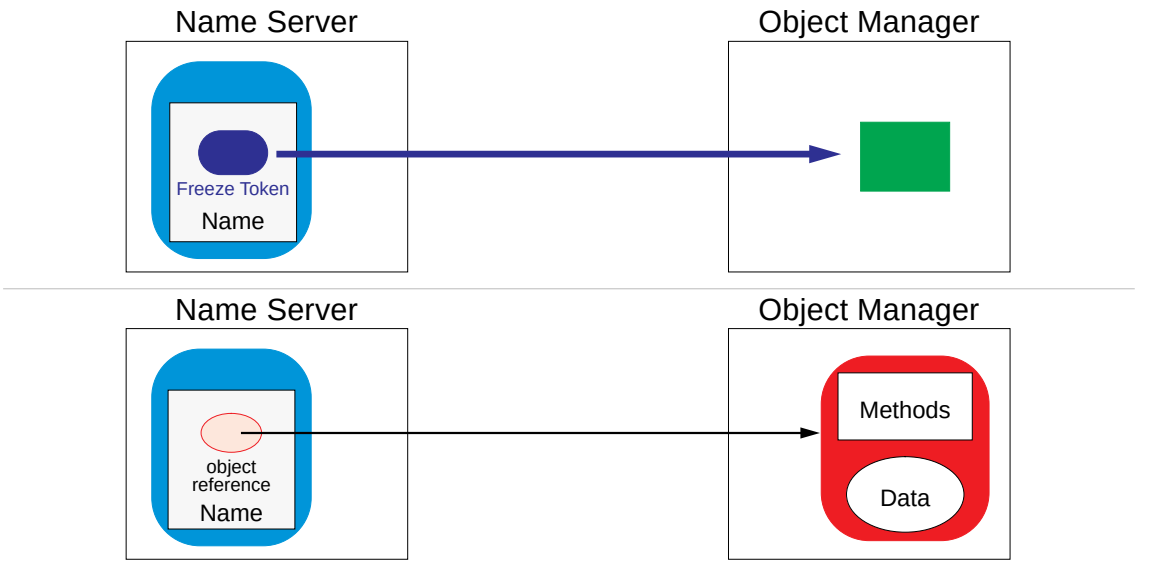
In general, pickling is the responsibility of the object manager. It produces some form of the object that can persist and arranges for it to be stored someplace that survives crashes.

Freezing and pickling can be combined. For example, a freeze token might itself contain the pickled form of the object.

*Notes:*

## Persistence

Melting




Spring

The Name Service Slide 14 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

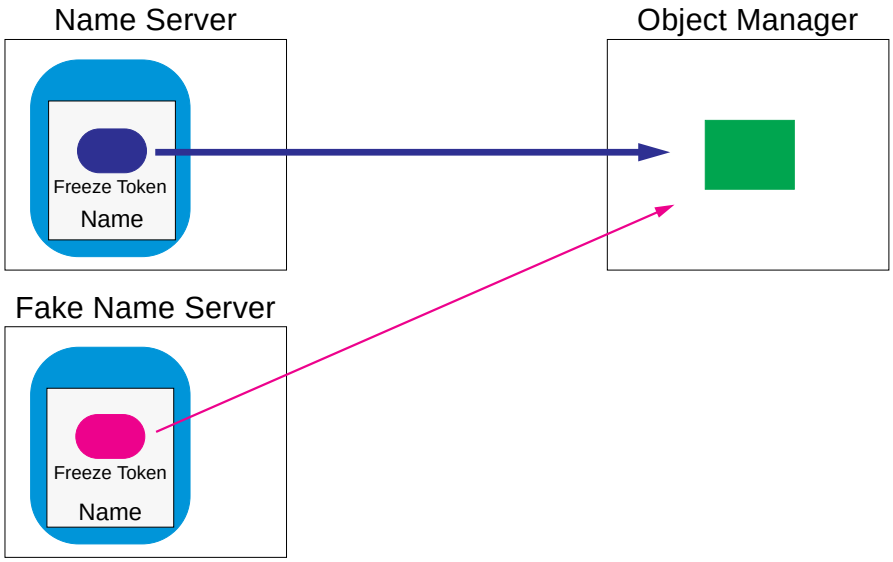
### Explanation:

Melting a freeze token results in the recreation of the object reference. If a pickled object is also associated with it, then the object is recreated as well. The active participation of the object manager is required for both melting and unpickling.

*Notes:*

## Persistence

Impostors




Spring

The Name Service Slide 15 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

### Explanation:

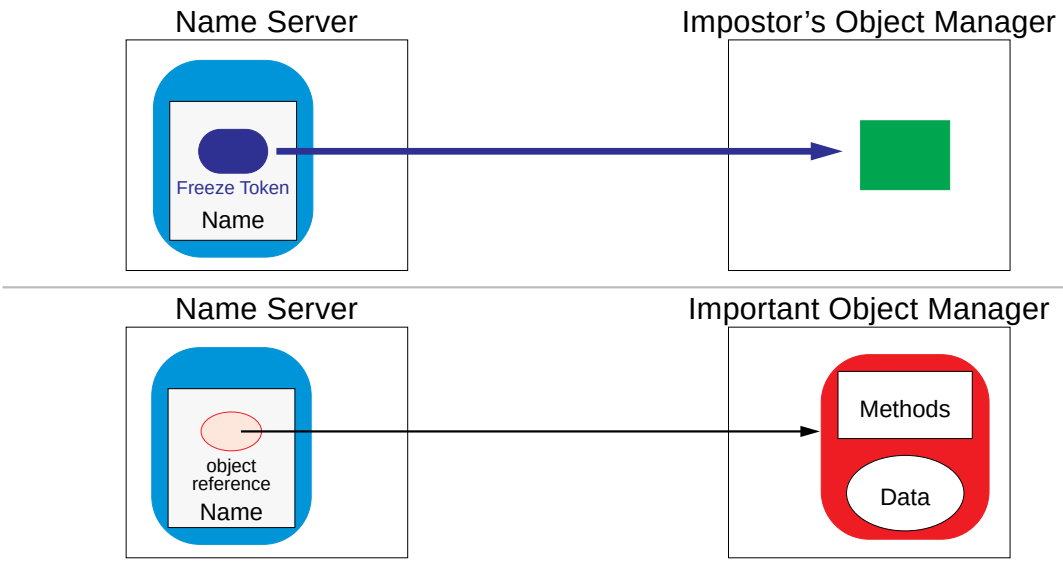
There are some security concerns associated with freezing and melting. A freeze token either contains or refers to a pickled object. But, as we've already mentioned, the security that comes along with the use of object references is not there for freeze tokens. There is nothing preventing an unscrupulous individual from guessing (or stealing) a freeze token. The individual might then present the freeze token to the object manager and get the object manager to produce an object reference. Something must be done to allow the object manager to make certain that it only melts freeze tokens for the agent that possessed the object reference in the first place.

An object manager might insist that it melt only those freeze tokens presented to it by a trusted name server. But does this solve our problem?

*Notes:*

## Persistence

A Smarter Impostor




Spring

The Name Service Slide 16 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

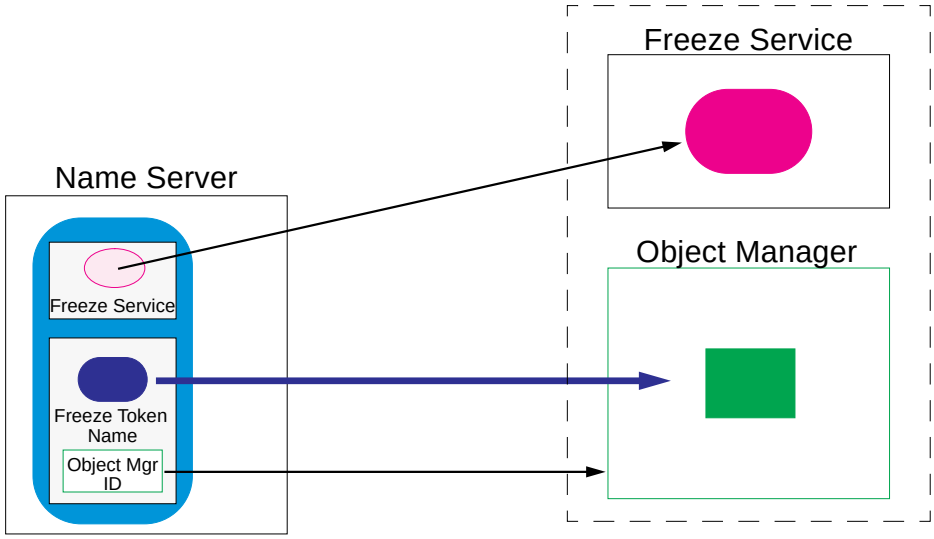
### Explanation:

Suppose that freezing is accomplished by invoking a *freeze* method on the object being frozen. A clever impostor might realize that either contained in or associated with the freeze token must be the identity of the object manager, since the name server must know to which object manager to go in order to melt the freeze token. So our impostor provides its own object manager and has it bind a reference to one of its object in the name server. The name server then contacts the impostor's object manager to freeze the reference, but this object manager produces a freeze token that looks like a freeze token of some important object manager, and includes with it the ID of the important object manager. Then, when the name server handles a resolve (from our sneaky impostor), it goes to the important object manager to melt the freeze token. The important object manager, trusting the name server, melts the freeze token and creates a reference to an important object, which is then given to the impostor.

*Notes:*

## Persistence

Freeze Service




Spring

The Name Service Slide 17 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

## Explanation:

Our solution involves the further cooperation of the name server and object manager, as well as the formalization of the *freeze service*: When a binding is established, the name server adds to it the identification of the object manager. This is provided by invoking the method *object_manager_id*, which is a standard method included with each Spring object. A reference to the object manager's freeze server is bound to a name (the object manager's ID) in a special context managed by the name server.

So this time, rather than trusting a freeze method on the object itself, the name server looks up the object manager's freeze service and calls it to freeze the object reference. When it comes time to melt the freeze token, the name service goes to the same freeze service it went to in the first place to freeze the original object reference. Thus we are assured that the same agent who did the freezing also does the melting.

