Separating Presentation from Interface in RPC and IDLs

Bryan Ford Mike Hibler Jay Lepreau
University of Utah

Abstract

In RPC-based communication, we term the **interface** *the set of remote procedures and the types of their arguments; the* **presentation** *is the way these procedures and types are mapped to the target language environment in a particular client or server, including semantic requirements. For example, presentation includes the local names assigned to RPC stubs, the physical representation of a logical block of data (e.g., in-line, out-of-line, linked blocks), and trust requirements (e.g., integrity, security). In existing systems, the presentation of a given RPC construct is largely fixed.*

Separating presentation from interface, both in the interface definition language (IDL) itself and in the RPC implementation, is the key to **interoperability***, with many benefits in the area of elegance, as well. This separation and resulting cleanliness makes it manageable to generate specialized kernel code paths for each type of client-server pair. This is a key element of* **end-to-end optimization***. The separation should also allow the integration of disparate RPC optimization techniques, such as those applied in LRPC[2] and fbufs[6], into a single system, in a uniform and fully interoperable way. In initial work we demonstrate a variant of threaded code generation and two presentation-based optimizations, transparently activated by the RPC system. Each of these optimizations speeds up local RPC by approximately 25%.*[1]

1 Introduction

Remote procedure call (RPC) and interface definition language (IDL)[10] compilers were first introduced primarily as programming shortcuts: tools to help automate the production of networking code. However, with the growth of client-server computing and large object-oriented distributed systems, RPC interfaces have changed from a programming convenience to abstractions useful in themselves to promote software modularity and interoperability. Entire software systems, such as Spring[9], are designed and built in terms of the RPC interfaces between their components. IDLs have been elevated from a shorthand language for networking code to a *formalization* of interfaces between modules.

[1] Contact author: Jay Lepreau, `lepreau@cs.utah.edu`

However, the fundamental design of the IDLs and RPC implementations we use today, even the IDLs and object invocation used in modern object-oriented systems, still reflect the original purpose of RPC. Our solution to this mismatch revolves around a basic concept: a notion of *presentation* defined separately from RPC interface.

1.1 Presentation

In a typical RPC implementation, client and server stubs, written automatically by an IDL compiler, cooperate with an underlying data transport mechanism to encapsulate communication across protection and network boundaries.[2] The stubs "present" interprocess communication (IPC) in a convenient, high-level form, easily accessible from the client or server's general-purpose language. The exact linguistic rules by which the stubs interact with application code, and the RPC-related semantic requirements of the application, are what we call the *presentation*.

In conventional RPC systems, the presentation of an RPC interface for a particular target language is largely fixed, defined either explicitly in the RPC system specification, or implicitly by the stubs produced by the IDL compiler. For example, if an RPC involves transmitting a block of data, the RPC stubs expect the data to be provided in a certain way, such as in a continuous buffer with its length specified as an additional argument to the stub.

1.2 Flexible Presentation

We believe that much greater presentation flexibility can be provided by the IDL, and that it can be efficiently supported in an optimized microkernel RPC implementation. A flexible presentation enhances *interoperability* and should bring a number of benefits: (i) it makes IDLs simpler by separating out presentation constructs; (ii) it makes RPC interfaces "narrower" by eliminating the need for multiple variants of operations that differ only in presentation; (iii) it makes the programmer's job easier by offloading more of the work of using RPC from the general-purpose language to the RPC system; (iv) it gives the RPC system as a whole (both the IDL compiler and the underlying IPC mechanism) more opportunity for IPC special-

[2] While RPC is usually synchronous, the concepts described here apply to any flavor of IPC that allows a language layer above it, including asynchronous communication and even multicast.

ization and optimization by providing additional semantic information in declarative form.

In the rest of this paper we first discuss examples of presentation and related work, in Section 3 we examine the IDL aspects of separating presentation, and in Section 4 we do the same for RPC and our implementation.

2 Related Work and Examples

The term "presentation" comes from the OSI networking model, of course, which defines a *presentation layer* responsible for providing applications with data in the locally-preferred representation. The primary presentation emphasis in the OSI model is on data format and exclude semantic attributes, whereas we are concerned with both types.

