Interactively Testing Remote Servers Using the Python Programming Language

Guido van Rossum
CWI, dept. CST; Kruislaan 413
1098 SJ Amsterdam, The Netherlands
E-mail: guido@cwi.nl

Jelke de Boer
HIO Enschede; P.O.Box 1326
7500 BH Enschede, The Netherlands

This paper describes how two tools that were developed quite independently gained in power by a well-designed connection between them. The tools are Python, an interpreted prototyping language, and AIL, a Remote Procedure Call stub generator. The context is Amoeba, a well-known distributed operating system developed jointly by the Free University and CWI in Amsterdam. As a consequence of their integration, both tools have profited: Python gained usability when used with Amoeba — for which it was not specifically developed — and AIL users now have a powerful interactive tool to test servers and to experiment with new client/server interfaces.[1]

1 Introduction

Remote Procedure Call (RPC) interfaces, used in distributed systems like Amoeba [6, 7], have a much more concrete character than local procedure call interfaces in traditional systems. Because clients and servers may run on different machines, with possibly different word size, byte order, etc., much care is needed to describe interfaces exactly and to implement them in such a way that they continue to work when a client or server is moved to a different machine. Since machines may fail independently, error handling must also be treated more carefully.

A common approach to such problems is to use a *stub generator*. This is a program that takes an interface description and transforms it into functions that must be compiled and linked with client and server applications. These functions are called by the application code to take care of details of interfacing to the system's RPC layer, to implement transformations between data representations of different machines, to check for errors, etc. They are called 'stubs' because

[1] An earlier version of this paper was presented at the Spring 1991 EurOpen Conference in Tromsø under the title "Linking a Stub Generator (AIL) to a Prototyping Language (Python)."

they don't actually perform the action that they are called for but only relay the parameters to the server [2].

Amoeba's stub generator is called AIL, which stands for Amoeba Interface Language [9]. The first version of AIL generated only C functions, but an explicit goal of AIL's design was *retargetability*: it should be possible to add back-ends that generate stubs for different languages from the same interface descriptions. Moreover, the stubs generated by different back-ends must be *interoperable*: a client written in Modula-3, say, should be able to use a server written in C, and vice versa.

This interoperability is the key to the success of the marriage between AIL and Python. Python is a versatile interpreted language developed by the first author. Originally intended as an alternative for the kind of odd jobs that are traditionally solved by a mixture of shell scripts, manually given shell commands, and an occasional ad hoc C program, Python has evolved into a general interactive prototyping language. It has been applied to a wide range of problems, from replacements for large shell scripts to fancy graphics demos and multimedia applications.

One of Python's strengths is the ability for the user to type in some code and immediately run it: no compilation or linking is necessary. Interactive performance is further enhanced by Python's concise, clear syntax, its very-high-level data types, and its lack of declarations (which is compensated by run-time type checking). All this makes programming in Python feel like a leisure trip compared to the hard work involved in writing and debugging even a smallish C program.

It should be clear by now that Python will be the ideal tool to test servers and their interfaces. Especially during the development of a complex server, one often needs to generate test requests on an ad hoc basis, to answer questions like "what happens if request X arrives when the server is in state Y," to test the behavior of the server with requests that touch its limitations, to check server responses to all sorts of wrong requests, etc. Python's ability to immediately execute 'improvised' code makes it a much better tool for this situation than C.

The link to AIL extends Python with the necessary functionality to connect to arbitrary servers, making the server testbed sketched above a reality. Python's high-level data types, general programming features, and system interface ensure that it has all the power and flexibility needed for the job.

One could go even further than this. Current distributed operating systems, based on client-server interaction, all lack a good command language or 'shell' to give adequate access to available services. Python has considerable potential for becoming such a shell.

1.1 Overview of this Paper

The rest of this paper contains three major sections and a conclusion. First an overview of the Python programming language is given. Next comes a short description of AIL, together with some relevant details about Amoeba. Finally, the design and construction of the link between Python and AIL is described in much detail. The conclusion looks back at the work and points out weaknesses and strengths of Python and AIL that were discovered in the process.

2 An Overview of Python

Python[2] owes much to ABC [4], a language developed at CWI as a programming language for non-expert computer users. Python borrows freely from ABC's syntax and data types, but adds modules, exceptions and classes, extensibility, and the ability to call system functions. The concepts of modules, exceptions and (to some extent) classes are influenced strongly by their occurrence in Modula-3 [3].

Although Python resembles ABC in many ways, there is a a clear difference in application domain. ABC is intended to be the only programming language for those who use a computer as a tool, but occasionally need to write a program. For this reason, ABC is not just a programming language but also a programming environment, which comes with an integrated syntax-directed editor and some source manipulation commands. Python, on the other hand, aims to be a tool for professional (system) programmers, for whom having a choice of languages with different feature sets makes it possible to choose 'the right tool for the job.' The features added to Python make it more useful than ABC in an environment where access to system functions (such as file and directory manipulations) are common. They also support the building of larger systems and libraries. The Python implementation offers little in the way of a programming environment, but is designed to integrate seamlessly with existing programming environments (e.g. UNIX and Emacs).

Perhaps the best introduction to Python is a short example. The following is a complete Python program to list the contents of a UNIX directory.

```
import sys, posix

def ls(dirname):     # Print sorted directory contents
    names = posix.listdir(dirname)
    names.sort()
    for name in names:
        if name[0] != '.': print name

ls(sys.argv[1])
```

The largest part of this program, in the middle starting with `def`, is a function definition. It defines a function named `ls` with a single parameter called `dirname`. (Comments in Python start with '#' and extend to the end of the line.) The function body is indented: Python uses indentation for statement grouping instead of braces or begin/end keywords. This is shorter to type and avoids frustrating mismatches between the perception of grouping by the user and the parser. Python accepts one statement per line; long statements may be broken in pieces using the standard backslash convention. If the body of a compound statement is a single, simple statement, it may be placed on the same line as the head.

