A Python-based Framework for Distributed Programming and Rapid Prototyping of Distributed Programming Models

by

ALEXEY S. FEDOSOV

B.S. (University of San Francisco) 2001

THESIS
Submitted in partial satisfaction of the requirements for the degree of
MASTER OF SCIENCE

in the

DEPARTMENT OF COMPUTER SCIENCE
of the
UNIVERSITY OF SAN FRANCISCO

Approved:

2009

A Python-based Framework for Distributed Programming and Rapid Prototyping of Distributed Programming Models

Abstract

River is a Python-based framework for rapid prototyping of reliable parallel and distributed run-time systems. The current quest for new parallel programming models is hampered, in part, by the time and complexity required to develop dynamic run-time support and network communication. The simplicity of the River core combined with Python's dynamic typing and concise notation makes it possible to go from a design idea to a working implementation in a matter of days or even hours. With the ability to test and throw away several implementations River allows researchers to explore a large design space. In addition, the River core and new extensions can be used directly to develop parallel and distributed applications in Python. This thesis describes the River system and its core interface for process creation, naming, discovery, and message passing. We also discuss various River extensions, such as Remote Access and Invocation (RAI) extension, the Trickle task-farming programming model, and a River MPI implementation.

Soli Deo Gloria

Contents

List of Figures

List of Tables

Acknowledgements

First and foremost, I would like to thank the person without whom I simply would not be where I am: my teacher, my colleague, my mentor, my friend, my advisor — Professor Greg Benson. None of this would exist without his support and motivation. Greg, I am forever grateful to you for getting me here.

I also owe a great deal of gratitude to my thesis committee, Professors Chris Brooks and Peter Pacheco. Thank you for putting up with this three-year ordeal and for your insightful comments and feedback.

I would like to also thank Yiting Wu, who has written most of the code for rMPI, and the rest of the members of the River Team during the 2006-2007 academic year, who have spent countless hours working on this project: Joseph Gutierrez, Brian Hardie, Tony Ngo, and Jennifer Reyes.

Finally, I would like to thank my family — my wife, my mother, and my grandmother — for their continuous support, encouragement and belief in me. I made a promise that I would finish this monumental work and I am glad that it is finally over. I really should have done it sooner.

Chapter 1

Introduction

The River framework [66] is a parallel and distributed programming environment[1] written in Python [62] that targets conventional applications and parallel scripting. The River core interface is based on a few fundamental concepts that enable the execution of code on multiple machines and provide a flexible mechanism for communication among them. Compared to previous parallel and distributed programming systems, River is unique in its integration with a dynamically-typed scripting language, its simplicity in implementation and usage, and its ability to be used as a prototyping tool for new parallel programming models. This thesis describes the River programming model, usage of River to construct applications and new programming models and the River implementation. In addition, we present both a quantitative and a qualitative evaluation of River and our River extensions.

The main features of River include a simple programming model, easy program deployment, and a novel communication mechanism. The goal is to allow a user to leverage multiple machines, i.e., desktops and laptops, to improve the performance of everyday computing tasks. Parallel application programming can be done directly with the River API or simplified further by using higher-level constructs provided by River extensions.

A River application is made up of one or more Virtual Resources (VRs), which are similar to processes. By naming Virtual Resources with universally unique identifiers, or UUIDs, River isolates name references from physical identifiers (such as an IP address) or socket and file descriptors. Thus, to communicate among themselves Virtual Resources need to refer to each other only by their UUID. No other identification information (such as an IP address or a hostname) is necessary.

We have created a novel communication mechanism called Super Flexible Messaging (SFM) for basic communication between processes, both at the system and application levels. The main feature of this mechanism is the ability to easily send dynamically-typed attribute-value pairs from one Virtual Resource

[1]River is a parallel and distributed system but for brevity we use the term *parallel* in this thesis to mean parallel and distributed.

to another. All the internal communication within River, such as machine discovery, allocation, and process deployment is achieved through the use of SFM. Any possible conflicts between system and application packets are avoided by giving application processes different names (UUIDs) from the system processes. The flexibility and ease of use of this model combined with powerful and elegant semantics of the Python programming language made the rapid prototyping of our system extremely efficient and allowed us to explore many different approaches for which we otherwise would not have had time.

Another important goal of River is support of rapid prototyping of new parallel and distributed programming models. For instance, the flexibility afforded by the River core has allowed us to quickly implement several small to medium scale extensions to the River framework that provide run-time support for different programming models. In particular, the first extension to River we developed is Remote Access and Invocation (RAI), which provides transparent access to remote methods and data for River applications. River makes it easy not only to develop and debug extensions, but also to implement several variations of an extension. For example, we have used an object-oriented approach to implement MPI [45, 48] on top of the River core. Our version allows for incorporating an increasing amount of functionality by subclassing different feature sets. The base implementation can be extended to support derived data types, non-blocking communication, and optimized collective communication. Additionally, our Trickle [8] extension for task farming and work distribution has gone through several iterations in which we have experimented with different semantics.

The implementation of the River run-time system consists of a River Virtual Machine (VM), which provides support for communication, discovery, allocation, and deployment; the Virtual Resource (VR) base class, which serves as the primary River API; and our messaging framework, composed of serialization routines and a message queue. Underneath the hood the River VM is comprised of several building blocks that work together to support running VRs. We have a unified network connection model that can utilize multiple network interfaces and protocols, a connection caching mechanism to support an arbitrary number of simultaneous connections, a name (UUID) resolution protocol, and low-level data transmission routines. Our messaging framework automatically handles all the low-level details, such as serializing data, and allows a programmer to send structured data without any additional interface specification.

We present a quantitative evaluation of River using both micro-benchmarks and small-scale applications. It is possible to write sophisticated parallel Python applications using the River core and the River extensions, and such applications can be used to evaluate different programming models or to solve real problems. Of course, such programs will be subject to Python's sequential performance. To improve performance, the River core and extensions can be coupled with high-performance C and C++ based libraries such as NumPy [47] and the Python Imaging Library [52]. Because benchmarks alone do not properly convey the advantages of River as a framework for building distributed systems and programming models, we provide a qualitative evaluation as well. Trickle, for example, has been used locally to teach parallel programming, as well as for movie creation and data analysis.

The rest of this thesis is organized as follows. Chapter 2 provides motivation for River and surveys previous developments in the field of distributed programming languages and libraries. Chapter 3 explains the River Core API and demonstrates how to use it to develop distributed applications. Chapter 4 discusses our implementation of the River run-time system and its components. The next three chapters serve as small case studies for using River to develop higher-level functionality and to prototype different parallel programming models. Chapter 5 presents the Remote Access and Invocation (RAI) framework, an extension to the River Core. Chapter 6 discusses the Trickle programming model. Chapter 7 presents the design and implementation of our River-based MPI library. Finally, Chapter 8 presents some concluding remarks and offers direction for future work.

Chapter 2

Background

In designing River we leverage past and current developments in the field of parallel and distributed computing. This chapter provides the context for the ideas behind River. First, we provide our motivation for the creation of the River framework. Then we survey past work in the field of distributed languages and libraries. We also present different communication mechanisms that have been developed for use in distributed systems. Finally, we discuss Python, the language we chose for the implementation of River, and the important features it offers for distributed programming.

2.1 Motivation

In accordance with Moore's law the number of transistors on a processor increases exponentially by doubling every two years [44, 43]. At the same time, processor technology appears to have reached a frequency limit. As such, performance improvements are now obtained by packing multiple cores on a single CPU. Software, however, has not kept pace with hardware advancements and very few applications have progressed beyond sequential computing to exploit the parallelism present in today's desktops and laptops. Applications must now maximize usage of all available cores to achieve the best performance.

In the future we can only expect the utilization/idle ratio to get worse as more and more cores are packed into a single chip. Application programmers realize they have to utilize multiple processors, yet are reluctant to parallelize their software because parallel programming is difficult and brings with it new issues, problems, and complexities not present in sequential programming. There is no silver bullet yet; the holy grail of automatically parallelizing an arbitrary sequential program has not been attained. No matter which language or library is used for the development of a parallel program, a programmer still requires the knowledge and skills of operating in a parallel environment where many things are happening at once, be it multiple threads on a single system or multiple processes on a cluster.

While there exist several languages and libraries for parallel scientific computing, they have not gained traction for general purpose productivity applications due to programming complexity and significant administrative requirements. The solution is to find the next set of parallel programming models or incrementally improve current ones, as well as provide a simple and robust underlying framework to allow developers to take advantage of inherent parallelism in their systems (networked desktops and laptops) without requiring dedicated supercomputers and staff to support them.

Simplifying parallel programming of large scale computations is an area of vital importance to the future of supercomputing. Conventional clusters, big iron, grids, and now multi-core personal computers require increased programmer productivity in order to effectively harness multiple processors. In addition to the general challenge of writing correct parallel programs, newly-trained parallel programmers are forced to contend with languages (like C and Fortran) that seem primitive compared to those they have been trained in, such as Java or Python. In order to bridge this gap, new parallel languages need to provide broad-market features that do not undermine the goals of parallel computing [16]. Thus, much effort in the research community is focused on improving development languages and distributed run-time systems.

While ultimately using conventional languages for implementing production systems is necessary for performance, our belief is that using such languages for prototyping can restrict exploration of a larger set of designs. We speculate that the current practice for parallel language and library design, however, has typically involved a long development cycle, consisting of a specification by a committee or a group, a prototype in a conventional language, feedback from and improvement upon the prototype. Going from a specification to a prototype implementation is a time-consuming and error-prone task if conventional languages (e.g., C and Java) and conventional communication mechanisms (e.g., sockets and MPI message passing) are used. This is a key problem and a major roadblock to parallel language development. Our belief based on empirical experience is that prototyping with conventional languages results in design decisions locked in at an early stage because making changes to the core of a run-time system can be a substantial engineering effort. Furthermore, our belief is based on the fact that using a higher-level language results in fewer lines of code and therefore less complexity.

However, much can be learned from a working implementation, especially one that can run real applications on real parallel hardware. It is possible to build prototypes in existing parallel and distributed languages, but such languages are often designed to support the development of applications, not new parallel constructs and interfaces. By decreasing the time to prototype and experiment with new ideas via a working system, more alternatives can be examined and tested. A shorter feedback loop can open up a much larger design space.

The River framework was developed to allow an application to easily exploit multiple CPUs and systems for increased performance and reliability. We intentionally constrained the core of the system to have a small and easy to use interface. River's other goal is to facilitate rapid prototyping of parallel programming systems and constructs. We implemented River entirely in Python [62], chosen for its high productivity and rapid development characteristics.

In some sense, River is another attempt to bring supercomputing to the masses. Our first experience with popularizing supercomputing was the FlashMob event [24] at the University of San Francisco, which assembled a massive supercomputer from 700 desktops and laptops. Unlike traditional supercomputers, which are expensive and not accessible to the general public, a FlashMob supercomputer is temporary, made up of ordinary computers, and is built to work on a specific problem. This elaborate experiment proved that instant supercomputing is viable.