We know of two IDLs in which the notion of separate presentation exists to some degree. DCE IDL's "Application Configuration File" allows specification, separate from the interface definition file, of a few attributes such as binding protocol, error condition treatment, and data type equivalence. The Concert system's[1] "endpoint modifier" supports more presentation attributes. Since Concert does not have a separate IDL, the endpoint modifier is automatically generated by each language's compiler and only handles the vagaries of the language mapping.

The Subcontract work[9], an elegant and general method for extending and specializing object communication mechanisms and semantics, is largely complementary to our work. Their Spring system has separate stub, subcontract, and kernel levels, with the language-level stubs and kernel IPC mechanism remaining entirely generic, and all specialization restricted to the subcontract layer. Therefore, in the general case, the exact details of presentation to the application code cannot be specified, varied, or controlled. However, when an *object* is a parameter to a call, that object's subcontract can specialize the object's marshalling and a few other aspects of presentation. However, it apparently cannot specialize these based on which method of the target object is being invoked, losing much of its potential power. Also, end-end optimization is not possible since the kernel IPC mechanism is not specialized.

Our language model for presentation specification is based on the general concept of subtyping, found in object-oriented languages as inheritance. We introduce the concept of "presentation subtypes," which are types relevant only to the local scope of a particular software module; the boundary between interface and presentation types is the boundary at which type mismatches no longer affect interoperability. Also, in some sense we are separating "implementation from interface," where the "implementation" is our "presentation" of the interface. However, our initial exploration of the deeper language issues shows that the issue is complex, as Cardelli[3] discusses, and is worthy of a research effort in its own right.

Clark et al[5] emphasize the importance of optimizing the presentation layer in traditional networking, showing that it can dominate processing time. They also emphasize that for performance reasons, the necessity of non-contiguous data location in the recipient is a critical architectural constraint. This can occur not only in RPC, where each parameter is scattered in memory, but in integrated-layer processing of stream-based protocols, due to dropped packets. Their emphasis is on the latter, ours the former. Our optimization effort applies the end-to-end argument[14], which emphasizes the importance, in a multitude of domains, of considering the total path in design and implementation.

Operating system interfaces are replete with special calls whose only purpose is to accommodate slight variance in presentation. For example, for performance reasons OSF added a number of system calls to the Mach kernel, differing from other calls only in presentation. `Device_read_overwrite`, `mach_msg_overwrite`, `vn_read_overwrite`, and `vm_remap` only modify memory allocation semantics of parameters. OSF has also added new IDL attributes which by their nature affect only presentation, but since the concepts are not separated, affect the entire interface: `PhysicalCopy`, `OverWrite`, and `SameCount`. That these were found necessary shows the importance of presentation in real-world situations, but all of these calls and options can be subsumed by our design.

3 Presentation in IDLs

In theory, there is exactly one interface definition file per defined interface, and all clients and servers supporting that interface use that one file. However, in practice it is often necessary to effect minor changes to the standard interface, in order to support particular clients and servers. For example, in the Mach/MIG/OSF-1 environment such changes are made for a number of reasons. Some of them are to avoid naming conflicts by renaming the generated stub routines, to specify user-defined marshaling and unmarshaling functions to be invoked by the stubs on certain data types, to modify the IDL compiler's behavior based on whether the generated stubs will be used inside or outside of the kernel, and to provide the server with message sequencing information.

These local deviations do not affect the information transferred across the RPC interface, only the way in which the RPC stubs interact with the local client or server implementation: in other words, the presentation.

What is needed is a way to specify presentation separately from interface so that the former can be varied while the latter remains stable. Since existing IDLs have no direct support for this separation, ad hoc solutions are used instead. Typically, modifying the interface's presentation is accomplished either by maintaining a variant copy of the IDL file, or by embedding special conditional constructs (#ifdef's) in the global interface definition file and using externally-specified actions (makefile rules) to select the behavior of a particular run of the IDL compiler. These often extensive rules and conditions (over 400 lines in the standard Mach system) are essentially presentation definitions. However, because the presentation is provided in a completely ad hoc way, its use and maintenance is infamously difficult. These awkward mechanisms are nevertheless used extensively, demonstrating the inherent need to control presentation separately from interface.

