PyRemote: Object mobility in the Python programming language

by

Petter Häggholm

B.Sc., Bishop's University, 2005

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

August 2007

Abstract

The current trend in computation is one of concurrency and multiprocessors. Large supercomputers have long been eclipsed in popularity by cheaper clusters of small computers. In recent years, desktop processors with multiple cores have become commonplace. A plethora of languages, tools, and techniques for dealing with concurrency already exist. Where concurrency and multiprocessors meet modern, highly dynamic languages, however, is uncharted territory.

Traditional distributed systems, however complex, tend to be simplified by assumptions of type consistency. Even in systems where *types* and not merely instances and primitive objects can be serialised and distributed, it is usually the case that such types are assumed to be static. The Python programming language, as an example of a modern language with highly dynamic types, presents novel challenges, in that classes may be altered at runtime, both through the addition, removal, or modification of attributes such as member variables and methods, and through modifications to the type's inheritance hierarchy.

This thesis presents a system called PyRemote which aims to explore some of the issues surrounding type semantics in this environment.

Contents

Abstract ii

Contents iii

List of Tables vi

Listings vii

Acknowledgments viii

Dedication ix

Chapter 1 Introduction 1

1.1 Secondary objective 2

Chapter 2 Related work 4

2.1 Distributed programming languages 4

2.1.1 Emerald 4

2.1.2 Distributed Smalltalk 5

2.1.3 Erlang 6

2.1.4 RTSync 6

2.2 Distributed toolkits for Python 7

2.2.1 PYRO 7
2.2.2 RPyC 9
2.2.3 PyRCall 10
2.2.4 Thoughts on distributed toolkits 10

Chapter 3 The CPython implementation 12

3.1 Python objects 12
3.1.1 `PyObject` 12
3.1.2 `PyTypeObject` 14
3.2 Name resolution 16
3.2.1 Method resolution order 16

Chapter 4 Object mobility design 18

4.1 PyRemote in action 19
4.1.1 Example 1: Remote calls 19
4.1.2 Example 2: Object mobility 21
4.1.3 Example 3: Object consistency 21
4.2 Copying objects 22
4.2.1 Immutable objects 23
4.3 Moving objects 24
4.4 Proxies 25
4.4.1 Client-side proxies 26
4.4.2 Server-side object stubs 28
4.4.3 Method lookup and proxies 30
4.5 Augmentation of Python objects 30
4.6 Type and object consistency 31

4.6.1 Real object consistency 32

Chapter 5 Results and evaluation 34

5.1 Design choices and tradeoffs 35

5.1.1 Immutable types 35

5.1.2 Proxy consistency policies 35

5.1.3 Moving objects 36

5.1.4 GUIDs 37

5.2 Local performance 38

5.2.1 PyBench 39

5.2.2 Pyrex 40

5.3 Distributed performance 41

Chapter 6 Conclusion and discussion 44

6.1 The challenge of CPython 44

6.1.1 Type equivalence 45

6.1.2 Pickling limitations 46

6.1.3 Old and new classes 46

6.1.4 Unsupported operations 47

6.2 Future work 47

6.2.1 Garbage collection 47

6.2.2 Object mobility policies 48

6.2.3 Transactional consistency semantics 49

6.2.4 Nameserver 49

6.3 Final thoughts 50

6.4 Obtaining the code 50

List of Tables

5.1 PyBench results 39
5.2 Cost of `PyGuid` functions 40
5.3 Pyrex timings 41
5.4 Distributed performance 43

Listings

3.1 `PyObject` 13

3.2 `PyTypeObject` 15

4.1 PyRemote example 1: Remote calls 20

4.2 PyRemote example 2: Object mobility 21

4.3 PyRemote example 3: Object mobility 21

4.4 `PyObjectProxy_Type` 28

4.5 `PyObjectProxy` 28

4.6 Server side dispatch 29

4.7 New `PyObject_HEAD` macro 31

4.8 Type reassignment 33

5.1 Pyrex testing code 40

Acknowledgments

This space would not be complete without thanks to one's supervisor, and due credit is *definitely* due to Norm Hutchinson, who was not only able to draw on his own experiences in the field to guide me through some of the shoals and reefs as I was trying to navigate the conceptual waters of object mobility, but also understanding during some of the rockier spots of my graduate studies. Many thanks also go to Brett A. Cannon, because having a friend and housemate who is a member of the Python development team was a tremendous help in getting started and getting over the warts and bumps of the CPython source.

Traditionally, this space is also used to thank one's parents (*Thank you, Mum!*), but while I don't want to seem filially ungrateful, I'd like to break away from the trend and give some particular credit to those who so rarely *get* any in these hallowed thesis spaces: My grandparents, without whose love and assistance I might never have made it to Canada at all, and my friends (also a distributed system, and more fault tolerant than any ever deployed on computers), without whose support I could not have stayed (more or less) sane during the difficult times since, let alone whilst working on my thesis (if sanity this can be called).

To my dad, who first got me interested in computers.

I wish you were here to read this.

Chapter 1

Introduction

The problem of object distribution and distributed computations is in no way a novel one. However, few applications are very widely known and used. The reasons for this are manifold. The most obvious and critical one is that it is an area riddled with very *difficult* problems that have perhaps never been solved satisfactorily. This thesis does not aspire to address these fundamental issues of object distribution.

However, other reasons why distributed objects have never 'caught on' may include the fact that most attempts at tackling the problem have been either very academic in nature, or very finely tuned to highly specific applications. While object distribution has been implemented in programming languages, such as Distributed Smalltalk or Emerald, in the past, these are not widely used languages and so have not been able to harness a large developer base, nor the benefits of pre-existing, powerful tools and libraries. There are also *libraries* for existing languages that aspire to provide distributed object capabilities. However, these are inevitably restricted by the fundamental limitations of the languages, and even language implementations, along which they are used.

The PyRemote project aims to tackle an old problem from a different di-

rection: Neither the creation of a novel language, which—though it may be better designed from the ground up to be compatible with distribution requirements—suffers the inertia of any new programming language, most of which never meet with wide acceptance; nor the addition of a library, which will generally have no syntactic support and be *limited* by the underlying implementation; but by the *modification* of an existing, dynamic, widely used programming language, Python, to support the necessary features and capabilities.

1.1 Secondary objective

A secondary objective of the PyRemote project is to address certain limitations within the Python language, particularly in its canonical interpretation; the Python interpreter, written in C (and so usually referred to as *CPython*) available from `http://python.org`. Although the Python standard library contains mechanisms for dealing with multiple threads of control, the ability of a Python program to scale with multiple processors or CPU cores is limited by the *global interpreter lock*, or GIL, which effectively prevents the underlying operating system from running more than one interpreter thread at once even where multiple execution units are available. Past attempts to eliminate the controversial GIL have been either prohibitively difficult or so inefficient that the effort has been counterproductive. There are also arguments that multi*threading* is not the optimal form of concurrency, and that a system based on multiple concurrent processes offers many benefits—in terms of stability, in terms of scaling on multi-core or multi-CPU systems where processors may not have symmetrical access to memory, et cetera.

Although there is nothing to prevent Python programs from executing in a multi-process context, an obvious and basic requirement to use this mechanism for

distributed computation is a method of interprocess communication. Barring the socket interface and a few XML marshalling libraries, no such method exists in the standard Python distribution; in particular, there is nothing to provide transparent inter-process communication (which is to say, convenient, consistent with ordinary Python syntax and semantics, and intuitive). By providing a location-agnostic method of IPC, the PyRemote system attempts to fill this role, and thereby make it easy—or at least easier—to write Python programs that scale to take advantage of multiple processors.

Chapter 2

Related work

2.1 Distributed programming languages

2.1.1 Emerald

The Emerald language [1, 2] was developed in the early 1980's with the original goal of addressing shortcomings in the Eden system, which inspired it. Its primary goals were to offer object-based distributed computation with *uniform semantics* for local and remote objects, and to do so efficiently, in stark contrast to existing systems, where distributed objects tended to be extremely heavy-weight.

Emerald achieved its stated goals, and produced several incidental novelties in terms of its type structure, which can be broadly characterised as *prototype*-based rather than class-based, and its use of type specifications to which objects conformed without concern to internal implementation details (that is, what would today be called *interfaces*)—it is even argued that *true* polymorphism is only achieved when a type specification is polymorphic with respect to implementation [2].