River, however, takes a somewhat different approach. Just like FlashMob, we want to bring distributed computing to the people, but rather than create massive instant supercomputers, we aim to provide a usable parallel framework for an average user who has access to a few desktops and laptops. We want to allow the user to exploit the inherent parallelism present in today's systems. Our goal is to give the user a programming model and run-time system that enable parallel programming with little difficulty.

River is based on a few fundamental concepts that enable the execution of code on multiple virtual machines and provide a flexible mechanism for communication. These concepts are supported by the River run-time system, which manages automatic discovery, connection management, naming, process creation, and message passing. The simplicity and elegance of the River core combined with Python's dynamic typing and concise notation make it easy to rapidly develop a variety of parallel run-time systems.

River introduces a novel dynamically typed communication mechanism, called *super flexible messaging* that makes it easy to send arbitrary data as attribute-value pairs and can selectively receive based on matching attribute names and subsets of attribute values [23]. This powerful mechanism eliminates the need to define a fixed packet structure or specify fixed remote interfaces.

The rest of this chapter describes previous and related work that we leverage in the field of distributed languages and libraries, various communication mechanisms available to parallel programmers, and parallel and distributed extensions developed specifically for the Python language.

2.2 Parallel and Distributed Languages

The design of River is influenced by past work on the design of programming languages for parallel and distributed computing. In [4] Bal, *et al* provide a comprehensive survey of various research languages and libraries. Albeit this work was published in 1989, the issues discussed are still pertinent in the field of distributed computing today. A distributed computing system, as defined by Bal, consists of multiple autonomous processors that do not share primary memory, but cooperate by sending messages over a communications network.

Message passing as a programming language primitive was introduced by C.A.R. Hoare in a language called Communicating Sequential Processes (CSP) [30]. The CSP model consists of a fixed number of sequential processes that communicate only through synchronous message passing. This resulted in many subsequent languages supporting some form of message passing, like Occam [41]. These languages were used for programming distributed systems as well as shared-memory multiprocessors. The next step was the development of higher-level paradigms and interprocess communication, such as rendezvous, remote procedure call (RPC), and distributed data structures [4]. River continues this tradition by adopting message passing as the basic communication primitive. Like past systems, River builds more advanced constructs, such as RPC, on top of message passing.

However, in CSP all interprocess communication is done using synchronous receive and send. Both simple and structured data may be communicated as long as the value sent is of the same type as the variable receiving it. The structured data can be given a name. An empty constructor may be used to synchronize two processes without transferring any real data, i.e., with an empty message. River (and our SFM framework specifically) extends this model by making receives asynchronous and removing all restrictions on the data types. Any type (both simple and structured) can be named and communicated.

Additionally, both CSP and Occam are static in the sense that they do not allow dynamic creation of new processes during the execution of a program. River allows machines to be discovered dynamically, thus enabling the addition of new processes during program execution. We have designed a flexible *discover-allocate-deploy* mechanism, described in Chapter 3, to support this idea.

In most procedural languages for distributed programming, parallelism is based on the notion of a process [4]. For example, the Network Implementation Language (NIL) [71] system consists of a network of dynamically created processes that communicate only by message passing over communication channels. In NIL, a process is not only the unit of parallelism but also the unit of modularity. This is similar to the *Virtual Resource*, a process-like abstraction found in River. Similar to NIL, each process in River may also be a different module, providing a unit of modularity.

For situations where message passing is inadequate as the basic communication primitive, other higher-level mechanisms may be employed. For instance, processors may communicate through some generalized form of remote procedure call [10]. RPC was first introduced by Brinch Hansen for his language Distributed Processes (DP) [28]. DP processes communicate by calling one another's common procedures. Another possibility is to use a distributed data structure. One example of this is Linda [15], which supports an abstract global memory called the *Tuple Space*, rather than using shared variables or message passing for communication.

Numerous languages support object-based distributed computing such as Emerald [35] and

Orca [3]. In Emerald, objects are units of programming and distribution and the entities between which communication takes place. The language is strongly typed and has explicit notions of location and mobility. Emerald supports concurrency both between objects and within an object. Orca [3], on the other hand, allows processes to share variables of abstract data types (objects). Orca provides an explicit fork primitive for spawning a new child process and passing parameters to the new process.

River also borrows a lot of concepts from the Synchronizing Resources (SR) language, developed for programming distributed operating systems and applications [1]. Like SR, River consists of one or more parametrized *resources*, which constitute the main building block of the system. In both cases a resource is a module run on one physical node. Resources are dynamically created and optionally assigned to run on a specific machine. SR resources interact by means of *operations*, which generalize procedures. Operations are invoked by means of synchronous **call** or asynchronous **send** and implemented by procedure-like **procs** or by **in** statements. In contrast, River Virtual Resources (VRs) communicate by explicit message passing using **send** and **recv**. The VM and invocation handles concept from our Trickle framework (described in Chapter 6) are similar in function to VM identities and operation capabilities found in SR.

SR's input statements combine aspects of both Ada's **select** statement [68] and CSP's guarded input statement. However, both SR's input statements and River's SFM model are more powerful, since the expression may reference formal parameters and selection can be based on parameter values. Unlike SR, which is a standalone distributed language and run-time system, River uses a general-purpose language and targets conventional computer systems. Also, the compile-link-deploy cycle in SR is somewhat heavy-weight, whereas River's use of Python allows the programmer to immediately run the code without any intermediate steps. Additionally, SR is not object-oriented, and not as portable as Python or Java.

The JR [38] language extends Java [27] with the concurrency model provided by SR in a manner that retains the feel of Java. JR is intended as a research and teaching tool. Since JR extends the syntax of Java, and requires an extra step to translate JR programs into standard Java code. Unlike JR, River is built in a conventional language and does not introduce any new syntax, which makes it an ideal platform for developing and prototyping distributed systems and applications.

Several other extensions to Java attempt to modify its concurrency model. For example, Ajents [32] provides remote object creation, asynchronous RMI, and object migration through a collection of Java classes without any modification to the language. JavaParty [51] allows transparent remote objects and instantiation on remote hosts. Communicating Java Threads [29] extends the Java concurrency model by providing communication between threads based on the CSP [30] paradigm.

The complexities and limitations found in the dominant languages and interfaces for parallel computation – Fortran, C, and MPI – have sparked a quest for the next generation of HPC development environments [21, 40]. Recent large-scale efforts include X10, Fortress, and Chapel. Such languages are attempting to make explicit parallel programming more manageable.

X10 [77, 18, 67] is designed specifically for parallel programming. It is an extended subset of the Java programming language, strongly resembling it in most aspects, but featuring additional support for arrays and concurrency. X10 uses a partitioned global address space model. It supports both object-oriented and non-object-oriented programming paradigms. It is still under development but several releases are available.

Fortress [25] is a draft specification for a programming language for high-performance computation that provides abstraction and type safety on par with modern programming language principles and includes support for implicit parallelism, transactions and static type checking. However, as of the writing of this thesis, it has not progressed beyond a reference interpreter implementation.

Chapel [17, 13, 16] is designed to serve as a parallel programming model that can be used on commodity clusters and desktop multicore systems. Chapel supports a multithreaded execution model via high-level abstractions for data parallelism, task parallelism, concurrency, and nested parallelism. It strives to vastly improve the programmability of large-scale parallel computers while matching or beating the performance and portability of current programming models like MPI. A prototype compiler is currently available.

We believe that these next-generation HPC development environments could greatly benefit from the River framework, as it could be used for rapid prototyping and testing of new parallel programming constructs and interfaces. Furthermore, River opens up the design space and shortens the development cycle, allowing programmers to experiment with a wide variety of approaches.

2.3 Communication Libraries and Mechanisms

Interfaces for data communication are crucial to the development and execution of distributed-memory programs. As such, a multitude of communication mechanisms are available to programmers. Simple communication actions can be easily expressed through function calls of standard communication libraries, such as the ubiquitous sockets interface for TCP/IP [70]. It is provided by most operating systems and is exposed as a higher-level module in most programming languages (e.g., the standard Python `socket` module [5] and the Java `net.Socket` class [31]). However, problems arise, for example, when a process wants to selectively receive a message from one of a number of other processes, where the selection criteria depends on the state of the receiver and the contents of the message [4].

As the operating system does not know how data structures are represented, it is unable to pack a data structure into a network packet. Higher-level middleware mechanisms such as remote procedure call and remote method invocation are built on top of a sockets interface. Such mechanisms include Sun RPC, Java RMI, CORBA remote objects, SOAP, and XML-RPC [9].

Java provides a message service API [33] that allows point-to-point and publish/subscribe messaging. This support, however, is directed at loosely-coupled enterprise solutions, rather than tightly-coupled concurrent programs. Message destinations are created and maintained using administrative tools and, as such, cannot be considered a language-level mechanism, as they require the programmer to work outside of the language.

CORBA [19] provides push/pull messaging through event channels. Its greatest strengths are its language and architecture independence. CORBA is often used for loosely-coupled distributed programs, however, it requires a separate interface definition and corresponding compiler to maintain language independence.

In high-performance computing, the Message Passing Interface (MPI) [45, 48] provides a standard set of mechanisms for point-to-point and collective communication. Finally, as surveyed in [4], several parallel and distributed programming languages incorporate dedicated syntax and semantics for achieving communication between remote processes.

All existing mechanisms are useful in appropriate contexts and they all have advantages and disadvantages. Our observation is that the low-level mechanisms provide simplicity in concept, but require detailed handling of data. Likewise, the higher-level mechanisms hide some of the complexities of the underlying low-level mechanism, but introduce new burdens on the programmer. For example, TCP sockets require a receiver to handle partial reads, a situation when less data is available than has been requested. Also, explicit serialization is required for sending structured data. In contrast, complete objects can be accessed using a remote object system such as Java RMI [33]. Such systems require some form of interface definition, separate compilation tools, and a configured execution environment. Elegant mechanisms exist in dedicated distributed programming languages, but their usage is limited because their implementations are typically not widely supported and mainstream developers tend to use more general purpose programming languages.

A message received during the course of execution of a program may come from an arbitrary source. Like most imperative programming languages, we use a form of a *select* statement [68] as a construct for controlling such non-determinism [4]. Our novel communication mechanism, SFM, allows programmers to *selectively* receive messages based on not only their source but any other parameters specified within the message.

The SFM model provides a simple way to send arbitrary, dynamically typed messages between remote processes. Like traditional low-level mechanisms, SFM allows for the explicit transfer of data between processes, and like higher-level mechanisms, SFM automatically serializes structured data. SFM handles all the low-level details and allows a programmer to send structured data without additional interface specification. In SFM the location in the source code where `send()` is invoked serves as the specification. The sender and receiver implicitly agree on the type of data transferred. SFM can be viewed as an extension of function invocation semantics in dynamically typed languages. In such languages, the caller and callee agree on the types of data passed in and returned by convention. No static type checking is performed. Type errors are determined at run time.

