Subcontract: A Flexible Base for Distributed Programming

Graham Hamilton
Michael L. Powell
James G. Mitchell

SMLI TR-93-13 April 1993

Abstract:

A key problem in operating systems is permitting the orderly introduction of new properties and new implementation techniques. We describe a mechanism, subcontract, that within the context of an object-oriented distributed system permits application programmers control over fundamental object mechanisms. This allows programmers to define new object communication without modifying the base system. We describe how new subcontracts can be introduced as alternative communication mechanisms in the place of existing subcontracts. We also briefly descibe some of the uses we have made of the subcontract mechanism to support caching, crash recovery, and replication.

Sun Microsystems Laboratories, Inc.
A Sun Microsystems, Inc. Business
M/S 29-01
2550 Garcia Avenue
Mountain View, CA 94043

email addresses:
graham.hamilton@eng.sun.com
michael.powell@eng.sun.com
jim.mitchell@eng.sun.com

Subcontract: A Flexible Base for Distributed Programming

Graham Hamilton, Michael L. Powell, James G. Mitchell

Sun Microsystems Laboratories, Inc.
2550 Garcia Avenue
Mountain View, CA 94043

1 Introduction

It has become common to provide remote procedure call facilities that extend the semantics of local procedure calls to distributed systems [Birrell & Nelson 1984]. This often takes the form of remote object invocation [Almes et al 1985], [Black et al 1987].

However, rather than there being a single set of obvious semantics for all remote objects, there appears to be a wide range of possible object semantics, often reflecting different application requirements. For example, there are RPC systems that include integrated support for replication [Birman & Joseph 1987], atomic transactions [Liskov 1988], object migration [Schuller et al 1992], and persistence [OMG 1991]. But there are also RPC systems that provide only minimal features and instead concentrate on high performance [Schroeder & Burrows 1990], [Bershad et al 1990].

One possible reaction to this diversity would be to attempt to invent a single mechanism for remote objects that includes all possible features. Unfortunately the list of possible properties is continually expanding and not all features are necessarily compatible. For example, a high performance system may not want its objects to include support for persistence or atomicity. Moreover, there are often a variety of different ways of implementing a given set of semantics. Having a single RPC system prevents applications exploiting new and improved mechanisms that may better reflect their real needs.

For example, say that we wish to support object replication so as to increase reliability. We do not want client application code to need to do extra work simply to talk to replicated objects, so we would prefer to support replication underneath the covers, as part of the object invocation mechanism. But there are many different ways of implementing replication and it seems undesirable to build in support for some particular set of mechanisms while implicitly rejecting others. If an application developer discovers a more efficient way of managing replicated objects within their application then we would like them to be able to start using this new mechanism without having to change the base RPC system.

In the Spring system we have taken an approach of welcoming diversity, rather than trying to implement a single catch-all RPC mechanism. We provide a framework which makes it easy for different RPC mechanisms to work together and for implementors to add new mechanisms in a consistent, compatible, way. Replaceable modules known as subcontracts are given control of the basic mechanisms of object invocation and argument passing.

Subcontracts are separated from object interfaces and object implementations. It is easy for object implementors to either select and use an existing subcontract or to implement a new subcontract. Correspondingly, application level programmers need not be aware of the specific subcontracts that are being used for particular objects.

The rest of the paper is structured as follows: Section 2 describes related work in the distributed systems and language communities. Section 3 gives an overview of the Spring system, which is the context for this work. Sections 4, 5 and 6 describe subcontract and how it is conventionally used. Section 7 gives an example of the operation of a subcontract-based object. Section 8 describes some uses of subcontract. Finally Section 9 reflects on our experience with subcontract.

2 Related work

2.1 Language-level support for remote operations

Techniques for providing a language-level veneer for remote operations have been in use for some time [Nelson 1981], [Birrell & Nelson 1984]. Typically, a remote interface is defined in some language. Then a pair of *stubs* are generated from this interface. The *client stub* runs in one machine and presents a language level interface that is derived from the remote interface. The *server stub* runs in some other machine and invokes a language-level interface that is derived from the remote interface. To perform a remote operation, a client application programmer invokes the client stub, which *marshals* the arguments into network buffers and transmits them to the server stub, which *unmarshals* the arguments from the network buffers and calls into the server application. Similarly, when the server application returns, the results are marshalled by the server stub and returned to the client stub, which unmarshals the results and returns them to the client application.

When the arguments or results are simple values such as integers or strings, the business of marshalling and unmarshalling is reasonably straightforward. The stubs will normally simply put the literal value of the argument into the network buffer. However, in languages that support either abstract data types or objects, marshalling becomes rather more interesting.

One solution is for the stubs to marshal the internal data structures of the object and then to unmarshal this data back into a new object. This has several fairly serious deficiencies. First, it is a violation of abstraction, since the stubs have no business knowing about the internals of objects. Second, it requires that the server and client implementations of the object use the same internal layout for their data structures. Third, it may involve marshalling large amounts of unnecessary data since not all of the internal state of the object may really need to be transmitted to the other machine.

