AIL - a Class-Oriented RPC Stub Generator for Amoeba

Guido van Rossum

CWI, Center for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
E-mail: guido@cwi.nl or mcvax!guido

ABSTRACT

AIL – an acronym for Amoeba Interface Language – is a class-oriented RPC stub generator, used with Amoeba's RPC primitives. Together with Amoeba's facilities for manipulating capabilities (bit patterns that are unforgeable references to objects maintained by servers anywhere on a network), AIL provides a completely object-oriented view of a distributed operating system.

Input to AIL consists of class and type definitions and generator directives; output are several files containing function definitions to be compiled and linked with clients and servers. Class definitions consist mainly of function headers (specifying parameter types, etc.). Classes can inherit multiple other classes. AIL can (in principle) generate stubs for different programming languages, so clients and servers need not be written in the same language.

Introduction

Modern distributed operating systems use Remote Procedure Call (RPC) as their primary communication paradigm. The principle of RPC is as follows: A client program calls a *stub routine* which marshals its parameters into a buffer, ships the buffer over the network to a server, and waits for a reply. The server that receives the buffer unmarshals the parameters and calls the implementation procedure with these parameters. When the implementation procedure returns, the server marshals its output parameters into a buffer and ships it back to the client. The client stub awakes once it receives the reply buffer, unmarshals the output parameters, and returns to its caller.

The attractive value of the RPC paradigm lies in its resemblance with ordinary, local procedure calls: the client just calls a procedure to execute a remote operation, and the operation is done when the procedure returns. Since ordinary compilers do not generate code to call remote procedures, a tool is needed to generate the stub routines and the corresponding server code (like unmarshaling the parameters). Such a tool is called a *stub compiler* or stub generator. Stub compilers are used for several reasons: to simplify the interface to the underlying communication primitives, to cope with heterogeneity in the environment (like byte order problems), and to support interface abstraction.

This paper describes AIL, a stub compiler for Amoeba,[1] a capability-based distributed operating system. Besides doing the jobs that stub compilers normally do, AIL provides a powerful mechanism to define common subsets of interfaces in the form of class inheritance.

Amoeba RPC

Amoeba provides a highly efficient RPC mechanism. A client process sends a request to a server by calling

```
trans(&reqhdr, reqbuf, reqbufsize,
      &replhdr, replbuf, replbufsize).
```

A server process waits for a request by calling

```
getreq(&header, buffer, buffersize)
```

and sends a reply with the call

```
putrep(&header, buffer, buffersize)
```

(not necessarily using the same parameter values). A header is a data structure defined in a standard definitions file containing space for a capability (which includes a 64-bit service port number identifying the service), a few numeric parameters to the request, and some extra information that can be used for access rights checking. A buffer is an array of memory bytes and is transferred uninterpreted. Amoeba's RPC layer locates the server based on a match (involving a one-way function to prevent forgery) between the ports specified by client and server in their request headers, and ensures that at most one server gets the request and that the reply is sent back to the correct client.

In common usage a server, say a file server or a time server, recognizes a number of request types: e.g., a file server would support the requests create-file, write-file, read-file, and delete-file, and a time server would support the requests get-time and set-time. All requests to one server are normally directed at the same port; the server differentiates between request types by inspecting a command code in the request header.

To provide a nice interface to the service for client programmers, stub routines for a service are normally provided in a library. Generally, a stub routine copies its input parameters into a request header and/or buffer, sets the request code in the header, does the transaction with the server, and returns relevant fields from the reply header and/or buffer to its caller through output parameters or the function result.

Why A Stub Generator

There are at least two reasons why it is desirable to automate the generation of client stubs and request decoding code for servers.

First, such code almost always follows a standard pattern, so it is boring to write (and hence mistakes such as unchecked error conditions slip in easily).

Second, a stub generator can more easily ensure that servers support exactly the same interface where this is required: for instance, multiple file server implementations should show the same face to their clients, so clients needn't know on which server their files reside.