*Notes:*

# *The Spring Nucleus* 4

## The Spring Nucleus

Outline

- Doors
- Threads
- Invocation

Spring

The Spring Nucleus Slide 1 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

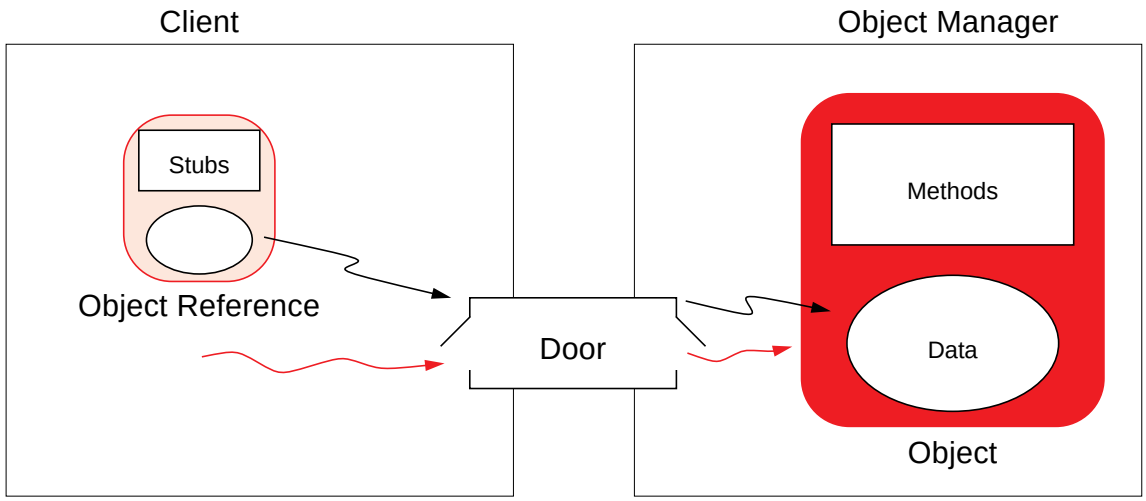
### Explanation:

We now examine what goes on inside the Spring nucleus. In particular, we look at the implementation of doors and threads and discuss how cross-address-space invocation takes place.

*Notes:*

## Doors

Model




Spring

The Spring Nucleus Slide 2 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

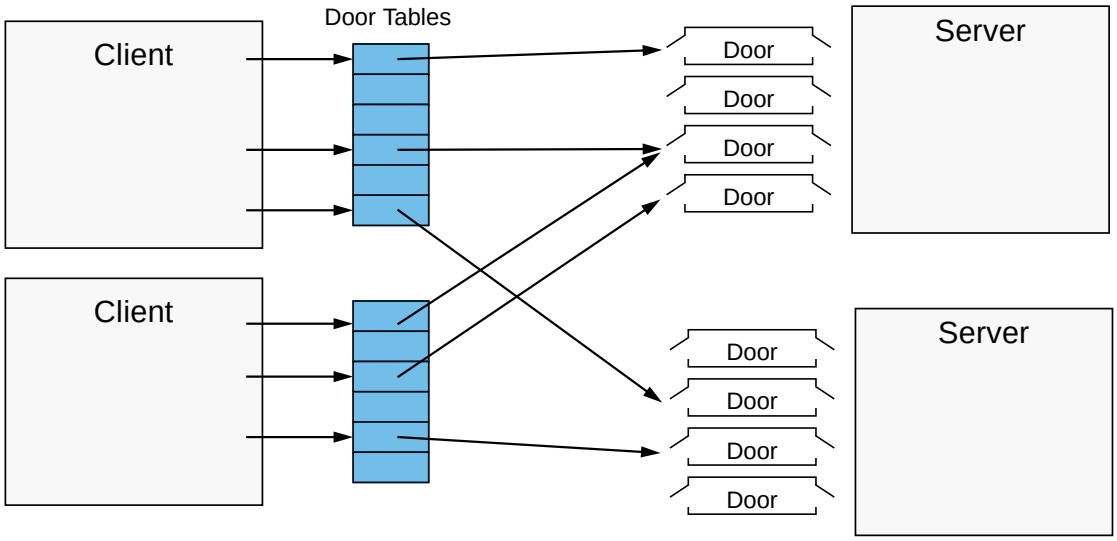
### Explanation:

The idea behind *doors* is that they provide a protected entry into servers through which clients can make invocations. A client thread invokes an operation through the client end of the door; a server thread emerges from the other end and calls the appropriate method. When done, the server thread returns back into the door, and the client thread, in turn, returns from the door.

*Notes:*

## Doors

Implementation




Spring

The Spring Nucleus Slide 3 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

### Explanation:

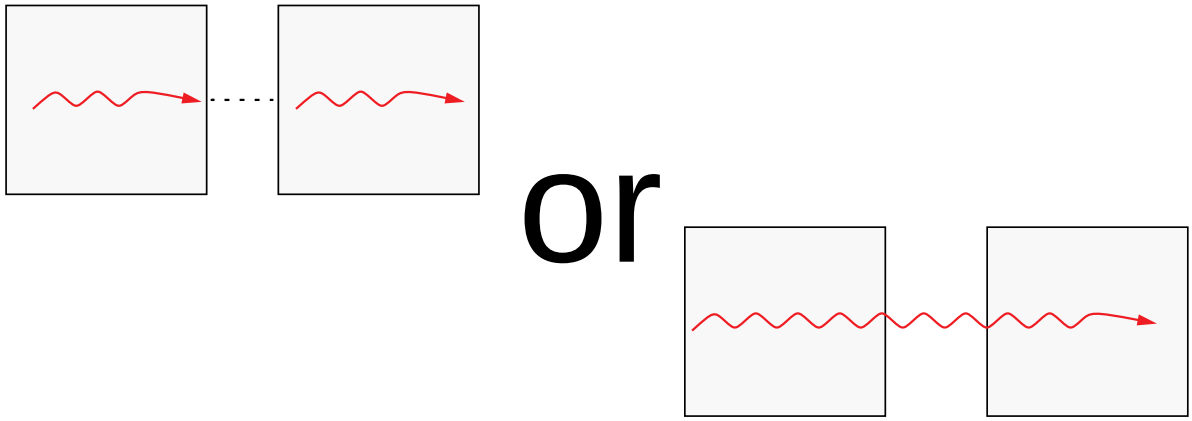
It is important that doors be secure: a client must not be able to use a door to which it has not been explicitly granted access. Doors are implemented in the kernel and this security is easy to provide. Each door is attached to the process that owns it. Clients refer to doors via *door tables*, which are arrays of pointers to doors. At the user level, a thread in a client process refers to a door by the index in the door table of the pointer to the door. Thus it is not possible for a client to refer to a door that has not been given to it.

In the singleton subcontract, a marshaled object reference is the door identifier—the index of the entry in the door table. When this is passed into the kernel, the kernel marshals it further into the pointer to the door. When such a reference is received by a process, it is first unmarshaled in the kernel by allocating a slot in the receiver's door table, yielding a door identifier (the index into the door table). This is unmarshaled further by user-level code into an object reference.

*Notes:*

## Threads

Cross-Process Calls




Spring

The Spring Nucleus Slide 4 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

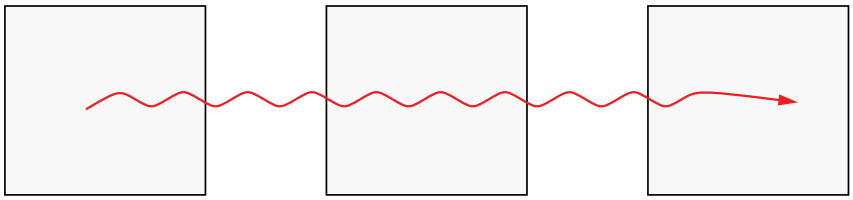
### Explanation:

What exactly happens when a thread invokes an operation through a door? From the point of view of the users, the client thread travels across processes and enters the server by emerging through the door. This could be implemented in exactly this way, but a number of problems dictated against it: if the client is being debugged, it would not be advisable for it to retain control of the thread in the server. For example, that thread might be holding a lock when the client-side debugger stops it. Rather than deal with this and other problems, separate server and client threads handle the call.

*Notes:*

## Threads

Shuttles



User View

Nucleus View

Spring

The Spring Nucleus Slide 5 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

### Explanation:

It is important, however, for scheduling information, such as time remaining in time slice, priority, etc., to be passed from client thread to server thread so that the call has the appearance of being handled by a single thread. This information is encapsulated in a *shuttle* which carries the state of the calling thread to the server thread and to further server threads that may be called upon.

*Notes:*

## Invocation

Approaches

- Fast path
- Vanilla path
- Bulk path



### Explanation:

Our primary concern for call invocation is how parameters are transferred across address spaces. As shown in the slide, three approaches are used, depending on the size of the arguments.

*Notes:*

## Invocation

Fast Path

- Data transferred from sender to receiver via registers
  - limited to 16 bytes
  - most important case for performance
- Performance (SPARC instructions):

| Action | call | return | total |
|---|---|---|---|
| Fiddling with register windows | 10 | 9 | 19 |
| Switching thread / process | 12 | 11 | 23 |
| Getting target info | 7 | 0 | 23 |
| Reference-counting target door | 3 | 4 | 7 |
| Saving / restoring return info | 6 | 4 | 10 |
| Switching VM context | 7 | 7 | 14 |
| Miscellany | 13 | 9 | 22 |
| Total | 58 | 44 | 102 |

Spring

The Spring Nucleus Slide 7 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

### Explanation:

*Fast path* is used when a small amount of data is being transferred: in the SPARC implementation, no more than sixteen bytes. Most of calls fall into this category, so it is important for performance. The table shows the number of instructions required.

*Notes:*

## Invocation

Vanilla Path

- Data is copied from sender to kernel to receiver

Spring

The Spring Nucleus Slide 8 Usenix OSDI

SunSoft
A Sun Microsystems, Inc. Business

### Explanation:

*Vanilla path* is the most straightforward means for transferring data across address spaces. The data is simply copied from sender to kernel and then from kernel to receiver.

*Notes:*

## Invocation

Bulk Path

- Data is page-aligned
- Pages transferred between sender and receiver address spaces
  - copy-on-write for copied arguments
  - unmapped in sender's address space for non-copied arguments



### Explanation:

For large amount of data, the *bulk path* approach is used. The data must be page-aligned and a multiple of a page size in length. The pages of the sender are simply mapped into the receiver's address space. If the parameter-passing mode requires copying of the argument, the mapping is copy-on-write. Otherwise the pages are unmapped from the sender's address space.

*Notes:*

# *Virtual Memory and the File System* 5

## Virtual Memory and the File System

Outline

- Virtual-Memory Model
- Virtual-Memory Implementation
- File System



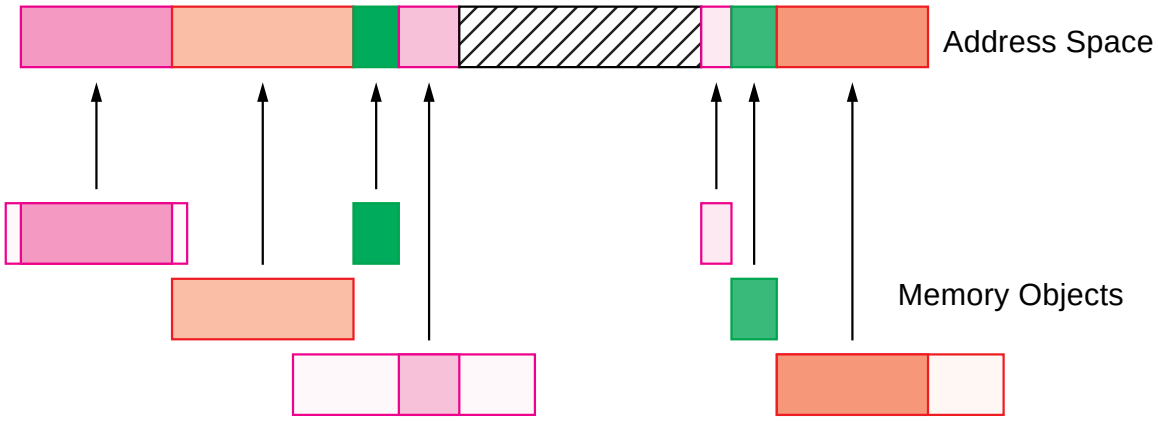
### Explanation:

In this final module we cover virtual memory in Spring and its relation to file systems.