The first statement of the function body calls the function `listdir` defined in the module `posix`. This function returns a list of strings representing the

[2] Named after the funny TV show, not the nasty reptile.

contents of the directory name passed as a string argument, here the argument `dirname`. If `dirname` were not a valid directory name, or perhaps not even a string, `listdir` would raise an exception and the next statement would never be reached. (Exceptions can be caught in Python; see later.) Assuming `listdir` returns normally, its result is assigned to the local variable `names`.

The second statement calls the method `sort` of the variable `names`. This method is defined for all lists in Python and does the obvious thing: the elements of the list are reordered according to their natural ordering relationship. Since in our example the list contains strings, they are sorted in ascending ASCII order.

The last two lines of the function contain a loop that prints all elements of the list whose first character isn't a period. In each iteration, the `for` statement assigns an element of the list to the local variable `name`. The `print` statement is intended for simple-minded output; more elaborate formatting is possible with Python's string handling functions.

The other two parts of the program are easily explained. The first line is an `import` statement that tells the interpreter to import the modules `sys` and `posix`. As it happens these are both built into the interpreter. Importing a module (built-in or otherwise) only makes the module name available in the current scope; functions and data defined in the module are accessed through the dot notation as in `posix.listdir`. The scope rules of Python are such that the imported module name `posix` is also available in the function `ls` (this will be discussed in more detail later).

Finally, the last line of the program calls the `ls` function with a definite argument. It must be last since Python objects must be defined before they can be used; in particular, the function `ls` must be defined before it can be called. The argument to `ls` is `sys.argv[1]`, which happens to be the Python equivalent of `$1` in a shell script or `argv[1]` in a C program's `main` function.

2.1 Python Data Types

(This and the following subsections describe Python in quite a lot of detail. If you are more interested in AIL, Amoeba and how they are linked with Python, you can skip to section 3 now.)

Python's syntax may not have big surprises (which is exactly as it should be), but its data types are quite different from what is found in languages like C, Ada or Modula-3. All data types in Python, even integers, are 'objects'. All objects participate in a common garbage collection scheme (currently implemented using reference counting). Assignment is cheap, independent of object size and type: only a pointer to the assigned object is stored in the assigned-to variable. No type checking is performed on assignment; only specific operations like addition test for particular operand types.

The basic object types in Python are numbers, strings, tuples, lists and dictionaries. Some other object types are open files, functions, modules, classes, and class instances; even types themselves are represented as objects. Extension modules written in C can define additional object types; examples are objects representing windows and Amoeba capabilities. Finally, the implementation itself makes heavy use of objects, and defines some private object types that aren't normally visible to the user. There is no explicit pointer type in Python.

Numbers, both integers and floating point, are pretty straightforward. The notation for numeric literals is the same as in C, including octal and hexadecimal integers; precision is the same as `long` or `double` in C. A third numeric type, 'long integer', written with an 'L' suffix, can be used for arbitrary precision calculations. All arithmetic, shifting and masking operations from C are supported.

Strings are 'primitive' objects just like numbers. String literals are written between single quotes, using similar escape sequences as in C. Operations are built into the language to concatenate and to replicate strings, to extract substrings, etc. There is no limit to the length of the strings created by a program. There is no separate character data type; strings of length one do nicely.

Tuples are a way to 'pack' small amounts of heterogeneous data together and carry them around as a unit. Unlike structure members in C, tuple items are nameless. Packing and unpacking assignments allow access to the items, for example:

```
x = 'Hi', (1, 2), 'World'   # x is a 3-item tuple,
                            # its middle item is (1, 2)
p, q, r = x                 # unpack x into p, q and r
a, b = q                    # unpack q into a and b
```

A combination of packing and unpacking assignment can be used as parallel assignment, and is idiom for permutations, e.g.:

```
p, q = q, p                 # swap without temporary
a, b, c = b, c, a           # cyclic permutation
```

Tuples are also used for function argument lists if there is more than one argument. A tuple object, once created, cannot be modified; but it is easy enough to unpack it and create a new, modified tuple from the unpacked items and assign this to the variable that held the original tuple object (which will then be garbage-collected).

Lists are array-like objects. List items may be arbitrary objects and can be accessed and changed using standard subscription notation. Lists support item insertion and deletion, and can therefore be used as queues, stacks etc.; there is no limit to their size.

Strings, tuples and lists together are *sequence* types. These share a common notation for generic operations on sequences such as subscription, concatenation, slicing (taking subsequences) and membership tests. As in C, subscripts start at 0.

Dictionaries are 'mappings' from one domain to another. The basic operations on dictionaries are item insertion, extraction and deletion, using subscript notation with the key as subscript. (The current implementation allows only strings in the key domain, but a future version of the language may remove this restriction.)

2.2 Statements

Python has various kinds of simple statements, such as assignments and `print` statements, and several kinds of compound statements, like `if` and `for` state-

ments. Formally, function definitions and `import` statements are also statements, and there are no restrictions on the ordering of statements or their nesting: `import` may be used inside a function, functions may be defined conditionally using an `if` statement, etc. The effect of a declaration-like statement takes place only when it is executed.

All statements except assignments and expression statements begin with a keyword: this makes the language easy to parse. An overview of the most common statement forms in Python follows.

An *assignment* has the general form

variable = *variable* = ... = *variable* = *expression*

It assigns the value of the expression to all listed variables. (As shown in the section on tuples, variables and expressions can in fact be comma-separated lists.) The assignment operator is not an expression operator; there are no horrible things in Python like

```
while (p = p->next) { ... }
```

Expression syntax is mostly straightforward and will not be explained in detail here.