An alternative solution is that when an object is marshalled, none of its internal state is transmitted. Instead an identifying token is generated for the object and this token is transmitted. For example in the Eden system [Lazowska et al 1981], [Almes et al 1985], objects are assigned names and when an object is marshalled, then its name rather than its actual representation is marshalled. Subsequently when remote machines wish to operate on this object, they must use the name to locate the original site of the object and transmit their invocations to that site. This mechanism is appropriate for heavyweight objects, such as files or databases, but it is often inappropriate for lightweight abstractions, such as an object representing a cartesian coordinate pair, where it would have been better to marshal the real state of the object.

Finally, some object-oriented programming systems provide the means for an object implementation to control how its arguments are marshalled and unmarshalled. For example, in the Argus system [Herlihy & Liskov 1982] object implementors can provide functions to map between their internal representation and a specific, concrete, external representation. The Argus stubs will invoke the appropriate mapping functions when marshalling and unmarshalling objects so that it is the external representation rather than any particular internal representation that is transmitted.

These different solutions all either impose a single standard marshalling policy for all objects, or require that individual object implementors take responsibility for the details of transforming an object into a marshalled form.

2.2 Reflection in object-oriented languages

Some object-oriented languages provide the ability for object implementors to gain control of some of the fundamental object mechanisms. This technique is known as *reflection*. For example in the 3-KRS language [Maes 1987] objects can have meta-objects associated with them. A meta-object provides methods specifying how an object inherits information, how an object is printed, how objects are created, how message passing (that is, object invocation) is implemented, etc. 3-KRS does not however provide any control over argument passing.

By providing reflective object invocation in Smalltalk-80 it was possible to implement objects which are automatically locked during invocation and objects which only compute a value when they are first read [Foote & Johnson 1989].

We were interested in applying these notions of reflective control of fundamental object mechanisms to the particular problems of distributed computing.

2.3 Object adaptors in CORBA

The Common Object Request Broker Architecture (CORBA) [OMG 1991] defines a notion of *object adaptors*, which is based, in part, on some of our early experience with subcontract. Object adaptors provide a limited set of choices about the server-side object mechanisms. However, all object adaptors are supplied as part of the basic object machinery and it is not possible for application writers to implement new object adaptors, or for the object machinery to discover and install new object adaptors at run-time.

3 Overview of Spring

Spring is an experimental distributed environment. Its current incarnation includes a distributed operating system and a support framework for distributed applications.

Spring is focused on providing interfaces rather than simply on providing implementations. We aim to encourage the coexistence of radically different implementations of a given interface within a single system. We have reinforced this view of a strong separation of implementations and interfaces by using the object-oriented notions of data encapsulation and interface inheritance.

3.1 The interface definition language

The unifying principle of Spring is that all the key interfaces are defined in an interface definition language called IDL [OMG 1991]. This language is object-oriented and includes support for multiple inheritance. It is purely concerned with interface properties and does not provide any implementation information.

From the IDL interfaces it is possible to generate language-specific stubs. These stubs provide a language-specific mapping to the Spring interfaces. For example, in our main implementation language, C++, Spring objects are represented by C++ objects. When a method on a stub object is invoked it will either perform a local call within the current address space or forward the call to another address space, which may be on a different machine.

3.2 The Spring object model

Spring has a slightly different way of viewing objects from other distributed object-oriented systems and it is necessary to clarify this before discussing the details of subcontract.

Most distributed systems present a model that objects reside at server machines and client machines possess object references that point to the object at the server. (See Figure 1.) So clients pass around object references rather than passing around actual objects.

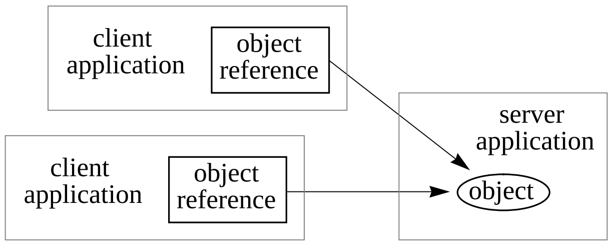

FIGURE 1. Systems distinguishing object references and objects

Spring presents a model that clients are operating directly on objects, rather than on object references. However some of these objects happen to keep all their interesting state at some remote site, so that their local state merely consists of a handle to this remote state. (See Figure 2.)

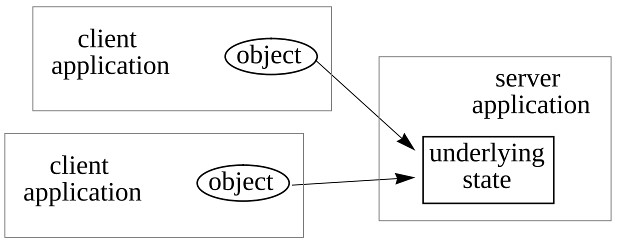

FIGURE 2. Objects in Spring

An object can only exist in one place at a time, so if we transmit an object to someone else then we cease to have the object ourselves. However, we can also copy the object before transmitting it, which might be implemented such that there are now two distinct objects pointing to the same underlying state.