The PyRemote project empathically does not aspire in any academic or theoretical way to transcend Emerald, which presented a very clean solution to problems

in distribution. Rather, it is one of few known attempts to provide a reasonable subset of its features—uniform local and remote semantics, object mobility, et cetera—in a more widely used programming language; languages born in academia, such as Emerald, may boast pure and elegant solutions to theoretical problems, but are often stillborn in terms of widespread adoption, as any new programming language necessarily lacks comprehensive libraries, platform bindings, and so forth.

Emerald was one of the primary inspirations for the PyRemote project.

2.1.2 Distributed Smalltalk

John Bennett's work on Distributed Smalltalk ('DS') [3] represents one of the projects most significant to the PyRemote project, and raised many of the issues that still face designers of distributed systems today. It provides transparent remote invocation, automatic marshalling, object mobility (by means of move and copy primitives), and distributed garbage collection, among many other features. The one important feature that DS appears to *lack*, as far as [3] is concerned (and in our view), is class sharing: Objects and their classes must be co-resident; an object's class must be present at a node where it is moved; and no support for class sharing is provided, beyond a weak guarantee that identically named classes, *if already present* on both nodes in a transaction, are compatible.

DS was particularly interesting, given the goals of the PyRemote project, in that it too was an endeavour to add distributed computation facilities to an *existing* language. (It was arguably a simpler task: Smalltalk semantics are explicitly centered around message passing, and the interpreter already had an object reference indirection table.)

2.1.3 Erlang

Erlang is a distributed programming language developed by the telecommunications company, Ericsson, to support large-scale, soft realtime applications [4]. Its syntax and semantics, in its 'local' subset, are declarative rather than imperative; it is historically based on Prolog [5], but is in appearance similar to functional programming languages. Concurrency is strictly explicit.

An interesting point to note is that declarative languages by 'nature' and by design tend to have very few statements with side effects—which is to say that they tend to modify very few data structures save perhaps by assignment from return values. This also makes them intrinsically easier to modify to be distributed languages: Most of the problems that make distributed programming so hard stem from such side effects and the difficulties of keeping data structures synchronised across peers in a distributed system; non-existent problems in a purely functional language except perhaps in maintaining a namespace hierarchy. Functional subsets of languages are therefore interesting sections to inspect for possible exploitation in parallel and distributed languages.

2.1.4 RTSync

The RTSync project [6] represented an attempt to create a programming language offering soft real-time constraints in a distributed system. It was implemented by Petter Häggholm and Scott Stoddard in the summer of 2004 to support theoretical work by Dr. Stefan D. Bruda. It offered a multi-threaded, actor-based programming environment with object messages passed through *synchronizer* entities. Programmers can also express end-to-end timing constraints; synchronizer entities schedule message passing in order to satisfy these constraints, and programmers may specify

the consequences of timing failures. The RTSync compiler infers, wherever possible, local 'sub-constraints'.

Although the language itself (if not the runtime system!) was fairly simple, it is mentioned here as representing the author's earliest experience with implementing a distributed programming language.

2.2 Distributed toolkits for Python

There exist numerous tools for providing distributed computation in Python (in fact, several new ones have emerged since the PyRemote project was begun). Typically, however, they are limited in scope, in particular as pertains to dealing with *types*, and they tend to be written purely in Python. From an implementation point of view this is a considerably simpler task, but it may come at a cost since Python code will tend to run more slowly. (Although it is often said that Python code will run more slowly than C or C++ code by more than an order of magnitude, properly written Python can be quite efficient—sometimes even faster than Java and almost as fast as C++, if startup costs are ignored or amortised [7]. However, the issue here is that implementing proxies in Python would involve more lookups and indirections than code that can be injected directly into the type structures of the interpreter.)

2.2.1 PYRO

PYRO (Python Remote Objects) is a distributed object system written in pure Python [8]. It appears to present a very functional distributed object system (by far the most substantial project we found when we surveyed the field to see what efforts had been made in this area for the Python language). It supports sharing of arbitrary objects, so long as they can be "pickled" by the standard `pickle` module

(for object serialisation), and handles semantics in an intuitive fashion (there are no special restrictions on objects as function parameters, et cetera). Exceptions are handled elegantly: Not only are they propagated to the caller, but the stack trace (attached to Python exceptions) is transmitted as well, to avoid confusing a programmer attempting to debug software with a stack trace in which the 'bottom' frame is a proxy call, with no hint as to the stack on the node where the actual invocation happened.

PYRO provides an event server system (publisher/subscriber) to ease event-driven programming. There is also support for asynchronous function calls. To provide some security, PYRO offers *connection validators* to restrict clients allowed to connect to services, and *code validators* to control whether mobile code is allowed from any given client. It should be noted, however, that Python itself is not a secure programming language.

Limitations

PYRO has two different kinds of proxies: 'Simple' proxies, which are the default type, and 'dynamic' proxies, which support attribute access. It follows, therefore, that unless one explicitly requests such a proxy, the operations performed on proxies are constrained. The distinction exists because dynamic proxies are much slower; however, this may confuse programmers and makes PYRO code much less transparent. Proxies also cannot be shared between threads. It is not clear from the documentation why this is so. Furthermore, while in a sense arbitrary objects can be shared, objects must be either derived from a PYRO base class or decorated with a PYRO delegate object. There are some exceptions: Not all objects can be 'pickled'. For example, objects such as file and socket objects cannot be shared at

all.

Although PYRO has limited support for mobility, it is not without problems and subtleties. For example, dynamic proxy objects can cause unexpected exceptions to be raised if types the involved are not present on the caller side of a remote invocation: There is no support for automatic transmission of classes not present on one peer in a distributed system. (Classes that are present but not loaded are imported automatically; this is a regular feature of Python's `pickle` module.) Classes can be shared across different nodes, but only if they are available in separate modules—classes in the `__builtin__` namespace cannot be shared at all. More unexpected problems arise when accessing nested attributes (`an_ob.an_attr.sub_attr`).

Very importantly, there is no guarantee of consistency. Once a class is copied from one host to another, PYRO will do nothing to ensure that the two stay consistent if one side is altered.

2.2.2 RPyC

RPyC (Remote Python Call) is a Python library providing remote calls [9]. It aims to provide (largely) transparent access to a remote 'slave' interpreter, though it does not provide any form of code mobility. Remote namespaces are brought into variables in the local namespace, allowing invocations to be made to slave interpreters. Remote objects are represented by proxies, or (in the case of immutable objects) copied by value. Exceptions thrown by remote calls propagate to the caller side in an intuitive manner.

RPyC also supports *asynchronous* calling, where the results of a remote call may be polled to determine completion.

Limitations

RPyC does not (and does not aspire to) offer any kind of object or code mobility; it is a remote call system and no more. Furthermore, it interacts poorly with threaded programs.

2.2.3 PyRCall

PyRCall (Python Remote Call Module) is another remote call library for Python [10]. (Unfortunately, the documentation we were able to find was somewhat sparse.) Similar to RPyC, PyRCall offers remote access to objects. Compared to RPyC, the remote calls are much more explicit in that access to remote names involves explicit PyRCall function calls rather than importing namespaces. Security is offered through SSL connections and through authorisation keys whereby access to server objects may be limited.

Limitations

PyRCall appears to offer no support for code or object mobility.

2.2.4 Thoughts on distributed toolkits

Although the toolkits discussed in previous sections are quite functional, it is our opinion that none of them are sufficiently transparent to be entirely convenient to programmers. Two of them are in no significant way comparable to the aims of the PyRemote project, as they provide only remote function calls, rather than full-featured mobile objects. (There are, in fact, some smaller or less mature toolkits that do much the same, not surveyed here. None of them, as far as we could find, are more ambitious.) PYRO goes much further. However, none of the toolkits found

or investigated offered all of the following features, which we consider critical to transparency:

- Proxy representations of objects
- Object mobility
- Marshalling and automatic transmission of types
- A single set of uniform semantics on proxies and objects

The PyRemote project provides all of these features.

Chapter 3

The CPython implementation

PyRemote builds upon the canonical Python implementation, 'CPython', which is written in C. (The version written for this thesis is based on the version 2.4.3 source code.) Although CPython is a fairly large piece of software, consisting of some 366,000 lines of code, much of this resides in modules not relevant to the PyRemote implementation. No changes were made to the parser, bytecode compiler, stack machine, and so forth, and although some understanding of some of these components was necessary in order to design PyRemote and decide where best to implement changes, they have no part in the implementation. This chapter thus discusses some key facets of the CPython implementation that that are fundamental to understand the low-level design of the PyRemote project.