An *expression statement* is just an expression on a line by itself. This writes the value of the expression to standard output, in a suitably unambiguous way, unless it is a 'procedure call' (a function call that returns no value). Writing the value is useful when Python is used in 'calculator mode', and reminds the programmer not to ignore function results.

The `if` statement allows conditional execution. It has optional `elif` and `else` parts; a construct like `if...elif...elif...elif...else` can be used to compensate for the absence of a *switch* or *case* statement.

Looping is done with `while` and `for` statements. The latter (demonstrated in the 'ls' example earlier) iterates over the elements of a 'sequence' (see the discussion of data types below). It is possible to terminate a loop with a `break` statement or to start the next iteration with `continue`. Both looping statements have an optional `else` clause which is executed after the loop is terminated normally, but skipped when it is terminated by `break`. This can be handy for searches, to handle the case that the item is not found.

Python's *exception* mechanism is modelled after that of Modula-3. Exceptions are raised by the interpreter when an illegal operation is tried. It is also possible to explicitly raise an exception with the `raise` statement:

`raise` *expression*, *expression*

The first expression identifies which exception should be raised; there are several built-in exceptions and the user may define additional ones. The second, optional expression is passed to the handler, e.g. as a detailed error message.

Exceptions may be handled (caught) with the `try` statement, which has the following general form:

```
try:   block
except expression, variable:   block
except expression, variable:   block
...
except:   block
```

When an exception is raised during execution of the first block, a search for an exception handler starts. The first `except` clause whose *expression* matches the exception is executed. The expression may specify a list of exceptions to match against. A handler without an expression serves as a 'catch-all'. If there is no match, the search for a handler continues with outer `try` statements; if no match is found on the entire invocation stack, an error message and stack trace are printed, and the program is terminated (interactively, the interpreter returns to its main loop).

Note that the form of the `except` clauses encourages a style of programming whereby only selected exceptions are caught, passing unanticipated exceptions on to the caller and ultimately to the user. This is preferable over a simpler 'catch-all' error handling mechanism, where a simplistic handler intended to catch a single type of error like 'file not found' can easily mask genuine programming errors — especially in a language like Python which relies strongly on run-time checking and allows the catching of almost any type of error.

Other common statement forms, which we have already encountered, are function definitions, `import` statements and `print` statements. There is also a `del` statement to delete one or more variables, a `return` statement to return from a function, and a `global` statement to allow assignments to global variables. Finally, the `pass` statement is a no-op.

2.3 Execution Model

A Python program is executed by a stack-based interpreter.

When a function is called, a new 'execution environment' for it is pushed onto the stack. An execution environment contains (among other data) pointers to two 'symbol tables' that are used to hold variables: the local and the global symbol table. The local symbol table contains local variables of the current function invocation (including the function arguments); the global symbol table contains variables defined in the module containing the current function.

The 'global' symbol table is thus only global with respect to the current function. There are no system-wide global variables; using the `import` statement it is easy enough to reference variables that are defined in other modules. A system-wide read-only symbol table is used for built-in functions and constants though.

On assignment to a variable, by default an entry for it is made in the local symbol table of the current execution environment. The `global` command can override this (it is not enough that a global variable by the same name already exists). When a variable's value is needed, it is searched first in the local symbol table, then in the global one, and finally in the symbol table containing built-in functions and constants.

The term 'variable' in this context refers to any name: functions and imported modules are searched in exactly the same way.

Names defined in a module's symbol table survive until the end of the program. This approximates the semantics of file-static global variables in C or module variables in Modula-3. A module is initialized the first time it is imported, by executing the text of the module as a parameterless function whose local and global symbol tables are the same, so names are defined in module's symbol table. (Modules implemented in C have another way to define symbols.)

A Python main program is read from standard input or from a script file passed as an argument to the interpreter. It is executed as if an anonymous module was imported. Since `import` statements are executed like all other statements, the initialization order of the modules used in a program is defined by the flow of control through the program.

The 'attribute' notation *m.name*, where *m* is a module, accesses the symbol *name* in that module's symbol table. It can be assigned to as well. This is in fact a special case of the construct *x.name* where *x* denotes an arbitrary object; the type of *x* determines how this is to be interpreted, and what assignment to it means.

For instance, when `a` is a list object, `a.append` yields a built-in 'method' object which, when called, appends an item to `a`. (If `a` and `b` are distinct list objects, `a.append` and `b.append` are distinguishable method objects.) Normally, in statements like `a.append(x)`, the method object `a.append` is called and then discarded, but this is a matter of convention.

List attributes are read-only — the user cannot define new list methods. Some objects, like numbers and strings, have no attributes at all. Like all type checking in Python, the meaning of an attribute is determined at run-time — when the parser sees *x.name*, it has no idea of the type of *x*. Note that *x* here does not have to be a variable — it can be an arbitrary (perhaps parenthesized) expression.

Given the flexibility of the attribute notation, one is tempted to use methods to replace all standard operations. Yet, Python has kept a small repertoire of built-in functions like `len()` and `abs()`. The reason is that in some cases the function notation is more familiar than the method notation; just like programs would become less readable if all infix operators were replaced by function calls, they would become less readable if all function calls had to be replaced by method calls (and vice versa!).

The choice whether to make something a built-in function or a method is a matter of taste. For arithmetic and string operations, function notation is preferred, since frequently the argument to such an operation is an expression using infix notation, as in `abs(a+b)`; this definitely looks better than `(a+b).abs()`. The choice between make something a built-in function or a function defined in a built-in method (requiring `import`) is similarly guided by intuition; all in all, only functions needed by 'general' programming techniques are built-in functions.

2.4 Classes

Python has a class mechanism distinct from the object-orientation already explained. A class in Python is not much more than a collection of methods and a way to create class instances. Class methods are ordinary functions whose first parameter is the class instance; they are called using the method notation.