*Notes:*

## Virtual-Memory Model

Virtual Memory




Spring

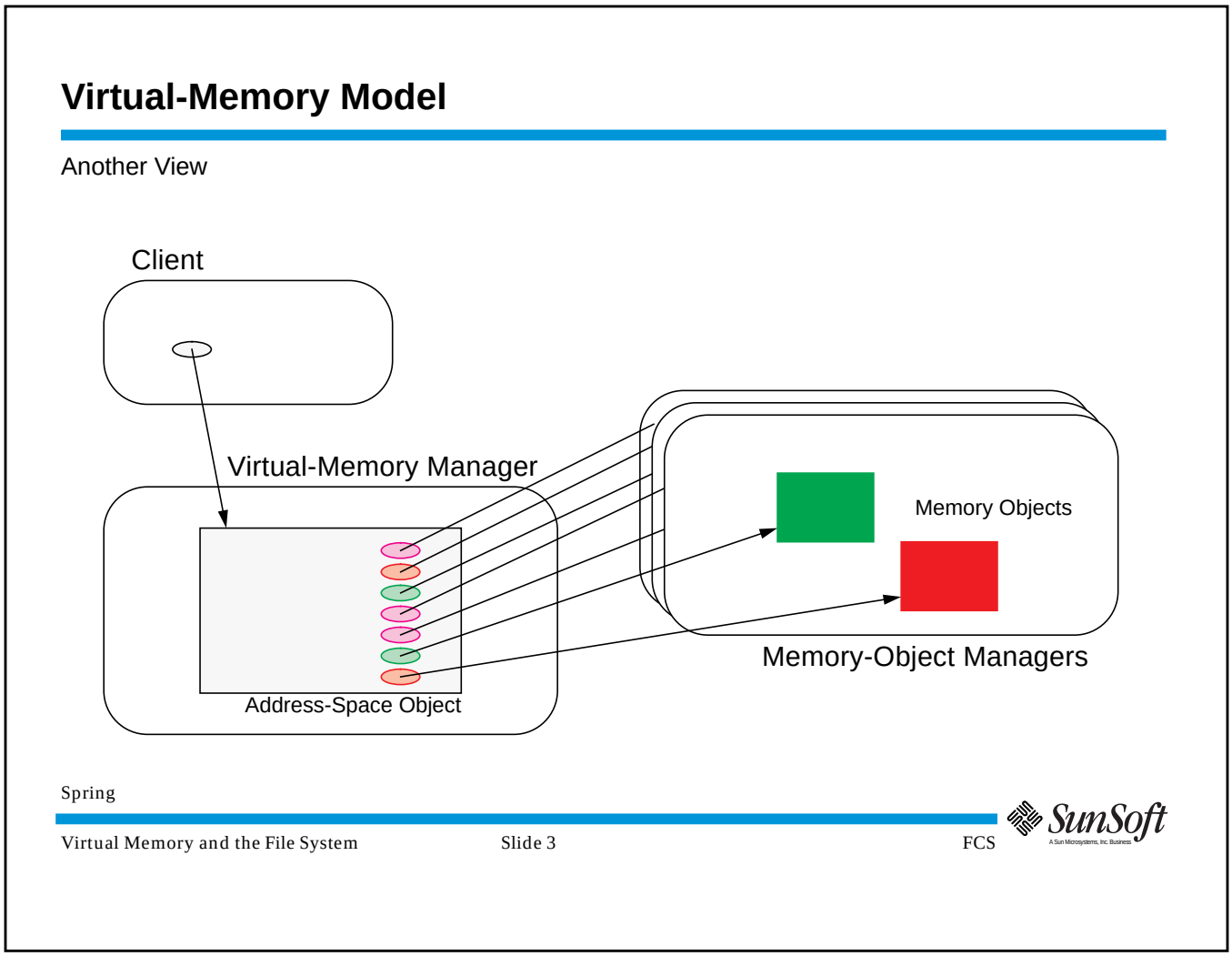
Virtual Memory and the File System Slide 2 FCS

SunSoft
A Sun Microsystems, Inc. Business

### Explanation:

The virtual-memory model of Spring is straightforward: certain “things” are mapped into the address space. The things that can be mapped are *memory objects*, which can be mapped whole or in part.

*Notes:*




## Explanation:

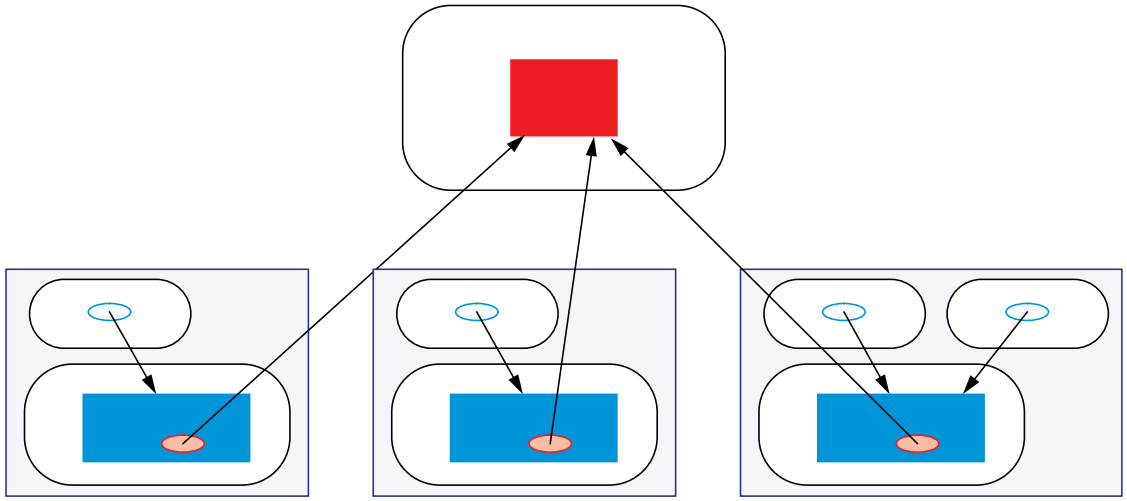
Here we have a different view of virtual memory, this time in terms of objects and object references. The *virtual-memory* manager is a component of the kernel; it manages *address-space objects*. Each process possesses a reference to the object for its address space, which it uses to perform mapping operations. Contained in each address-space object are the references to the memory objects that have been mapped into the address space. These objects are managed by *memory-object managers*.

The virtual-memory manager is responsible for maintaining each address space, for managing physical storage, and for making page-in and page-out decisions. The memory-object manager is responsible for the contents of the memory object. It must provide a backing store, if necessary, as well as insure persistence, if necessary.

*Notes:*

## Virtual-Memory Model

And Yet Another View



Spring

Virtual Memory and the File System Slide 4 FCS

SunSoft
A Sun Microsystems, Inc. Business

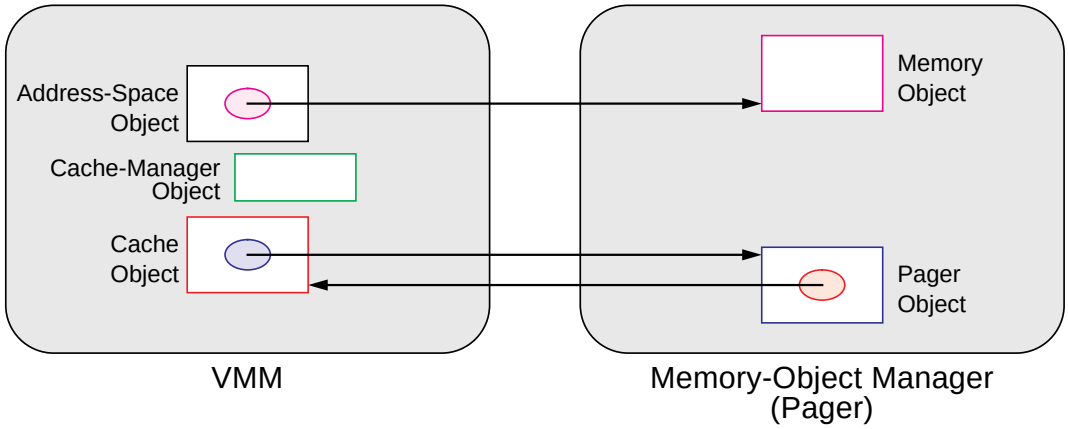
### Explanation:

In this view we see the sorts of sharing that are supported. Each of the large shaded boxes at the bottom of the slide represents separate computers, each with its own virtual-memory manager. On each of the computers, one or two processes have mapped into their address spaces one particular object, which is being shared by the four processes in the slide. As we are about to discuss, the individual processes's views of the memory object are made consistent with one another's via cooperation between the virtual-memory managers and the object managers.

*Notes:*

## Virtual-Memory Implementation

Implementation Components




Spring

Virtual Memory and the File System Slide 5 FCS

SunSoft

### Explanation:

As part of supporting a mapping, the virtual-memory manager (VMM) and memory-object manager (henceforth called *pager*) interact with each other. The VMM is itself represented as an object, the *cache-manager object*. As explained on page 154, it places a reference to this object in the name space. If a memory object is mapped in the address space, a reference to the memory object is placed in the address-space object. This object reference serves to identify what is mapped, but, as discussed on page 148, is not used to operate on the object. Instead, the pager provides a reference to a *pager object* to the VMM, which is used to do the page-in and page-out requests.

The pager also needs a handle to operate on its pages in the address space. So, associated with each mapping of a memory object is a *cache object*, managed by the VMM. This object encapsulates the pages of the memory object that are currently cached by the VMM.