2.4 The Python Language

The River run-time system, as well as its applications, are written in Python [62], a general-purpose high-level programming language. Its design philosophy emphasizes code readability. Python's core syntax and semantics are minimalistic, while the standard library is large and comprehensive. Python supports multiple programming paradigms (primarily object-oriented, imperative, and functional) and features a fully dynamic type system and automatic memory management, similar to Perl, Ruby, Scheme, and Tcl.

Python has gained in popularity due, in part, to its object-oriented programming model and system-independent nature. However, its concurrency model is not very flexible, and provided mostly by third-party packages. Python does support threads, but in practice they do not improve performance as only one thread at a time may hold Python's global interpreter lock. River's goal is to extend Python's rapid development capabilities to distributed systems. We aim to facilitate shorter design/implementation cycles by opening up the design space and enabling prototypes to run on real hardware. This allows programmers to quickly demonstrate scalability and feasibility.

Python uses dynamic typing and a combination of reference counting and a cycle-detecting garbage collector for memory management. Our RAI (Remote Access and Invocation) extension takes advantage of the garbage collector to deallocate remote objects when their local references go out of scope. An important feature of Python is dynamic name resolution (late binding), which binds method and variable names during program execution. For a thorough introduction to Python, we refer the reader to [5]; however, we believe that there are several language features that merit further explanation: keyword arguments, introspection, and the Python object model.

The River framework, and specifically the SFM mechanism, relies on Python's support for keyword arguments passed to a function. In general, an argument list passed to a function must have any positional arguments followed by any keyword arguments of the form `name=value`, where the keywords must be chosen from the formal parameter names. It is not important whether a formal parameter has a default value or not. No argument may receive a value more than once and formal parameter names corresponding to positional arguments cannot be used as keywords in the same calls [63].

When a final formal parameter of the form `**kwargs` is present, it receives a dictionary containing all keyword arguments except for those corresponding to a formal positional parameter. This may be combined with a formal parameter of the form `*args` which receives a tuple containing the positional arguments beyond the formal parameter list. This makes it possible to capture arbitrary attribute-value pairs by declaring a function that takes a single argument: `**kwargs`. Usage of this feature is shown in Figure 2.1.

```
def foo(*args, **kwargs):
  print args, kwargs

a = 1 ; b = 2 ; c = 3

foo(a, b, c)
foo(x = 9, y = 8, z = 7)
foo(a, b, c, x = 9, y = 8, z = 7)

$ python foo.py
(1, 2, 3) {}
() {'y': 8, 'x': 9, 'z': 7}
(1, 2, 3) {'y': 8, 'x': 9, 'z': 7}
```

Figure 2.1: Using Python Keyword Arguments

Another important feature of the Python language is its Python object model [61]. All data in a Python program is represented by objects or by relations between objects. (In conformance to Von Neumann's model of a stored program computer, code is also represented by objects.) Every object has an identity, a type, and a value. An object's identity never changes once it has been created; one may think of it as the object's address in memory. An object's type determines the operations that the object supports and also defines the possible values for objects of that type. The value of some objects can change. Objects whose value can change are said to be mutable; objects whose value is unchangeable once they are created are called immutable. Some objects contain references to other objects; these are called containers. Examples of containers are tuples, lists and dictionaries.

The two types used extensively in River are mapping types and callable types. Mapping types (dictionary is the only current example) represent finite sets of objects indexed by arbitrary index sets. The subscript notation `a[k]` selects the item indexed by `k` from the mapping `a`; this can be used in expressions and as the target of assignments. To support this interface, a class must define two methods: `__getitem__` to get an item, and `__setitem__` to set an item. Several classes in River define this interface and serve as containers for other objects.

A callable type is a type to which the function call operation (parentheses) can be applied. These are user-defined functions and methods, as well as class types (used when creating new objects of a class) and class instances. Class instances are callable only when the class has a `__call__` method; `foo(arguments)` is a shorthand for `foo.__call__(arguments)`. The Remote Access and Invocation extension, described in Chapter 5, makes use of this feature to allow function calls on remote objects.

A class has a namespace implemented by a dictionary object. Class attribute references are translated to lookups in this dictionary. When the attribute name is not found there, the attribute search continues in the base classes. A class may define a special method, `__getattr__`, which will be called upon each attribute access, *if* the specified attribute is not already defined as a member of the class or any of its parents. An exception may be raised to indicate an absence of such attribute. Otherwise, the returned value will be used as the value of the specified attribute. River, and specifically our SFM implementation, described in Chapter 3, depends on this feature to allow arbitrary attribute names within a message.

A special Python module called `inspect` [57] exists to support introspection at runtime. It provides several useful functions to help get information about live objects such as modules, classes, methods, functions, etc. For example, it can be used to examine the contents of a class or retrieve the source code of a method. We use this functionality to import code and data into remote Python interpreters in our Trickle programming model, described in Chapter 6.

With River we are using Python in a novel way to build parallel run-time systems. While Python has proven to be a high-productivity programming language in a wide variety of domains such as web services, program steering, data analysis, and conventional scripting, due to performance issues it is not typically used as a systems language. However, Python's concise syntax combined with dynamic typing, built-in data structures, and a rich standard library enables rapid program development. Also, because it is an interpretive language, one avoids the edit, compile, run cycle: changes are instantaneous. We leverage all of these benefits to use Python as a form of executable specification for run-time systems. Python's flexible syntax and introspection features allow developers to strike a balance between language-based and library-based programming models.

2.5 Distributed Programming in Python

The standard Python distribution comes with several library modules for network communication and Internet protocols, such as the `socket` module [64], or the `xmlrpclib` module [65]. 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 [49, 50] project is most similar to River, and specifically the Trickle extension described in Chapter 6. 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. Unlike IPython, River is a relatively concise extension for distributed programming. We have deliberately constrained the River programming model to make it easy to learn and use. See Section 6.5 for a comparison of Trickle and IPython.

PYRO [20] brings to Python a more traditional form of distributed objects based on a client/server model, similar to our Remote Access and Invocation (RAI) extension presented in Chapter 5. However, PYRO does not have direct support for asynchronous invocation and dynamic scheduling. Python support for the Linda [14] parallel programming model and its *Tuple Space* distributed data structure is provided by the PyLinda [76] module.

Finally, there exist several projects that provide Python wrappers for the standard MPI interface [45, 48]. These include PyMPI [42], MYMPI [37, 36], and Pypar [46]. Our implementation of MPI, built on top of River, is described in Chapter 7. Unlike other Python MPI interfaces, it is written entirely in Python and does not require additional libraries, such as a C implementation of MPI. At the same time, we believe that River conforms to the C MPI specifications more closely than the other Python interfaces. See Section 7.4 for a comparison of rMPI to existing Python-based MPI interfaces.

Chapter 3

Programming with River

River is a set of Python classes that provide a core interface and an underlying run-time system. A River program consists of one or more *virtual resources* (VRs) that execute on River virtual machines (VMs); VMs can exist locally or remotely and VRs are named using universally unique identifiers, or UUIDs [39]. A VM is simply a Python interpreter running the River run-time system. An initiating VR can discover existing VMs and deploy itself or other VRs onto selected VMs. Once running, VRs communicate with each other using a mechanism called *super flexible messaging* (SFM) for sending and receiving dynamically typed messages. This chapter explains these concepts and presents the River programming model and its API. In addition, several examples are presented to demonstrate how to build River applications. The implementation of River is described in Chapter 4.

3.1 Virtual Resources

A virtual resource (VR) is a process-like abstraction for Python programs. A VR runs on a River Virtual Machine (VM), which provides run-time support for deployment and communication. Each VR has its own thread of execution, its own message queue, and is named with a UUID (see Figure 3.1). A VR is specified by subclassing `VirtualResource`; therefore, a VR encapsulates state using the standard Python class type. The UUID serves as a unique VR identifier, automatically generated at VR creation time. In practice, a developer never needs to know the UUID value itself; it can be determined and used implicitly. Having a soft name, such as a UUID, as opposed to an IP address, allows VRs to potentially migrate between VMs. While not implemented in the version of River described in this thesis, we discuss migration and fault tolerance in Section 4.5. The message queue associated with each VR is used to support our *super flexible messaging* mechanism, described in Section 3.5, which VRs use to communicate with each other.

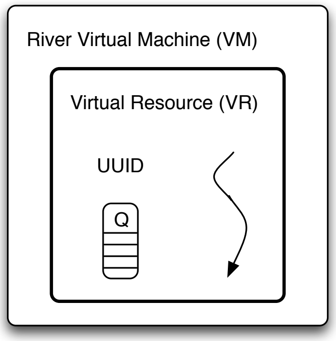

Figure 3.1: River Virtual Resource

The `VirtualResource` base class provides methods for VM discovery, VR creation, and message passing; Table 3.1 lists all the methods available to River programmers. Additionally, the `VirtualResource` class provides several member variables that are of use to the programmer and are assigned by the run-time system during VR deployment. These are listed in Table 3.2.

Method	Description
`send(dest=<UUID> [, <attribute>=<value> ] ...)`	Send method. (Section 3.5)
`recv([<attribute>=<value>, ...])`	Generic receive; returns first matching message. Several receive variations exist. (Section 3.5)
`discover([<attribute>=<value>, ...])`	Discovers VMs running on the local network. Returns a list of discovered VMs. (Section 3.4)
`allocate(<vmlist>)`	Allocates VMs given in a list for an application. Returns a list of allocated VMs. (Section 3.4)
`deploy(<vmlist>, <module> [, <func> [, <args>]])`	Deploys a given module on the VMs in vmlist. Optionally, the function to start execution may be specified, along with its arguments. Returns a list of VMs on which the module was successfully deployed. (Section 3.4)
`setVMattr(<name>, <value>)`	Sets an attribute with a given name and value on the VM where the current VR is executing. (Section 3.2)

Table 3.1: `VirtualResource` Methods

Name	Description
`uuid`	UUID of this VR.
`parent`	UUID of the parent VR, that is, the VR that created this one. In case this is the launcher, or initiator VR, this value is `None`.
`args`	Arguments passed to the VR, either during deployment, or in the case of the initiator, on the command line.
`label`	Label given to this VM. `None` by default.
`user`	Username of the VM owner.

Table 3.2: `VirtualResource` Data Members

3.2 Virtual Machines

A Virtual Machine is a Python interpreter executing the River run-time system. Multiple VMs can be started on the same host to take advantage of multiprocessor and multi-core systems. As each VM is a standalone process executing a separate Python interpreter, modern operating systems will execute them in parallel on different cores or processors. A VM generally executes only one *application* VR at a time: when that VR finishes, the VM becomes available for another VR. However, there is no hard limit on the number of VRs that may execute on a VM at the same time since each VR runs in its own thread.

