Python-based Distributed Programming with Trickle

Gregory D. Benson and Alexey S. Fedosov
Department of Computer Science
University of San Francisco
San Francisco, CA, USA
{benson, fedosov}@cs.usfca.edu

Abstract *Trickle is a an extension to the Python programming language that provides explicit but simple mechanisms to write distributed scripts and programs. Trickle links together remote Python interpreters running on heterogeneous machines so that work can be deployed and results collected. A Trickle program interacts with remote interpreters by injecting functions or classes. Remote objects can be instantiated and invoked synchronously or asynchronously. Also, the injected code need not reside on the remote interpreters; code is dynamically transferred as needed. Trickle leverages off of Python's list comprehensions and generators to simplify parallel invocation, result gathering, and dynamic scheduling. The Trickle run-time system uses a broadcast mechanism to find eligible Trickle virtual machines. Python programmers can immediately use Trickle to dispatch work to idle machines with minimal setup and easy to learn mechanisms. This paper describes the Trickle extension interface, its implementation, and presents an example application and its performance.*

Keywords: Distributed programming, Python

1 Introduction

The Python programming language [15] has proven to be effective in a variety of domains including scripting, web services, data analysis, simulation, and prototyping. Python's compact yet expressive syntax and dynamic typing combined with powerful built-in data types and a comprehensive standard library allow for rapid program development. Python programs tend to be shorter and easier to read than similar programs written in general-purpose, statically-typed languages such as C++, Java, and C#.

Trickle is a Python extension that allows programmers to easily execute code on remote Python interpreters, or, as we call them, Trickle virtual machines (VMs). The Trickle interface is explicit, but simple; a programmer can easily express parallel execution of coarse-grained work. Programmers can Trickle-enable existing Python code or write new parallel Python code. Furthermore, enabling a computer to accept Trickle requests is as simple as starting a single executable. The only software requirement is a modern Python installation, which comes standard on Mac OS X and most Linux distributions.

The Trickle programming model incorporates three basic concepts: injection, remote access, and asynchronous invocation. An initiating Trickle program can inject local functions and classes into remote Trickle VMs. Once injected, remote objects can be accessed and invoked transparently with local VM handles. Using fork/join parallelism, remote code can be invoked asynchronously on remote VMs for true parallel execution. Finally, Trickle provides a simple mechanism for dynamically scheduling work to remote VMs. This mechanism simplifies the use of networked machines of varying performance.

The standard Python distribution comes with several library modules for network communication and Internet protocols. By themselves, these modules do not readily allow a programmer to easily design and develop parallel Python programs. As such, there exist several Python extensions designed to allow programs to cooperate in a distributed environment. The IPython [13, 14] project is most similar to Trickle. It allows interactive coordination of distributed Python interpreters. However, its feature set is rather large and is aimed at coordination and development of high-performance parallel programs. PYRO [5] brings to Python a more traditional form of distributed objects based on a client/server model. However, PYRO does not have direct support for asynchronous invocation and dynamic scheduling. Finally, there exist several projects that provide Python wrappers for the standard MPI interface [12]. Trickle is much simpler than the full MPI implementation and provides a MPMD (multiple program, multiple data) model rather than a SPMD (single program, multiple data) model.

The rest of this paper is organized as follows. Section 2 describes the Trickle programming model and basic Trickle mechanisms. Section 3 presents a complete Trickle program and its performance. Section 4 provides some details of our Trickle implementation. Section 5 reviews related work. Section 6 makes some concluding remarks and gives directions for future work.

2 Programming Model

The Trickle programming model extends the standard Python execution environment by allowing multiple Python interpreters to interact in a coordinated manner. A Trickle *initiator* process can connect with one or more remote Trickle *virtual machines* (VMs). A Trickle VM is simply a Python interpreter running Trickle run-time code that waits to be discovered for use by an initiator. A user will start Trickle VMs on each machine that can be involved in a distributed computation. This is a matter of invoking a single executable, which requires no configuration. Trickle VMs can be added to startup scripts or as a startup item; they can also be deployed onto the nodes of a dedicated cluster.