3.1 Python objects

3.1.1 `PyObject`

Because the implementation language is C, and because Python is an object-oriented language with polymorphism and other features associated with OOP, CPython

Listing 3.1: The basic Python object

```
/*
 * This is a simplified version of the PyObject_HEAD macro
 * See Include/object.h
 */
#define PyObject_HEAD           \
        int ob_refcnt;          \
        struct _typeobject *ob_type;

typedef struct _object {
    PyObject_HEAD
} PyObject;
```

implements inheritance "by hand". The preprocessor macro `PyObject_HEAD` defines the minimal information that all objects need, such as reference count, object type, and (depending on compilation options) certain optional fields to do with garbage collection and reference tracing. The basic object type in CPython, `PyObject`, contains only this information. Listing 3.1 provides an approximate idea of what the `PyObject` structure looks like (the real code is complicated by more, and more sophisticated, preprocessor macros for debugging and variable-size objects). By making every object (including type objects) contain `PyObject_HEAD` as its first item, it is then possible to perform generic operations uniformly on all objects by casting them as `PyObject` pointers. Furthermore, since the object type is one of the basic data whose offset is known in all type structures, type lookup can also be performed.

Objects may be either pure Python objects, or 'C objects'. The core built-in types such as `type`, the `object` base class, `int`, `str`, and so forth, are all implemented in C. The C interface allows programmers to write Python extensions in C, which are largely transparent to the Python programmer[1].

[1]Typically, C types are somewhat 'less mutable' than pure Python types; so for instance the methods of built-in types cannot be altered.

3.1.2 `PyTypeObject`

The *type* of a Python object is defined either in a structure generated at runtime (in the case of all user-defined, and many library classes), or in a (generally static) structure directly in the C code. The `PyTypeObject` is an structure containing all of the information pertaining to a type, such as the name, allocation information, and pointers to all of the operations on objects of this type that are implemented in C. We shall use the convention of referring to these as *built-in* methods, as opposed to regular methods, which are implemented in Python. Built-in methods include allocation and deallocation, as well as common, low-level operations such as calling (on callable objects such as functions and function objects), comparisons, hashing, and producing a string representation. (They are not 'common' in the sense of being common to *all* objects, but rather in the sense of being sufficiently common, and sufficiently commonly *invoked*, to merit special, efficient lookup.)

Optionally, the structure may also contain sub-structures of common operations to arithmetic types, *sequence* types (such as tuples and lists), *mapping* types (such as dictionaries), and buffer types (providing raw memory access). See Listing 3.2 for a truncated view of the `PyTypeObject` structure. The real `PyTypeObject` structure contains many more types of function pointers, more method suites, and many more built-in methods, but the example should suffice to make the general idea clear.

Method lookup for a built-in method of an object o follows the code path outlined (in rough terms) below:

- The object is passed to an interpreter function that acts on `PyObject` pointers, such as `PyObject_Repr()`, which returns a (Python) string representation of

Listing 3.2: The object type object

```
/* These are only excerpts; refer to Include/object.h */
typedef PyObject * (*unaryfunc)(PyObject *);
typedef PyObject * (*binaryfunc)(PyObject *, PyObject *);
typedef PyObject *(*getattrfunc)(PyObject *, char *);
typedef int (*setattrfunc)(PyObject *, char *, PyObject *);

typedef struct {
    binaryfunc nb_add;
    binaryfunc nb_subtract;
    unaryfunc nb_positive;
} PyNumberMethods;

typedef struct _typeobject {
    PyObject_VAR_HEAD
    char *tp_name;
    destructor tp_dealloc;
    getattrfunc tp_getattr;
    setattrfunc tp_setattr;
    PyNumberMethods *tp_as_number;
    PySequenceMethods *tp_as_sequence;
} PyTypeObject;
```

its object.

- The function looks up the object's type object:

  ```
  PyTypeObject *tp = o->ob_type
  ```

- Generic operations are invoked through the intrinsic method pointers, whose offsets are known (they are fields in `PyTypeObject`).

- Invoke the function pointer with o as its first parameter:

  ```
  PyObject *result = tp->tp_str(o);
  ```

Although this is a very simplified account (partially for historical reasons, attribute lookup in CPython can internally be rather complex), it is sufficient to explain how PyRemote exploits these structures. It should be noted that there is an entirely different, more general code path for Python methods that are *not*

necessarily built-in operations. Below is an example for lookup up some method m of an object o:

- The interpreter looks up the object's dictionary, `o.__dict__`
- The method `m` is looked up through an ordinary dictionary lookup:
  `met = o.__dict__[m]`
- `met`'s `__call__()` method is invoked with the proper parameters.

Although more general, this codepath is of little interest to PyRemote, since it is handled automatically, so long as the low-level method lookups providing services such as dictionary lookup and the `__call__()` method are forwarded as appropriate. These lower-level lookups correspond to the first code path outline.

3.2 Name resolution

Name resolution in Python is generally performed at runtime, using dictionaries. This is an extremely general approach and is reflected frequently throughout the system; so for instance a local variable within a function is looked up in the function's dictionary, global variables in a global dictionary, an object's attributes in the object's dictionary, et cetera. Although not very fast, this mechanism enables Python to be a highly dynamic language, and manipulating key facets is often as simple as altering values in a dictionary.

3.2.1 Method resolution order

When a method on a Python object is invoked—or indeed any attribute lookup is performed—the interpreter potentially performs a *series* of lookups in object dictio-

naries. Initially, the dictionary of the object itself is queried; if the desired attribute is not found, its *type* is queried; if again it is not found, then the superclasses of this type are queried. The precise lookup order is non-trivial. "Old-style" Python classes (see Section 6.1.3) used a simple algorithm; superclasses were queried depth-first in the inheritance graph, left to right. However, this approach is problematic in that it is not always monotonic. "New-style" Python classes, since version 2.3, use a more sophisticated ordering which guarantees monotonicity [11]. The general principle, however, is much the same; only the *order* of the dictionary lookups is changed.

Chapter 4

Object mobility design

When we speak of 'object mobility' and the concept of 'moving' an object from one computer to another, there are several semantic interpretations of what it means for host A (the client in this particular transaction) to request an object o from host B (the server). Possibilities include

- Sending a *copy*, where A receives a copy o_A completely independent of o;
- Sending a *proxy*, where the object remains on B and any operations A performs on it are performed remotely, on B;
- Actually *moving* the object, where A now acts as a server for o and B is left with a proxy p_o, referring to o—this is equivalent to a copy and a deletion;
- *Replicating* the object, where A and B each hold a 'real' object (as opposed to a proxy) but, unlike in the copy operation, they conceptually refer to the same object, so that any changes to o_A are reflected in o_B.

The default mechanism for object mobility in PyRemote is proxy semantics, but there is also support for copy and move semantics—although due to limitations

of the implementation and the constraints of working with the existing CPython code base, not all objects can be be moved, and some (though fewer) objects cannot be copied. Immutable objects comprise an exception to this general rule as they are always copied; see Section 4.2.1. Meanwhile, there are some objects, primarily resource handles, for which move operations do not make much sense (although remote access through proxies is still relevant and functional). *Types* are also handled differently, as it is not feasible to have strictly remote type objects, as they are enmeshed rather deeply in the type system, and as they are typically read vastly more often than they are modified; it is more sensible to *replicate* rather than move them.

4.1 PyRemote in action

For readers unfamiliar with our use of terminology, inspired primarily by the Emerald language, this section provides a number of examples derived from the PyRemote test code to demonstrate how its facilities may be used. In order to show both code and results, the Python interpreter's interactive mode was used. Lines in the examples indented with >>> or ... are code entered into the interpreter; lines not so prefixed are results printed to the standard output.

We do not show server code; the server side of these transactions consists solely of launching a server thread and publishing an entry object.

4.1.1 Example 1: Remote calls

A client connects to a server and (in the current, ad-hoc system) receives an advertised object. We may then perform operations on it as though it were a regular Python object, including type lookup, member listings, and method calls; see sec-

tion 4.4. Although the object appears in every respect identical to local object, the `isproxy()` function is provided to identify proxies.

It is even possible to modify the type of the remote object.

Listing 4.1: PyRemote example 1: Remote calls

```
>>> import pyremote
>>> # Obtain a remote object proxy:
>>> remote_ob = pyremote.connect()
>>> # Inspect the proxy:
... dir(remote_ob)
['__add__', '__class__', '__delattr__', '__dict__', '__doc__', '__getattribute__',
 '__hash__', '__init__', '__module__', '__new__', '__reduce__', '__reduce_ex__',
 '__repr__', '__setattr__', '__str__', '__weakref__', 'another_foo', 'class_name',
 'dict_str', 'dir_str', 'foo_class', 'frobble', 'named_foo', 'some_list', 'ugly_class',
 'x']
>>> type(remote_ob)
<class 'testclass.Foo'>
>>> isproxy(remote_ob)
True
>>> # Remote method call:
... remote_ob.frobble()
-14
>>> # We may modify the remote type:
... type(remote_ob).frobble = lambda self: self.x
>>> remote_ob.x = 22
>>> remote_ob.frobble()
22
```

4.1.2 Example 2: Object mobility

Given a remote object, we may request the real object by means of the new function `real()`, wherein the object is moved to the local peer; see sections 4.2 and 4.3. This is a 'soft request'; if the object is immovable, it will return a proxy object (the argument). Note that at present, this only works for objects in certain namespaces (see section 5.1.3).