*Notes:*

## Virtual-Memory Implementation

Design Issues

1) Separation of *memory object* from *paging object*
2) Memory objects encapsulate access rights to storage
   - different memory objects referring to same storage share cached pages when used on the same machine
3) Not all memory-object managers are created equal
   - some are trusted by the kernel; most are not



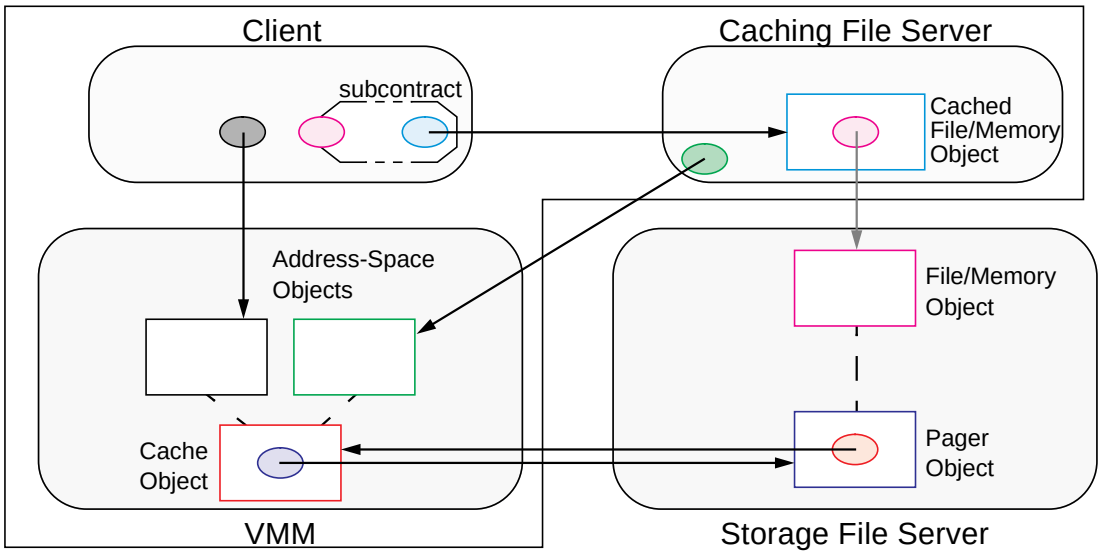
### Explanation:

A few design issues must be explained before we look further at the virtual-memory implementation.

*Notes:*

## Virtual-Memory Implementation

Separation of Memory Object From Paging Object




Spring

Virtual Memory and the File System Slide 7 FCS

SunSoft
A Sun Microsystems, Inc. Business

### Explanation:

It is important to separate the notion of *memory object* from the notion of *paging object*, particularly for the case of *file objects*, which are derived from memory objects. This allows the implementations of the two objects to reside in separate processes, perhaps on separate machines.

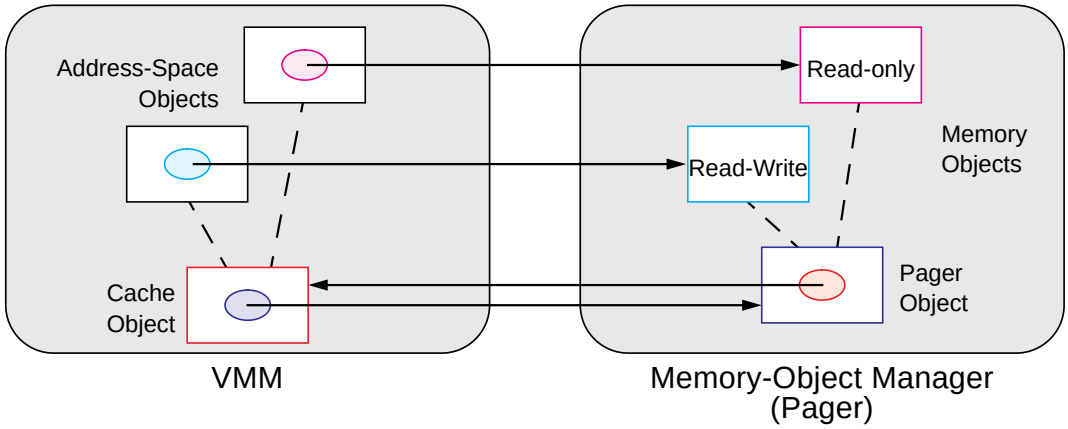
The slide gives a simplification of what is covered in the latter part of this module: we are using the *cacher subcontract* in conjunction with the *storage file server* (*SFS*), where files reside permanently, and the *caching file server* (*CFS*), which serves as a means for caching file data and attributes. Files may be accessed in Spring either by direct *read* and *write* operations or by *mapping* the file (utilizing its memory-object base type) into an address space. Operations on the file object (i.e., the memory object), such as reads, writes, and maps, go to the local CFS, while page-ins and page-outs go directly to the SFS. The CFS itself maps the file into its address space, and thus it can satisfy reads and writes using data cached by the

*Notes:*

VMM, yet use its own copies of file attributes. Once a file has been cached by a local CFS, further mapping operations on the file can be handled via the cached information in the CFS, without need for a remote request to the SFS.

## Virtual-Memory Implementation

Memory Objects Encapsulate Access Rights to Storage




Spring

Virtual Memory and the File System Slide 8 FCS

SunSoft
A Sun Microsystems, Inc. Business

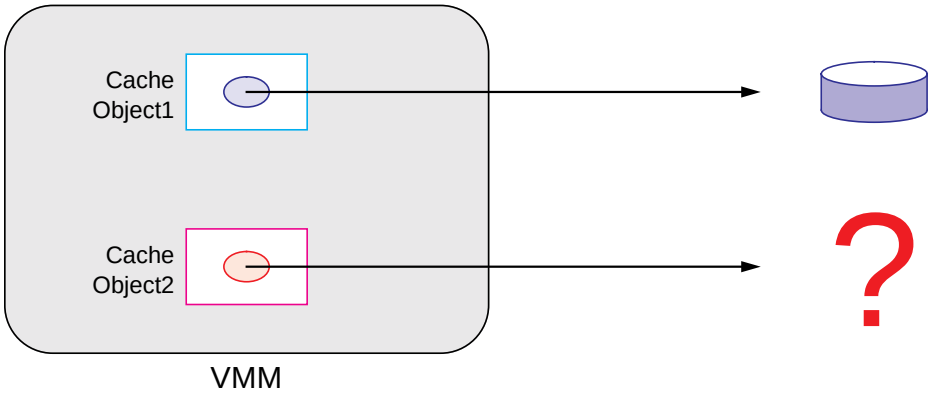
### Explanation:

A memory object represents not only storage (e.g., a file) but also one's access rights to that storage. For example, one process may have mapped a file read-write, while another process on the same machine may have mapped it read-only. Each process has a separate memory object representing its access to the file. It would be foolish to have two (duplicate) collections of pages on the machine to represent the two mappings. There should be only one collection; the VMM can give read-write access to the pages to one process and read-only access to the other.

*Notes:*

## Virtual-Memory Implementation

Not All Memory-Object Managers Are Created Equal




Spring

Virtual Memory and the File System Slide 9 FCS

SunSoft
A Sun Microsystems, Inc. Business

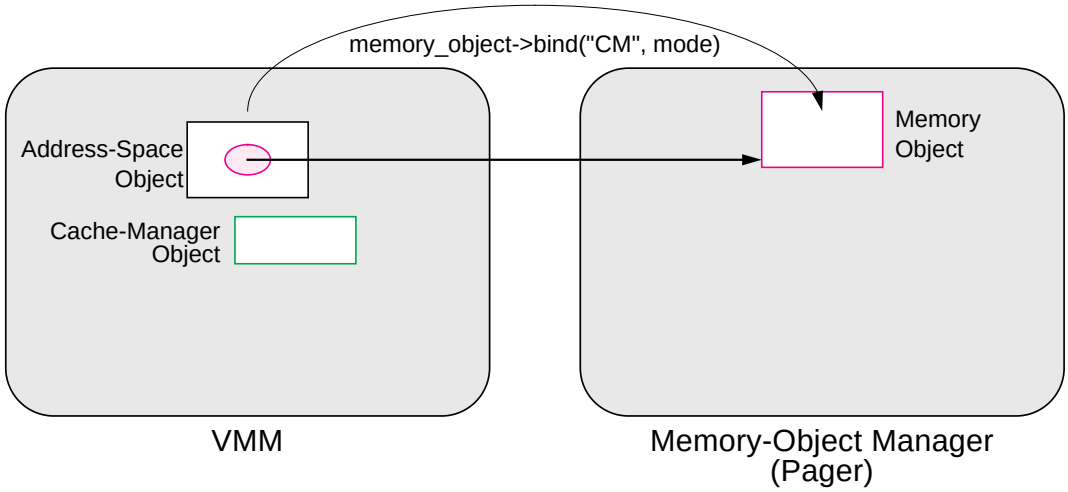
### Explanation:

Our final concern involves trust. Suppose that a VMM is running out of page frames and is attempting to page out some pages to a pager. There is probably one pager, the *default pager*, that is really part of the “system” and that the VMM can depend on to honor all requests quickly. Other pagers might not be quite so reliable (for example, they might be the result of an introductory course on Spring). If a page-out to a less reliable pager is taking too long, the VMM can always page out the pages to temporary space managed by the default pager. If the default pager is taking too long, the VMM has no choice but to wait until it completes its requests. It thus needs some means for learning and representing which are the reliable pagers and which are not.

*Notes:*

## Virtual-Memory Implementation

Bind Protocol (1)




Spring

Virtual Memory and the File System Slide 10 FCS

SunSoft A Sun Microsystems, Inc. Business

## Explanation:

In response to a user process's request to map a memory object, the VMM invokes the *bind* method of the memory object, passing it the name, in the name space, to which is bound a reference to the *cache-manager object*. What the VMM is trying to determine is whether or not it has already mapped the storage behind this memory object, perhaps via a different memory object (referring to the same storage). The only way it can find out is to ask the pager. If the underlying storage is already mapped, the pager will immediately return an (indirect) reference to the cache object. Otherwise the pager calls back to the VMM, via the cache-manager object, to establish a cache object, and then returns a reference to the cache-object. We go through these steps in the next few slides.

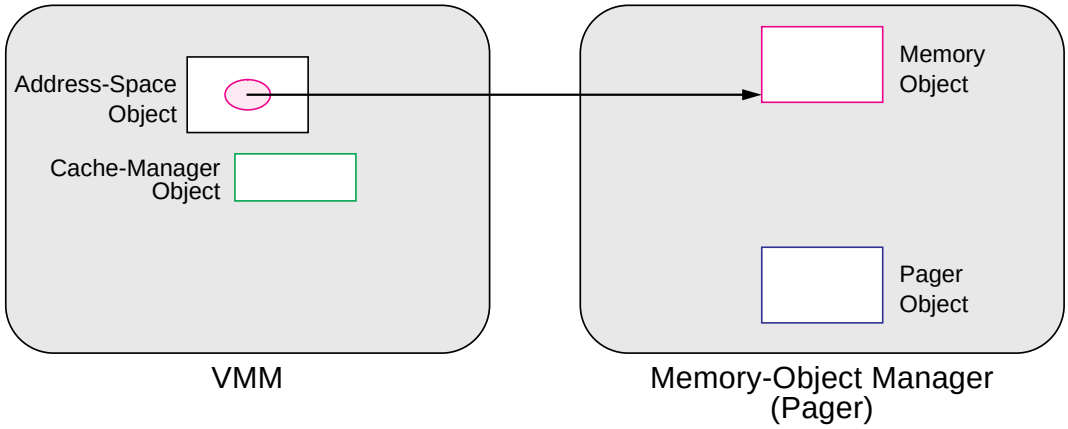
At issue here is the "privilege level" of the pager. By passing the name of the cache-manager object to the pager, rather than simply passing an object reference, the VMM forces the pager to do an authenticated lookup in the name service. Reliable pagers (e.g., the default pager) can prove themselves by resolving a spe-

*Notes:*

cial cache-manager name bound to an object reference that encapsulates their high privilege level. Pagers can cache the cache-manager-name-to-object-reference translation, so the overhead for this degree of indirection is not large.