The Trickle initiator can issue a command to *connect* to remote Trickle VMs. Once connected, the initiator can *inject* data, functions, or classes into remote VMs. Access to remote data or code is achieved through a remote access mechanism. This mechanism is a generalization of remote procedure call and remote method invocation. Remote access is seamless, so a remote request appears as a local request. Consider the simple Trickle program presented in Example 1:

Example 1 A Simple Trickle Program

```
1 def foo(x):
2     return x + 10
3
4 vmlist = connect()
5 inject(vmlist, foo)
6 results = [vm.foo(10) for vm in vmlist]
7 print results
```

```
$ trickle exsimple.py
[trickle: discovered 4 VMs]
[20, 20, 20, 20]
```

This code connects to the available Trickle VMs using the `connect()` function. In a standard Ethernet-based network a multicast send is used to discover remote VMs. A list of VM handles is returned from `connect()`; in this case 4 handles are returned. The handles are used for invoking remote operations. Initially, each connected VM is idle and contains a default Python run-time environment. In order to use a VM, the initiator must use the `inject()` function on a VM handle or a list of VM handles to transfer local Python objects from the initiator's environment to the remote VM environments. In addition to VM handles, a Python object must be passed as a parameter to `inject()`. Possible Python objects include data objects, functions, and classes. Using `inject()`, the initiator can fill each VM with any data and code necessary to carry out a remote computation. In this example, we inject the function `foo()` into each discovered VM.

Once objects have been injected into remote VMs, they can be accessed using the VM handles. We can invoke remote operations synchronously or asynchronously. In this example, `foo()` is invoked synchronously on each remote VM using Trickle remote access (Example 1, line 6). Note that remote access looks similar to local access except for the VM handle prefix. Also, this example uses a list comprehension to collect the results of each remote invocation. Since we are using synchronous invocation, each remote call to `foo()` must complete before the next call is issued.

Notice that this simple program is completely self-contained. It is not necessary to have any code on the remote VMs prior to execution. The `inject()` method takes care of transferring objects and code from the initiator to the VMs. This approach makes it is easy to organize small parallel Trickle programs. Also, compared to an SPMD programming model, the Trickle model is very explicit: a programmer only injects necessary code into remote VMs. While subtle, we believe many programmers will find this model more natural than a model like MPI, in which distributed process differentiate themselves via a rank value. Also, unlike distributed object systems, Trickle does not require separate client and server code.

The rest of this section describes the main Trickle concepts and interfaces in detail. First, we explain connecting and injecting. Second, we cover synchronous remote access. Third, we present different forms of asynchronous invocation. Fourth, we explain the Trickle dynamic scheduling mechanism. Finally, we describe some practical considerations for starting Trickle VMs and accessing file systems.

2.1 Connecting and Injecting

As described previously, a Trickle initiator uses `connect()` to discover remote Trickle VMs. The return value of `connect()` is a list of VM handles (`vmlist`). Therefore, the number of available remote servers is computed by `len(vmlist)`. It is possible to specify a maximum number of VMs needed for a particular computation by passing a `max` argument to `connect()`. For example, `connect(4)` requests a maximum of 4 VMs. If 4 VMs are not located, an exception is raised. Restricting the number of required VMs also allows a user to run multiple Trickle programs simultaneously as a single initiator may not need all available VMs.

The valid forms of Trickle `inject()` are:

```
inject(vm, obj0 [, obj1] ...)
inject(vmlist, obj0 [, obj1] ...)
```

Example 2 Trickle Injection

```
table = { 1 : 'a', 2 : 'b'}

def find(name, namelist):
    for i, n in enumerate(namelist):
        if n == name:
            return i

class foo(object):
    def update(x, y):
        self.x = x
        self.y = y

vmlist = connect()
inject(vmlist, table, find, foo)
```

Trickle injection explicitly places Python objects into remote VMs. It is possible to inject data, functions, and classes. The `inject()` function can take a single VM handle or a VM handle list as the first argument. The remaining arguments are objects to be injected. If a VM handle list is provided, then the object arguments are injected into each VM in the VM handle list. The injected objects are copies of the original initiator objects and they can be accessed transparently using a VM handle. The access is similar to a local access. When an initiator completes execution, injected objects are removed from the remote Trickle VMs. See Example 2 for different forms of injection.

2.2 Synchronous Remote Access

Once code and data have been injected into remote VMs, these objects can be accessed transparently using synchronous remote access via VM handles. Each remote access invocation is blocking; an invocation must complete before the program can continue execution. In addition to injected objects, all Python built-in functions are available for remote invocation.