Listing 4.2: PyRemote example 2: Object mobility

```
>>> import pyremote
>>> remote_ob = pyremote.connect()
>>> local_ob = real(remote_ob.another_foo())
>>> isproxy(local_ob)
False
>>> type(local_ob)
<class '.Foo'>
>>> # Note (above) that although we have copied the class, it is now a member of no
... # module, since the module was remote and is not copied.
...
>>> local_ob.frobble()
-14
```

4.1.3 Example 3: Object consistency

If a type is distributed over multiple machines, as it must be when real objects are copied, it should be kept synchronised. PyRemote does this, although it is a broadcast system that does not guarantee absolute consistency (there are no strict transactional semantics).

Listing 4.3: PyRemote example 3: Object mobility

```
>>> import pyremote
>>> remote_ob = pyremote.connect()
>>> # We obtain a purely local copy with the same type
... local_ob = real(remote_ob.another_foo())
>>> type(local_ob).fruble = lambda self: 2
>>> remote_ob.fruble()
2
>>> type(local_ob).fruble = lambda self: 1
>>> remote_ub.fruble()
1
```

4.2 Copying objects

Copying objects is simple and, in general, achieved by using Python's built-in functionality through such tools as the `pickle` and `marshal` modules. `pickle` [12] is the primary tool used by Python programmers for marshalling and serialisation; it uses an extremely simple stack-based language to encode objects in a portable format suitable for transmission, and can handle quite arbitrary (though not all) objects as well as recursive structures. Copying is therefore—as far as implementation is concerned—a fairly trivial issue.

The main exception to this rule is the copying of *types*, which cannot normally be either 'pickled'[1] or 'marshalled'; however, it is not generally difficult to serialise and transfer their *components* (class name, methods, and so forth). Al-

[1]When the Python `pickle` module encounters a type, it inserts a reference by fully qualified name. To 'unpickle' it, therefore, it *imports* it, requiring the module where the type is defined to be already present. This is a space efficient way of serialising objects with known types, but provides no facility for type transmission.

though intrinsic facilities do not exist to do this, the `marshal` module is capable of marshalling Python bytecode, including the `code` objects representing the bytecode of functions.

Built-in types, of course, do not need to be copied (and in fact never are copied), as they exist on each peer in a PyRemote system, and share the same static identifier. Extension modules (not part of the standard library) implemented in C (or another compiled language, such as C++) present a special problem. If they are present on both nodes involved in a copy operation, they can be looked up and imported by name—this is done automatically. However, if only one of the nodes has a copy of the extension module, the copy operation will fail. It is not possible, in the general case, to copy binary modules from one host to another. (We do not assume that peers in a system are in any way homogenous, save in running compatible Python/PyRemote interpreter versions.)

We shall discuss further limitations of this system in Section 6.1.2.

4.2.1 Immutable objects

Immutable objects present a special case: Although Python is highly dynamic, there are some built-in types whose objects cannot in any way be modified. These include arithmetic and Boolean types, tuples (although tuple *items* can be modified), and strings. Since they cannot be modified, it is semantically equivalent to hold a proxy to a remote `int` object and to hold a local copy with the same 'unique' identifier.

Another set of objects considered immutable are built-in types, which *must* be present at both peers. (Strictly speaking, it is necessary that the `type` type exist and be functionally equivalent; proxy types cannot be constructed without using `type` as a base type. To simplify and streamline the bootstrap process, we assume

that peers in a PyRemote system will have compatible versions of all the built-in types.)

4.3 Moving objects

To *move* an object, it must be copied to the destination and deleted from the source. Copying is described in Section 4.2; move operations use the same mechanism. The *deletion* poses new challenges in terms of the implementation, because CPython was not originally designed with this in mind. In particular, CPython object references are simple pointers and may be held by references in dictionaries representing variable mappings as well as internal structures and functions, there is no trivial mechanism to trace them. This is unlike languages such as Distributed Smalltalk, which has an object indirection table (although not originally for the purpose of mobility), allowing the mobility system to simply modify this indirection entry to point to a proxy instead of a real object [3].

This problem should not be under-emphasised, nor its impact on the implementation underestimated. In languages *designed* for object distribution, such as Emerald, object references may be made globally unique, and probably *should* be made so [2]. CPython, on the other hand, is designed to work within a single address space. It was therefore necessary to add to every object a globally unique identifier. However, this identifier cannot be used for local object lookups: Both in terms of performance and implementation effort, the consequences would be disastrous. The version 2.4.3 source code of the CPython interpreter contains 20,035 mentions of `PyObject` pointers (quite apart from pointers to *specific* types of objects!). However, the vast majority of objects on a given node will never be addressed by another node. Only those objects that are actually *used* in a 'distributed manner'—that is,

those which are sent as proxies, sent as copies, or distributed to other machines—truly *need* more sophisticated references. (Then, of course, the pointer value is no longer sufficient as an identifier as two different objects in different address spaces may have the same memory address.)

The approach taken to the object-move problem is fairly straightforward: PyRemote attempts to replace any 'external' references to an object—that is, references not held by PyRemote internal structures—by references to a proxy. If all external references can be resolved, the object is considered 'mobile' and is moved; if some references remain unresolved, then the object is considered 'immovable' and the request to move it is refused.

In searching for references to the object, there are various locations to consider (see Section 3.2 on name resolution in Python). Each *module* currently loaded has a dictionary mapping strings (identifiers) to Python objects. The dictionaries of all loaded modules may thus be searched (linearly) for keys corresponding to the target object, and all such references can be replaced. Similarly, we may scan the dictionaries of objects (and type objects) containing their attributes, and the dictionaries of closure objects belonging to active functions and methods. See Section 5.1.3 for a discussion on the possible policies.

4.4 Proxies

When a host A operates on some object o that is not local—that is, it resides wholly on some remote node B and is not copied to A—it is represented on A by a *proxy object* p_o, whose task is to forward any requests made of it to its 'target' object o. This structure is similar to classical remote procedure call implementations—indeed, the basic idea of the structure dates back to at least the 1980's [13]. The primary

concern in the design of the proxy objects of the PyRemote project was to achieve *transparency*—as much as possible, it should not be (and is not) visible to a user whether an object is a local, 'real' object or a proxy to a remote object. Not only are all requests (attribute lookups, method calls, et cetera) forwarded to the target; an object o of type T is logically considered to have the exact same type as the proxy p_o referring to o. A new built-in function, `isproxy()`, is made available to distinguish proxies from 'real' objects.

4.4.1 Client-side proxies

The basic mechanism of implementing proxies in CPython is relatively straightforward in principle, if not in detail. Every object contains a reference to its *type structure*, as described in Section 3.1.2. The type object tracks built-in operations by function pointers, and functions that operate on generic `PyObject` pointers do so by performing function pointer lookups.

PyRemote works by effectively intercepting these lookups. In principle—an idealised implementation, such as might have been used in a language designed from scratch—a proxy object would simply refer to a generic `Proxy` type. The generic operation 'slots'[2] in this ideal class contain pointers to *proxy* functions, which marshal the arguments and send them to the appropriate peer, which would unmarshal them and, via a generic object *stub* function, dispatch them to the real object. The results may then be marshalled and returned to the proxy.

If a remote invocation causes an exception to be thrown, this too should be intercepted, marshalled, and re-raised on the caller side. (That is, if p_o on host A

[2]We here use the word 'slots' in a rather generic way; this is not to be confused with the Python concept of slots for a fixed set of user-defined variables, saving space by using a fixed-size `__slots__` object member rather than a variable-sized dictionary member, `__dict__`. (See `http://docs.python.org/ref/slots.html`.)

invokes an operation on o on host B, causing an exception to be raised, the caller on host A should be the one to deal with it—not host B.) The current PyRemote implementation does intercept, marshal, and transmit exceptions, although some traceback information is lost since the current implementation of `pickle` does not support pickling of `traceback` objects (see Section 6.1.2).