## Virtual-Memory Implementation

Bind Protocol (2)




Spring

Virtual Memory and the File System Slide 11 FCS

SunSoft A Sun Microsystems, Inc. Business

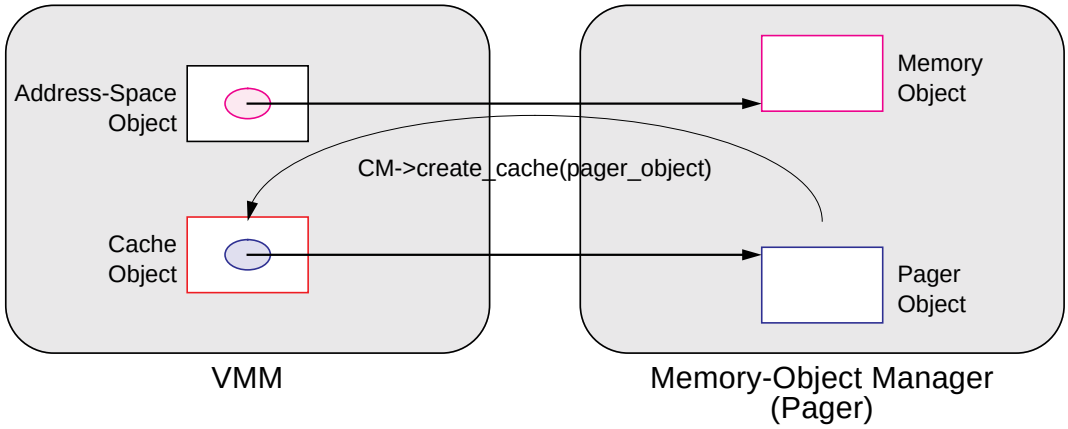
### Explanation:

In response to the bind request, the pager checks to see if the storage underlying the memory object is already mapped in the requesting cache manager (and hence machine). If so, there already exists a pager object, in the pager, and a cache object, in the VMM, for the mapping. The pager then returns a reference to a *cache_rights* object, as explained on page 162, which identifies the VMM-implemented cache already being used for this memory object. This insures that the VMM will maintain only one copy of the pages from any one memory object. If the memory object is not already in use on the requesting machine, the pager creates a *pager object* and proceeds with the steps covered in the next few slides.

*Notes:*

## Virtual-Memory Implementation

Bind Protocol (3)





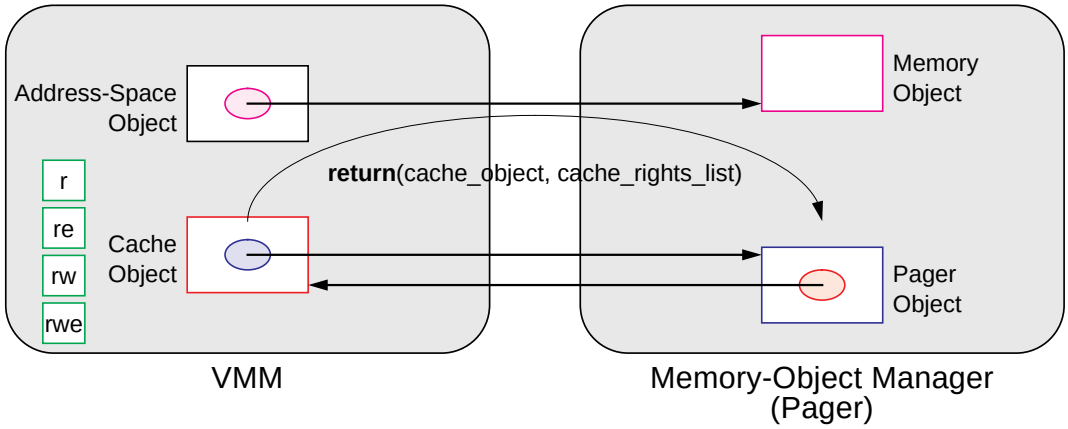

### Explanation:

Next, the pager invokes the cache manager's *create_cache* method to establish a new *cache object* in the requestor's VMM. It passes to the cache manager a reference to the pager object, which is stored in the cache object.

*Notes:*

## Virtual-Memory Implementation

Bind Protocol (4)





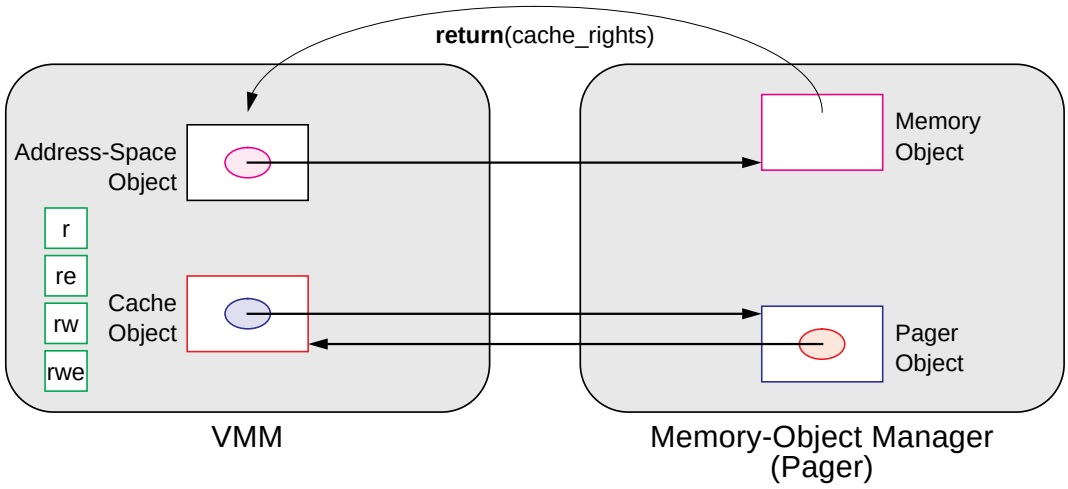

### Explanation:

As part of establishing the cache object, the cache manager puts together a list of *cache_rights* objects called the *cache_rights_list*. Each of these objects serves to indicate one of the four possible access rights to the memory object (read-only, read-execute, read-write, read-write-execute) and hence to the pages cached via the cache object. These objects are used subsequently by the pager to identify to the VMM the cache object and access rights required.

*Notes:*

## Virtual-Memory Implementation

Bind Protocol (5)






### Explanation:

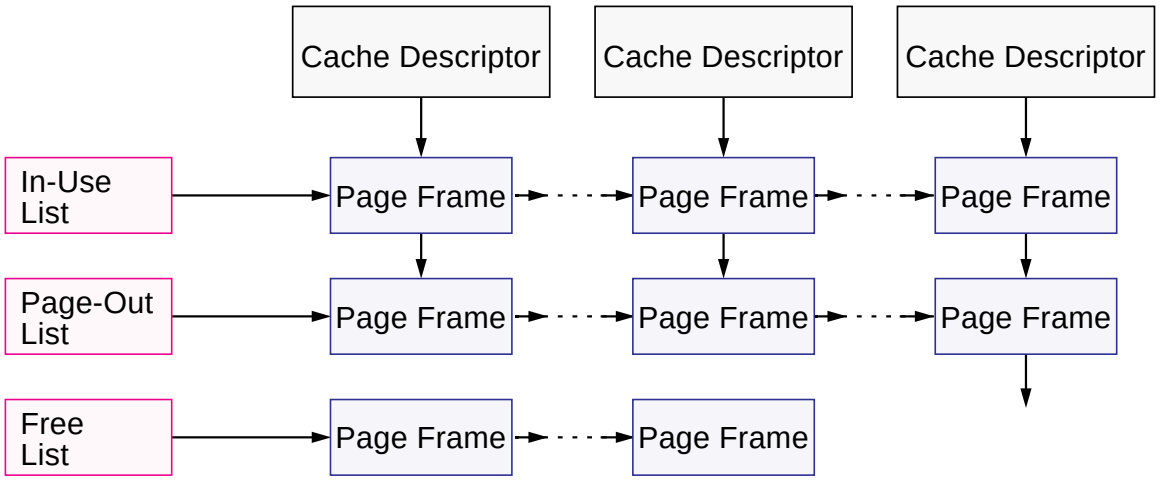
Finally, the VMM's original call to the bind method of the memory object returns a reference to a *cache_rights* object. As discussed with the previous slide, this object identifies a cache object in the VMM and the access rights required. For example, if the memory object is being mapped read-only, then the *cache_rights* object reference returned refers the VMM to the (now) already established cache object and, furthermore, instructs it that the user of this mapping is allowed only to read, and not to modify, the pages.

The rationale for returning a *cache_rights* object and not a reference to either the cache object or the pager object is security: the *cache_rights* object is not a capability—it confers no additional rights to the receiver. A malicious process might try to force the pager to create a cache object on a legitimate kernel, then have the pager return to the process rights that the process shouldn't have. But what is returned will be of no use to it whatsoever, since it merely refers the receiver to something that the receiver should possess already.

*Notes:*

## Virtual-Memory Implementation

Page-Level Management




Spring

Virtual Memory and the File System Slide 15 FCS

SunSoft

### Explanation:

The implementation of page-level management is straightforward. Associated with each cache object is a cache descriptor that heads a list of references to the page frames holding the cached pages. Page frames are also on other lists: a list of in-use pages, ordered in (approximate) least-recently-used order, a list of pages that are being paged out (but are still in some cache), and a free list.