Example 3 shows how to access remotely injected data. In this case, we have injected a list into the remote VMs. We can transparently perform updates on the remote lists (line 5). This also works on remote dictionaries and any remote object that supports the `__setitem__/__getitem__` interface.

Example 3 Synchronous Data Access

```
1 vmlist = connect()
2 names = ['Alex', 'Sami', 'Greg', 'Peter']
3 inject(vmlist, names)
4 for i, vm in enumerate(vmlist):
5     vm.names[i] = '*NONE*'
6     print vm.names

$ trickle exsynchdata.py
[trickle: discovered 4 VMs]
['*NONE*', 'Sami', 'Greg', 'Peter']
['Alex', '*NONE*', 'Greg', 'Peter']
['Alex', 'Sami', '*NONE*', 'Peter']
['Alex', 'Sami', 'Greg', '*NONE*']
```

Example 4 Synchronous Function Invocation

```
def factorial(x):
    if x == 0: return 1
    else: return x * factorial(x-1)

vmlist = connect()
inject(vmlist, factorial)
for i, vm in enumerate(vmlist):
    print vm.range(i+1), vm.factorial(i)

$ trickle exsyncfunc.py
[trickle: discovered 4 VMs]
[0]   1
[0, 1]   1
[0, 1, 2]   2
[0, 1, 2, 3]   6
```

Both injected functions and Python built-in functions can be invoked remotely using VM handles. Possible parameter values for remote invocation are only limited to Python data types that can be pickled (serialized). Example 4 shows how to call an injected `factorial()` function and the `range()` built-in function.

Remote object creation and invocation is demonstrated in Example 5. In this case, we are only creating a remote object on a single Trickle VM (see line 13). We invoke the remote object just as we would a local object (see lines 14-16). A remote object will exist on a remote VM as long as there is a local reference to the proxy object.

2.3 Asynchronous Remote Invocation

Parallel execution is achieved in Trickle with asynchronous remote invocation. Trickle uses a fork/join paradigm to invoke remote functions or methods asynchronously. A Trickle `fork()` function begins the execution of a remote function and returns immediately with a handle. The handle is later used by a `join()` call to synchronize with the remote invocation. The basic `fork()` and `join()` functions have the following forms:

Example 5 Synchronous Object Invocation

```
class Stack(object):
    def __init__(self):
      self.stack = []
  def push(self, x):
      self.stack.append(x)
  def pop(self):
      return self.stack.pop()
  def __str__(self):
      return self.stack.__str__()

vmlist = connect()
inject(vmlist, Stack)
s = vmlist[0].Stack()
s.push('A') ; s.push('B'); s.push('C')
s.pop()
print s

$ trickle exsynchclass.py
[trickle: discovered 4 VMs]
['A', 'B']
```

```
h     = fork(vm, func, arg0 [, arg1 ] ...])
hlist = fork(vmlist, func, arg0 [, arg1 ] ...])
hlist = fork(vmlist, funclist, arg0 [, arg1 ] ...])
hlist = fork(vmlist, funclist, arglist)

r     = join(h)
rlist = join(hlist)
h, r  = joinany(hlist)
```

These functions are quite flexible; they can be used to fork a single function on a single VM or invoke a function on multiple VMs. In addition, it is possible to map a list of functions to a list of VMs. In this case, each VM is invoked with a different function in the function list. The same arguments can be passed to the function(s) or an argument list can be used to map different arguments to different functions in a function list. The `join()` function waits for all handles provided. However, `joinany()` waits for the completion of a single invocation from list of two or more handles; it returns a value and the associated handle. Example 6 presents different uses of `fork()` and `join()`.

Basic usage of `fork()` and `join()` is shown on lines 10 and 11. A single function is forked on a single VM with a single argument. The returned handles are used by `join()` on line 11 to wait for the invocation to finish execution. Lines 14 and 15 show how to map a list of functions and a list of arguments to a list of VMs to fork multiple computations in a single call. Finally, lines 18 and 19 show how to map a single function to multiple VMs and an argument list so that the function is invoked with different parameter values on different VMs.

As will be explained in greater detail in Section 4, `fork()` does not use local threads to achieve asynchronous invocation, rather, a non-blocking send issues the remote invocation request. So, each Trickle VM needs only one computation thread.