In reality, because CPython was not designed with remote invocation considerations in mind, the implementation is more complex. Type structures do *not* contain only function pointers, but also data, which are specific to types, such as the type name. It is therefore necessary to create a proxy type corresponding to *each* real type; because it would be prohibitively (and unnecessarily) expensive to do so on interpreter startup (or class creation time), this is done on demand. In general, however, these proxy types work much as the ideal `Proxy` type outlined above; indeed the creation of most of the proxy dispatch functions was automated. The difference is strictly the caching of type-specific data.

PyRemote does, in fact, have an ideal proxy type (`PyObjectProxy_Type`), although no objects of this type are ever created. Instead, it is used as a *template* for specific proxy types. A partial listing of this class is given in Listing 4.4. When creating a real proxy type, a new `PyTypeObject` is allocated, and its fields are copied from either `PyObjectProxy_Type`, for all forwarding methods, or the concrete type (the 'target' of the proxy), for data fields such as the name. PyRemote is thus able to create, at runtime, a proxy representation of any type, whether it is implemented in a C extension or in Python code.

Actual proxy objects (as opposed to proxy *type* objects) are extremely simple; they contain effectively no data (since their task is to forward requests to remote locations where the data actually reside). The full layout of a `PyObjectProxy` object

Listing 4.4: The object proxy type

```
PyTypeObject PyObjectProxy_Type = {
    PyObject_HEAD_INIT(&PyType_Type)
    0,                      /* ob_size */
    NULL,                   /* tp_name; provided by target */
    sizeof(PyObjectProxy), /* tp_basicsize */
    0,                      /* tp_itemsize */
    proxy_tp_dealloc,       /* tp_dealloc */
    NULL,                   /* tp_print (deprecated); unsupported */
    proxy_tp_getattr,       /* tp_getattr */
    proxy_tp_setattr,       /* tp_setattr */
    ...
};
```

Listing 4.5: The proxy object

```
typedef struct {
    PyObject_HEAD;
    PyWeakReference *real_object;
} PyObjectProxy;
```

is given in Listing 4.5. For efficiency reasons, and because there are circumstances under which proxies may exist to objects residing on the *local* machine, proxy objects contain *weak references*[3] to targets residing in the same address space; these allow much more efficient lookup of such objects, while adding only a negligible cost (compared to the IPC mechanism) to remote lookups.

4.4.2 Server-side object stubs

The server side of remote invocations is considerably simpler. Conceptually, we might envision an object *stub* for each and every type for which a proxy type exists; much as we have a generic `PyObjectProxy_Type`, we might have a generic

[3]Weak references are objects that look like ordinary references, but do not add to an object's reference count. If the target disappears, an attempt to use the weak reference will raise an exception. Weak reference objects can also be queried efficiently; PyRemote will not raise and catch exceptions every time a remote proxy is referenced.

Listing 4.6: Server side dispatch

```
PyObject *server_dispatch(PyObject *remote_tuple) {
    ...
    switch(opcode) {
    case MP_LENGTH:
        if(PyRemote_ExtractArgs(remote_tuple, "O", &u) != 0)
            goto error;
        if(u->is_proxy)
            goto not_here;
        x = PyMapping_Length(u);
        res = PyRemote_BuildArgs(opcode | REMOTE_RESPONSE, "i", x);
        break;
    case ...
        ... break;
    error:
    default:
        Py_XDECREF(res);
        res = NULL;
        if (!PyErr_Occurred())
            PyErr_Format(PyExc_RuntimeError, "Object stub error");
        break;
    }

    ...
    if(PyErr_Occurred())
        res = bundle_exception();
    ...
    return res;
}
```

`PyObjectStub_Type`, with an instance for each remotely referenced object. It turns out, however, that this is not necessary, since all lookups on the conceptual server stubs are *function* lookups rather than data lookups (as data lookups cannot be forwarded and must therefore be handled by mirroring data in each proxy). A single `server_dispatch()` function can therefore unmarshal requests and dispatch them through the appropriate function. Listing 4.6 presents a snippet of the dispatch code.

4.4.3 Method lookup and proxies

Due to the nature of Python's method resolution order (see Section 3.2.1) and method lookup process, it is frequently the case that method lookups require access not only to an object's dictionary, but also to its type object's dictionary, and potentially a number of supertype dictionaries. This could easily lead to very large numbers of lookups, which might get prohibitively expensive if each lookup involved a pair of remote messages. However, each proxy contains a shallow copy of its target's dictionary. (Dictionary keys are always copied; this is never problematic since Python requires dictionary keys to be immutable objects, and PyRemote already copies immutable objects. The dictionary *values* are copied by proxy.) As such, although *invoking* a method will cause remote messages to be sent, querying an object for the method proxy is a purely local operation.

4.5 Augmentation of Python objects

Although PyRemote mainly *added* to CPython rather than modifying it, some modifications were necessary. One of the more significant is that the `PyObject_HEAD` macro has been modified to contain information necessary for remote objects (see Listing 4.7): A flag (`is_proxy`) was added, as well as a field for the globally unique identifier (GUID) necessary since an address unique within a single address spaces may clash with an address on another node. This increases the size of the basic Python object by `sizeof(int)+N` for an N-byte GUID. The current implementation uses a 128-bit (16-byte) GUID.

Another (more minor) modification was a conditional dispatch entered into the attribute accessor and mutator functions of `PyTypeObject`, the type of types:

Listing 4.7: New `PyObject_HEAD` macro

```
#define PyObject_HEAD           \
    int ob_refcnt;              \
    struct _typeobject *ob_type; \
    PyGuid ob_id;               \
    int is_proxy;
```

In the general case, the type of a proxy is itself a special proxy type, but this regression must terminate, and the type of the special proxy type itself is simply `type`, represented internally by `PyTypeObject`; so although there *exists* a proxy for this type, its functions must be invoked through regular type operations.

4.6 Type and object consistency

A crucial problem in distributed systems in general, and one of the issues which this project aimed to explore, is the notion of *consistency* when objects are either copy-replicated or distributed by proxies: What changes made on one node need be reflected on other nodes, and to what degree is this achievable?

The problem is particularly poignant in Python, due to the extremely dynamic semantics of the language. Unlike many other programming languages, Python allows the creation of types at runtime; furthermore (and more problematically), types are first-class, *mutable* objects; finally, object type assignments are mutable—to say that object o has type t means only that o's type reference points to t; it may be reassigned to some other type t'. Thus, at runtime, an object may have its type changed. For types implemented in Python, the *type* of an object does not affect its layout (all variables are stored by name in the object's `__dict__` member). (See Listing 4.8 for an example.) Since, in general, an object's attributes are simply stored in its dictionary (this applies to type objects as well as to other objects),

it is similarly the case that classes may acquire methods (and class attributes) at runtime, and even type names may change.

For proxies, the solution is quite straightforward: The generic mechanism for modifying an object is a special method called `__setattribute__()`, which the proxy mechanism already intercepts. (Additionally, data members such as `__dict__`, the member dictionary, may be modified; however, these can already be represented as dictionary *proxies*. For regular objects, the simple process outlined here is sufficient. The difficulty arises, once again, with type objects; once again, because only method calls, not data accesses, are forwarded. When the server modifies the real type object, the information cached by proxies becomes stale. A type object that is updated should send an update message to all of its proxies (or broadcast an update message, if it does not know where they are). The node *requesting* the change receives update information in the reply. See Section 5.1.2 for a further discussion of this subject.

4.6.1 Real object consistency

Besides proxies, we also need to ensure that *types* are kept consistent across the system, lest we encounter unexpected behaviour when moving or copying an object from one host to another where the type *exists* on both nodes, but is not equivalent. It would perhaps be desirable to also be able to keep objects consistent—in other words, to be able to distribute an object, in order to make it faster to access and more resilient to failure—but this would be very difficult to efficiently implement; it is also our opinion that this is much less important than maintaining type consistency. It is particularly important that types be kept consistent because they may be copied implicitly by PyRemote support routines (for example, if a node A

Listing 4.8: Type reassignment

```
class Foo(object):
    def __init__(self):
        self.x = 5
    def foo(self):
        return self.x * self.x

class Bar(object):
    def __init__(self):
        self.x = 'hello'
    def foo(self):
        return self.x

f = Foo()              # x = 5, foo() -> x * x
f.foo()                # returns 25
f.__class__ = Bar      # foo() -> x
f.foo()                # returns 5

b = Bar()              # x = 'hello', foo() -> x
b.foo()                # returns 'hello'
b.__class__ = Foo      # foo() -> x * x
b.foo()                # TypeError: can't multiply sequence by non-int
```

requests a real object o from another node B to replace a proxy p_o, its type is copied from B to A, invisibly to the user). Obviously, it would be highly undesirable for implicit operations that the programmer may not be aware of to risk introducing inconsistencies.