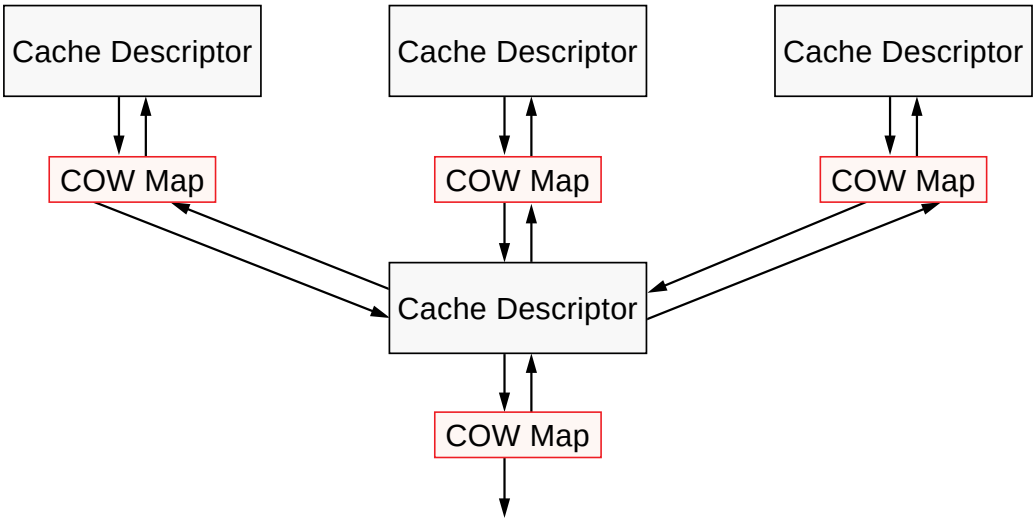
Four classes of threads are employed to manage the page frames:

- *pushers*: responsible for the page-out list
- *prefetchers*: responsible for prefetching pages into page frames
- *sweeper*: a single thread that implements the usual two-handed clock algorithm for maintaining an approximate notion of the least-recently-used pages
- *cache reclaimer*: a single thread responsible for deleting unattached caches

*Notes:*

## Virtual-Memory Implementation

Copy on Write




Spring

Virtual Memory and the File System Slide 16 FCS

SunSoft A Sun Microsystems, Inc. Business

### Explanation:

The data structures for maintaining the various copy-on-write relationships are shown in the slide. When a region is created as a copy of another region, the first (target) shares its pages with the second (source). As long as pages in either region are merely read, there is no need for copying. If, however, a page is modified in either region, then a copy of that page must be made, so that the other has the original.

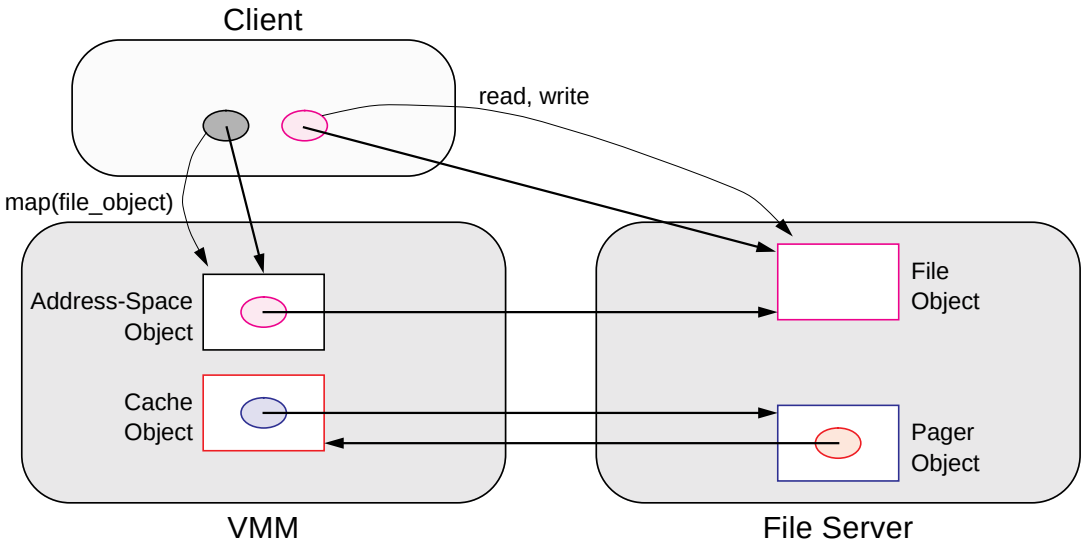
A region set up as a copy of another has a *COW map* attached to its cache descriptor. This indicates the status of the pages of the region: have they been copied from the source, or are they still shared with the source? If a target region (possessing one of the upper cache descriptors in the slide) modifies a page, it obtains a copy of the page from the source (which possesses the lower cache descriptor). If the source modifies a page, it first copies the page into the caches of each of the targets.

*Notes:*

This arrangement nests to arbitrary numbers of levels—the source in the picture could be a target of a lower-level source.

## File System

Naive Approach




Spring

Virtual Memory and the File System Slide 17 FCS

SunSoft A Sun Microsystems, Inc. Business

### Explanation:

We now look at the interactions between file systems and virtual memory. The situation we want to avoid is illustrated in the slide. Here a file (which inherits from *memory object*) is mapped into a process, while at the same time the file's *read* and *write* methods are being invoked directly. The problem is that pages of the file modified via virtual memory are cached in the cache object, while portions of the file modified directly via *write* are not cached (or at least not cached in the cache object). This leads to inconsistency.

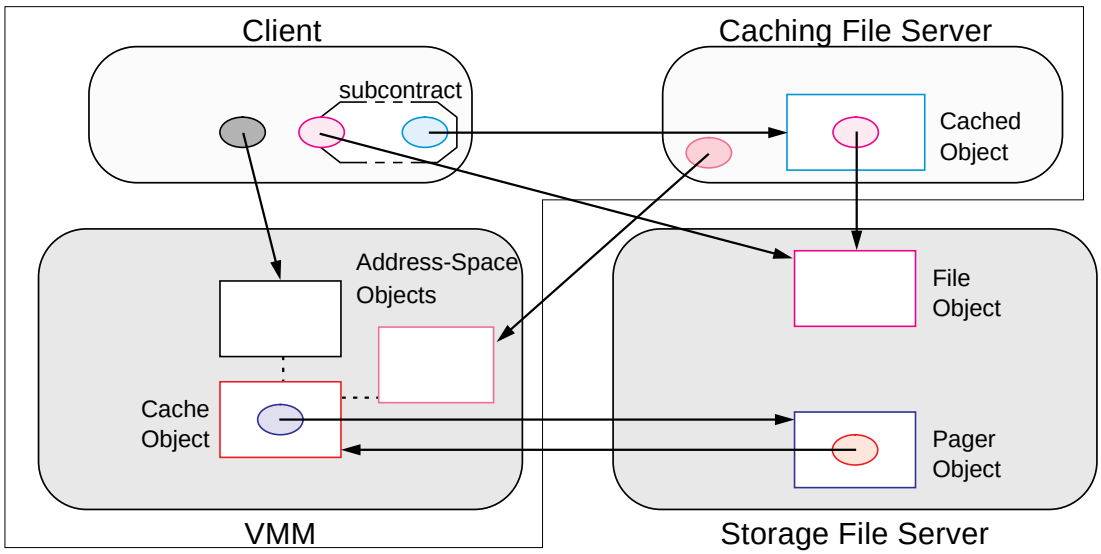
Another problem with this approach is that all the attributes of the file (e.g., size and time of last modification) are stored only at the file server. All requests to obtain these attributes (such as with the UNIX *stat* call) are remote requests. Since attribute requests typically occur frequently, their cumulative cost is quite high.

Finally, all requests to map a file involve *bind* calls to the pager-implemented file object, despite the fact that the file might already be mapped locally.

*Notes:*

## File System

The Caching File System (1)




Spring

Virtual Memory and the File System Slide 18 FCS

SunSoft
A Sun Microsystems, Inc. Business

### Explanation:

The solution is to use the cacher subcontract (see page 76), carefully integrated with the virtual memory system.

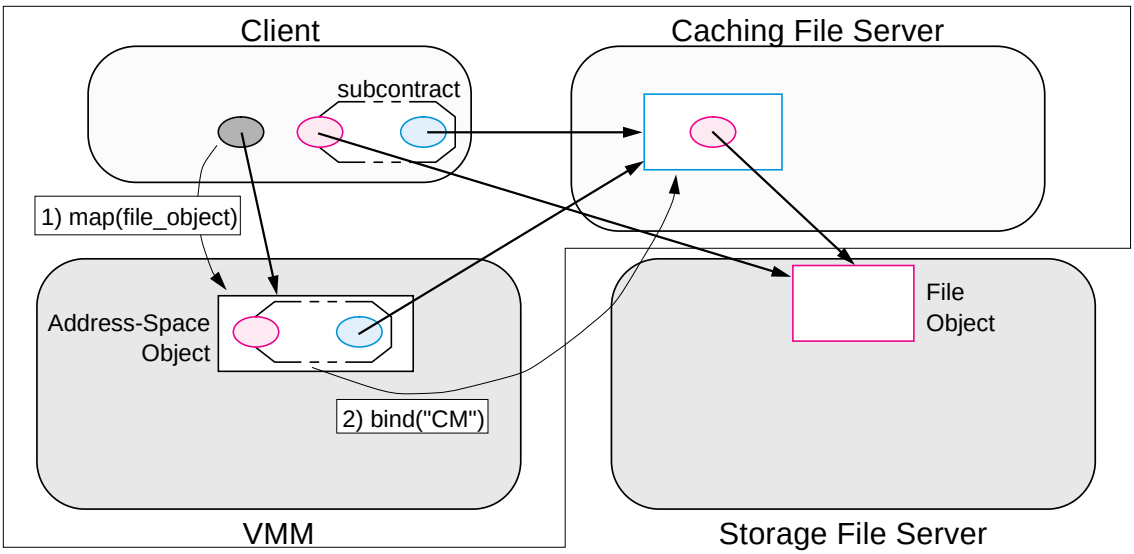
We use two sorts of file servers—*storage file servers* (*SFS*), which are our usual idea of file servers (they maintain files on disks), and *caching file servers* (*CFS*), which implement local caches. The desired situation is partially shown in the slide. The CFS positions itself between clients and file objects: thus operations by a client on the file object are handled by the local CFS, which maps the file into its address space to handle reads and writes. If a client has also mapped the file into its address space, the VMM makes certain (as discussed on page 156) that all users of the file share the same cache object (and thus share the pages from the file). The CFS maintains a copy of the attributes of the file, so it can respond quickly to requests for file attributes. Furthermore, it intercepts requests to bind files into clients' address spaces (again, via the actions of the cacher subcontract), so it can

*Notes:*

determine whether a file is currently mapped and refers the VMM directly to the cache object rather than having to make a remote call to the SFS (which acts as the pager).

