Using Annotated Interface Definitions to Optimize RPC

Bryan Ford Mike Hibler Jay Lepreau

Department of Computer Science, University of Utah

http://www.cs.utah.edu/projects/flux/

Abstract[1]

In a typical remote procedure call (RPC) system, interfaces between clients and servers are defined explicitly in an interface definition language (IDL), and the IDL file is processed by a stub generator to produce client and server stubs. The primary purpose of the IDL file is to define the "network contract" between the client and the server: what operations can be invoked and what information must be passed across the network on an invocation. However, in most RPC systems, the IDL file also indirectly defines the "programmer's contract" between the stubs and the programmer: how parameters are passed to the stub, who allocates storage for the parameters, etc. For example, consider the following CORBA IDL fragment:

```
interface SysLog {
                  void write_msg(in string msg);
                };
```

Given this interface definition, a CORBA-compliant stub compiler for C will *always* produce a stub with the following C function prototype, with the `msg` parameter assumed to be null-terminated:

```
void SysLog_write_msg(SysLog object, CORBA_Exception *ex,
                      char *msg);
```

However, the stub could just as easily conform to the following function prototype instead, taking the length of the string explicitly through the `length` parameter:

```
void SysLog_write_msg(SysLog object, CORBA_Exception *ex,
                      char *msg, int length);
```

This difference should not affect the protocol between the client and the server: a client stub using the former prototype should still be able to invoke a server stub using the latter, because the C calling convention is merely a local language issue. In our terminology (adopted from the OSI networking model), these function prototypes represent alternate *presentations* of the same interface. The former is the *standard* presentation, but by no means the only possible one.

While the example was drawn from CORBA, this restriction occurs in most existing RPC systems, because they support only a single fixed presentation for any given interface definition. However, a few (OSF's DCE, IBM's Concert system) allow the presentation to be varied independently for a given client or server. In DCE, a few presentation attributes can be specified explicitly, separately from the IDL file defining the interface, in a supplemental file known as an *application control file* (ACF). Thus, while all clients and servers using a particular RPC interface generally share the same IDL file, each can have its own ACF and thus specify its own presentation annotations for its stubs.

[1] This research was supported in part by the Advanced Research Projects Agency under grant number DABT63-94-C-0058 and by the Hewlett-Packard Research Grants Program.

SIGOPS '95 12/95 CO, USA
© 1995 ACM 0-89791-715-4/95/0012...$3.50

Up until now, the primary motivation for flexible presentation has been for programmer convenience and improved interoperability. However, we have found flexible presentation also to be useful for optimization of RPC, and in many cases *necessary* to achieving maximal performance without throwing out the RPC system and resorting to hand-coded stubs. In this paper we provide seven examples demonstrating this point for a number of different operating systems and IPC transport mechanisms, with RPC performance improvements ranging from 5% to an order of magnitude. To demonstrate the broad applicability of this concept, we implemented the examples in a variety of environments and transport protocols. Among the example annotations were parameter storage allocation control (reads on a pipe server; optimizing same-domain communication in CORBA), controlling copy vs. borrow semantics of parameters, varying degrees of trust between client and server by controlling integrity and confidentiality, and avoiding extra copies (Sun RPC in NFS).

In doing this work, we designed and implemented a new RPC stub compiler called Flick (FLexible IDL Compiler Kit) that supports flexible presentation while retaining compatibility with existing RPC systems. The stub compiler is cleanly separated into front-ends and back-ends so that it can both read multiple existing IDLs as its input and generate stubs for various operating systems and transport protocols as its output. Currently we have Mach MIG, CORBA IDL, and Sun front-ends, and back-ends for Sun RPC/XDR in a Unix environment and several Mach-based protocols (transport mechanisms). Flexible presentation is supported in our system by adding a third compiler stage between the front-end and back-end, in which the presentation of an RPC interface is modified declaratively through the use of a *presentation definition language* (PDL). Nothing declared in the PDL file can affect the contract between client and server; thus, while all clients and servers using a particular RPC interface will generally share the same IDL file defining that interface, each can have its own PDL file. [The Flick compiler source and binaries are freely available from http://www.cs.utah.edu/projects/flux/flick.html]

Our results show that flexible presentation benefits RPC performance because it is necessary to create optimal stubs: any fixed presentation is the wrong one some of the time, causing unnecessary data copying in either the user code, the stubs, or both. For example, if a client wants to read data through RPC into a particular buffer, but the RPC stub insists on allocating a new buffer for the returned data, the client will have to perform an extra copy—often uselessly, because the stub could just as easily have unmarshaled the data into the client's buffer in the first place.

In general, we observe that the more efficient the underlying IPC transport mechanism is, the more important it is for the RPC system to support flexible presentation, in order to avoid unnecessary user-space overhead. Therefore, we believe that flexible presentation support will be most important in two domains: highly decomposed and microkernel-based operating systems that support extremely fast IPC mechanisms, and in very high speed networking.