PyRemote's method to cope with this is to mark all type objects that are distributed across two or more nodes as being *synchronised.* Whenever a synchronised object's attributes are modified (through the `__getattribute__` method), the update is pushed onto the network in a message that is broadcast to all the peers that the current node are aware of. (Since the current PyRemote implementation uses TCP/IP v4, it is not a true broadcast message, but rather an asynchronous message sent iteratively over each peer connection.)

Chapter 5

Results and evaluation

The PyRemote system offers most of the features it was envisioned to have. Primarily, it provides object sharing and remote method invocation through the use of proxy objects, in a manner entirely transparent to the programmer. Beyond initialising the system (with a single call) and connecting to a remote peer to obtain a reference to a remote object, a programmer does not need to use any non-standard syntax, semantics, or function calls (although functions to provide additional, *optional* functionality, such as requesting references to real objects—'pull' requests—are provided). With very few exceptions, primarily ones regarding raw memory access (see Section 6.1.4), arbitrary operations are permitted on arbitrary types. Some objects naturally cannot be moved—such as handles to open resources, e.g., files and sockets—but they can nevertheless be accessed remotely, by proxy. When proxies to objects of unfamiliar types are requested, the necessary type information is conveyed automatically and transparently.

5.1 Design choices and tradeoffs

During the implementation of the PyRemote system, some questions arose to which the proper answers at present are unclear and merit further research. In this section we shall discuss a few important policies that have been implemented, but are not necessarily the only—and not obviously the correct—choices.

5.1.1 Immutable types

As noted in Section 4.2.1, PyRemote never creates proxies for certain immutable types, as they can be freely copied. One potential drawback of this—or at least of applying it as 'indiscriminately' as it is presently applied—is that if such objects are merely passed back and forth, large amounts of data may be copied. It might be prudent to perform a check and still present proxy objects for objects such as very large strings, which might represent entire text files. This is not currently done in PyRemote: It would add considerable extra complexity in that there are many operations on string objects that *require* them to be locally present, and it is not clear how the implementation would know when it is necessary to fetch large strings, and when it would be appropriate to leave them where they are.

5.1.2 Proxy consistency policies

It is not exactly clear how the proxy type data cache update should take place, and here is an important tradeoff between efficiency and strict semantic coherence. In order to guarantee total coherence—'Once the type object is considered updated, it is updated everywhere'—we would need a transactional system where, at the very least, all the stale caches are marked stale before the operation takes place and returns an ACK. PyRemote does not implement such a transactional system,

although it would be interesting to investigate how much effort it would take to create it, and to see exactly what the tradeoffs would be. This sounds prohibitively expensive, and indeed we do not expect that a distributed system where *every* remote message obeys transactional semantics would be efficient, it should be kept in mind that this problem in PyRemote exists only for *type* objects, and although they present great flexibility as well as the greater challenge in terms of coherence, we may well expect that messages that modify type objects represent a very small subset of the messages sent during the run of a typical Python program. A transactional system, although expensive, might not turn out to be *prohibitively* expensive.

5.1.3 Moving objects

As discussed in Section 4.3, moving an object is implemented with the usual copy operation and detaching the object from plain object references by replacing all such references with references to a proxy object in lookup dictionaries where it is present. We may envision a number of policies for this process. For example, we might search only the module corresponding to the script with which the Python interpreter is invoked (`__main__`); we might search all 'user' modules (as opposed to modules known to correspond to the standard library); we might search *all* loaded modules; or we might choose some intermediate option (e.g. excluding built-in modules[1]). Other places where references may be held include *object* dictionaries (of attributes: 'member variables') and closures of function (and method) objects.

The current implementation replaces references only in `__main__`[2], and does

[1] 'Built-in' modules are modules that are implemented in C; they are generally fairly 'immutable' and we do not expect to be able to perform the same kind of substitutions in these as in pure Python modules.

[2] Actually, the current implementation, for testing purposes only, attempts to replace references also in a module named `testclass`, if present and loaded.

not look at any classes or objects, but this should be considered an intermediate state of development rather than a final solution. We expect that it is probably not the optimal solution—rather, we expect that replacing references in the `__main__` module would be wise, at the very least, and quite possibly all 'user' modules.

To perform further searches and replacements would be straightforward and simple, although it is not done at the present state of development. The question is not *whether* or *how* this *could* be done, but *if* and *to what degree* it *should* be done. A trade-off must be found between efficiency (both in terms of work on the interpreter, and in runtime performance, as all of the structures mentioned must be searched linearly) and completeness. On one extreme, no objects might ever be moved at all; on the other, every structure capable of holding a reference must be examined; the optimal approach lies obviously somewhere in between, but it is not clear where. To find this general optimum requires profiling of realistic applications making use of the PyRemote framework—and of course, no such applications presently exist.

5.1.4 GUIDs

The present implementation of GUIDs is quite primitive and suitable only for small-scale testing; a 128-byte fingerprint is generated simply by repeatedly calling the system `rand()` function. Although this suffices for the current implementation, deployment on a larger system would need a more intelligent fingerprint that provides (at the least) a reasonably strong probabilistic guarantee that GUIDs will truly be unique.

More problematically, a GUID is generated *every time* an object is created. This is a very wasteful policy: The identifier is only necessary when an object is involved in mobility. Deferred GUID generation has been implemented, but enabling

it causes an extremely obscure bug, possibly caused by a primitive copy within the interpreter resulting in multiple identical objects with the same 'null' identifier. As the system now stands, it would be expensive for a system that allocates large numbers of small objects.

5.2 Local performance

A series of tests were run to examine the performance of the modified interpreter on code that does *not* make use of any distributed features. We consider it essential that the modifications thus made not incur prohibitive costs to normal Python programs. Ideally, no additional cost should be incurred; however, we may expect that a project of this nature (and with only one programmer) should incur some costs on a first attempt. Please note that no major efforts have yet been made to optimise the system.

The system used to run the tests was an Athlon 64 3000+ (2.00 GHz, 512 kB cache) with 1 GB of RAM, running Gentoo Linux (kernel version 2.6.21). The baseline Python interpreter used for comparison is the version that the PyRemote code was based on, version 2.4.3. All interpreters were compiled with `gcc` version 4.1.2. Unless otherwise specified, compilation options were left at the defaults.

The original intent was to run, at the very least, tests using two well-known Python benchmarking suites, PyBench and ParrotBench, as well as Pyrex as an example of a real-world application. Unfortunately, ParrotBench, specifically designed to be extremely demanding in terms of language compliance and used as much as a compliance test as a performance measure, failed to run using our modified interpreter due to an unexpected hash value. It is currently unclear whether this represents an *invalid* value, or an unexpected value that ParrotBench simply does

	Run 1	Run 2	Run 3	Run 4	Run 5	Average
PyRemote	3604	3625	3621	3626	3605	3616
CPython 2.4.3	3285	3275	3300	3239	3277	3275

Table 5.1: PyBench results *(times in milliseconds)*

not accommodate.

5.2.1 PyBench

PyBench is a benchmark suite for Python created by eGenix [14]. It performs a wide variety of tests. Each invocation of PyBench actually runs ten iterations of the entire test suite, producing detailed information as well as an over-all average. Scores here represent averages, and our tests represent entire sets of iterations. Table 5.1 summarises the tests. On average, PyRemote is 10% slower than the stock CPython interpreter. It is not entirely clear exactly why the penalty is so great, since the changes to the interpreter were very small—note that no distributed computation was performed in this benchmark suite.

In order to trace the slowdown, the PyRemote interpreter running pybench was analysed using the `gprof` profiling tool included with the `gcc` compiler suite [15]. In section 5.1.4, we expressed a concern that the lack of deferred GUID initialisation would slow down execution. We therefore examined the profile, extracting the data for initialisation and copying of GUIDs. As seen in Table 5.2, functions responsible for initialising and copying GUIDs accounted for about 1.26% of the program's total running time. This is not a very large number, but the cost of `PyGuid_Make()` is incurred at object creation time, and a test or application that spends more time creating small objects will therefore incur a proportionally higher cost. It is a problem that should be eliminated—but it does not account for the entire slowdown.