3.1 Adding Presentation Support to an IDL

To demonstrate the benefits of fully supporting separate specification of presentation and interface, we are creating a new Mach RPC system, based on the CORBA IDL[13], Sun's freely-available IDL compiler, our own IDL compiler back-end, and an extension of our migrating threads RPC mechanism[7].

An interface specification is generally intended to be independent of the languages used to write its client programs. However, presentation specification is fundamentally language-specific, because it defines the interaction between client or server code and the IPC system (RPC stubs). For example, an attribute specifying data buffer deallocation semantics would make sense in C, but not in Lisp, which has automatic storage reclamation. Therefore, it is important to cleanly separate interface and presentation support, both in the IDL syntax and in its compiler implementation.

In the CORBA IDL, the basic unit of interface definition is the *type*. Both concrete data types such as integers and structures, and entire interfaces (sets of RPC declarations), are considered types. In our extended version of the CORBA IDL, defining presentation for already-defined types involves creating *presentation subtypes* of the "pure" interface types. A presentation subtype is invisible to all external (remote) software components, appearing identical to the pure base type it was derived from. However, it is distinct in the view of the local program for which the presentation is defined. For example, in Figure 1, two local presentation subtypes, `SimpleData` and `FbufData`, specify two different presentations for the pure interface type `DataBlock`. All three types are distinct and possibly incompatible within the client or server using them, but programs on the "other end" of an RPC interface only see the base type, `DataBlock` (or their own independent presentations of that base type).

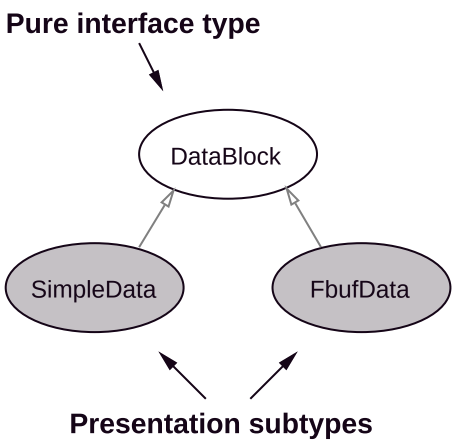

Figure 1: Presentation Subtypes

4 Presentation and Interface in RPC Implementation

Many powerful techniques for optimizing IPC have been demonstrated, such as data transfer with page remapping[4, 15], copy-on-write, pairwise shared memory buffers[2], and memory buffers group-wise shared across full data paths, leveraging weakened semantics when possible[6]. These techniques impose various restrictions on the situations in which they can be used and on the way the client and server code must be written. They give up generality in return for better performance in specific common cases.

None of these techniques, used alone, will result in an IPC system that is both fast in the common cases and general enough to work throughout a diverse distributed system. Only by integrating a variety of these techniques together in one RPC system, taking advantage of the strengths of each in their applicable areas, can a truly general-purpose, high-performance RPC system be built. Finding ways to make optimizations work together has become a more important challenge than simply finding optimizations. It is not necessary for every possible combination to be fast (only the common cases have to be fast), but every possible combination must *work*.

4.1 How Presentation Helps

One of the primary reasons integrating optimized RPC mechanisms is difficult is because each has different expectations and requirements in terms of presentation. For example, fbufs can provide extremely low-overhead data transfer across an unlimited number of protection boundaries, but for optimal performance it requires all participating clients and servers to handle data blocks scattered through multiple disjoint buffers, as well as knowledge of weakened semantics when they exist (e.g. mutable data). Because this presentation does not match the "default" presentation assumed by typical RPC systems (i.e. continuous data buffers, completely restricted access), and the IDL does not provide any way to select between the two presentations locally, the fbufs optimizations can only be supported by adding a more-or-less separate, noninteroperable "subsystem" to the basic RPC facility. The lack of presentation support means that a software component's *interface*, and not just its presentation, must be modified in order to allow the optimization to be applied, thereby making the program incompatible with other programs that do not use fbufs.