Each running River VM maintains a set of key-value attributes that provide VM-specific information, such as the VM's current status (e.g., busy or available), its operating system, hostname, CPU speed, amount of memory, and River version. Default attributes are presented in Table 3.3. Additionally, a programmer may set his own attributes using the `setVMattr()` method call.

Name	Description
status	Status of the VM: busy or available.
appVRs	Number of non-system VRs currently running on the VM.
mem	Amount of memory on the remote system in megabytes.
cpu	CPU of the remote system.
cvr	UUID of the ControlVR on the remote VM.
host	Host name of the remote VM.
label	Label given to the remote VM.
pyver	Python version of the remote VM.
allVRs	Total count of all VRs (system and app) on the remote VM.
version	River version of the remote VM.
os	Operating systems of the remote VM.
user	Username of the VM owner.
caps	Capabilities of the remote VM.
arch	Platform architecture of the remote VM.

Table 3.3: Default VM Attributes

A special attribute, named `label`, exists to provide a way to name a VM or a set of VMs. For instance, a group of VMs running the same application may be labeled with the name of the application; this provides a logical separation between multiple groups running multiple applications. When all those VMs are running under the same user, each group may be independently discovered by its label; Section 3.7 explains how this can be done. For ease of assigning a label, it may be specified as a command-line argument when the VM is started.

Any custom attributes added will also be present in the VM's dictionary of attributes. Note that the names of all custom attributes will appear as the value of *caps* (short for capabilities), separated by whitespace. This makes it possible to find out all the custom attribute names at once.

3.3 VR Execution

To execute a River program, one or more VMs must be running locally or remotely on a local network. The VMs can be running on the nodes of a cluster, similar to MPD [11] found in the MPICH implementation of MPI. River VMs, launched by running `river` with no arguments, become available for use by future applications[1]. A River program is started by running `river` with the name of the application as an argument. After that, the *initiator*, a special instance of a VR, begins the computation. Figure 3.2 shows a possible River setup on a network.

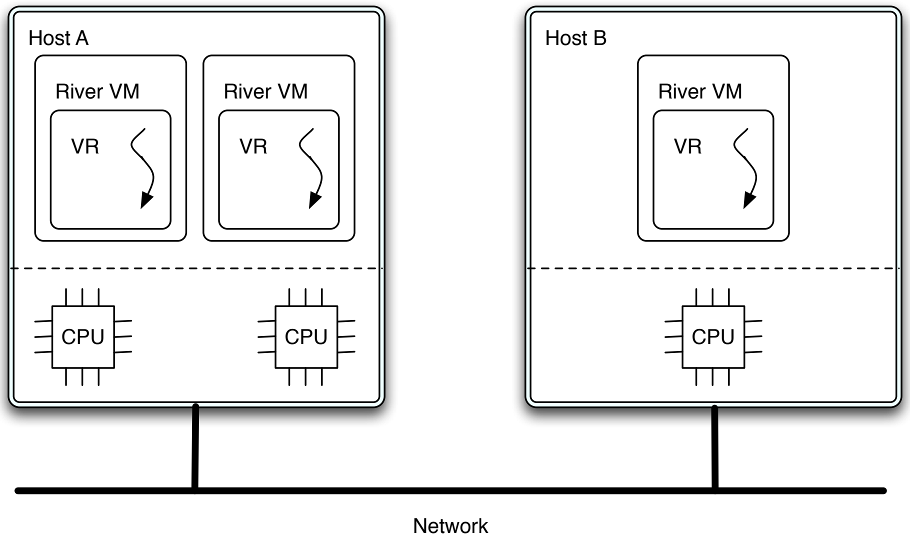

Figure 3.2: River Virtual Machines on a Network

In the case of the initiator, a new River VM is started and immediately begins executing the specified Python source file (River module). On the initiator only, the execution begins in the `vr_init()` method. The `vr_init()` method is invoked before `main()` in the initiator. This approach, in which a VM is created for the initiator, supports a wide range of startup strategies and allows for custom startup code that can be easily tailored for different parallel programming model extensions. The `vr_init()` method is responsible for locating running River VMs on the network, allocating them for an application and deploying itself (or another module) onto them. The *discover-allocate-deploy* sequence is discussed in detail in Section 3.4.

[1]The exact startup procedure of River VMs is dependent upon the execution environment. For instance, they may be launched as startup items at login on a desktop, or by special management tools on a cluster.

Non-initiator VRs begin execution at the method specified during deployment; by convention this method is called `main()`.

If the `vr_init()` method succeeds, it must return `True`. This indicates to the launcher code that deployment was successful and that the application should proceed. At this point the launcher invokes the VR's `main()` method. After the `main()` method completes, the VR finishes executing and exits, terminating the thread. The VM then becomes available for additional work. Figure 3.3 illustrates this execution sequence.

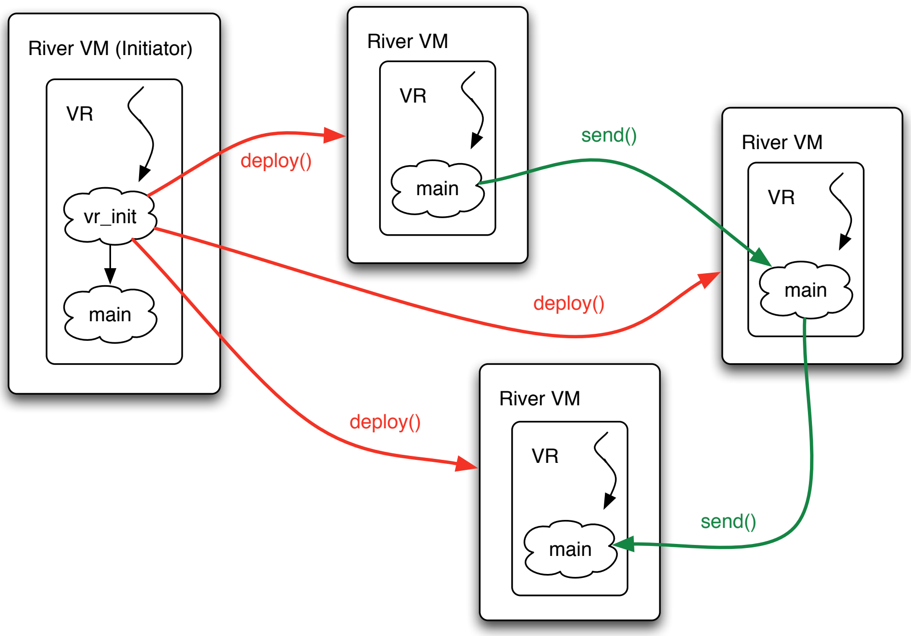

Figure 3.3: River Program in Execution

3.4 Discovery, Allocation, and Deployment

Starting new VRs on remote VMs is a three step sequence, involving three distinct method calls in this order: `discover()`, `allocate()`, and `deploy()`.

First, the `discover()` call is used to locate available VMs on the network. In its default form, invoked without arguments, it will return a list of *VM descriptors* representing all running VMs that are available for use and are running under the same username as the initiator. We use the username string, rather than numeric user id's, to differentiate between users, since the same person may have a different user id on two separate systems, but is likely to have the same username[2]. The implementation of `discover()` is described in detail in Section 4.4.

A VM descriptor is a dictionary mapping VM attributes to their respective values. The descriptor contents may be examined to select a subset of VMs, based on their attributes, for execution. In the case where a user wants to execute an application on all available VMs that were discovered, the list of descriptors need not be examined or modified.

The next step is to pass a list of the VM descriptors corresponding to the VMs on which the application will be executed to the `allocate()` method. In its simplest form, the call to `allocate()` takes the return value from `discover()` as its only argument. At this point each remote VM ”reserves a spot” for a new VR, generates a new UUID for it, and returns the UUID back to the allocator to be incorporated into the VM descriptor. The allocation step allows River to avoid a possible race condition where multiple initiators are trying to use the same VM for execution. The allocation requests are serviced on a first-come-first-serve basis, so if several initiators attempt to allocate the same VM, only the first one will succeed and the rest will fail. Since UUIDs are assigned to participating VRs ahead of deployment, each new VR can be given a list of its peers. This is useful when many VRs are executing the same code in a SPMD fashion. The `allocate()` method also returns a list of VM descriptors corresponding to the allocated VMs. If allocation failed on one or more VMs, a `RiverError` exception is raised. This allows the programmer to retry discovery and allocation, perhaps requesting fewer VMs.

Finally `deploy()` is invoked to begin execution of VRs on the allocated VMs. The `deploy()` function takes four arguments:

- A list of allocated VMs onto which to deploy.
- The River VR module to deploy.
- Optionally, which function in the module should start execution.
- Optionally, the arguments to `main()` or the specified start function.

If the start function is not specified, execution will begin at the `main()` method of the module, invoked without arguments. In theory, every deployed VR can execute a different function or even a different module, however, in practice that rarely makes sense and usually all VRs will execute the same module but perhaps a different function, depending on the work distribution paradigm used. Thus, programs using the *manager-worker* paradigm may choose to have the worker VMs execute one function and the initiator

[2]In this prototype of River we assume an open network and do not employ strong authentication or provide encryption. However, River could be deployed on top of a virtual private network (VPN) to achieve stronger security.

(manager) VM another. On the other hand, programs following the SPMD paradigm of all nodes executing the same code will elect to call the same function across all participating VRs.

The return value of the `deploy()` method is another list of VM descriptors, this time corresponding to the VMs on which the deployment was successful. However, a new attribute, `uuid`, is added to each descriptor, which specifies the UUID of the newly created VR. As such, the list of all `uuid` attributes from every descriptor in the *deployed* list represents all peers participating in the running application. A `RiverError` exception is raised if deployment failed on one or more VMs.

The initiating VR thus deploys itself or other VRs onto the remote VMs. This *discover-allocate-deploy* approach is flexible enough to handle a wide range of process creation semantics. Furthermore, these interfaces can be invoked at any time during program execution, thus allowing River programs to dynamically utilize new machines (VMs) as they become available. Note that it is not necessary to have River application code reside locally on each machine, as it will be automatically included with the *deploy* request.

3.5 Super Flexible Messaging

In River we introduce a novel message passing mechanism called *super flexible messaging* (SFM). Our mechanism leverages Python's dynamic typing and named arguments for sending structured data as attribute-value pairs between VRs. SFM allows for an arbitrary number of attributes and the values can be any Python data type that can be serialized using the `pickle` module. Here is the syntax for sending and receiving:

```
send(dest=<UUID> [, <attribute>=<value> ] ...)
m = recv([<attribute>=<value> [, <attribute>=<value>] ...])
```

Note that the attribute names need only be specified at the call sites of `send()` and `recv()`. This mimics normal Python function invocation with named parameters and does not require any additional specification of the message structure elsewhere in the code. This allows developers to send arbitrary data without declaring the names and types of message fields and without explicit serialization. The `recv()` mechanism allows for selective message retrieval by specifying specific required attributes and messages to received. The fields of a received message are accessed using member dot notation: `m.attribute` or via a standard dictionary lookup: `m['attribute']`. The latter construct is useful when dynamically generating attribute names, e.g., in a loop. The list of attribute names may also be obtained by calling the `m.keys()` method.