Function	% running time	Seconds	# calls
PyGuid_Make	0.97	0.30	83,111,266
PyGuid_Copy	0.29	0.09	No data
PyGuid_Hash	0.00	0.00	2231

Table 5.2: Cost of PyGuid functions

Listing 5.1: Pyrex testing code

```
#!/bin/sh

for i in 1 2 3 4 5
do
    # Run 10 loops, repeat the whole exercise 5 times
    ./python Lib/timeit.py -n 10 -r 5 \
        -s 'from Pyrex.Compiler.Main import compile' \
           'for x in xrange(100):' \
           '    compile("../pyrex-demos/spam.pyx")' \
           '    compile("../pyrex-demos/numeric_demo.pyx")' \
           '    compile("../pyrex-demos/primes.pyx")'
done
```

Branching was added to some built-in `type` operations (see Section 4.6), but profiling information does not indicate that any significant time was spent in these functions.

5.2.2 Pyrex

Pyrex is a programming language created to aid the development of Python modules, allowing a mixture of Python classes with C data types and structures, finally compiled into C extension modules with no burden on the programmer to explicitly create glue code. In our tests, Pyrex was used to compile the bundled example programs: `numeric_demo.pyx`, `spam.pyx`, and `primes.pyx`. As these programs are all very small, each run of the test compiles each program 100 times.

Timing was performed using the standard Python module, `timeit`, which attempts to use the best timing facilities available on any given target platform [16].

	Run 1	Run 2	Run 3	Run 4	Run 5	Average
PyRemote	4.49	4.48	4.52	4.50	4.51	4.50
CPython 2.4.3	7.63	7.58	7.61	7.63	7.59	7.60

Table 5.3: Pyrex timings *(seconds per* `timeit` *loop)*

The script presented in Listing 5.1 was used to invoke the tests. Results are given in Table 5.3. Curiously, PyRemote actually appears to run significantly *faster* than the stock Python interpreter—by as much as 69%! Since no effort has been made to optimise the interpreter, and since there is code in PyRemote that adds more code to be executed (`PyGuid` function calls, etc.; see Section 5.2.1), it seems unlikely that this reflects a true performance increase. If it represents an error in the tests, it is unclear wherein this error consists: Identical output is generated by the Pyrex compiler regardless of which interpreter is used to run it, and the timings obtained by the `timeit` module are consistent with those obtained using the standard UNIX timing utility, `time`.

Although we cannot explain these numerical results, they are presented for completeness and as a curiosity. We recommend that the PyBench results be viewed as representative of the true cost of PyRemote—and a sign that optimisation work should be done.

5.3 Distributed performance

To test the overhead of PyRemote alone, we ran a distributed system of two nodes on the same computer—the results thus include the overhead of the TCP/IP stack, but not network transmission. The test program called a function of a remote object 100 times; the function did nothing but return an integer (which was returned by

copy; see Section 4.2.1). The cost of evaluating the function call itself was negligible (see Table 5.4).

The distributed test system was analysed using the `profile` module of the Python standard library [16]. This particular profiler measures only time spent in Python code, not the lower-level C libraries; total CPU time in this test run (slowed down by the profiler) was measured as 75.58 seconds. The bottleneck is the client; the server spent the vast majority of its time waiting for user input (the test server is terminated by a key press). On the client side, although a few percent of the time was spent in networking code, output formatting, etc., the overwhelmingly largest culprits were the pickling code (≈19.6 seconds; 26%) and the `acquire()` method of lock objects (≈12.9 seconds; 17%). The data are very approximate, and we do not wish to cite detailed data and give the impression of more confidence in these numbers, but should be taken as an indication of problem areas.

It should be noted that Python actually contains *two* pickling modules: `pickle`, which is implemented purely in Python, and `cPickle`, which is implemented in C and, although somewhat less customiseable, is a very great deal more efficient—it is up to 1,000 times faster than the Python implementation [16]. Originally, the PyRemote implementation used the Python `pickle` module for ease of implementation, as some (very minor) changes had to be made; we expect that the system is more efficient using the `cPickle` module. It turns out, however, that the variance of the timings is extremely high, rendering this comparison useless; probably due to non-deterministic interaction of locks and threads (see Table 5.4). We do not show average values for the remote-call runs; the variance is too great for mean values to be meaningful. We may say with some confidence that the cost of a remote invocation—apart from network traffic, which is unpredictable—is *on the*

	Run 1	Run 2	Run 3	Run 4	Run 5	Average
Remote (`pickle`)	138	546	506	155	174	N/A
Remote (`cPickle`)	533	485	532	128	405	N/A
Regular call	0.00845	0.00900	0.00829	0.00823	0.00826	0.00845

Table 5.4: Distributed performance *(seconds per 10,000 calls)*

order of 35 milliseconds (±25 milliseconds, or more).

Running `gprof` to find the C profile of the same code reveals that almost 10% of the program's time was spent in a dictionary lookup method `lookdict_string()`, mostly when called from `GenericGetAttr()`. This is not necessarily indicative of any problems; rather we should expect attribute lookup to consume a great deal of time, since the test involves the repeated lookup and call of a function which itself involves very little processing. (The single most 'expensive' function was the interpreter function `PyEval_EvalFrame()`, which evaluates a stack frame object; it accounted for some 34% of the runtime. It may be worth noting that it occupied over 54% of the time in the PyBench test.)

We conclude that the PyRemote source code would benefit from retooling in terms of the locking scheme. Also, the data marshalling, even apart from switching to a version of the more efficient `cPickle` module, is unoptimised and can probably be made much more efficient. The locking scheme, however, is likely to be the greater bottleneck in more complex scenarios where more threads compete for locks on resources related to greater numbers of remote peers.

Chapter 6

Conclusion and discussion

6.1 The challenge of CPython

The PyRemote project aspired to add capabilities to the Python programming language. This has obvious benefits—it harnesses the power of an existing and well-tested programming language with a powerful standard library, many third-party libraries, existing applications, and developers. However, it also creates many challenges in that the pre-existing system was in no way designed to support the type of modifications that were necessary, and in some cases was not easily amenable to these changes.

Although Python, as a programming language, is simple, consistent, and employs an explicit policy that "There should be one, and preferably only one, obvious way of doing something"[1], CPython is often rather confusing 'under the hood'; so for instance there are several ways in which a method call may be resolved, and several places where something so basic as an object attribute may be looked

[1]From *The Zen of Python* by Tim Peters; see `http://www.python.org/doc/Humor.html#zen`.

up. In the simplest case, many attributes may be held either in 'slots' in the type object *or* in the object's attribute dictionary; the method call resolution is still more complex.

6.1.1 Type equivalence

Some of the more insidious bugs that were uncovered in the course of implementing the PyRemote project were related to the notion of *type equivalence*. Internally, built-in types in CPython tend to have two utility macros for efficient type checking, e.g., for the `dict` type (`PyDictObject`), `PyDict_Check(o)` to check if an object o is a dictionary, and `PyDict_CheckExact(o)` to check if an object o has *exactly* the type `dict`. The former accepts a subclass; the latter does not. Although we consider a PyRemote proxy type *logically* equivalent to its target type, `PyDict_CheckExact()` looks for a reference to the precise C structure representing the type object. The consequence of this is that a type that passes `PyDict_CheckExact()` may be safely assumed to have the exact memory layout and data members specified by `PyDictObject`. A type that only passes `PyDict_Check()`, therefore, should have the methods specified by the type object, but may not have the exact same memory layout.

Unfortunately, some of the code in the `dictobject` module assumed precisely this, and therefore failed (in unexpected ways) when faced with dictionary proxies (which obviously do *not* have the same memory layout as actual dictionary objects!)—that is, `PyDict_Check()` was used where `PyDict_CheckExact()` was needed. Several such bugs were found, and we suspect that there may be more that PyRemote tests have not uncovered (and that PyRemote may be entirely unaffected by). Since the code chiefly deals with internal structures, which in the normal course

of execution will never be anything *but* precisely `dict` objects, it is not likely that they will be encountered except by modification of the sort we have done.[2]

Errors of this type, whether caused—as in this example—by bugs in the CPython interpreter, or—as in many, many cases—by programming error in writing PyRemote due to an incomplete or faulty understanding of the CPython codebase caused the implementation to consume much more time than initially estimated.

6.1.2 Pickling limitations

PyRemote uses Python's existing object marshalling facilities, the `pickle` and `marshal` modules. Although minor additions were made (e.g., the ability to marshal types as described in Section 4.2), there are certain objects that are presently 'unpicklable'. This is here considered to be more in the nature of a limitation of the library than of PyRemote: The solution is deferred to improving the `pickle` library rather than working around it.