There are other desirable properties of good client-server interfaces that can benefit from mechanical assistance: when client and server reside on machines with different architectures, transformation of data between client and server representations may be necessary. Amoeba automatically translates the integers in the request and reply headers when necessary, but data in the buffer is copied as-is. Hence, sending an array of integers from a little-endian machine such as a VAX to a big-endian machine like a 68000 requires swapping the bytes in each integer. Transfer of floating point values between machines poses even bigger problems. Inclusion of proper data conversion code is better done mechanically, especially since its absence often isn't spotted in tests.

Design Goals

The AIL stub generator was designed with the following goals in mind:

- Generate client stubs and server main loops for the majority of server interfaces envisioned for Amoeba.
- Support easy incorporation of predefined standard interfaces into specific interfaces.
- Support multiple implementations of the same server interface.
- Support clients and servers written in different programming languages.
- Support clients and servers running on different machine architectures.
- No changes to the Amoeba kernel.
- As efficient as equivalent handwritten code.

AIL does not attempt to solve the following issues:

- Name binding: the basic binding of clients to servers is done by Amoeba's 'locate' mechanism (on a per-request basis); an additional level of naming is available through the directory service.
- Transparency between local and remote calls: because of the inherently different semantics, we find it a bad idea to try and make remote calls look exactly like as local calls.
- Backward compatibility with existing server interfaces in Amoeba: most were such a mess that they weren't worth saving anyway.

There are some beneficial side effects from using AIL as well:

- AIL helps to enforce uniform error handling.
- AIL can output the data packing and unpacking code (together called marshaling code) by itself, providing a simple way for applications to save their data structures to disk in a portable yet binary (and hence efficient) format.

Input Language

Initially, the idea was to put special markers in C source and preprocess this. This soon proved impractical, especially in the light of the multi-language goal, so we decided to design a language specialized to the specification of RPC interfaces. Although we made up the combination, most language building blocks were borrowed from other languages:

- the lexical elements, source preprocessing mechanism and type system was borrowed from C (perhaps not the ideal choice but with the advantage of compatibility with existing software in the Amoeba project);
- the class concept (although extensively modified) and the function prototype notation were borrowed from C++;[2]
- the notion of separate in and out parameters was borrowed from Ada and Modula-2+.[3]

There are good reasons for borrowing language elements when designing a special-purpose language like this one:

- the advantages and disadvantages of concepts and notations are well understood;
- users need little time to learn the new language (if they know the language borrowed from);
- because we chose C's preprocessor, lexical elements and type system, we can actually

include C definition files to import types that are needed by AIL interfaces.

Classes
The class concept in AIL differs quite a bit from that in C++. A class in AIL can contain only constant and type definitions and function prototypes; there are no data members. AIL classes specify only public information; there are no private definitions as in C++.

Functions And Parameters
Interface stub functions generated by AIL always return an error code as their function result and have a capability as their first parameter. Thus, AIL function prototypes do not specify a return type, and the first parameter is represented by a single '*'. Each following parameter must be completely specified by listing its transfer direction, type and name.

- The transfer direction can be 'in' (which is the default), 'out', or 'in out'; it indicates whether the parameter is input to the server, output from the server, or both. There is a fourth transfer direction, 'var in', which means (in C) a parameter that is input to the server but passed to the stub by reference instead of by value. In other languages this may be the same as 'in'.
- The type of a parameter determines the type of expressions or variables acceptable as actual parameters, including properties like array size.
- The name is used only for documentary purposes.

Since the client stubs are intended to be called by programs written in C or Pascal (for example), the class member call notation from C++ cannot be used to call client stubs generated by AIL. Instead of writing

```
object->member_function(argument1, argument2, ...)
```

the user must write

```
member_function(object, argument1, argument2, ...)
```

To make this work, the client stubs must have global scope (unless the language supports modules, like Modula-2). A consequence of this restriction is that there may be no overloading of function names used in different interfaces. A simple convention to prevent name clashes in practice is to prefix all function names with an abbreviation of their class name, followed by an underscore.

Class Inheritance
The power of the class mechanism lies in the possibility to extend existing classes by creating *derived classes*. A derived class has all the properties of its *base class(es)*, plus any properties added by its own definition. In C++, a derived class must be derived from exactly one base class; in AIL, a class can be derived from multiple base classes. This property is called *multiple inheritance*.