Sent messages are delivered to the message queue of the destination VR. The `recv()` implementation uses the specified attribute-value pairs to select messages from the queue (see Chapter 4 for more details on the implementation). The `send()` function is non-blocking and `recv()` is blocking[3]. We also provide a non-blocking receive (`recv_nb()`) and a `peek()` predicate for querying the message queue. A single message queue per virtual resource is used to simplify the interface and to more easily support state management. Higher-level message passing mechanisms such as mailboxes can be built using the basic SFM primitives.

As previously stated, each VR has a receive queue and a UUID. The SFM `send()` operation must explicitly specify a UUID as the destination. On the receiving side SFM uses the destination UUID to associate a message with the destination VR and deposit it on the correct receive queue. The `recv()` operation retrieves qualifying messages from the receive queue.

An SFM message is a set of attribute-value pairs. An attribute name is always a string, whereas a value can be of any type, such as integer, string, list, object, etc. Two special attributes are reserved and have special meaning: `src` and `dest`. The `dest` attribute is required when sending and specifies the UUID of the destination VR. The `src` attribute is added automatically by the run-time system and contains the UUID of the originating VR. If the programmer leaves out `dest` or attempts to provide `src` in a `send()` operation, a `RiverError` exception is raised. Thus, sending a message can be as simple as passing all the desired attributes and their values as named parameters to the `send()` function. The syntax for `send()` is:

```
send(dest=<VR> [, <attribute>=<value> ] ...)
```

Typically, at least one attribute-value pair is provided in addition to the `dest=<VR>` pair. However, it is possible to send an empty message for synchronization purposes. Consider the following example, where the destination UUID is stored in a variable `B`:

```
send(dest=B, tag='input', data=[1,2,3,4,5])
```

Note that the attribute names are completely arbitrary. Also, data can come from the surrounding scope:

```
rec = { name : 'Dave', id : 24 }
send(dest=newdest, protocol='db', idrec=rec)
```

[3]Our current implementation depends on the sockets library for buffering sends.

As mentioned previously, attribute values can be any Python type that can be pickled. This includes most Python types, e.g., most immutable types, lists, dictionaries, and user-defined objects.

The SFM `recv()` function is similar to the `send()` function in that it takes a variable number of attribute-value pairs as arguments. However, the attribute-value pairs constrain which messages should be received. Each received message is guaranteed to have at least the attribute-value pairs specified in the `recv()` call. The return value is an SFM message object whose attributes can be accessed using the member variable dot notation. The default `recv()` function is blocking; thus, if no messages match the specified criteria, the caller is blocked until a matching message arrives. The syntax of `recv()` is:

```
message = recv([, <attribute>=<value> ] ...)
```

In its simplest form, receiving without any arguments will return the first message in the queue in first-in, first-out order:

```
msg = recv()
```

A more common scenario is to receive a message from a particular source (note that the `src` attribute will appear in all messages, even though it is not explicitly specified by the sender):

```
m = recv(src=A)
```

Matching can be done on any number of attributes but the following conditions have to be satisfied: all attributes must appear in the message and their values must be equal to the values specified in the `recv()` call. Note that this does not mean the message must only have the specified attributes, just that the specified attributes must be present in the message. Here we ask for messages that came from VR `A` and have the `protocol` attribute set to the string value `input`:

```
m = recv(src=A, protocol='input')
```

Another possibility is to receive messages from any source as long as the `protocol` attribute is equal to `db`:

```
m = recv(protocol='db')
```

Since the attribute names are completely arbitrary, matching can be done on any attribute. If a message does not have one or more attributes specified in `recv()`, then the message will not be accepted (it will remain in the queue to be matched later). As shown in Figure 3.4, accessing the attribute values of a message is identical to accessing member variables of Python objects.

```
# VR A
mylist = range(10)
send(dest=B, mytag=42, input=mylist)

# VR B
m = recv(mytag=42)
print m
print m.src
print m.input
print m.mytag

# Output
Message: {’dest’: ’B’, ’src’: ’A’, ’mytag’: 42, \
        ’input’: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]}
A
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
42
```

Figure 3.4: Simple SFM Example

Figure 3.5 shows how different invocations of `recv()` can match different messages in the queue, and how the same message may be matched by specifying different arguments to `recv()`. The first matching message will be returned. Note that we just show sample invocations and the messages they will match in the queue, not a sequence of steps.

Additionally, an attribute passed to `recv()` may be paired with a lambda expression or a named function rather than a value. This can be done to restrict attribute values to an arbitrary subset, (e.g., `id = (lambda x : x > 100)`). In this case, no direct comparison is done but instead that function is invoked with the attribute value (from the message) as the only argument if the attribute exists in the message. If the function returns `True`, the attribute is considered to match. If all other attributes match, the message is returned. This allows the user to perform more sophisticated matching such as simulating a logical OR expression, or arbitrary string matching. For example, the following code will match a message if it contains a `tag` attribute equal to one of the two specified values:

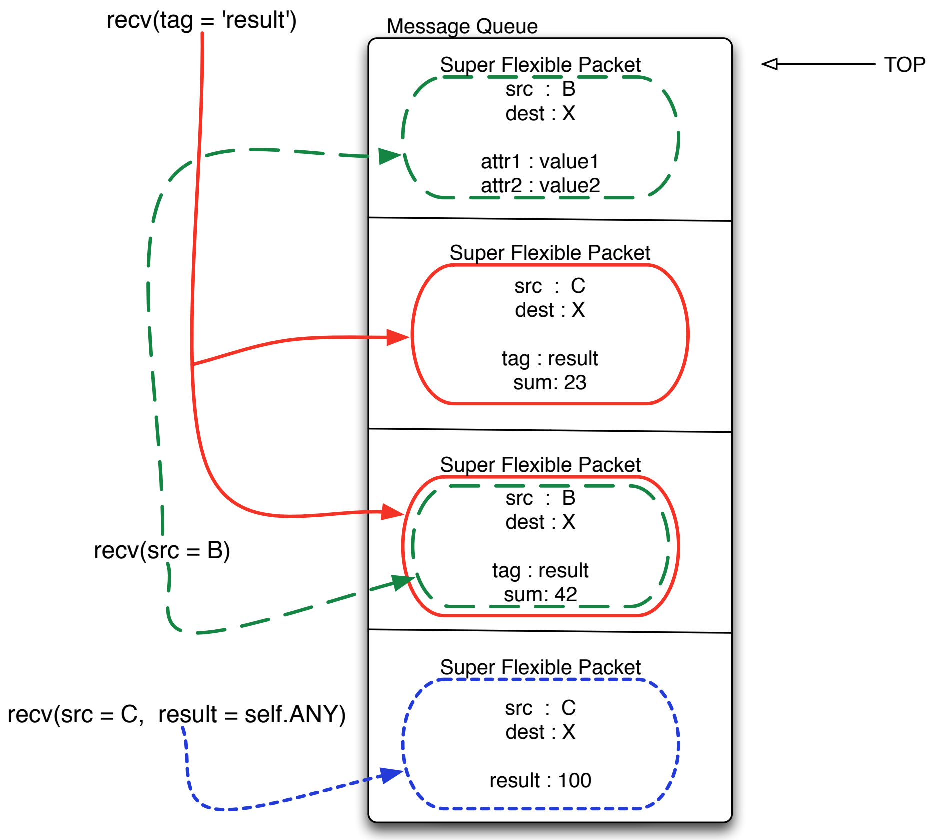

Figure 3.5: SFM Matching

```
m = recv(tag=lambda x: x == 23 or x == 42)
```

Finally, to receive a message that contains a certain attribute, regardless of its value, the `ANY` constant (`self.ANY` within VR code) can be specified to achieve a wildcard effect[4]. For instance, the following code will match any message as long as it has a `tag` attribute, no matter what its value is:

```
m = recv(tag=self.ANY)
```

As mentioned above, several variants of the `recv()` function exist. All of them take a variable number of attribute-value pairs as arguments but vary in their blocking effect and return value. Table 3.4 summarizes them.

The difference between the default `recv()` and `recv_nb()` functions is that `recv()` blocks when

[4]`self.ANY` is equivalent to `lambda x: True`.

Name	Description
recv	Blocking receive (most commonly used).
recv_nb	Non-blocking receive. Raises an exception if no matches are found.
peek	Non-blocking peek. Returns `True` or `False` based on match result.
regcb	Registers a callback function to be invoked when a message matches the specified attributes.
unregcb	Unregisters a callback function.

Table 3.4: Various Receive Functions

no message matching the specified attributes has been found, whereas `recv_nb()` raises a `QueueError` exception instead of blocking. The `peek()` function works in exactly the same way but returns `True` or `False`, indicating whether a match has been found without removing the message from the queue. It is useful to avoid blocking; for instance, when we expect a number of messages from our peers with a given attribute, but we do not know exactly how many messages there will be. In this case, we can invoke `peek()` in a loop and as long as it returns `True`, there will be a matching message on the queue, so we can then call `recv()` without blocking:

```
while peek(tag='result'):
    p = recv(tag='result')
```

Finally, an alternate method of receiving a message is registering a user-defined callback function with the queue. Only one callback function may be registered. The function will be invoked asynchronously when a matching message arrives. The syntax is the same as for the other SFM functions except that a `callback` keyword argument must be provided, giving the reference to the callback function that should be invoked. The callback function must take a single argument: the matching message.

```
def mycbfunc(p):
    # process message

regcb(src=A, tag='result', callback=mycbfunc)
```

The callback function needs to be registered only once but it will continue to be called for every matching message until unregistered. The `unregcb()` arguments must match the `regcb()` arguments exactly.

```
unregcb(src=A, tag='result', callback=mycbfunc)
```

3.6 A Complete River Program

Building on the concepts explained in previous sections, Figure 3.6 presents a simple but complete River example. This program attempts to discover all running VMs on the network and deploy itself onto them. All VRs then send a message to the initiator with their hostname, and the initiator prints them out.

```
 1 from socket import gethostname
 2 from river.core.vr import VirtualResource
 3
 4 class Hello (VirtualResource):
 5     def vr_init(self):
 6         discovered = self.discover()
 7         allocated = self.allocate(discovered)
 8         deployed = self.deploy(allocated, module=self.__module__)
 9         self.vrlist = [vm['uuid'] for vm in deployed]
10         return True
11
12     def main(self):
13         if self.parent is None:
14             for vr in self.vrlist:
15                 msg = self.recv(src=vr)
16                 print '%s says hello' % msg.myname
17         else:
18             self.send(dest=self.parent, myname=gethostname())
```

Figure 3.6: First River Program

The `Hello` class is declared as a subclass of `VirtualResource` on line 4. Note that the same module is executed on all peers. Recall that the `vr_init()` function is invoked only on the initiator. Next, all available VMs running under the same username (the default behavior) are discovered on line 6. All the discovered VMs are allocated on line 7 and the module is then deployed to each of them on line 8. A list of all peers is then saved on line 9. Once `vr_init()` returns, control passes to the `main()` function on the initiator. All the deployed VRs start executing `main()` as well, since no other function was specified in the `deploy()` call.