For instance, a class can be defined as follows:

```
class Foo:
   def meth1(self, arg): ...
   def meth2(self): ...
```

A class instance is created by `x = Foo()` and its methods can be called thus:

```
x.meth1('Hi There!')
x.meth2()
```

The functions used as methods are also available as attributes of the class object, and the above method calls could also have been written as follows:

```
Foo.meth1(x, 'Hi There!')
Foo.meth2(x)
```

Class methods can store instance data by assigning to instance data attributes, e.g.:

```
self.size = 100
self.title = 'Dear John'
```

Data attributes do not have to be declared; as with local variables, they spring into existence when assigned to. It is a matter of discretion to avoid name conflicts with method names. This facility is also available to class users; instances of a method-less class can be used as records with named fields.

There is no built-in mechanism for instance initialization. Classes by convention provide an `init()` method which initializes the instance and then returns it, so the user can write

```
x = Foo().init('Dr. Strangelove')
```

Any user-defined class can be used as a base class to derive other classes. However, built-in types like lists cannot be used as base classes. (Incidentally, the same is true in C++ and Modula-3.) A class may override any method of its base classes. Instance methods are first searched in the method list of their class, and then, recursively, in the method lists of their base class. Initialization methods of derived classes should explicitly call the initialization methods of their base class.

A simple form of multiple inheritance is also supported: a class can have multiple base classes, but the language rules for resolving name conflicts are somewhat simplistic, and consequently the feature has so far found little usage.

2.5 The Python Library

Python comes with an extensive library, structured as a collection of modules. A few modules are built into the interpreter: these generally provide access to system libraries implemented in C such as mathematical functions or operating system calls. Two built-in modules provide access to internals of the interpreter and its environment. Even abusing these internals will at most cause an exception in the Python program; the interpreter will not dump core because of errors in Python code.

Most modules however are written in Python and distributed with the interpreter; they provide general programming tools like string operations and random number generators, provide more convenient interfaces to some built-in modules, or provide specialized services like a *getopt*-style command line option processor for stand-alone scripts.

There are also some modules written in Python that dig deep in the internals of the interpreter; there is a module to browse the stack backtrace when an unhandled exception has occurred, one to disassemble the internal representation of Python code, and even an interactive source code debugger which can trace Python code, set breakpoints, etc.

2.6 Extensibility

It is easy to add new built-in modules written in C to the Python interpreter. Extensions appear to the Python user as built-in modules. Using a built-in module is no different from using a module written in Python, but obviously the author of a built-in module can do things that cannot be implemented purely in Python.

In particular, built-in modules can contain Python-callable functions that call functions from particular system libraries ('wrapper functions'), and they can define new object types. In general, if a built-in module defines a new object type, it should also provide at least one function that creates such objects. Attributes of such object types are also implemented in C; they can return data associated with the object or methods, implemented as C functions.

For instance, an extension was created for Amoeba: it provides wrapper functions for the basic Amoeba name server functions, and defines a 'capability' object type, whose methods are file server operations. Another extension is a built-in module called `posix`; it provides wrappers around post UNIX system calls. Extension modules also provide access to two different windowing/graphics interfaces: STDWIN [8] (which connects to X11 on UNIX and to the Mac Toolbox on the Macintosh), and the Graphics Library (GL) for Silicon Graphics machines.

Any function in an extension module is supposed to type-check its arguments; the interpreter contains a convenience function to facilitate extracting C values from arguments and type-checking them at the same time. Returning values is also painless, using standard functions to create Python objects from C values.

On some systems extension modules may be dynamically loaded, thus avoiding the need to maintain a private copy of the Python interpreter in order to use a private extension.

3 A Short Description of AIL and Amoeba

An RPC stub generator takes an interface description as input. The designer of a stub generator has at least two choices for the input language: use a suitably restricted version of the target language, or design a new language. The first solution was chosen, for instance, by the designers of Flume, the stub generator for the Topaz distributed operating system built at DEC SRC [1, 5].

Flume's one and only target language is Modula-2+ (the predecessor of Modula-3). Modula-2+, like Modula-N for any N, has an interface syntax that is well suited as a stub generator input language: an interface module declares the functions that are 'exported' by a module implementation, with their parameter and return types, plus the types and constants used for the parameters. Therefore, the input to Flume is simply a Modula-2+ interface module. But even in this ideal situation, an RPC stub generator needs to know things about functions that are not stated explicitly in the interface module: for instance, the transfer direction of VAR parameters (IN, OUT or both) is not given. Flume solves this and other problems by a mixture of directives hidden in comments and a convention for the names of objects. Thus, one could say that the designers of Flume really created a new language, even though it looks remarkably like their target language.

3.1 The AIL Input Language

Amoeba uses C as its primary programming language. C function declarations (at least in 'Classic' C) don't specify the types of the parameters, let alone their transfer direction. Using this as input for a stub generator would require almost all information for the stub generator to be hidden inside comments, which would require a rather contorted scanner. Therefore we decided to design the input syntax for Amoeba's stub generator 'from scratch'. This gave us the liberty to invent proper syntax not only for the transfer direction of parameters, but also for variable-length arrays.

On the other hand we decided not to abuse our freedom, and borrowed as much from C as we could. For instance, AIL runs its input through the C preprocessor, so we get macros, include files and conditional compilation for free. AIL's type declaration syntax is a superset of C's, so the user can include C header files to use the types declared there as function parameter types — which are declared using function prototypes as in C++ or Standard C. It should be clear by now that AIL's lexical conventions are also identical to C's. The same is true for its expression syntax.