## File System

The Caching File System (2)




Spring

Virtual Memory and the File System Slide 19 FCS

SunSoft
A Sun Microsystems, Inc. Business

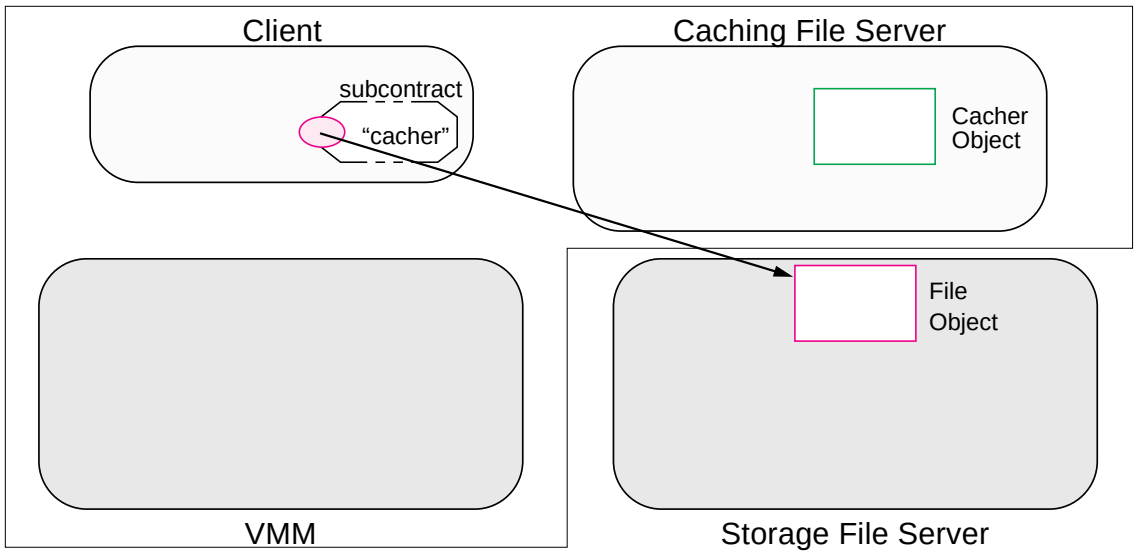
### Explanation:

Suppose that a client maps a file into its address space. It goes through the steps discussed starting on page 154, except that the memory object (in this case the file object) uses the cacher subcontract (discussed in detail starting with the next slide). With this subcontract, all invocations on the file-object reference are sent not to the file object, but to the cached object (in the caching file server). Thus the invocation of *bind* by the VMM goes to the CFS.

*Notes:*

## File System

The Cacher Subcontract (1)




Spring

Virtual Memory and the File System Slide 20 FCS

SunSoft
A Sun Microsystems, Inc. Business

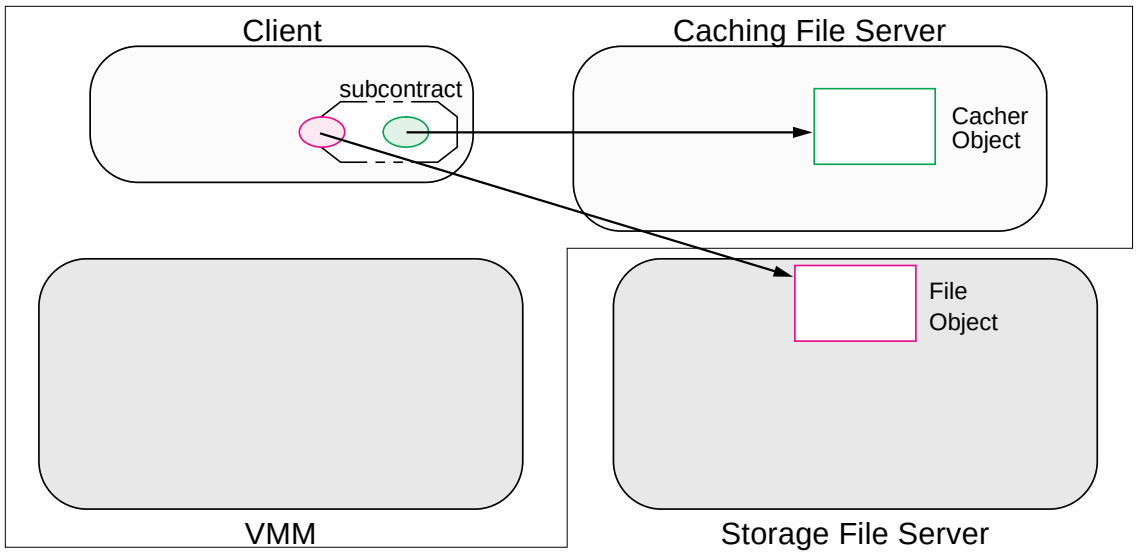
### Explanation:

We make a brief digression to discuss the cacher subcontract. When cacheable-object references (e.g., our file objects) are marshaled, the subcontract includes in the representation the *cacher name*, which is the name to which each local CFS binds a reference to its *cacher object*. Thus, in the case of the file system, each CFS binds its cacher object to the cacher name in the local machine's name space.

*Notes:*

## File System

The Cacher Subcontract (2)





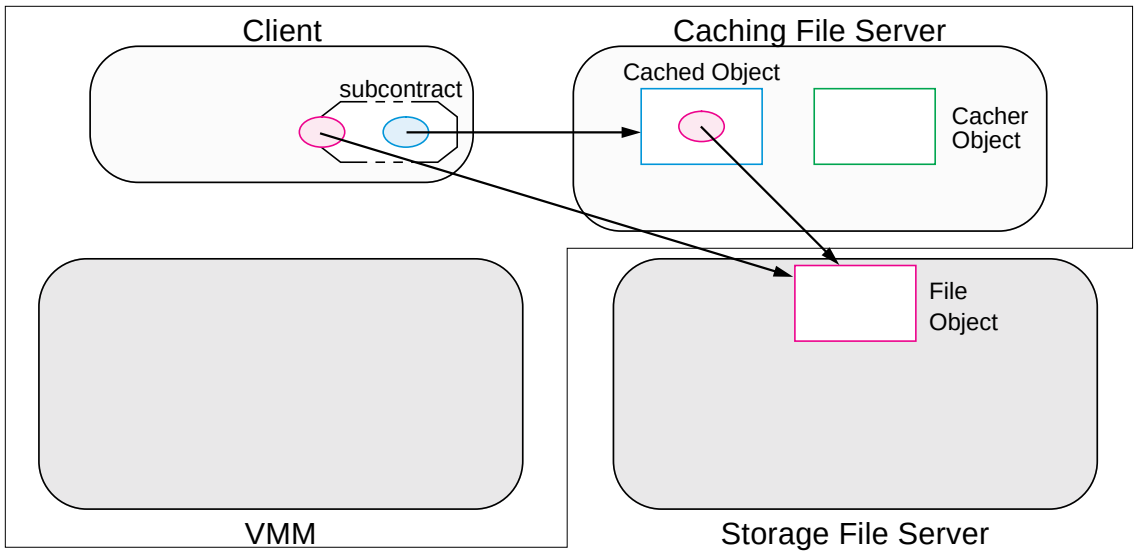

### Explanation:

When a cacheable-object reference is unmarshaled, first the cacher name is resolved to the CFS's cacher object, on the machine at which the cacheable-object reference is being used. A call is then placed to the cacher object's *get_cached_obj* method, which is passed the cacheable-object reference (in this case, the reference to the file object).

*Notes:*

## File System

The Cacher Subcontract (3)




Spring

Virtual Memory and the File System Slide 22 FCS

SunSoft
A Sun Microsystems, Inc. Business

### Explanation:

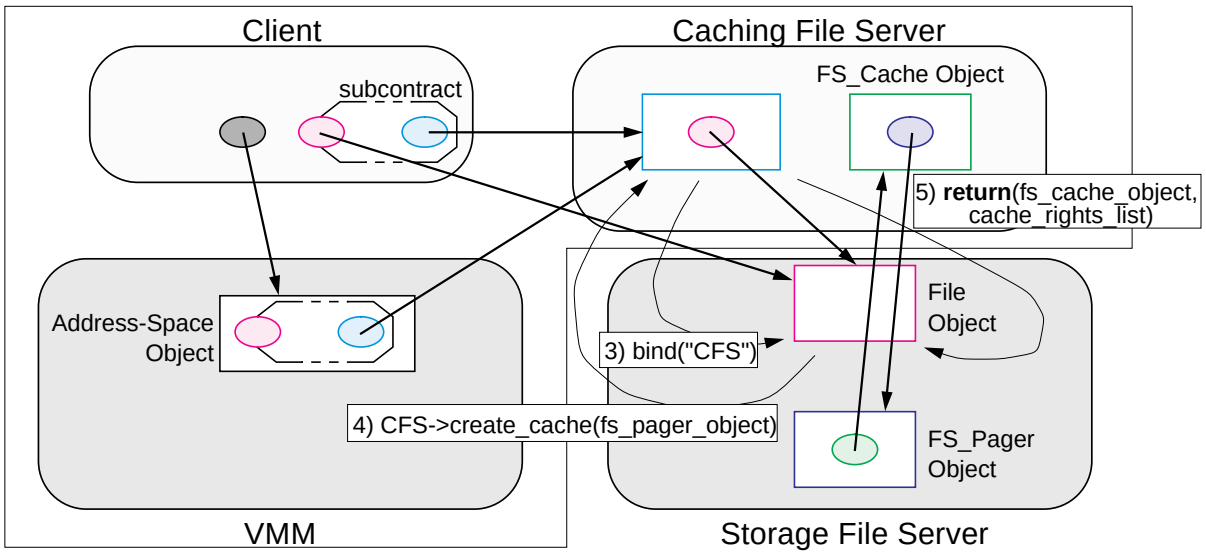
If the cacheable object is not already cached at the CFS, the CFS creates a *cached object* and passes a reference to this back to the caller (i.e., the cacher subcontract in the client which is unmarshaling the reference to the file object). The cached object is now the object to which operations on the file object by the client are directed. Some side effects are discussed in the following slides.

If the cacheable object has already been cached at the CFS, the existing cached object is returned to the caller.

*Notes:*

## File System

The Caching File System (3)




Spring

Virtual Memory and the File System Slide 23 FCS

SunSoft A Sun Microsystems, Inc. Business

### Explanation:

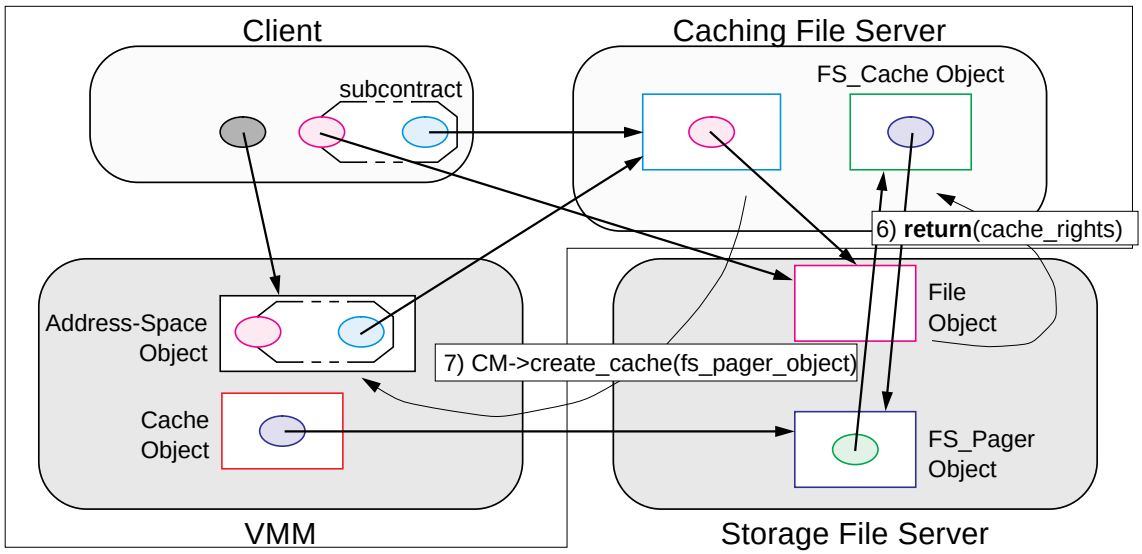
We now return to our discussion of the file system.