Recall that `self.parent` contains the UUID of the initiator in all the deployed VRs, but is equal to `None` in the initiator itself. In this way we can distinguish the initiator from the rest of its peers at run time. In this example, every child VR sends a message to the initiator on line 18, specifying its hostname as the only message attribute, `myname`. The initiator has a list of all its child VRs, and waits to receive a message from each on line 15. The hostname of the child VR is then printed on line 16. Note the usage of the dot notation to access the message attribute and the attribute name is identical to what the originating VR specifies in the `send()` call.

3.7 Selective Discovery

So far we have shown only trivial usage of the discover, allocate, deploy sequence: all available VMs are discovered and used to execute the example in Figure 3.6. When invoked without any arguments, `discover()` will return a list of available VMs belonging to the user. However, it is possible to further restrict or expand the list of discovered VMs. This can be done by passing arguments to `discover()` in the form of attribute-value pairs, much like in the case of SFM. However, in this case the specified arguments will be matched against the attributes of the remote VMs.

A wildcard `self.ANY` can be specified as the attribute value in the `discover()` call to indicate that any value of that attribute in the remote VM is acceptable, as long as that value is present. This allows testing for the *presence* of an attribute, ignoring its value.

For instance, Figure 3.7 shows how *all* running VMs, regardless of their status, be it available or busy, can be discovered. Note that although every VM can be discovered, not every VM may be allocated or used for deployment. The same default rules apply to allocation and deployment as to discovery: a VM must be available and must belong to the same user as the one attempting to allocate it for an application.

```
from river.core.vr import VirtualResource

class Foo(VirtualResource):
   def main(self):
       discovered = self.discover(status=self.ANY)
       print 'Discovered %d VMs:' % len(discovered)
       for vm in discovered:
           print '%s %s' % (vm['host'], vm['status'])
```

Figure 3.7: Broader Discovery

Furthermore, just like with SFM, it is possible to specify a `lambda` expression or a named function as an attribute value. In this case, the function will be invoked by `discover()` during the matching phase with the actual value as the argument. If the function returns `True`, the match is successful, and the corresponding VM is included in the returned list. Figure 3.8 shows a `discover()` call that looks for VMs running on Intel architecture. Note that we present only the relevant method call, as the rest of the code is identical to Figure 3.7.

```
discovered = self.discover(arch = lambda x: x in ['i386', 'x86_64'])
```

Figure 3.8: Selective Discovery with Functions

3.8 Examples

This section presents some sample River programs: `MCPi`, a program that calculates the value of π using the Monte-Carlo method; `WFreq`, a word frequency counter, and parallel dot product and matrix multiplications functions from a conjugate gradient solver.

3.8.1 Monte Carlo Pi

Figure 3.9 shows an implementation for calculating Pi using the Monte Carlo method and a manager-worker paradigm. As always, `vr_init()` method executes first on the initiator to set up the execution environment, where available VMs are discovered and deployed on lines 6-7. A list of participating VMs is gathered on line 8. Next, by default, control passes to `main()` on the initiator, and to `worker()`, the function specified during deployment, on the worker VMs.

The remainder of the program is driven by the manager, contained in the `main()` function. The number of iterations is passed by the user on the command line and is available as `self.args` variable. The manager then computes and sends out the number of iterations per worker on line 17. On lines 20-21 the manager collects the results from all workers and computes the final result.

A worker receives the number of iterations on line 29, performs the work and sends back the result on line 38. The UUID of the manager is available to the worker as the `self.parent` variable, which contains the UUID of the initiator.

A more detailed version of this example is presented in Appendix A.1.

```
import random
from river.core.vr import VirtualResource

class MCpi (VirtualResource):
    def vr_init(self):
        VMs = self.allocate(self.discover())
        VMs = self.deploy(VMs=VMs, module=self.__module__, func='worker')
        self.workers = [vm['uuid'] for vm in VMs]

         return True

    def main(self):
        n = int(self.args[1])
        numworkers = len(self.workers)

        for uuid in self.workers:
            self.send(dest=uuid, tag='n', n=n/numworkers)

        in_circle = 0
        while numworkers > 0:
            p = self.recv(tag='result')
            in_circle += p.result
            numworkers -= 1

        pi = 4.0 * in_circle / n
        print 'Our approximation of Pi = %f' % pi

    def worker(self):
        p = self.recv(src=self.parent, tag='n')

        in_circle = 0
        for i in range(0, p.n):
            x = 2.0 * random.random() - 1.0
            y = 2.0 * random.random() - 1.0
            length_sqr = x * x +  y * y
            if length_sqr <= 1.0: in_circle += 1

            self.send(dest=self.parent, result=in_circle, tag='result')
```

Figure 3.9: Monte Carlo Pi

3.8.2 Distributed Word Frequency Count

Figure 3.10 shows a distributed word count program that computes word frequencies across multiple files. We again use a manager-worker paradigm to distribute the work, and assume that there are enough files to create equal-sized chunks of work for each worker VR. The general structure of the program is the same as the previous example, so we omit most of the skeleton code like the `vr_init()` function. Complete source code appears in Appendix A.2.

Each worker receives a list of files to process, loops through them, opens each file, tokenizes it, and counts the number of times each word occurs. The results are saved in a dictionary, which is sent back to the manager. The manager begins by splitting all input files (passed on the command line) into work slices, one for each worker. The work (a list of file names) is then sent to the worker, and the manager then waits for the results. As the results come in, the manager combines the dictionary from each worker into one final result.

```
def manager(self):
    for w in self.workers:
        self.send(dest=w, tag='files', files=fnames[w])

    m = {}
    for w in self.workers:
        p = self.recv(tag='result')
        rm = p.result
        for i in rm:
            m[i] = m.get(i, 0) + rm[i]

def worker(self):
    p = self.recv(src=self.parent, tag='files')

    m = {}
    for f in p.files:
        wordcount(f, m)

    self.send(dest=self.parent, tag='result', result=m)
```

Figure 3.10: Word Count

3.8.3 Parallel Dot Product and Matrix Multiplication

Figure 3.11 illustrates a simple way of implementing parallel dot-product computation and parallel matrix multiplication. This excerpt is taken from the River version of a parallel conjugate gradient (CG) solver. Due to space considerations, the entire program code is not shown, but appears in Appendix A.3.

First, the `pdot()` function is listed, which includes a naive implementation of the MPI `Allreduce()` operation. The local dot product is calculated on line 2, then all worker VRs send their local result to the parent VR on line 5. The parent collects the local results from all the workers on line 11 and sums them up, then sends the global sum back to each worker on line 15. Finally, all the worker VRs receive and return the global result on line 6.

Next, in the `pmatmul()` function we emulate the MPI `Allgather()` operation. All processes except the parent VR send their local part of vector `x` to the parent on line 21 and then block in `recv()` until the parent sends back the entire global `x` on line 22. Meanwhile, the parent receives everyone's part on line 28 and creates the global version of the vector, which is then propagated to all nodes on line 32. Finally, all nodes perform the multiplication of the entire vector `x` and their piece of the matrix `A` and return the result on line 34.

```python
def pdot(self, x, y):
    result = dot(x, y)

    if self.uuid != self.parent:
        self.send(dest=self.parent, result=result, tag='pdot')
        m = self.recv(src=self.parent, tag='allreduce')
        return m.result

    else:
        for w in self.workers:
            m = self.recv(tag='pdot')
            result += m.result

        for w in self.workers:
            self.send(dest=w, result=result, tag='allreduce')

    return result

def pmatmul(self, A, x):
    if self.uuid != self.parent:
        self.send(dest=self.parent, x=x, tag='pmatmul')
        p = self.recv(src=self.parent, tag='allgather')
        globalx = p.globalx

    else:
        globalx = x[:]
        for w in self.peers:
            p = self.recv(src=w, tag='pmatmul')
            globalx += p.x

        for w in self.peers:
            self.send(dest=w, globalx=globalx, tag='allgather')

    return matmul(A, globalx)
```

Figure 3.11: Parallel Dot-Product and Matrix Multiplication

3.9 Extensions

The River interface combined with Python's syntax and data types can also be used to implement alternate parallel programming models. The benefit of developing a programming model in River is that development time is spent on the interesting parts of the dynamic run-time support, rather than on mundane low-level details that would be required in languages such as C, C++, and Java. We also find Python's syntax significantly rich to explore different parallel constructs; to demonstrate this we have developed several alternate programming models. Chapters 5, 6, and 7 describe the extensions included with the River run-time system. This section discusses how a programmer can develop an extension to River and illustrates this with an example, a remote dictionary extension. The amount of time it took to develop this extension serves as a testament to productivity afforded by the combination of River and Python — it took less than two hours to design and implement a working model.

We start by examining a possible sample program built upon this extension. Figure 3.12 presents a variation of our word frequency counter from Section 3.8 that is based on the remote dictionary extension. Note that only the `wordcount` function remains unchanged, but everything else is different. Most importantly, there is no reference to any of the River Core API methods, or even the `VirtualResource` base class. Instead we derive our class from the `RemoteDictionary` extension class. The interface provided by this extension class is the two dictionary methods (`__getitem__` and `__setitem__`), which allow usage of the bracket operators on the `self` object. Thus we pass `self` to `wordcount` as the dictionary in which to store results.

```
from remdict import RemoteDictionary

class WFreq (RemoteDictionary):
    def main(self):
        for f in self.args:
            self.wordcount(f, self)

    def wordcount(self, filename, m):
        f = open(filename)
        for l in f:
            for t in l.split():
                m[t] = m[t] + 1
        f.close()
```

Figure 3.12: Word Frequency Example with the Remote Dictionary Extension

A partial implementation of the `RemoteDictionary` extension class is shown in Figure 3.13;

see Appendix A.4 for the complete source code. The most important methods here are `__getitem__` and `__setitem__` to get and set items in the dictionary, respectively. On a get, we send a request to look up the specified key remotely, wait for a reply, and return the result to the caller. On a set, we simply send the key and value pair to the remote machine.