Where does AIL differ from C, then? Function declarations in AIL are grouped in *classes*. Classes in AIL are mostly intended as a grouping mechanism: all functions implemented by a server are grouped together in a class. Inheritance is used to form new groups by adding elements to existing groups; multiple inheritance is supported to join groups together. Classes can also contain constant and type definitions, and one form of output that AIL can generate is a header file for use by C programmers who wish to use functions from a particular AIL class.

Let's have a look at some (unrealistically simple) class definitions:

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

class standard_ops [1000 .. 1999] {
    /* Operations supported by most interfaces */
    std_info(*, out char buf[size:100], out int size);
    std_destroy(*);
};
```

This defines a class called 'standard_ops' whose request codes are chosen by AIL from the range 1000-1999. Request codes are small integers used to identify remote operations. The author of the class must specify a range from which AIL chooses, and class authors must make sure they avoid conflicts, e.g. by using an 'assigned number administration office'. In the example, 'std_info' will be assigned request code 1000 and 'std_destroy' will get code 1001. There is also an option to explicitly assign request codes, for compatibility with servers with manually written interfaces.

The class 'standard_ops' defines two operations, 'std_info' and 'std_destroy'. The first parameter of each operation is a star ('*'); this is a placeholder for a capability that must be passed when the operation is called. The description of Amoeba below explains the meaning and usage of capabilities; for now, it is sufficient to know that a capability is a small structure that uniquely identifies an object and a server or service.

The standard operation 'std_info' has two output parameters: a variable-size character buffer (which will be filled with a short descriptive string of the object to which the operation is applied) and an integer giving the length of this string. The standard operation 'std_destroy' has no further parameters — it just destroys the object, if the caller has the right to do so.

The next class is called 'tty':

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

The request codes for operations defined in this class lie in the range 2000-2099; inherited operations use the request codes already assigned to them. The operations defined by this class are 'tty_read' and 'tty_write', which pass variable-sized data buffers between client and server. Class 'tty' inherits class 'standard_ops', so tty objects also support the operations 'std_info' and 'std_destroy'.

Only the *interface* for 'std_info' and 'std_destroy' is shared between tty objects and other objects whose interface inherits 'standard_ops'; the implementation may differ. Even multiple implementations of the 'tty' interface may exist, e.g. a driver for a console terminal and a terminal emulator in a window. To expand on the latter example, consider:

```
class window [2100 .. 2199] {
    inherit standard_ops;
    win_create(*, int x, int y, int width, int height,
                  out capability win_cap);
    win_reconfigure(*, int x, int y, int width, int height);
};

class tty_emulator [2200 .. 2299] {
    inherit tty, window;
};
```

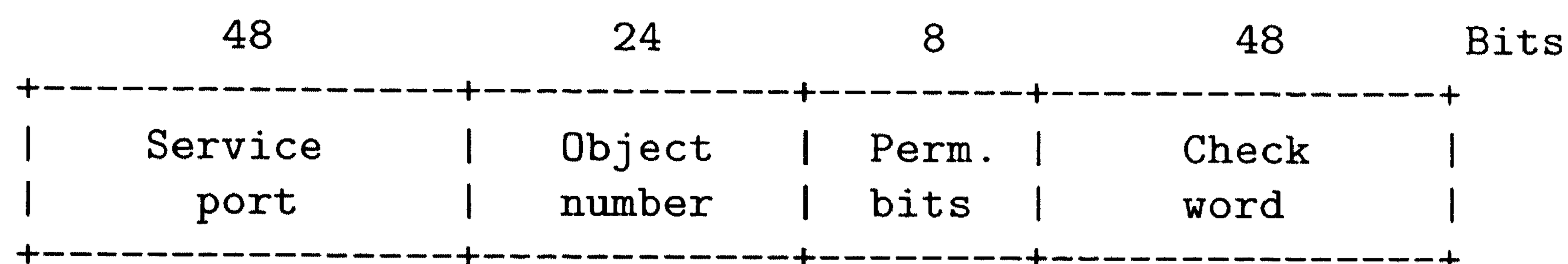
Here two new interface classes are defined. Class 'window' could be used for creating and manipulating windows. Note that 'win_create' returns a capability for the new window. This request should probably should be sent to a generic window server capability, or it might create a subwindow when applied to a window object.

Class 'tty_emulator' demonstrates the essence of multiple inheritance. It is presumably the interface to a window-based terminal emulator. Inheritance is transitive, so 'tty_emulator' also implicitly inherits 'standard_ops'. In fact, it inherits it twice: once via 'tty' and once via 'window'. Since AIL class inheritance only means interface sharing, not implementation sharing, inheriting the same class multiple times is never a problem and has the same effect as inheriting it once.

Note that the power of AIL classes doesn't go as far as C++. AIL classes cannot have data members, and there is no mechanism for a server that implements a derived class to inherit the implementation of the base class — other than copying the source code. The syntax for class definitions and inheritance is also different.

3.2 Amoeba

The smell of 'object-orientedness' that the use of classes in AIL creates matches nicely with Amoeba's object-oriented approach to RPC. In Amoeba, almost all operating system entities (files, directories, processes, devices etc.) are implemented as *objects*. Objects are managed by *services* and represented by *capabilities*. A capability gives its holder access to the object it represents. Capabilities are protected cryptographically against forgery and can thus be kept in user space. A capability is a 128-bit binary string, subdivided as follows:



```
         48                 24           8            48          Bits
+----------------+------------+--------+----------------+
|    Service     |   Object   | Perm.  |     Check      |
|     port       |   number   | bits   |     word       |
+----------------+------------+--------+----------------+
```

The service port is used by the RPC implementation in the Amoeba kernel to locate a server implementing the service that manages the object. In many cases there is a one-to-one correspondence between servers and services (each service is implemented by exactly one server process), but some services are replicated. For instance, Amoeba's directory service, which is crucial for gaining access to most other services, is implemented by two servers that listen on the same port and know about exactly the same objects.

The object number in the capability is used by the server receiving the request for identifying the object to which the operation applies. The permission bits specify which operations the holder of the capability may apply. The last part of a capability is a 48-bit long 'check word', which is used to prevent forgery. The check word is computed by the server based upon the permission bits and a

random key per object that it keeps secret. If you change the permission bits you must compute the proper check word or else the server will refuse the capability. Due to the size of the check word and the nature of the cryptographic 'one-way function' used to compute it, inverting this function is impractical, so forging capabilities is impossible.[3]

A working Amoeba system is a collection of diverse servers, managing files, directories, processes, devices etc. While most servers have their own interface, there are some requests that make sense for some or all object types. For instance, the *std_info()* request, which returns a short descriptive string, applies to all object types. Likewise, *std_destroy()* applies to files, directories and processes, but not to devices.

Similarly, different file server implementations may want to offer the same interface for operations like *read()* and *write()* to their clients. AIL's grouping of requests intc classes is ideally suited to describe this kind of interface sharing, and a class hierarchy results which clearly shows the similarities between server interfaces (not necessarily their implementations!).

The base class of all classes defines the *std_info()* request. Most server interfaces actually inherit a derived class that also defines *std_destroy()*. File servers inherit a class that defines the common operations on files, etc.

3.3 How AIL Works

The AIL stub generator functions in three phases:

- parsing,
- strategy determination,
- code generation.

Phase one parses the input and builds a symbol table containing everything it knows about the classes and other definitions found in the input.

Phase two determines the strategy to use for each function declaration in turn and decides upon the request and reply message formats. This is not a simple matter, because of various optimization attempts. Amoeba's kernel interface for RPC requests takes a fixed-size header and one arbitrary-size buffer. A large part of the header holds the capability of the object to which the request is directed, but there is some space left for a few integer parameters whose interpretation is left up to the server. AIL tries to use these slots for simple integer parameters, for two reasons.

First, unlike the buffer, header fields are byte-swapped by the RPC layer in the kernel if necessary, so it saves a few byte swapping instructions in the user code. Second, and more important, a common form of request transfers a few integers and one large buffer to or from a server. The *read()* and *write()* requests of most file servers have this form, for instance. If it is possible to place all integer parameters in the header, the address of the buffer parameter can be passed directly to the kernel RPC layer. While AIL is perfectly capable of handling

[3]As computers become faster, inverting the one-way function becomes less impractical. Therefore, a next version of Amoeba will have 64-bit check words.

requests that do not fit this format, the resulting code involves allocating a new buffer and copying all parameters into it. It is a top priority to avoid this copying ('marshalling') if at all possible, in order to maintain Amoeba's famous RPC performance.

When AIL resorts to copying parameters into a buffer, it reorders them so that integers indicating the lengths of variable-size arrays are placed in the buffer before the arrays they describe, since otherwise decoding the request would be impossible. It also adds occasional padding bytes to ensure integers are aligned properly in the buffer — this can speed up (un)marshalling.

Phase three is the code generator, or back-end. There are in fact many different back-ends that may be called in a single run to generate different types of output. The most important output types are header files (for inclusion by the clients of an interface), client stubs, and 'server main loop' code. The latter decodes incoming requests in the server. The generated code depends on the programming language requested, and there are separate back-ends for each supported language.

It is important that the strategy chosen by phase two is independent of the language requested for phase three — otherwise the interoperability of servers and clients written in different languages would be compromised.

4 Linking AIL to Python

From the previous section it can be concluded that linking AIL to Python is a matter of writing a back-end for Python. This is indeed what we did.

Considerable time went into the design of the back-end in order to make the resulting RPC interface for Python fit as smoothly as possible in Python's programming style. For instance, the issues of parameter transfer, variable-size arrays, error handling, and call syntax were all solved in a manner that favors ease of use in Python rather than strict correspondence with the stubs generated for C, without compromising network-level compatibility.

4.1 Mapping AIL Entities to Python

For each programming language that AIL is to support, a mapping must be designed between the data types in AIL and those in that language. Other aspects of the programming languages, such as differences in function call semantics, must also be taken care of.

While the mapping for C is mostly straightforward, the mapping for Python requires a little thinking to get the best results for Python programmers.

4.1.1 Parameter Transfer Direction Perhaps the simplest issue is that of parameter transfer direction. Parameters of functions declared in AIL are categorized as being of type `in`, `out` or `in out` (the same distinction as made in Ada). Python only has call-by-value parameter semantics; functions can return multiple values as a tuple. This means that, unlike the C back-end, the Python back-end cannot always generate Python functions with exactly the same parameter list as the AIL functions.

Instead, the Python parameter list consists of all `in` and `in out` parameters, in the order in which they occur in the AIL parameter list; similarly, the

Python function returns a tuple containing all `in out` and `out` parameters. In fact Python packs function parameters into a tuple as well, stressing the symmetry between parameters and return value. For example, a stub with this AIL parameter list:

```
(*, in int p1, in out int p2, in int p3, out int p4)
```

will have the following parameter list and return values in Python:

```
(p1, p2, p3)  ->  (p2, p4)
```

4.1.2 Variable-size Entities The support for variable-size objects in AIL is strongly guided by the limitations of C in this matter. Basically, AIL allows what is feasible in C: functions may have variable-size arrays as parameters (both input or output), provided their length is passed separately. In practice this is narrowed to the following rule: for each variable-size array parameter, there must be an integer parameter giving its length. (An exception for null-terminated strings is planned but not yet realized.)

Variable-size arrays in AIL or C correspond to *sequences* in Python: lists, tuples or strings. These are much easier to use than their C counterparts. Given a sequence object in Python, it is always possible to determine its size: the built-in function `len()` returns it. It would be annoying to require the caller of an RPC stub with a variable-size parameter to also pass a parameter that explicitly gives its size. Therefore we eliminate all parameters from the Python parameter list whose value is used as the size of a variable-size array. Such parameters are easily found: the array bound expression contains the name of the parameter giving its size. This requires the stub code to work harder (it has to recover the value for size parameters from the corresponding sequence parameter), but at least part of this work would otherwise be needed as well, to check that the given and actual sizes match.

Because of the symmetry in Python between the parameter list and the return value of a function, the same elimination is performed on return values containing variable-size arrays: integers returned solely to tell the client the size of a returned array are not returned explicitly to the caller in Python.

4.1.3 Error Handling Another point where Python is really better than C is the issue of error handling. It is a fact of life that everything involving RPC may fail, for a variety of reasons outside the user's control: the network may be disconnected, the server may be down, etc. Clients must be prepared to handle such failures and recover from them, or at least print an error message and die. In C this means that every function returns an error status that must be checked by the caller, causing programs to be cluttered with error checks — or worse, programs that ignore errors and carry on working with garbage data.

In Python, errors are generally indicated by exceptions, which can be handled out of line from the main flow of control if necessary, and cause immediate program termination (with a stack trace) if ignored. To profit from this feature, all RPC errors that may be encountered by AIL-generated stubs in Python are turned into exceptions. An extra value passed together with the exception is

used to relay the error code returned by the server to the handler. Since in general RPC failures are rare, Python test programs can usually ignore exceptions — making the program simpler — without the risk of occasional errors going undetected. (I still remember the embarrassment a hundredfold speed improvement reported, long, long, ago, about a new version of a certain program, which later had to be attributed to a benchmark that silently dumped core...)

4.1.4 Function Call Syntax Amoeba RPC operations always need a capability parameter (this is what the '*' in the AIL function templates stands for); the service is identified by the port field of the capability. In C, the capability must always be the first parameter of the stub function, but in Python we can do better.

A Python capability is an opaque object type in its own right, which is used, for instance, as parameter to and return value from Amoeba's name server functions. Python objects can have methods, so it is convenient to make all AIL-generated stubs methods of capabilities instead of just functions. Therefore, instead of writing

```
some_stub(cap, other_parameters)
```

as in C, Python programmers can write

```
cap.some_stub(other_parameters)
```

This is better because it reduces name conflicts: in Python, no confusion is possible between a stub and a local or global variable or user-defined function with the same name.

4.1.5 Example All the preceding principles can be seen at work in the following example. Suppose a function is declared in AIL as follows:

```
some_stub(*, in char buf[size:1000], in int size,
             out int n_done, out int status);
```

In C it might be called by the following code (including declarations, for clarity, but not initializations):

```
int err, n_done, status;
capability cap;
char buf[500];
...
err = some_stub(&cap, buf, sizeof buf, &n_done, &status);
if (err != 0) return err;
printf("%d done; status = %d\n", n_done, status);
```

Equivalent code in Python might be the following:

```
cap = ...
buf = ...
n_done, status = cap.some_stub(buf)
print n_done, 'done;', 'status =', status
```

No explicit error check is required in Python: if the RPC fails, an exception is raised so the `print` statement is never reached.

4.2 The Implementation

More or less orthogonal to the issue of how to map AIL operations to the Python language is the question of how they should be implemented.

In principle it would be possible to use the same strategy that is used for C: add an interface to Amoeba's low-level RPC primitives to Python and generate Python code to marshal parameters into and out of a buffer. However, Python's high-level data types are not well suited for marshalling: byte-level operations are clumsy and expensive, with the result that marshalling a single byte of data can take several Python statements. This would mean that a large amount of code would be needed to implement a stub, which would cost a lot of time to parse and take up a lot of space in 'compiled' form (as parse tree or pseudo code). Execution of the marshalling code would be sluggish as well.

We therefore chose an alternate approach, writing the marshalling in C, which is efficient at such byte-level operations. While it is easy enough to generate C code that can be linked with the Python interpreter, it would obviously not stimulate the use of Python for server testing if each change to an interface required relinking the interpreter (dynamic loading of C code is not yet available on Amoeba). This is circumvented by the following solution: the marshalling is handled by a simple *virtual machine*, and AIL generates instructions for this machine. An interpreter for the machine is linked into the Python interpreter and reads its instructions from a file written by AIL.

The machine language for our virtual machine is dubbed *Stubcode*. Stubcode is a super-specialized language. There are two sets of of about a dozen instructions each: one set marshals Python objects representing parameters into a buffer, the other set (similar but not quite symmetric) unmarshals results from a buffer into Python objects. The Stubcode interpreter uses a stack to hold Python intermediate results. Other state elements are an Amoeba header and buffer, a pointer indicating the current position in the buffer, and of course a program counter. Besides (un)marshalling, the virtual machine must also implement type checking, and raise a Python exception when a parameter does not have the expected type.

The Stubcode interpreter marshals Python data types very efficiently, since each instruction can marshal a large amount of data. For instance, a whole Python string is marshalled by a single Stubcode instruction, which (after some checking) executes the most efficient byte-copying loop possible — it calls `memcpy()`.

Construction details of the Stubcode interpreter are straightforward. Most complications are caused by the peculiarities of AIL's strategy module and Python's type system. By far the most complex single instruction is the 'loop' instruction, which is used to marshal arrays.

As an example, here is the complete Stubcode program (with spaces and comments added for clarity) generated for the function `some_stub()` of the example above. The stack contains pointers to Python objects, and its initial contents is the parameter to the function, the string `buf`. The final stack contents will be the function return value, the tuple `(n_done, status)`. The name `header` refers to the fixed size Amoeba RPC header structure.