Example 6 Asynchronous Invocation

```
 1 def foo(x):
 2     return x + 10
 3
 4 def bar(x):
 5    return x + 20
 6
 7 vmlist = connect()
 8 inject(vmlist, foo, bar)
 9
10 h0 = fork(vmlist[0], foo, 1); h1 = fork(vmlist[1], bar, 2)
11 r0 = join(h0); r1 = join(h1)
12 print r0, r1
13
14 hlist = fork(vmlist[0:2], [foo, bar], [1, 2])
15 rlist = join(hlist)
16 print rlist
17
18 hlist = fork(vmlist, foo, range(len(vmlist)))
19 print join(hlist)

$ trickle exasynch.py
[trickle: discovered 4 VMs]
11   22
[11, 22]
[10, 11, 12, 13]
```

2.4 Dynamic Scheduling

The Trickle `fork()` and `join()` functions are general enough to implement a variety of approaches for scheduling work on remote VMs. However, a common idiom is to construct a list of work and repeatedly assign portions of work to be consumed by code on remote VMs. Trickle provides the `forkwork()` function to accomplish this task:

```
rlist = forkwork(vmlist, func, worklist [, chunksize=n])
```

The programmer provides a list of VMs, a worker function, and a list of work. Each work element can be any Python object. The worker function must be implemented to correctly process the work element object type or a list of such objects. The `forkwork()` function will issue work to remote VMs in a dynamic fashion until all the work is complete. This idiom can be used to take advantage of remote VMs running on machines of different speeds. The optional `chunksize` parameter is used to send work elements in chunks to remote VMs to reduce network overhead. See Section 3 for a larger example that uses `forkwork()`.

Trickle also provides a Python generator called `forkgen()` for scheduling work dynamically. Using the generator, partial results can be processed as they become available.

2.5 Execution Environment

A Trickle program can connect to any number of Trickle VMs that exist on machines in a local area network. A broadcast mechanism is used to dynamically discover Trickle VMs at the time the `connect()` function is invoked. As such, a user will need to start Trickle VMs on machines in the local network. If a small number of VMs is required, then doing this manually is straightforward. It is also possible to enable the Trickle VM as a startup item. Trickle can also be started automatically in a cluster environment similar to how MPI is started.

Each Trickle VM has access to all resources available to the owner of the Trickle VM process. This means that a Trickle VM can access the local file system or any network-mounted file systems available to the user.

3 Example

In this section we present a complete working Trickle program and its performance on a varying number of machines. Example 7 is a distributed word frequency counter called `wfreq`. Given a list of file names, `wfreq` computes the total count of each unique word found in all the given files. This is a common operation used in document indexing and document analysis. If the data files exist on all the remote machines, then the counting phase runs in parallel. Only the work distribution and merging phases are sequential.

The `wfreq` program consists of three parts: a worker function, a merge function, and the Trickle code to distribute the work. The worker function `mergecounts()` accepts a list of file names and computes a single dictionary that maps words to their corresponding frequencies in all the files provided as input. The `mergecounts()` function simply combines all of the remotely computed word frequency dictionaries. Finally, the main Trickle code connects to the available Trickle VMs, injects the `wordcount()` function, then issues `forkwork()` to dynamically schedule provided file names to the worker function.

We ran the `wfreq` Trickle program on a small cluster to see what kind of speedup is possible. The cluster consists of 8 machines with dual Opteron 270 processors and 4 GB of RAM, connected to an isolated Gigabit Ethernet network. We used a data set with 63,362 files for a total of 285 MB of data. Each node was given a complete copy of the data set. We ran `wfreq` with a chunk size of 100. Thus, each Trickle VM works on 100 files at a time. Table 1 shows the results of running on 1, 2, 4, and 8 processors. As can be seen in the results, the serial portions of the code limit the speedup. However, these results show that it is relatively easy to leverage multiple machines with Trickle.

Example 7 Document Word Frequency Counter

```
import sys, time, glob

def wordcount(files):
    m = {}
    for filename in files:
        f = open(filename)
        for l in f:
            for t in l.split():
                m[t] = m.get(t,0) + 1
        f.close()
    return m

def mergecounts(dlist):
    m = {}
    for d in dlist:
        for w in d:
            m[w] = m.get(w,0) + d[w]
    return m

if len(sys.argv) != 4:
    print 'usage: %s <vmcount> <chunksize> <pattern>' \
          % sys.argv[0]
    sys.exit(1)
n = int(sys.argv[1]); cs = int(sys.argv[2])
files = glob.glob(sys.argv[3] + '*')

stime = time.time()
vmlist = connect(n)
inject(vmlist, wordcount)
rlist = forkwork(vmlist, wordcount, files, chunksize=cs)
final = mergecounts(rlist)
ftime = time.time()
print '%s seconds' % (ftime - stime)
```