By allowing presentation to be explicitly declared in an IDL separately from the interface, the RPC system can enable individual IPC optimizations such as LRPC and fbufs on a connection-by-connection basis, depending on the presentations of the two endpoints. Since the presentations of the endpoints are independent from the point of view of the programs using the RPC system, only the basic interfaces must match in order to guarantee interoperability. IPC will be faster if the presentations match more closely, but it will always work. In summary, we believe that the separation of presentation from interface, and the addition of more declarative presentation flexibility, will aid IPC optimization by providing a generalized framework for combining a variety of optimization techniques in a fully interoperable way.

Implementation Complexity

If I is the number of "interface variants" supported (procedures, data types, etc.), and any interface variant can appear on each of two endpoints with P "presentation variants," the implementation complexity could be as large as IP^2. However, in practice, presentation variants are largely orthogonal to interface variants: for example, the way a block of memory is transferred is largely independent of what type of data the block holds. This reduces the complexity to the order of $I+P^2$. Finally, it is always possible to define some kind of *canonical format* for IPC data transfer, depending only on interface and not on presentation. The two "sides" of the RPC interface then become independent, each converting data to the common format, resulting in a basic complexity of only $I + P$.

Supporting sophisticated presentation-based optimizations of course requires deviating from the canonical format, but only the few common-case "cells" of the $P \times P$ matrix need to be optimized. The canonical format provides interoperability, while the common-case optimizations provide performance. Therefore, this kind of RPC system should be feasible, although still not "easy."

Basic Implementation Requirements

There are two main aspects to an RPC implementation with full presentation support. First, the IDL compiler must generate *type signatures* in addition to the normal stub routines. A type signature contains both interface and presentation information for a particular software component. Second, when a communication channel is initialized (bound) at run time, the type signatures of each endpoint must be available to the RPC mechanism. The RPC mechanism can then enable any IPC optimizations whose constraints are satisfied by the type signatures.

4.2 Prototype Implementation

We have implemented a prototype that embodies all of the basic characteristics of a flexible-presentation RPC system using binding-time path specialization. We demonstrate the value of such a system by measuring base performance and two classes of presentation-based optimizations, which yield additional improvement. In our implementation, the Mach kernel does most of the work of marshaling and unmarshaling data as well as transferring it across protection boundaries.

Before a Mach port can be used for communication, as part of binding the client and server must each *register* their respective type signatures with the kernel. The client's signature is attached to the *send right*, or object reference, while the server's signature is attached to the *port*, which is the kernel's representation of the object itself. Multiple clients that have references to a single server object can each register a different client presentation.

Once both the client and server type signatures are available, the kernel checks them against each other, verifying that the interfaces are compatible and searching for optimizations made possible by the specified presentations. It then builds a *combination signature*, a special type signature based on both input signatures. The combination signature is completely kernel-private and acts as a cache which keeps track of previously implemented RPC paths. Thus, the kernel only has to compare registered type signatures once; successive uses of a particular RPC path require no expensive computation. In fact, the combination signature include a block which threads[11] together small blocks of code which perform key parts of the RPC, such as register saving and restoring. It is important to realize that the binding time work needn't be done by the kernel; a privileged user process could equally well do it.

4.3 Results

Tests were done on a 66MHz HP 730 (PA-RISC). The times include all RPC costs, including both user-level stub code and kernel processing.

Basic threaded code performance: Null local RPC now takes 4.6 μsecs, and passing one 32-byte `in` parameter takes 5.9 μsecs. This null RPC is now nine times as fast as traditional Mach RPC and more than twice as fast as our migrating threads implementation[7] which relied on the traditional Mach message format, RPC is now twice as fast as system calls on native monolithic operating systems (BSD, HP-UX). This is because the latter are not, in general, specialized, but must set up exception handlers and other rarely used functions.