```
BufSize      1000            Allocate RPC buffer of 1000 bytes
Dup          1               Duplicate stack top
StringS                      Replace stack top by its string size
PutI         h_extra int32   Store top element in header.h_extra
TStringSlt   1000            Assert string size less than 1000
PutVS                        Marshal variable-size string

Trans        1234            Execute the RPC (request code 1234)

GetI         h_extra int32   Push integer from header.h_extra
GetI         h_size int32    Push integer from header.h_size
Pack         2               Pack top 2 elements into a tuple
```

As much work as possible is done by the Python back-end in AIL, rather than in the Stubcode interpreter, to make the latter both simple and fast. For instance, the decision to eliminate an array size parameter from the Python parameter list is taken by AIL, and Stubcode instructions are generated to recover the size from the actual parameter and to marshal it properly. Similarly, there is a special alignment instruction (not used in the example) to meet alignment requirements.

Communication between AIL and the Stubcode generator is via the file system. For each stub function, AIL creates a file in its output directory, named after the stub with a specific suffix. This file contains a machine-readable version of the Stubcode program for the stub. The Python user can specify a search path containing directories which the interpreter searches for a Stubcode file the first time the definition for a particular stub is needed.

The transformations on the parameter list and data types needed to map AIL data types to Python data types make it necessary to help the Python programmer a bit in figuring out the parameters to a call. Although in most cases the rules are simple enough, it is sometimes hard to figure out exactly what the parameter and return values of a particular stub are. There are two sources of help in this case: first, the exception contains enough information so that the user can figure what type was expected; second, AIL's Python back-end optionally generates a human-readable 'interface specification' file.

5 Conclusion

We have succeeded in creating a useful extension to Python that enables Amoeba server writers to test and experiment with their server in a much more interactive manner. We hope that this facility will add to the popularity of AIL amongst Amoeba programmers.

Python's extensibility was proven convincingly by the exercise (performed by the second author) of adding the Stubcode interpreter to Python. Standard data abstraction techniques are used to insulate extension modules from details of the rest of the Python interpreter. In the case of the Stubcode interpreter this worked well enough that it survived a major overhaul of the main Python interpreter virtually unchanged.

On the other hand, adding a new back-end to AIL turned out to be quite a

bit of work. One problem, specific to Python, was to be expected: Python's variable-size data types differ considerably from the C-derived data model that AIL favors. Two additional problems we encountered were the complexity of the interface between AIL's second and third phases, and a number of remaining bugs in the second phase that surfaced when the implementation of the Python back-end was tested. The bugs have been tracked down and fixed, but nothing has been done about the complexity of the interface.

5.1 Future Plans

AIL's C back-end generates server main loop code as well as client stubs. The Python back-end currently only generates client stubs, so it is not yet possible to write servers in Python. While it is clearly more important to be able to use Python as a client than as a server, the ability to write server prototypes in Python would be a valuable addition: it allows server designers to experiment with interfaces in a much earlier stage of the design, with a much smaller programming effort. This makes it possible to concentrate on concepts first, before worrying about efficient implementation.

The unmarshalling done in the server is almost symmetric with the marshalling in the client, and vice versa, so relative small extensions to the Stubcode virtual machine will allow its use in a server main loop. We hope to find the time to add this feature to a future version of Python.

6 AVAILABILITY

The Python source distribution is available to Internet users by anonymous ftp to site `ftp.cwi.nl` [IP address 192.16.184.180] from directory `/pub`, file name `python*.tar.Z` (where the * stands for a version number). This is a compressed UNIX tar file containing the C source and LaTeXdocumentation for the Python interpreter. It includes the Python library modules and the *Stubcode* interpreter, as well as many example Python programs. Total disk space occupied by the distribution is about 3 Mb; compilation requires 1-3 Mb depending on the configuration built, the compile options, etc.

REFERENCES

1. A.D. BIRRELL, E.D. LAZOWSKA, AND E. WOBBER (1987). *Flume — Remote Procedure Call Stub Generator for Modula-2+.* DEC SRC, (Topaz manual page), Palo Alto, CA.
2. A. D. BIRRELL AND B. J. NELSON (February 1984). Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems* 2, No. 1, 39–59.
3. LUCA CARDELLI ET AL. (November 1989). Modula-3 Report (revised). *Tech. Rep. 52*, DEC SRC, Palo Alto, CA.
4. LEO GEURTS, LAMBERT MEERTENS, AND STEVEN PEMBERTON (1990). *ABC Programmer's Handbook.* Prentice-Hall, London, ISBN 0-13-000027-2.
5. P.R. MCJONES AND G.F. SWART (September 1987). Evolving the UNIX System Interface to Support Multithreaded Programs. *Tech. Rep. 21*, DEC SRC, Palo Alto, CA.

6. S.J. MULLENDER, G. VAN ROSSUM, A.S. TANENBAUM, R. VAN RENESSE, AND J.M. VAN STAVEREN (May 1990). Amoeba: A Distributed Operating System for the 1990s. *IEEE Computer Magazine* 23, No. 5, 44–53.
7. A.S. TANENBAUM, R. VAN RENESSE, J.M. VAN STAVEREN, G.J. SHARP, S.J. MULLENDER, A.J. JANSEN, AND G. VAN ROSSUM (December 1990). Experiences with the Amoeba Distributed Operating System. *Communications of the ACM* 33, No. 12, 46–63.
8. G. VAN ROSSUM (April 1988). STDWIN — A Standard Window System Interface. *Tech. Rep. CS-R8817*, CWI, Amsterdam.
9. G. VAN ROSSUM (1990). AIL — A Class-Oriented Stub Generator for Amoeba, in *Workshop on Progress in Distributed Operating Systems and Distributed Systems Management*, 13–21, E W. Schröder-Preikschat and E. W. Zimmer (eds.), Springer Verlag.