VMs	1	2	4	8
Time (sec)	72.3	43.5	25.8	19.3

Table 1: Trickle `wfreq` results on 63,362 files (285 MB)

4 Implementation

Trickle is implemented on top of the River framework for distributed computing [16]. Both Trickle and River are implemented entirely in Python. The River core interface is based on a few fundamental abstractions that enable the execution of code on multiple virtual machines and provides a flexible mechanism for communication among them. Communication is achieved with a mechanism called Super Flexible Messaging (SFM) [6]. SFM provides a simple message-passing mechanism for transferring dynamically-typed messages between named processes. These abstractions are supported by the River execution engine, which manages automatic virtual machine discovery, connection management, and naming. While River can be used directly by application programmers, it also serves as a foundation on which higher-level programming models can be developed. We have developed Trickle as a River extension.

The main components of the Trickle implementation include a new remote access and invocation (RAI) mechanism, support for code injection, support for asynchronous invocation, and dynamic scheduling. All of these components are implemented in 768 lines of Python source code (the River core is approximately 3000 lines of Python source).

Support for remote object access is accomplished with the RAI mechanism. RAI is a generalization of remote data access, remote procedure call, remote object creation, and remote method invocation. Each Trickle VM instantiates a RAI server. In Trickle, the RAI server is configured to expose the standard Python environment. In this way, once a connection is established, a remote VM can access anything in the exported Python environment. The RAI server simply waits for incoming requests:

- **inject** Request contains source to be injected into VM.
- **call** Request contains function or method name and arguments to be invoked.
- **getattr** Request contains a request to access a remote object using dot (`.`) notation.
- **delete** Request to decrement reference count to a remote object.

A **call** request invokes the specified function or method. If an object is created by the invocation, it will be returned by reference. This allows for remote object creation. A **getattr** request looks in the VM namespace for an attribute reference. It allows for remote updates and access to global variables and objects. Both **call** and **getattr** reply to the sender with appropriate return values. If an exception is raised, it will be propagated back to the sender. The **delete** request is used to indicate that a remote proxy object is no longer accessible. This allows for objects to be reclaimed by the remote VM's garbage collector.

On the initiator end (client end) a special proxy object is used as a proxy for both the remote VM handles and remote objects. The initiator acquires VM proxy handles with the `connect()` function. The proxy object overrides many of the default object access methods, including `__getattr__`, `__setitem__`, `__getitem__`, and others. If an attribute does not exist in the local proxy object, it is assumed to be a remote request. In this way, Trickle can transparently propagate local invocations to remote VMs.

Code injection from the initiator VM to remote VMs is achieved with Python introspection. It is possible to obtain the source code for any function or class given a reference. The `inject()` function uses the specified reference to find the source code. The source is packaged up and sent to the remote VM. The RAI server accepts the **inject** request and uses the Python `exec` statement to introduce the source into the running VM.

Asynchronous invocation is supported by dividing a remote request into two parts: the send half and the receive half. A send half issues a request with a non-blocking send and returns a fork handle. The proxy object keeps track of outstanding send halves. Later, the fork handle is used to join with an outstanding asynchronous invocation. Note that each Trickle VM has a receive queue, so a remote VM can issue a reply at any time and it will be queued on the requesting VM.

The Trickle dynamic scheduling mechanism, `forkwork()` is built using basic asynchronous invocation and Python generators. Internally, `forkwork()` calls a generator called `forkgen()`:

```
def forkwork(self, vmlist, fname, work, chunksize=chunksize):
    results = []
    for rv in forkgen(vmlist, fname, work, chunksize=cs):
        results.append(rv)
    return results
```

The internal `forkgen()` routine dispatches work to available remote VMs and waits for invocations to finish. Each time a remote invocation completes, a new piece of work, if available, is dispatched to an available VM. Here is an abbreviated version, without the support for `chunksize`:

```
def forkgen(self, vmlist, fname, work):
    hlist = []
    while True:
        while len(work) > 0 and len(vmlist) > 0:
            w = work.pop()
            vm = vmlist.pop()
            hlist.append(fork(vm, fname, w))

        if len(hlist) > 0:
            h, rv = joinany(hlist)
            vmlist.append(h.vm)
            yield(rv)
        else:
            break
```

By implementing `forkgen()` with a Python generator (indicated by using the `yield()` statement) we simplify the implementation of `forkwork()` and also provide the programmer with a mechanism to process partial results while work is executing on remote VMs. This shows how Python generators can be used to extend the semantics of both the `for` statement and list comprehensions in a novel way.