6.1.3 Old and new classes

For historical reasons, the Python language actually supports *two* types of classes and instances. While largely transparent to users, the underlying implementations in CPython are completely different. The PyRemote implementation does not presently support the use of "old-style" classes. Although it could be done in much the same way as the proxy system for "new-style" classes, it was deemed an irrelevant duplication of effort for the purposes of this project; furthermore, the use in Python of "old-style" classes is discouraged. In future revisions of CPython, they

[2]We plan to examine the `PyDictObject` code for instances of this problem and submit patches when more time is available. It would probably not be relevant to do so based on the existing PyRemote code, however, as it is based on a CPython codebase several minor versions behind the current one.

will be removed.

6.1.4 Unsupported operations

Not quite all 'standard' operations are supported by PyRemote proxies, for the simple reason that the arguments cannot be transmitted. Certain entries in the type structure's table are functions that expect, as parameters, pointers to raw memory, which cannot intelligently be marshalled. Operations not supported by PyRemote proxies include the intrinsic `tp_print()` method (which is deprecated and should not pose a problem), *coercion* in numeric types (this also is not so great a problem as it may seem, as intrinsic number types are small, immutable objects which are always copied and never sent by proxy) and operations on the `buffer` type (whose purpose is precisely to expose object data as raw bytes).

6.2 Future work

PyRemote, in its present state, more or less represents a proof of concept and not a production ready system. Certain unsupported operations (see Section 6.1.4) should be implemented or at the very least handled more gracefully (for example, buffers might be copied automatically rather than having operations thereupon refused). Garbage collection is not presently implemented for remote objects. Finally, some features were left out due to simple lack of time and underestimations of the complexity of the implementation task.

6.2.1 Garbage collection

The present PyRemote implementation does not perform distributed garbage collection. To a certain extent, the problem is 'easy': The Python interpreter performs

reference counting (the default implementation uses a reference counting garbage collector with cycle detection), and the only thing that prevents an object, once proxies to it exist, from being properly garbage collected, is the existance of a reference to the object held by the PyRemote machinery, used to resolve remote requests. A garbage collection mechanism, therefore, would 'simply' need to track the number of remote peers that own *proxies* to the object; once this number reaches 0, unless there exist *local* proxies, the object may be removed from the PyRemote cache, and garbage collected if appropriate. However, this requires a proper distributed garbage collector to deal intelligently with 'ordinary' problems of the area. This is beyond the scope of this thesis.

Given a distributed garbage collection algorithm, it would probably be fairly straightforward to implement distributed garbage collection in PyRemote. The system already satisfies the local tracking criteria mentioned by Bennett in his design of the Distributed Smalltalk garbage collector [3]: Purely local garbage is collected, and references to remote objects are stored in a centralised location where a distributed collector can easily look them up.

6.2.2 Object mobility policies

For reasons laid out in Section 6.1, the PyRemote implementation does not support all the features that were originally planned. One of the more interesting features that were left out, inspired by the object mobility paper on Emerald [1], is the notion of need or usage pattern based object mobility policies. In effect, the system may *detect* whether an object, invoked remotely, is likely to be used enough (more accurately, whether the remote usage is likely to exceed the local usage by a sufficient margin) that it would be profitable to move it rather than to create a proxy.

This was, then, to be accompanied by a form of 'stickiness' property whereby a programmer might request that the object stay put. While this may seem micro-management, it may be relevant for objects where availability is a critical point and certain nodes in a network may be known or expected to be more reliable than others.

Unfortunately, there was not enough time to implement this feature, which could perhaps substantially improve performance of applications making heavy use of remote calls. It would be necessary to implement some limited form of dynamic profiling, and a policy to move objects based on these measured parameters.

Perhaps more useful still is the notion of using profiling (whether wholly dynamic or cached) to determine which portions of an application execute in isolation and so make good candidates for 'shipping off' to other nodes for processing. This feature is not in the current version of PyRemote, but the groundwork exists: In Python, due to its powerful introspection, and the profiling module present in the standard library; and in PyRemote, which can move code objects to remote nodes.

6.2.3 Transactional consistency semantics

As mentioned in Section 4.6, it could be interesting to investigate the possibility of implementing real transactional semantics for type consistency, rather than the more efficient but somewhat ad-hoc approach used in the current implementation.

6.2.4 Nameserver

The current PyRemote implementation does not have an elegant way of connecting to a remote peer and obtaining a directory of services provided. Indeed, the present service is rather ad-hoc in nature, where a server offers an arbitrary (user-specified)

object, and clients must connect to the server by IP address and port (although a default server port of 10101 is used). A useful addition to the system would be a more intelligent interface, and a more flexible encoding of hosts.

6.3 Final thoughts

It is perhaps in the nature of a thesis of this type that it is even more a learning experience of existing systems than it is a production of *novel* knowledge. When the PyRemote project was begun, the CPython interpreter was an opaque unknown, and the early stages of modification were very much in the nature of trial and error, where core features of the interpreter were overlooked or entirely unknown. Significant amounts of time (it is tempting to say 'inordinate') were spent (or lost) on not only fixing bugs, but refactoring or redesigning the core PyRemote modules as interactions between different interpreter components became more clear.

6.4 Obtaining the code

The code shown in examples and used to produce the cited test results was locally denoted as revision 215. It may be obtained as a patch set against Python version 2.4.3. It is a fairly large diff set which includes all modifications, test code, and various generation scripts. (The size of the diff, 37 MiB uncompressed, is rather disproportionate to the relatively smaller amount of changes to the code base.) If you want to hack on the latest version of PyRemote, a further diff is available of revision 243, which is current as of the finalisation of this thesis. This revision contains some incomplete work and may not be entirely functional, but in particular contains more and more robust consistency code.

- Python 2.4.3 source code (several download options, checksums listed):

  `http://python.org/download/releases/2.4.3/`

- PyRemote diff, revision 215 against Python 2.4.3 (6.0 MiB):

  `http://petterhaggholm.net/pyremote/pyremote-r215-diff.tar.bz2`

  md5sum: `bc47b0378859e3468cd0516a939e3acb`

- PyRemote diff, revision 243 against revision 215 (3.3 MiB):

  `http://petterhaggholm.net/pyremote/pyremote-r215_243-diff.tar.bz2`

  md5sum: `514651a78705a979b6f7e6062ecfd06c`

Bibliography

[1] E. Jul, H. Levy, N. Hutchinson, and A. Black, "Fine-grained mobility in the Emerald system," *ACM Transactions on Computer Systems*, vol. 6, pp. 109–133, February 1988.

[2] A. P. Black, N. C. Hutchinson, E. Jul, and H. M. Levy, "The development of the Emerald programming language," in *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, (New York, NY, USA), pp. 11–1–11–51, ACM Press, 2007.

[3] J. K. Bennett, "The design and implementation of Distributed Smalltalk," in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (N. Meyrowitz, ed.), vol. 22, (New York, NY), pp. 318–330, ACM Press, 1987.

[4] J. Armstrong, R. Virding, C. Wikström, and M. Williams, *Concurrent Programming in Erlang*. Prentice-Hall, second ed., 1996.

[5] J. L. Armstrong, "The development of Erlang," in *International Conference on Functional Programming*, pp. 196–203, 1997.

[6] S. D. Bruda, P. Häggholm, and S. Stoddard, "Distributed, real-time programming on commodity POSIX systems: A preliminary report," in *Proceedings of*

the 5th International Symposium on Parallel and Distributed Computing, IEEE Computer Press, 2006.
`http://turing.ubishops.ca/home/bruda/rtsync/`.

[7] L. Prechelt, "An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a search/string-processing program," in *Technical Report 2000-5*, Universität Karlsruhe, Fakultät für Informatik, 2000.

[8] I. de Jong, "Python Remote Objects."
`http://pyro.sourceforge.net/`.

[9] T. Filiba, "Remote Python Call."
`http://rpyc.wikispaces.com/`.

[10] G. Payer, "Python Remote Call Module."
`http://www.gernot-payer.de/pyrcall/`.

[11] M. Simionato, "The Python 2.3 method resolution order."
`http://www.python.org/download/releases/2.3/mro/`.

[12] A. Vassalotti, "Pickle: An interesting stack language."
`http://peadrop.com/blog/2007/06/18/pickle-an-interesting-stack-language/`.

[13] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Transactions on Computer Systems*, vol. 2, pp. 39–59, February 1984.

[14] M.-A. Lemburg, "Pybench."
`http://www.egenix.com/files/python/pybench-1.0.zip`.

[15] J. Fenlason and R. M. Stallman, "The GNU profiler."

http://www.gnu.org/software/binutils/manual/gprof-2.9.1/html_mono/gprof.html.

[16] G. van Rossum, "Python 2.3.4 library reference." http://www.python.org/doc/2.3.4/lib/lib.html.