For most server-based objects this distinction is mainly one of terminology. However, Spring also supports objects which are not server-based, and objects where the state of the object is split between the client and the server. In these cases it is much more convenient to regard the client as possessing the true object, rather than merely possessing a reference.

3.3 Doors

Spring applications run as separate *domains*. Each domain is an address space plus a collection of threads.

The Spring kernel provides an object-oriented inter-process communication mechanism called *doors* [Hamilton & Kougiouris 1993], which are analogous to ports in Mach [Acetta et al 1986], [Draves 1990]. A door is a communication endpoint, to which threads may execute cross address space calls. A domain that creates a door receives a *door identifier*, which it can pass to other domains so that they can issue calls to the associated door. The kernel manages all operations on doors and door identifiers, including their construction, destruction, copying and transmission. Door identifiers function as software capabilities since only the legitimate owner of a door identifier may issue a call on its associated door.

A set of network servers extend the door mechanism transparently over the network. This includes both forwarding door invocations over the network and also mapping door identifiers to and from an extended network form.

3.4 Status

Spring currently exists as a fairly complete prototype. The operating system is based around a minimal kernel, which provides basic object-oriented inter-process communication [Hamilton & Kougiouris 1993] and memory management [Khalidi & Nelson 1993A]. Functionality such as naming, paging, file systems, etc. are all provided as user-mode services outside of the basic kernel. The system also provides enough Unix® emulation to support standard utilities such as make, vi, csh, the X window system, etc. [Khalidi & Nelson 1993B]. All system interfaces are defined in IDL and all the inter-process communication uses our subcontract machinery.

4 Where subcontract fits in

A Spring object is perceived by a client as consisting of three things: 1) a *method table*, which contains an entry for each operation implied by the object's type definition; 2) a *subcontract operations vector* which specifies the basic subcontract operations described below in Section 5; and 3) some client-local private state, which is referred to as the object's *representation*.

A client employs a Spring object by invoking methods on what appears to be a C++ object. The code for this C++ object is automatically generated from an IDL interface description. This code transforms the method invocations into calls on either the object's regular method table or on its subcontract operations vector. How these methods achieve their effect is hidden from the client.

If the object is implemented by a remote server, then a typical arrangement will be that the subcontract implements the machinery for communicating with the remote server, while the method table consists of pointers to stub methods whose sole duty is to marshal the arguments into a buffer, call the subcontract to execute the remote call and then unmarshal any results from the reply buffer. Figure 3 shows the logical progression of a call to a server-based Spring object.

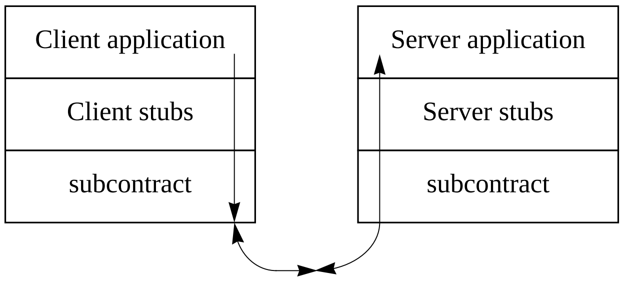

FIGURE 3. Invoking a method on a server-based object

In the remote server there will typically be some subcontract code to perform any subcontract work associated with incoming calls and some server side stub code that unmarshals the arguments for each operation and calls into the server application.

5 Basic subcontract mechanisms

To illustrate the subcontract mechanisms, we shall use as an example a subcontract called *replicon*, which is our simplest subcontract for supporting replication.

In replicon, a set of server domains conspire to maintain the underlying state associated with an object. Each server creates a kernel door (see 3.3 above) to accept incoming calls on that state. The client domains possess a set of door identifiers that they use to call through to server domains. In the case of replicon the clients are required to talk only to a single server and the servers are required to perform their own state synchronization. (Other subcontracts for replication use more elaborate rules.)

Thus a Spring object built on the replicon subcontract consists of a method table that consists entirely of stub methods, a replicon subcontract descriptor and a representation that consists of a set of kernel door identifiers, one per replica.

5.1 Client side subcontract interfaces

The principal client side subcontract operations are:

- marshal
- invoke
- unmarshal
- marshal_copy
- invoke_preamble

5.1.1 Marshal

The subcontract *marshal* operation is used by the stubs to transmit an object to another address space.

Marshal takes the current object and places enough information in a communications buffer so that an essentially identical object can be unmarshalled from this buffer in another domain. It then deletes all the local state associated with the object.

Replicon implements the marshal operation by marshalling the count of door identifiers and then marshalling each of its door identifiers in turn.

5.1.2 Unmarshal

The *unmarshal* operation is used when receiving an object from another domain. Its job is to fabricate a fully fledged Spring object, consisting of a method table, subcontract operations vector, and representation.

When a stub decides to read an object from a communications buffer, it must choose both an initial subcontract and an initial method table based on the expected type of the object. It then invokes the initial subcontract, passing it the initial method table and the buffer.