This excellent base performance was achieved through a range of optimization techniques (passing parameters in registers, migrating threads, avoiding register state saving and restoring) [12, 8, 7], as well as specialized code. With full knowledge of both client and server presentation, up into user code, more optimizations are possible on non-null RPCs.

Relaxing Mach's unique-name requirement: The standard Mach IPC system always enforces the semantic requirement that all references to a particular port from a particular protection domain (*task*) have only a single name within that task. This requirement is required in some situations, such as authentication, but is unnecessary for simple object invocation and slows down transfer of object references from one task to another. The single-name requirement is clearly a presentation feature, since it only affects the *appearance* of a port locally within a task. Leveraging the presentation semantic constraint of not needing unique port names resulted in a performance improvement of 24% when passing a single port (32.4 μsecs reduced to 24.7 μsecs).

Varying Trust Parameters: The trust relationships between clients and servers also count as "presentation" for our purposes, because they are useful in optimizing RPC but should not affect interoperability or the basic RPC interface. Therefore, in our RPC system, we allow each "side" of a connection to specify the degree to which it trusts the other side. Three levels of trust are provided: no trust, trust of confidentiality but not integrity, and full trust of both integrity and confidentiality. The middle level would typically apply between processes owned by the same user, which want to maintain protection boundaries but have no information to hide from each other. The full trust level could be used by clients communicating with privileged servers, such as a Unix single-server. We implemented relaxed trust levels by requiring less register saving, restoring, and clearing on the RPC path. Overall, we achieved 30% speedup with full trust, with intermediate trust gaining 8%.

5 Summary

Keeping presentation separate from interface is an important design element for both IDLs and communication systems. When done right, it provides enhanced cleanliness and interoperability, while offering the potential for end-to-end optimization of the entire communication path. This potential is due to the extension into user space of the kernel-known path, as well as the applicability of disparate IPC optimizations.

References

[1] J. S. Auerbach and J. R. Russell. The Concert signature representation: IDL as an intermediate language. In *Proc. of the Workshop on Interface Definition Languages*, pages 1–12, Jan. 1994.

[2] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. *ACM Trans. Comput. Syst.*, 8(1):37–55, Feb. 1990.

[3] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, Dec. 1985.

[4] D. R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, Mar. 1988.

[5] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the SIGCOMM '90 Symposium*, pages 200–208, 1990.

[6] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proc. of the 14th ACM Symposium on Operating Systems Principles*, pages 189–202, 1993.

[7] B. Ford and J. Lepreau. Evolving Mach 3.0 to a migrating thread model. In *Proc. of the Winter 1994 USENIX Conference*, Jan. 1994.

[8] G. Hamilton and P. Kougiouris. The Spring nucleus: a microkernel for objects. In *Proc. of the Summer 1993 USENIX Conference*, pages 147–159, Cincinnati, OH, June 1993.

[9] G. Hamilton, M. L. Powell, and J. J. Mitchell. Subcontract: A flexible base for distributed programming. In *Proc. of the 14th ACM Symposium on Operating Systems Principles*, pages 69–79, 1993.

[10] M. Jones, R. Rashid, and M. Thompson. Matchmaker: An interface specification language for distributed processing. In *Proc. of the ACM Symp. on Principles of Programming Languages*, Jan. 1985.

[11] P. Klint. Interpretation techniques. *Software — Practice and Experience*, 11(9):963–973, Sept. 1981.

[12] J. Liedtke. Improving IPC by kernel design. In *Proc. of the 14th ACM Symposium on Operating Systems Principles*, Asheville, NC, Dec. 1993.

[13] Object Management Group. The Common Object Request Broker: Architecture and specification. Draft 10 Rev 1.1 Doc # 91.12.1, OMG, Dec. 1991.

[14] J. Saltzer, D.P.Reed, and D.D.Clark. End to end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288, Nov. 1984.

[15] S.-Y. Tzou and D. Anderson. The performance of message passing using restricted virtual memory mapping. *Software — Practice and Experience*, 21(3):251–267, Mar. 1991.