As an example of an application of multiple inheritance, consider two interfaces: a class 'tty' implementing the operations write-to-tty and read-from-tty, and a class 'window' implementing create-window and move-window (and probably others). If we now want to implement a terminal emulator in a window, the interface to a terminal emulator window should support both the tty and the window interface. This is easily accomplished by creating a class 'tty-emulator-window' inheriting the classes tty and window.

Suppose that the tty and window classes both inherit the class 'standard'; this is now also inherited by the new class.

AIL does not need the concept of 'virtual functions' present in C++. The 'virtual' predicate in C++ means that a derived class can override the implementation of that function, and since AIL functions are implemented in servers, this is true for all AIL functions.

Variable Length Arrays

AIL's type system, mostly borrowed from C, has an extension to specify arrays of variable length. Variable length arrays are needed to provide reply buffers for stubs that can return a variable amount of data: the client has to specify a buffer containing space for the maximum amount expected, but the reply must indicate the actual amount returned.

By specifying two expressions for the size of an array, separated by a colon, an actual and a maximum size can be set. This construct may occur only in function prototypes (where both expressions may refer to other parameters) and in structure definitions (where the maximum must be constant but the actual size may refer to other members of the structure).

Overriding Marshalers

For data types containing pointers, C's type notation isn't powerful enough to derive adequate marshaling code: a pointer may point to exactly one object of the given type, or to an array containing a variable number of objects, whose length can in general only be determined by inspecting the data. For this purpose, amongst others, the user may provide the names of routines to do the actual marshaling for a particular type. The most trivial example is the C string data type: the library defines a marshaler to marshal characters up to the terminating zero. For other languages than C (assuming these support variable-length strings), marshalers must be provided that support the same network format. The syntax used to specify marshalers (an option tacked onto the typedef syntax) isn't wonderful, but additional marshalers are rarely needed in practice.

Examples

Here are some examples of AIL class definitions. These classes do not necessarily correspond to actual classes used in Amoeba. The numeric ranges in square brackets after the class name are used to generate request codes; for various reasons AIL cannot be trusted to choose request codes itself.

```
#include <amoeba.h>      /* Defines capability, etc. */

class standard [1000..1999] {
    std_info(*, out char info_buffer[buflen:100],
                in int buflen);
    std_restrict(*, int rights_mask,
                    out capability restricted_cap);
    std_destroy(*);
};

class tty [2000..2099] {
    inherit standard;
    const TTY_MAXBUF = 1000;
    tty_write(*, char buffer[size:TTY_MAXBUF],
                 int size);
    tty_read(*, out char buffer[size:TTY_MAXBUF],
                out int size);
};

class window [2100..2199] {
    inherit standard;
    win_create(*, int x, int y, int w, int h,
                  out capability win_cap);
    win_move_resize(*, int x, int y, int w, int h);
};

class tty_emulator_window [2200..2299] {
    inherit tty, window;
    /* No additional features */
};
```

The Stub Generator

AIL works in three phases.

- Phase one reads and checks all class definitions in the input.
- Phase two determines, for each function of each class, the request and reply message formats.
- Finally, phase three, instructed by generator directives in the input, writes source files in the desired language containing interface definitions, client stubs and/or server code.

The request and reply message formats determined in the second phase are a function only of the class definitions found in the input, not of the generator directives: if the stub generator used a different message format when directed to generate stubs for Pascal than when directed to generate C stubs, communication between clients and servers written in different languages would fail.

Server Structure

Amoeba servers generally create a number of threads, each executing the same server main loop (get request, process it, put reply). The details of thread creation are server-specific and not handled by AIL; it only generates the server main loop itself. For each function in the class that the server is to implement (including those in base classes), when a request for that specific function arrives, the server main loop calls a function 'impl_*function-name*' with the same parameters as the corresponding client stub, except that the first parameter (a capability in the client stub) points to the request header, to save copying a few bytes.

The server writer must provide implementation functions even for standard functions: although their meaning is standardized, their implementation is not. Every implementation function must return a status code indicating whether it succeeded or failed, which will become the return value of the client stub. Marshaling of return values is only done if the implementation function succeeds.