The subcontract must then fabricate an object based on the information in the communications buffer. This typically involves reading enough information from the communications buffer to be able to create a representation.

In the case of replicon, this normally involves reading a set of kernel door identifiers from the buffer and creating a representation to hold them. (Section 6 below discusses what happens in some abnormal cases.)

Finally the subcontract's unmarshal operation plugs together its own subcontract operations vector, the method table pointer and the representation to create a new Spring object.

5.1.3 Invoke

The *invoke* operation is used by the stubs to actually execute an object call, after the stubs have marshalled all the arguments. Invoke is given an argument buffer and is expected to return a result buffer.

Replicon implements the invoke operation using the kernel's door invocation mechanism. Replicon attempts to invoke each of its door identifiers in turn. If the door invocation fails due to a communications error, then replicon deletes that door identifier from its set of targets and proceeds to try the next door identifier. Otherwise, replicon returns the result of the door invocation back to the stubs.

The replicon invoke protocol also piggybacks some subcontract control information in the call and reply buffers. This is used to support changes to the replica set.

5.1.4 Invoke_preamble

The subcontract invoke operation is only executed after all the argument marshalling has already occurred. In practice it was noted that there are cases where an object's subcontract would like to become involved earlier in the process, so that it can either write some subcontract-level control information into the communications buffer or adjust the communications buffer to influence future marshalling.

For example we have some subcontracts that use shared memory regions to communicate with their servers. In this case when invoke_preamble is called, the subcontract can adjust the communications buffer to point into the shared memory region so that arguments are directly marshalled into the region, rather than having to be copied there after all marshalling is complete.

To enable the subcontract to set up any needed state, we introduced the subcontract operation *invoke_preamble*, which is called before any argument marshalling has begun.

5.1.5 Marshal_copy

Our interface definition language supports a parameter passing mode called *copy.* This mode implies that a copy of the argument object is transmitted, while the calling domain retains the original object.

This mode was originally implemented by first calling the subcontract copy operation (see 5.1.6 below) and then by calling the subcontract marshal operation on the copy. However, it was observed that this frequently led to redundant work in generating a copy that was immediately marshalled and then deleted.

The marshal_copy operation allows us to optimize this common case. It is defined to produce the effect of a copy followed by a marshal, but it is permitted to optimize out some of the intermediate steps.

5.1.6 Other client subcontract operations

The client subcontract provides operations for copying and deleting objects. This control over copy and delete is important for subcontracts that are maintaining dialogues between a server and each of its extant client objects and that may wish to notify the server of births and deaths.

The client subcontract also provides operations for performing run-time type queries and a few other minor operations.

5.2 Server side subcontract operations

Many subcontracts support client-server computing. We have described the client side view of subcontract, but for server-based objects there is also a certain amount of machinery on the server side.

On the client side, we attempt to hide the subcontract implementation from application programmers. However on the server side, we are prepared to allow server implementations to be more tightly coupled to particular subcontracts. For example a replicated subcontract may require special interfaces to the server application in order to support replication.

Thus the server side interfaces can vary considerably between subcontracts. However, there are three elements that are typically present: support for creating a Spring object from a language-level object, support for processing incoming calls, and support for revoking an object.

5.2.1 Creating a Spring object

Subcontracts must provide a way of creating Spring objects from language-level objects. This can take one of two forms.

The simplest form is that a subcontract creates a normal client side Spring object. This means that it must create some kind of communication endpoint (for example a kernel door) and fabricate a client side Spring object whose representation uses that endpoint.

However, some subcontracts provide special support for Spring objects that reside in the same address space as their server, by providing an optimized invocation mechanism for use within a single address space. When a Spring object is created using such a subcontract, it will typically fabricate an object using a special server-side subcontract operations vector and it will try to avoid paying the expense of creating resources required for cross-domain communication. When and if the object is actually marshalled for transmission to another domain, the subcontract will finally create these resources.

5.2.2 Processing incoming calls

Occasionally a subcontract will create a communications endpoint that delivers an incoming call directly to the server side stubs. More commonly, the subcontract will arrange that the incoming call arrives first in the subcontract, which then forwards the call to the stub level. This permits the server-side subcontract to maintain a dialogue with the corresponding client-side subcontract by piggybacking additional information on calls and replies.

5.2.3 Revoking an object

Occasionally a server will decide that it wishes to discard a piece of state, even though there are clients who currently have objects pointing at that state. This is particularly important for operating system services which may wish to reclaim resources without waiting for all their clients to consent. Thus typical server-side subcontracts provide a way for the server application to revoke an outstanding object. For example, this can be implemented by revoking any underlying kernel doors, which will effectively prevent further incoming kernel calls.

6 Subcontract conventions

6.1 Compatible subcontracts

Clearly it is desirable for different objects to have different subcontracts. In particular, two objects which are perceived by the client application as having the same type may in fact have different subcontracts.