The CFS forwards the bind request to the pager (here, the SFS), but it substitutes itself as the cache manager. The SFS, acting in its pager role (page 156), checks to see if the underlying file has already been mapped by the caller. In this case it hasn't, so the SFS creates an *FS_pager* object (derived from *pager*) and invokes the *create_cache* method of the CFS, passing it a reference to the *FS_pager* object. The CFS obediently creates an *FS_cache* object (derived from *cache*), sets up a *cache_rights_list*, and returns the list and the reference to the *FS_cache* object to the SFS.

*Notes:*

## File System

The Caching File System (4)




Spring

Virtual Memory and the File System Slide 24 FCS

SunSoft A Sun Microsystems, Inc. Business

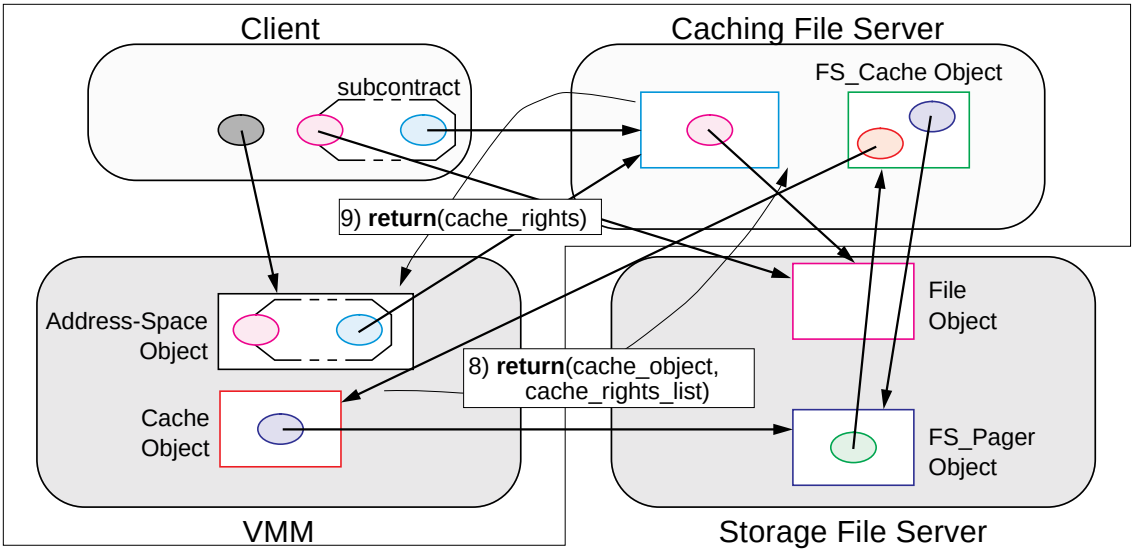
### Explanation:

The SFS now returns the appropriate *cache_rights* object reference to the CFS. The VMM, meanwhile, has been patiently waiting for a response to its bind request. The CFS, now playing the role of a pager, invokes the VMM cache manager's *create_cache* method, passing it a copy of the reference to the *fs_pager_object* of the SFS. The VMM establishes its own cache object and places inside of it the reference to the *FS_pager* object.

*Notes:*

## File System

The Caching File System (5)




Spring

Virtual Memory and the File System Slide 25 FCS

SunSoft A Sun Microsystems, Inc. Business

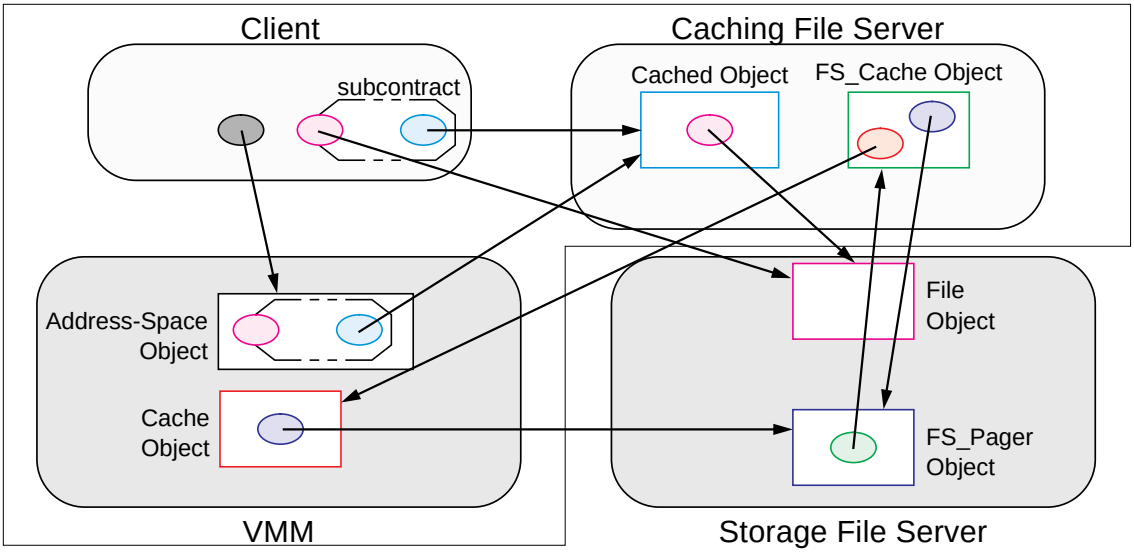
### Explanation:

The VMM, continuing with its responsibilities, creates the *cache_rights* list and passes it and the reference to its cache object back to the CFS, which promptly returns the appropriate *cache_rights* object reference to the VMM.

*Notes:*

## File System

The Caching File System: Final Setup ...




Spring

Virtual Memory and the File System Slide 26 FCS

SunSoft A Sun Microsystems, Inc. Business

### Explanation:

After the file object is mapped, the situation (finally) looks as shown in the slide. The *cache_object* in the VMM contains a reference to the *FS_pager* object of the SFS, so page-in and page-out requests are handled directly. Requests by the SFS to the *cache_object* (for maintaining cache coherency) are passed through the *FS_cache* object of the CFS.

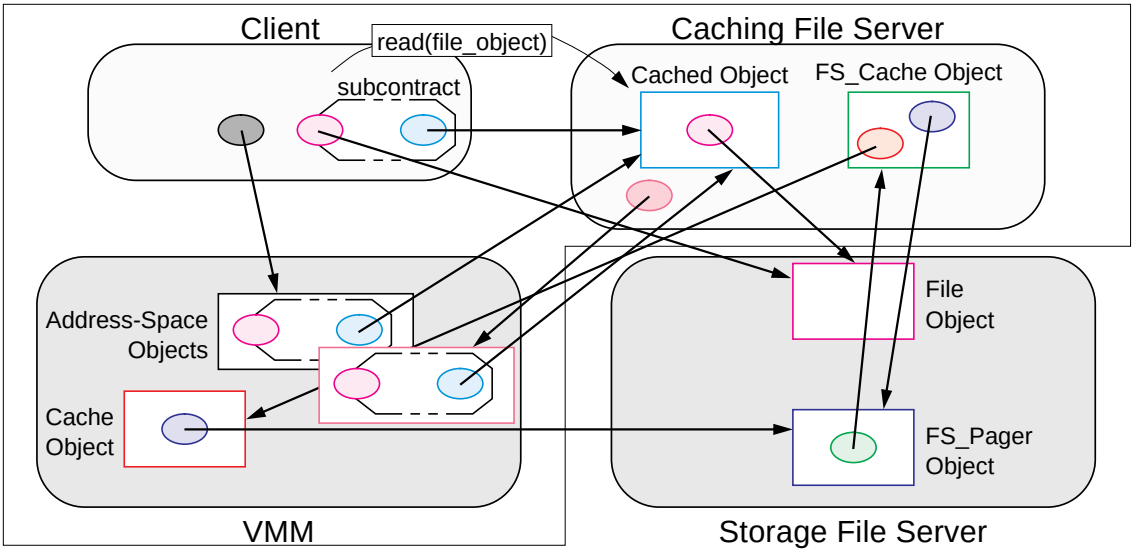
There are two reasons for this complicated-looking setup. The first is that the CFS is caching binds: without the CFS, each bind request by the VMM results in at least one call and return between the VMM and the SFS. Even if the VMM already has mapped the object, it still must contact the object's pager (SFS in this case) to find out which cache object to use. However, with the cacher subcontract, the VMM invocations of methods on the memory object are dealt with by the (local) CFS, which can tell the VMM which cache object to use without needing to communicate with the (remote) SFS.

*Notes:*

The other reason for the complicated setup is that, for file objects, the *attributes* of the file, in particular length and modification time, must be known on the client's machine. (For example, if the file has become shorter since its last use, a portion of the virtual memory into which the file has been mapped may no longer be valid.) So, when a file is mapped for which the bind no longer exists on the client machine, the CFS must obtain the current attributes of the file. It does this via an invocation of the *cached_bind* method of the *FS_pager* object, which returns the attributes of the file and an indication of whether these attributes may be cached.

## File System

The Caching File System: ... But Not Quite




Spring

Virtual Memory and the File System Slide 27 FCS

SunSoft
A Sun Microsystems, Inc. Business

### Explanation:

Now suppose that after a client has mapped a file, this client (or another one on the same machine) accesses the file by invoking its *read* or *write* methods. As mentioned on page 168, we must insure that the cache used for caching reads and writes is the same one used for caching the pages mapped into virtual memory. This can now be accomplished easily: The CFS maps the file (or a portion of it) into its address space. The VMM contacts the CFS as part of setting up the mapping. The CFS knows that the file is already mapped locally, so immediately returns to the VMM the *cache_rights*-object reference and *fs_pager*-object reference for the mapping. The CFS can now access file pages to satisfy read and write requests, and these will be the same pages used by other processes that have mapped the file.

*Notes:*