The same class performs the functions of the server hosting the dictionary. In fact, the actual dictionary will reside on the initiator, which will service get and set requests from other machines. After following the usual *discover-allocate-deploy* procedure in `vr_init()`, the initiator enters a loop, accepting and servicing dictionary requests. Once all the workers have exited, the final contents of the dictionary are printed out.

This is a trivial example that provides an interface in which multiple clients communicate with a single server. Access to the global dictionary results in communication with the initiator, where the actual dictionary is stored with no redundancy or fault tolerance. The performance is also very poor because each dictionary access results in network communication with no caching. However, this simple example illustrates how to build a River extension.

We include more sophisticated examples of River extensions in the following chapters. Chapter 5 presents our Remote Access and Invocation extension. Chapter 6 discusses Trickle, an explicit but simple distributed programming model. Chapter 7 presents a River version of the MPI programming library.

```
class RemoteDictionary (VirtualResource):
    def __getitem__(self, key):
        self.send(dest=self.parent, action='get', key=key)
        p = self.recv(src=self.parent, key=key)
        return p.value

    def __setitem__(self, key, value):
        self.send(dest=self.parent, action='set', key=key, value=value)

    def vr_init(self):
        self.m = {} ; deployed = [] ; end = 0

        VMs = self.allocate(self.discover())
        files = self.args[1:]
        slice = len(files) / len(VMs) ; extras = len(files) % len(VMs)

        for worker in VMs:
            beg = end ; end = beg+slice
            if extras: end += 1 ; extras -= 1

            deployed += self.deploy(VMs=[worker], module=self.__module__,
              func='rdmain', args=files[beg:end])

        self.peers = [vm['uuid'] for vm in deployed]
        self.startfunc = self.serve

        return True

    def rdmain(self):
        self.main()
        self.send(dest=self.parent, action='done')

    def serve(self):
        done = 0
        while done < len(self.peers):
            p = self.recv()
            if p.action == 'get':
                v = self.m.get(p.key, 0)
                self.send(dest=p.src, key=p.key, value=v)
            elif p.action == 'set':
                self.m[p.key] = p.value
            elif p.action == 'done':
                done += 1
```

Figure 3.13: Remote Dictionary Extension

Chapter 4

Design and Implementation

The River framework consists of run-time support code within the River Virtual Machine (VM) and `VirtualResource` base class. Internally the River VM is comprised of several components that work together to provide the VR execution environment. These include the unified network connection model, name resolution support, a connection caching mechanism, low-level data transmission routines, and our communication framework called *super flexible messaging* (SFM). This chapter describes the design and implementation of the River run-time system. The River Core source code is included in Appendix B.

4.1 Super Flexible Messaging

The SFM implementation consists of two main modules: the `packet` module (shown in Appendix B.4), which provides the basic encoding and decoding functionality, and the `queue` module (shown in Appendix B.5), which provides packet matching at the receiving side. We call an SFM message a *super flexible packet* (SFP). This section details encoding, decoding, and queue matching.

The SFM implementation does not include any low-level network communication routines. Instead, such communication is provided by the underlying environment, in this case, the River VM, as described in Section 4.2. SFM only deals with representing the data, rather than its transmission. In principle, SFM could be used on top of any network transport.

4.1.1 Encoding

Encoding consists of serializing the specified attribute-value pairs and preparing the serialized packet to be sent over the network (e.g., via a socket connection). An SFP is created by passing a list of named parameters to the `encode()` function, which first encodes them into a binary format. Since named parameters in Python appear as a dictionary, we accomplish this by serializing (or *pickling*, as it is called in Python [59]) the dictionary and adding a small header. The SFP header consists of two fields: a magic signature string, used for sanity checking, and the length of the serialized payload. As described in Chapter 3, an attribute is a string and its value can be of any serializable type. At least one attribute, the UUID of the destination VR, is required. Additionally, another attribute, the UUID of the source VR, is added to the dictionary automatically. A `RiverError` exception is raised if no destination is specified or if the user provides his own source. Figure 4.1 illustrates the encoding sequence.

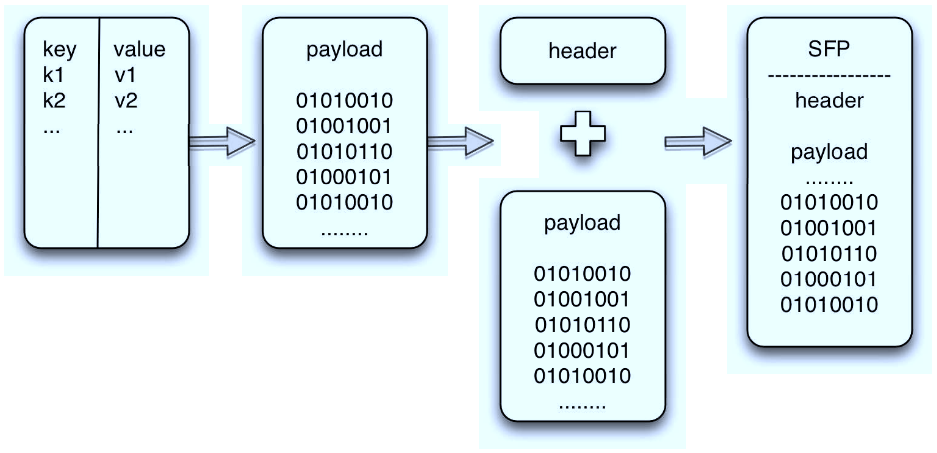

Figure 4.1: SFM Encoding Process

Serialization of data exacts a performance penalty because the data must be transformed and copied. Additionally, we incur an extra copy when we combine the payload with the SFM header. We show in Section 4.6.1 that the overall performance of River applications would greatly benefit from optimizing the encoding function. Appendix B.4 presents the source code to our encoding function.

4.1.2 Decoding

The `decode()` function operates as follows. First, the header is decoded and parsed to determine whether a packet has been received in its entirety and to ensure that it is a valid SFP. If the packet is incomplete, it is not processed until more data is available to complete the remaining part. Otherwise, the payload is deserialized into a Python dictionary and then used to construct a packet object that represents the SFP. The `Packet` class overrides the internal `__getattr__()` method, allowing users to access the packet's data by referencing attribute names using member field dot notation. This lets the user refer to the attributes contained within the packet using the same names as originally specified by the sender. Furthermore, the `Packet` class implements the standard dictionary methods such as `keys()`, `values()`, and `items()`.

Two items are returned: an unserialized packet object (or `None` if there was not enough data to complete a packet), and any leftover data, which could be an empty buffer, or the entire input buffer. Since the `decode()` function does not preserve any state, the leftover data should be saved by the caller and then prepended to the next sequence of bytes obtained from the network before the next round of decoding is attempted. If the `decode()` function determines that the input buffer does not contain a valid serialized SFP, an `SFMError` exception is raised. This should never be the case when a reliable low-level communication protocol, such as TCP, is employed. Appendix B.4 presents the source code to our decoding function.

4.1.3 Queue Matching

The message queue consists of an ordered list of received SFPs (in a first-in, first-out order) and a condition variable, which is used to block a VR when no packet can be returned. In its basic form the receive function (the queue get method), like the send function, takes a variable number of attribute-value pairs (again, as Python named parameters) but uses them as keys to match against packets in the queue. That is, for a packet to match, all attribute-value pairs passed to the receive function must appear in the packet and have the exact same value. To perform matching we linearly walk the queue of packets and check the specified attributes against every packet. If an attribute was specified to the receive function but does not appear in a packet or its value in a packet is different than the one specified, matching fails and we proceed to the next packet. Only the first matching packet is returned. If there are several matching packets, multiple invocations of the receive function are necessary to retrieve them.

The basic receive function is blocking and if no packets are present in the queue or none of them match, the calling process is put to sleep on a condition variable. Once a new packet is added to the queue, the VM will signal on the condition variable, waking up the sleeping process and allowing it to retry matching.

If the value of an attribute is a lambda expression or a function rather than a simple data type, no direct comparison is done but instead that function is invoked with the attribute value (from the packet) as the only argument. If the function returns `True`, the attribute is considered to match. If all other attributes match, the packet is returned. This allows the programmer to perform matching based on a set of values and simulate other logical constructs.

Additionally, we provide other variants of receive, such as a non-blocking receive that raises a `QueueError` exception if no packets were matched and a `peek()` function that returns a boolean value indicating whether a matching packet can be retrieved from the queue (see Section 3.5). The matching semantics are exactly the same as the basic receive: any number of attribute-value pairs used for matching are passed in as input parameters. The only difference is that the VR is not blocked on a condition variable if no matching packets are found. The source code for our queue implementation is shown in Appendix B.5.

4.2 River VM

In the common case, there are three threads of execution inside the River VM: the network receive thread, which is responsible for receiving and decoding incoming messages; the Control VR thread, which responds to discovery and deployment requests; and the application VR thread, which executes application code and provides VR support functions. These include methods to send data, register a VR, set VM attributes, obtain statistics, and shut down the VM. Figure 4.2 shows a general view of River VMs.

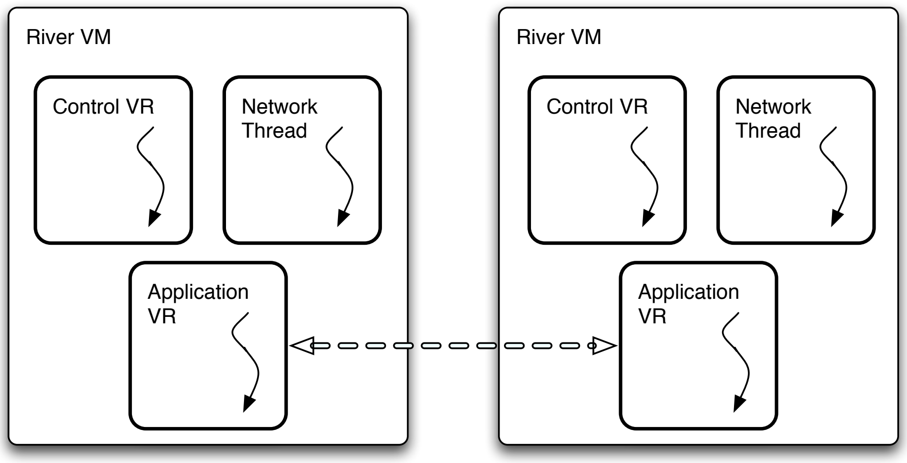

Figure 4.2: River Virtual Machines

We use a local socket for synchronization between network receive thread and application VR thread. That socket is monitored by the `select()` system call in the network receive thread and is employed for two purposes: to signal the network thread to shut down, and to indicate to it that a new connection has been established by the VR thread and the appropriate socket should be added to the `select()` read set.

We make the distinction between *system* and *application* VRs to differentiate between VRs that are part of the run-time system and VRs created by the user and running a River application, respectively. Unlike application VRs, there are no restrictions on the number of system VRs that can be created in a VM. The maximum number of application VRs is configurable, but in common usage the maximum is set to one.