For example, the standard type *file* is specified to use a simple subcontract called *singleton*. The type *cacheable_file* is a subtype of file, but instead uses the *caching* subcontract. So what happens when we send an object of type cacheable file where an object of type file is expected? Clearly if the receiver insists on unmarshalling the caching object as though it were a singleton, then it is going to be disappointed. For each type we can specify a default subcontract for use when talking to that type, but how do we cope when we actually need to use a different subcontract?

We solve this problem by introducing the notion of *compatible subcontracts*. Subcontract A is said to be *compatible* with subcontract B if the unmarshalling code for subcontract B can correctly cope with receiving an object of subcontract A.

The normal mechanism we use to implement compatible subcontracts is to include a subcontract identifier as part of the marshalled form of each object.

So a typical subcontract unmarshal operation starts by taking a peek at the expected subcontract identifier in the communications buffer. If it contains the expected identifier for the current subcontract, then the subcontract goes ahead and unmarshals the object. However if the unmarshal operation sees some other value then it calls into a registry to locate the correct code for that subcontract and then calls that subcontract to perform the unmarshalling.

6.2 Discovering new subcontracts

A program will typically be linked with a set of libraries that provide a set of standard subcontracts. However at run-time the program may encounter objects which use subcontracts that are not in its standard libraries.

We provide a mechanism to map from subcontract identifiers to library names and we support dynamic linking of libraries to obtain new subcontracts.

Say that a domain is expecting to receive an object of type file, using the singleton subcontract, but we instead send it an object of type replicated_file using the replicon subcontract. The singleton unmarshal operation will discover that it is dealing with a different subcontract and it will call into the domain's subcontract registry to find the correct subcontract code. The registry will discover that there is currently no suitable subcontract loaded, but it will then use a network naming context to map the subcontract identifier into a library name (e.g. replicon.so) and it will then dynamically link in that library to obtain the subcontract. Unmarshalling can then continue, using the newly linked subcontract code.

So even though the program had no concept of replicated objects and was not initially linked with any libraries that understood replicated objects, we were able to dynamically obtain the right code to talk to a replicated file object.

This mechanism means that it is possible to add new subcontracts and use them to talk to old applications without changing either the old applications or the standard libraries, provided only that we can make a suitable subcontract library available at run-time to the old programs.

Many domains, particularly systems servers, are reluctant to simply run some random dynamic library code nominated by a potentially malicious client. So, for security reasons the dynamic linker will only load libraries that are on a designated directory search-path of trustworthy locations. So it typically requires intervention by a privileged system administrator to install a new subcontract library in a standard directory which most domains will have on their search paths.

6.3 Subcontracts and object types

Subcontracts affect objects' semantics. Different implementations of a type such as file may use different subcontracts which provide substantially different semantics. It is necessary to provide a way for application programmers to determine the real semantics of particular objects.

Fortunately, this is easy to express in an object-oriented type system. We define standard base types with core semantics for various abstractions. Other types inherit from these standard types and add additional semantics, such as replication. It is the responsibility of each object implementor to select both a type for their object and a subcontract that meets the semantic commitments of that type. Clients may attempt to narrow an object's type at run-time to determine if a given object of a statically determined type, such as file, actually supports a subtype with richer semantics, such as replicated file.

7 The life-cycle of a Spring object

It may be useful to review the various points where subcontract is used during the lifetime of a particular object.

Say that a fileserver FS is exporting objects of type file, using the *simplex* subcontract. The simplex subcontract is a very simple client-server subcontract, using a single kernel door identifier to communicate with the server.

The fileserver starts with some internal state describing a file. It then uses the server-side code of the simplex subcontract to create a Spring object. At this point the server application must specify a method table, a set of server side stubs and a pointer to the internal file state. The simplex server code creates a kernel door to act as the entry point for this particular piece of file state and returns a new Spring object. This object consists of a pointer to the simplex subcontract, a pointer to the given method table, and a pointer to a representation consisting of a door identifier. See Figure 4.

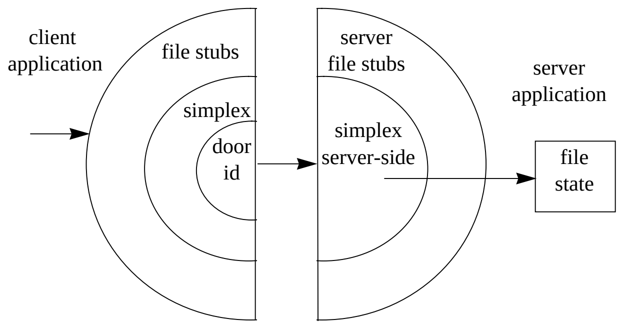

FIGURE 4. A file object using the simplex subcontract

Initially, the new file object is in the same address space as its creator. But let's say it gets passed to another address space as a result of an operation on a file_system object. When the file object is transmitted, the server-side file_system stubs will invoke the object's subcontract to marshal the object into a buffer. Simplex will deposit both its own subcontract ID and the object's door identifier in the buffer. At the receiving end, the client-side file_system stubs will attempt to unmarshal a file object. Since file's default subcontract is singleton, they will initially call the singleton unmarshal method. However singleton will quickly realize that the buffer holds a subcontract ID for another subcontract and will use the subcontract registry to locate the code for simplex and ask simplex to perform the unmarshalling. Simplex will read the subcontract ID and door identifier and create an appropriate Spring object.