Possible Extensions

The AIL stub generator is designed to support extensions: it is easy to add generator modules to support new languages, different stub types, or different RPC mechanisms. Some possible extensions are:

- Certain interfaces would like to have some objects maintained by a library inside the client program. Rather than starting a server thread in the client (possible, but expensive since the data will be copied at least three times), we could have AIL generate a test in the client stubs for a particular port, and call a local implementation routine instead of engaging in an RPC call.
- For C++ and other class-based languages (Modula-3?), it should be possible to generate client stubs adhering to the language's standard method-calling syntax.
- The input syntax could be augmented to tell more about the semantics of the remote functions, like idempotency, so a failed operation can be retried instead of returning an error immediately.

Performance

For many simple interfaces, AIL generates almost the same code as we would by hand. AIL-generated code never calls external functions besides user-specified marshalers and the RPC primitives, so there is no extra function call overhead.

However, there are a few tricks unknown to AIL when determining the optimal packing of parameters into header and buffer (e.g., to pack a long int in two shorts in the header), it does rather conservative error checking, and its server code always contains byte swapping code in case a request comes from a client on an 'other-endian' machine.

But the byte swapping code is never executed unnecessary: clients send and receive data in their native byte order, so the server needs to swap bytes only if their native byte order is different. All byte swapping is done in the server, to minimize the code size of the client stubs.

Comparison To Other Stub Generators

None of the RPC stub generators mentioned below support classes: each interface stands completely on its own.

- Sun RPC.[4] This is a stub generator for Sun RPC, which runs on SunOS (and other derivatives of 4.2 BSD Unix), and implements RPC on top of UDP/IP (unreliable datagrams). Like AIL, Sun RPC uses a special-purpose language derived mostly from C to specify server interfaces. The language is less powerful than ours, but directly supports the marshaling of discriminated unions. The input is not run through the C preprocessor. Rpcgen only generates C as output language.
- Flume.[5] This is an RPC stub generator for Modula-2+ (a derivative of Modula-2) under the Topaz operating system.[6] Flume reads Modula-2+ interface definition modules, but imposes some restrictions; because the modules must still be understandable by the Modula-2+ compiler, extra syntax for Flume's benefit could not be added, so certain options must be specified by embedding reserved strings in identifiers (a similar trick to lint's use of /*NOSTRICT*/ etc.). Flume tries to hide the difference between local and remote calls to the extreme: it even marshals recursive data structures. Since the same function name is used for the client stubs as for the implementation function in the server, a server cannot use its own client stubs to interface to another server (this would be useful in a 'gateway' service, for instance). Flume is completely bound to Modula-2+, and since (in practice) it is the only way to use Topaz RPC, programs writting in other languages cannot use RPC.

Conclusions

Here are a few points we have learned from our experience in building and using AIL:

- It is worthwhile to spend some time to design and implement a special-purpose language that is just right for your needs.
- Class-based interface definitions with multiple inheritance simplify the creation of new interfaces with a rich functionality.
- Automatically generated stubs can be as fast as handwritten ones.

This leads to:

- Every self-respecting distributed system should have an RPC stub generator.

And even:

- Every RPC stub generator should be class-based and support multiple inheritance.

Acknowledgements

I would like to thank first and foremost Siebren van der Zee, who has written the stub generator almost completely on his own, starting with a rather vague and incomplete description, and contributed quite a bit to the design in the process. Many discussions with Jack Jansen, Sape Mullender and Irvin Shizgal also contributed to the design. Robbert van Renesse's enthusiasm for the idea of a class-based stub compiler when I first proposed it gave me the little push that was needed to go forward with the project.

References

1. S. J. Mullender and A. S. Tanenbaum, "The Design of a Capability-Based Distributed Operating System," *The Computer Journal* **29**(4), pp. 289-300 (1986).
2. Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley (1986).
3. Paul Rovner, "Extending Modula-2 to Build Large, Integrated Systems," *IEEE*

Software **3**(6), pp. 46-57 (November 1986).
4. *rpcgen – an RPC protocol compiler*, Sun man page.
5. Andrew Birrell, Ed Lazowska, and Ted Wobber, *flume – remote procedure call (RPC) stub generator for Modula-2+*, Topaz manpage.
6. Paul R. McJones and Garret F. Swart, *Evolving the UNIX System Interface to Support Multithreaded Programs,* DEC SRC, Palo Alto, CA (1987).