5 Related Work

The design of Trickle leverages off past work on the design of programming languages for parallel and distributed computing [3]. Numerous languages support object-based distributed computing such as Emerald [7] and Orca [2]. In particular, the VM handles and invocation handles are similar in function to VM capabilities and operation capabilities found in the SR programming language [1].

As mentioned in Section 1, there are Python extensions that support distributed programming including IPython [13, 14] and PYRO [5]. In addition, PyLinda [17] provides Python support for the Linda [4] parallel programming model. A Python interface to the standard MPI library is provided by PyMPI [10], MYMPY [9, 8], and Pypar [11]. Trickle and the system it is built on, River, are both written entirely in Python and do not require additional libraries such as a C implementation of MPI. Also, unlike other Python extensions for parallel and distributed programming, Trickle is a relatively concise extension for farming tasks to remote machines. We have deliberately constrained the Trickle programming model to make it easy to learn and use.

6 Conclusions

Trickle is a simple extension to the Python programming language that enables heterogeneous distributed programming. The Trickle mechanisms are simple and can be learned quickly, which follows the Python philosophy. Unlike other distributed object systems for Python, Trickle provides a simple namespace that does not require a dedicated name server or an object broker. We have shown how Trickle can be used to distribute work statically or dynamically. Trickle is ideal for extending an existing Python program to take advantage of unused cycles in a network of machines. Finally, our experience with Trickle has been extremely positive. It is used locally to teach parallel programming, for movie creation, and data analysis.

For future work we plan to refine the Trickle interface to support dynamic discovery of VMs during program execution. Such a mechanism will allow a Trickle program to utilize additional computers as they become available. Similarly, we plan to provide a way to gracefully remove a VM from a Trickle computation. In addition, we plan to add various forms of fault tolerance to the Trickle run-time system. For example, if a VM crashes, the Trickle run-time system should be able to re-issue work on a different VM. We are also planning support for program checkpointing and VM migration.

Acknowledgments

We thank members of the River research group for their feedback and technical support: Brian Hardie, Tony Ngo, Jennifer Reyes, Joseph Gutierrez, and Yiting Wu. We also thank Terence Parr for reviewing an earlier draft of this paper.

This work was partially supported by a University of San Francisco faculty development research grant.

References

[1] G. R. Andrews, R. A. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend. An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, January 1988.

[2] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, SE-18(3):190–205, March 1992.

[3] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[4] N. Carriero and D. Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.

[5] Irmen de Jong. PYRO:python remote objects, 2007. http://pyro.sourceforge.net.

[6] A. S. Fedosov and G. D. Benson. Communication with super flexible messaging. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, USA, June 2007. CSREA Press.

[7] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[8] Timothy Kaiser, Leesa Brieger, and Sarah Healy. MYMPI - mpi programming in python. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 458–464, Las Vegas, Nevada, USA, June 2006. CSREA Press.

[9] Timothy H. Kaiser. MYMPI, 2007. http://peloton.sdsc.edu/ tkaiser/mympi/.

[10] Patrick Miller. Pympi: Putting the py in mpi, 2007. http://pympi.sourceforge.net/.

[11] Ole Nielsen. Pypar: Parallel programming in the spirit of python, 2007. http://datamining.anu.edu.au/ ole/pypar/.

[12] Peter S. Pacheco. *Parallel programming with MPI*. Morgan Kaufmann Publishers, 1997.

[13] Fernando Pérez et al. IPython: An enhanced interactive python. In *Proceedings of SciPy'03 – Python for Scientific Computing Workshop*, CalTech, Pasadena, CA, September 2003.

[14] Fernando Pérez et al. Ipython: An enhanced interactive python shell, 2007. http://ipython.scipy.org.

[15] The Python programming language. http://www.python.org.

[16] River. http://www.cs.usfca.edu/river.

[17] Andrew Wilkinson. Pylinda: Distributed computing made easy, 2007. http://www-users.cs.york.ac.uk/ aw/pylinda.