When a client attempts to invoke a method on the file object, they will invoke a method from the object's method table, which will be implemented by the client-side stubs. These stubs will start by calling the file object's subcontract invoke_preamble operation. In this case the simplex invoke_preamble does nothing and simply returns. After marshalling the arguments, the stubs will invoke the object's subcontract invoke operation. In this case the simplex invoke operation uses the kernel's door IPC mechanism to call through to the server. This propagates the call to the server-side simplex code, which forwards the call to the server-side file stubs and thence up into the server application. The reply similarly flows back through the server-side stubs, the server-side simplex code, the kernel, the client-side simplex code and back into the client-side stubs. Those stubs unmarshal any results and return them to the client application.

Another event that may befall our file object is that a client may want to copy the file object so they can send the copy on to some other application. (Note that this is a shallow copy, in which both the original and the copy refer to the same underlying state.) This can be done by invoking the object's subcontract copy method. In this case the simplex copy method will fabricate a new Spring object by simply copying its method table pointer and subcontract pointer and by asking the kernel to copy its door identifier.

Finally, alas, the client application may be finished with the file object. At this point it can invoke the object's subcontract consume method. In this case, simplex consume will simply tell the kernel to delete the door identifier and return. Later, when all active door identifiers for the server door have been deleted, the kernel will notify the door's target, in this case the server-side simplex code, so that it can clean up.

Notice how the simplex subcontract has been involved in all the key events of the object's life: its birth, its reproduction, its death, its transfer between address spaces, and finally and most importantly, simplex has been involved whenever any invocations occurred on the object. Simplex chooses to make only fairly limited use of these control points. Other subcontracts are rather more venturesome.

8 Example subcontracts

We have already described the replicon subcontract (in Section 5) and the simplex subcontract (in Section 7). We now provide a short overview of some other interesting subcontracts we have developed. For brevity, we merely provide simplified outlines of their key features and omit descriptions of error conditions and special optimizations.

8.1 The cluster subcontract

The simplex subcontract uses a distinct kernel door for each piece of server state that may be exposed as a separate Spring object. Since the kernel imposes a capability-like security model on door identifiers, this is a suitable implementation for any objects that are used to grant access to distinctly protected system resources. However some servers export large number of objects where if a client is granted access to any of the objects, it might as well be granted access to all of them. In this case a subcontract can reduce system overhead by using a single door to provide access to a set of objects.

The *cluster* subcontract supports this notion. Each cluster object is represented by the combination of a door identifier and an integer tag. The cluster invoke_preamble and invoke operations conspire to ship the tag along to the server when performing a cross-domain call on the door. The server-side cluster subcontract code uses this tag to dispatch to a particular object.

8.2 The caching subcontract

When a server is on a different machine from its clients, it is often useful to perform caching on the client machines. So when we transmit a cacheable object between machines, we'd like the receiving machine to register the received object with a local cache manager and access the object via the cache.

The caching subcontract, which was originally developed for the Spring file system [Nelson et al 1993], provides this functionality. The representation of a caching object includes a door identifier D1 that points to the server, a door identifier D2 that points to a local cache, and the name of a cache manager.

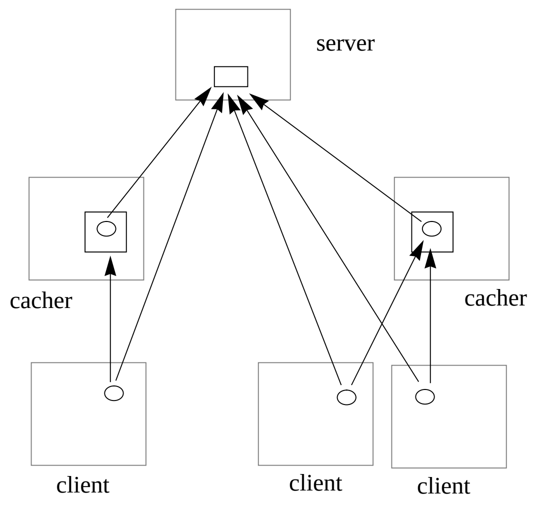

FIGURE 5. Three clients using the caching subcontract

When we transmit a caching object between machines, we only transmit the D1 door identifier and the cache manager name. The caching unmarshal code resolves the cache manager name in a machine-local context to discover a suitable local cache manager and then presents the D1 door identifier to the local cache manager and receives a new D2. Whenever the subcontract performs an invoke operation it uses the D2 door identifier. So all invocations on a cacheable object go to an appropriate cache manager on the local machine. (See Figure 5.)

8.3 The reconnectable subcontract

Some servers keep their state in stable storage. If a client has an object whose state is kept in such a server, it would like the object to be able to quietly recover from server crashes. Normal Spring door identifiers become invalid when a server crashes, so we need to add some new mechanism to allow a client to reconnect to a server.

The reconnectable subcontract uses a representation consisting of a normal door identifier, plus an object name.

Normally the recoverable subcontract's invoke code simply does a kernel door invocation on the door identifier. However, if this fails, the subcontract instead attempts to resolve the object name to obtain a new object and retries the operation on that. It retries periodically until it succeeds in getting a new valid object.

8.4 Future directions

In conjunction with some related projects, we are investigating a number of new subcontracts. One is to develop a subcontract that lets video objects encapsulate a specific network packet protocol for live video. Another is to develop a subcontract that transfers scheduling priority information between clients and servers for time-critical operations. Another is to transfer control information for atomic transactions at the subcontract level.

As base system implementors, we are only too happy that these techniques appear suitable for implementation by third party experts as new subcontracts and will not require modifications to the base RPC system.

8.5 Motivation for examples

This paper is not about replication or caching or crash recovery. What we hope to have established in this section is that the basic subcontract interfaces are sufficiently general that they can accommodate a wide range of possible solutions, while still providing a uniform application model.

9 Reflections on subcontract

One of the reasons that subcontract is effective is because it separates out the business of implementing objects from implementing object mechanisms. Subcontract implementors provide a set of interesting subcontracts that enable object implementors to chose from a range of different object policies without requiring that every object implementor must become familiar with the details of the object implementation machinery.

The set of operations that subcontract provides appear to be the right levers for obtaining control within a distributed environment. By design, all the key actions taken on remote objects will involve the object's subcontract in one way or another. Subcontract provides an effective way for plugging in different policies for different objects,

In practice subcontract has succeeded in reducing the functionality that must be provided by the base system. We have been able to implement a number of interesting new subcontracts without requiring any new facilities in the base system, including a number of new subcontracts (such as the caching subcontract) which we did not envisage in the original subcontract design.

9.1 Subcontracts versus specialized stubs

Our current system maintains a complete separation between stubs and subcontracts. Any set of stubs can work with any subcontract and vice versa. An alternative solution would have been to define a similar framework, but provide the subcontract functionality directly in the stubs, by implementing different sets of stubs for different subcontracts. There were two reasons for implementing subcontracts as separable modules.

First, we wanted subcontract writers to be able to develop new communication mechanisms without having to undertake the reasonably daunting task of developing (or modifying) a stub generator. This argues for a logical separation between stubs and subcontracts. However, this requirement could be satisfied by permitting subcontracts to exist as in-line procedures, which could be in-lined into general-purpose stubs.

Second, we wanted to avoid the need for client applications to load specialized stubs in order to talk to individual objects. If a program expects to receive an object of type foo through an RPC interface, then it can arrange to have the general-purpose foo stubs available and plug them together with appropriate subcontracts as needed. However, it is a much more difficult proposition to have all possible flavors of specialized foo stubs available. We expect the set of subcontracts to be much smaller than the set of types. While we are prepared to accept some moderate administrative hurdles for making a new subcontract available via dynamic linking, we are reluctant to pay this cost for all new combinations of types and subcontracts.

As a future direction, we are interested in providing specialized stubs for some particularly popular and performance-critical combinations of types and subcontracts. We would still keep the general purpose stubs available for these types, so as to be able to deal with different subcontracts, but when we were lucky enough to receive an object that happened to be of the right type and subcontract we would be able to use the specialized stubs.

9.2 The OS environment

Our subcontract work happens to have occurred in the context of a distributed operating system that provides an object-oriented IPC facility. Given this IPC mechanism, we have chosen to use it for a variety of our subcontracts and also to acknowledge its existence in our basic marshalling machinery. However this is mere happenstance. In different operating system environments it may be appropriate to use different IPC machinery for subcontracts or to operate at a lower level and build exclusively on raw network packets. Even in our environment it is possible to mix the use of the kernel's door mechanism with the use of raw IP packets, should one desire.

9.3 Performance overheads of subcontract

Subcontract has some performance costs over simpler RPC systems. Each object invocation always requires an additional two indirect procedure calls from the stubs into the client subcontract and typically requires a third indirect call from the server-side subcontract into the server stubs. Transmitting an object requires an extra pair of calls for marshalling and unmarshalling and typically also involves the cost of marshalling and unmarshalling a subcontract ID. However in our system we estimate that these costs add less than 2 microseconds (on a SPARCstation 2) to the costs for a minimal remote call or to the costs of a simple object transmission.

More significantly, subcontract makes it harder for the stubs to be optimized for particular sets of arguments and results, as the subcontract may be adding an indeterminate amount of data to the call or reply buffer. This is one of the factors causing us to investigate specialized combinations of stubs and subcontracts. (See Section 9.1 above.)

Individual subcontracts can be as efficient or as profligate as they wish. For example, the caching subcontract adds a significant overhead to object unmarshalling to achieve the useful optimization of local caching.

10 Conclusion

We have shown that it is possible to build distributed systems where a variety of different object mechanisms can co-exist. This enables the use of a range of lightweight and heavyweight mechanisms within the same system and avoids the need to choose a single mechanism for all occasions. Instead individual object implementations can select those features that they require and that they are prepared to pay for.

These different object mechanisms are all on a par with one another. There is no penalty for application writers who wish to use new subcontracts in place of the standard ones. A new type such as *replicated_file*, using a new subcontract, such a replicon, can be used in all the places where the earlier, simpler, type *file* was used.

It is therefore possible for object implementors to provide a wide range of different possibilities for security, for performance, for robustness, for scope of access, all behind the same set of application-visible interfaces.

11 Acknowledgments

We would like to acknowledge the contributions of Jon Gibbons, Ted Goldstein, Tommy Joseph, Peter Kessler, Yousef Khalidi, Panos Kougiouris, Mike Nelson, and Sanjay Radia to the evolution and implementation of subcontract and to the development of specific subcontracts. We would also like to thank Bob Hagmann, Roger Hayes, Peter Kessler, and Mike Nelson for their helpful comments on earlier drafts of this paper.

12 References

[Acetta et al 1986] M. Acceta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian & M. Young. "Mach: A New Kernel Foundation For UNIX Development." Summer USENIX Conference, Atlanta, June 1986.

[Almes et al 1985] Guy T. Almes, Andrew P. Black, Edward D. Lazowska, and Jerre D. Noe. The Eden System: A Technical Review." IEEE Transactions on Software Engineering, 11(1), January 1985.

[Bershad et al 1990] B. N. Bershad, T.E Anderson, E. D. Lazowska, and H. M. Levy. "Lightweight Remote Procedure Call." ACM Transactions on Computer Systems 8(1), February 1990.

[Birman & Joseph 1987] K. P. Birman and T. A. Joseph. "Exploiting Virtual Synchrony in Distributed Systems." Proc. of the 11th Symposium on Operating Systems Principles, Austin, Texas, December 1981.

[Birrell & Nelson 1984] A. D. Birrell and B. J. Nelson. "Implementing Remote Procedure Calls." ACM Trans. on Computer Systems, 2(1), February 1984.

[Black et al 1987] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. "Distribution and Abstract Types in Emerald." IEEE Trans on Software Engineering, 13(1), January 1987.

[Draves 1990] R. Draves. "A Revised IPC Interface." Proc. of the Usenix Mach Workshop, Burlington, Vermont, October 1990.

[Foote & Johnson 1989] Brian Foote and Ralph E. Johnson. "Reflective Facilities in Smalltalk-80." Proc. of the Conference on Object-Oriented Systems, Languages, and Applications, October 1989.

[Hamilton & Kougiouris 1993] G. Hamilton & P. Kougiouris. "The Spring nucleus: A microkernel for objects." Proc of the 1993 Summer Usenix conference, Cincinnati, June 1993.

[Herlihy & Liskov 1982] M. P. Herlihy and B. Liskov. "A value transmission mechanism for abstract data types." ACM Trans. on Programming Languages and Systems, 4(4), October 1982.

[Khalidi & Nelson 1993A] Y. A. Khalidi and M. N. Nelson. "The Spring Virtual Memory System." Sun Microsystems Laboratories, SMLI TR-93-9, March 1993.

[Khalidi & Nelson 1993B] Y. A. Khalidi and M. N. Nelson. "An Implementation of UNIX on an Object-oriented Operating System." Proc. of the 1993 Winter Usenix Conference, San Diego, January 1993.

[Lazowska et al 1981] E. Lazowska, H. Levy, G. Almes, M. Fischer, R. Fowler, and S. Vestal. "The Architecture of the Eden System." Proc. of the 8th Symposium on Operating Systems Principles, Pacific Grove, California, December 1981.

[Liskov 1988] B. Liskov, "Distributed programming in Argus." Communications of the ACM, 31(3), March 1988.

[Maes 1987] Pattie Maes. “Concepts and Experiments in Computational Reflection.” Proc. of the Conference on Object-Oriented Systems, Languages, and Applications, October 1987.

[Nelson 1981] B. J. Nelson. “Remote Procedure Call.” Tech report CSL-81-9, Xerox Palo Alto Research Center, Palo Alto, California, 1981.

[Nelson et al 1993] M. N. Nelson, Y. A. Khalidi, and P. W. Madany. “The Spring File System.” Sun Microsystems Laboratories, SMLI TR-93-10, March 1993.

[OMG 1991] Object Management Group. “Common Object Request Broker Architecture and Specification.” OMG Document Number 91.12.1.

[Schroeder & Burrows 1990] M. D. Schroeder and M. Burrows. “Performance of Firefly RPC.” ACM Transactions on Computer Systems 8(1), February 1990.

[Schuller et al 1992] P. Schuller, H. Hartig, W. E. Kuhnhauser, and H. Streich. “Performance of the BirliX Operating System.” Proc. of the Usenix Workshop on Micro-kernels and Other Kernel Architectures, Seattle, April 1992.