The most important task of the VM is the transmission of messages between VRs: sending on the source VR and receiving on the destination VM. The implementation of the River VM appears in Appendix B.2.

4.2.1 Sending Data

The `send()` function takes an arbitrary number of attribute-value pairs comprising a message, encodes them, and sends the resulting binary data to the destination VR. It executes in the context of a running VR and consists of the following steps:

1. Resolve the destination UUID into an (IP, port) tuple.
2. Retrieve a connection to the VM listening on that (IP, port) tuple.
3. SFM is used to encode the specified attribute-value pairs into a binary representation.
4. The resulting buffer is transmitted over the network.

Each of these steps is described in detail in the following sections. Figure 4.3 illustrates what happens when VR A sends a message to VR B. (1) The VM on which VR A is running needs to find the IP address and port number of the VM on which VR B is running, so it broadcasts an ARP request. (2) VR B's VM sends a reply with its information. (3) VR A's VM establishes a connection to VR B's VM. (4) This establishes a communication channel between VRs, and finally, the message is encoded and sent.

A special case of sending is when a message is transmitted to another VR on the same VM. In this case, a copy of the specified attribute-value pairs is made, which is then used to construct a `Packet` object. This SFP object is then directly added to the queue of the destination VR.

An `mcast()` function is provided to multicast a message to all VMs. Its semantics are identical to the `send()` function except that specifying a destination is no longer required and has no effect. Currently, the `mcast()` function is only used internally for discovery and address resolution and is not exposed to application VRs. Since there is no destination specified in a multicast message, the message will be placed in the queue of the Control VR on the receiving side.

4.2.2 Receiving Data

The sequence of receiving data can be broken down into the following steps:

1. Receive data from the network (e.g., a socket).
2. Decode the data into SFPs.
3. Add SFPs to the queue of the appropriate VR.
4. Perform matching to return an SFP back to the caller.

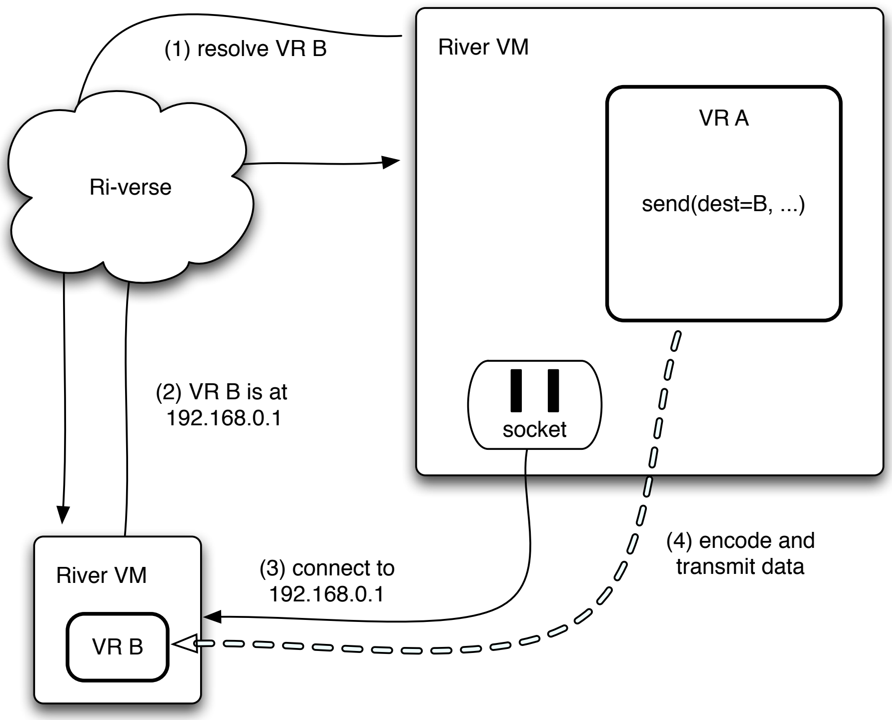
(1) resolve VR B
River VM
VR A
send(dest=B, ...)
Ri-verse
socket
(2) VR B is at 192.168.0.1
(4) encode and transmit data
River VM
(3) connect to 192.168.0.1
VR B

Figure 4.3: Sending Data

Unlike sending, there is no explicit low-level receive function available to read data from a socket and return SFP objects in one fell swoop. Again, this is because the SFM model has no provisions for low-level data transfer details but instead focuses on structured representation. Since data reception is an asynchronous event and may happen at any time, we have a separate network input thread responsible for centrally receiving all data from socket connections. The data is then decoded into SFPs and placed into the message queue of the appropriate VR, as specified by the `dest` attribute within the packet. Thus, the `recv()` function available to application programmers does not invoke any receive-related code within the VM but operates by retrieving matching packets from the queue. In the sequence given above all except the last steps are performed by the network receive thread.

The network receive thread runs an infinite loop that receives and processes network data. We use the `select()` system call to monitor multiple network connections simultaneously. The network receive code is tightly integrated with the connection pool (see Section 4.3) which maintains a set of open connections. The network thread detects incoming connections and adds them to the pool. Likewise, any dropped connections are removed from the pool. If there is pending data waiting on a connection, it is received and decoded into one or more SFPs. If an error occurs during decoding (that is, if the `SFMError` exception is caught), the data is discarded. It is not possible to recover from this condition but we have not seen it in practice.

Once the network thread has an SFP object from the socket, we determine the destination VR to which the packet was sent by using the included `dest` attribute. The packet is then added to the message queue of that VR, and the network thread performs a notify on a condition variable attached to the message queue being modified, in case a VR was blocked on a `recv()` invocation. Figure 4.4 illustrates data reception in River. The network receive thread also processes the registered callback functions, if any. The relevant source code is presented in Appendix B.2.

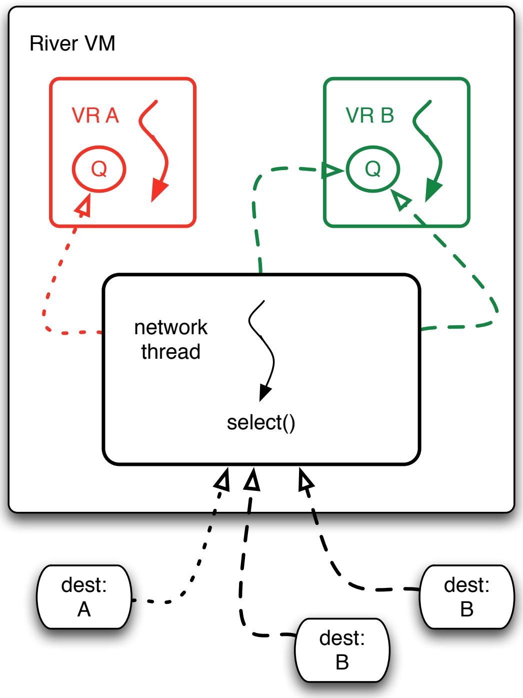

Figure 4.4: Receiving Data

Retrieval of the message by the application VR consists of the SFM queue matching operation that executes within the VR thread and is described in Section 4.1.

4.3 Connection Model and Caching

We have designed a unified connection model for low-level communication between River VMs (and in turn VRs). Currently we only support sockets but our model can be extended to support other APIs and interconnects, such as Inifiniband Verbs API [75] and Myrinet GM [26]. Our model provides a generic net connection object that represents a connection over which data can be sent and received. This connection object provides `send()` and `recv()` methods that allow for low-level data transmission and reception. In the case of sockets these directly map to the socket's `send()` and `recv()` methods, respectively. The connection class provides a uniform interface to both TCP and UDP connections, allowing the VM to treat unicast and multicast connections identically. The source code for our implementation of this model is shown in Appendix B.6.

We allow multiple VMs to run on the same host by using a different TCP port number for every VM. Thus each VM will be associated with a unique tuple containing its IP address and port number. Since all VR communication can be reduced to communication between VMs, we multiplex several logical connections (between VRs) over a single physical bi-directional (socket) connection between VMs. Thus, two distinct VRs communicating with another pair of VRs on a different VM will be using the same communication channel, as shown in Figure 4.5. Here VR A and VR B are executing one application, while VR X and VR Y are running another.

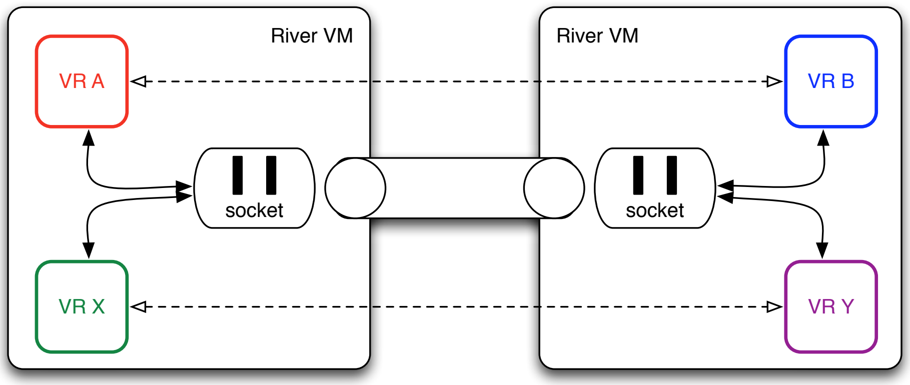

Figure 4.5: Multiplexing Connections

When a new connection object is created, the `__init__()` method of the connection class performs the low-level socket calls necessary to establish a connection between the two hosts. TCP sockets are used for low-level unicast communication between VMs (and in turn VRs), while UDP multicast sockets are used for broadcasting messages to all VMs.

A connection object maintains a receive buffer that stores any unprocessed (leftover) data and automatically prepends it to the new (incoming) data before attempting to decode the buffer into SFPs. Decoding is attempted at every `recv()` call after data has been read from the underlying socket. If no SFP is obtained, an empty list is returned to the caller, or a list of one or more valid SFPs upon success.

To support an arbitrarily large number of *simultaneous* connections we have implemented a caching connection pool. The purpose is two-fold: to allow more connections than the underlying OS would normally support and to avoid setting up a connection between every pair of VRs participating in a computation. The connection pool is integrated with the network connection object interface in such a way that the VM never needs to create connections directly but instead simply requests the necessary connection from the pool. If the pool does not currently have such a connection, a new connection object is implicitly created and returned. The pool object is a dictionary-like data structure that provides easy access using the standard Python dictionary lookup syntax with the [] (bracket) operator where the specified key is a tuple containing the IP address and port number of the VM to which a connection is requested. When the pool reaches its maximum size, connections are expunged on a least recently used (LRU) basis. If the expunged connection is later requested again, it will be re-established and a new connection object for it will be created. By default the cache holds 1024 connections.

Figure 4.6 shows that although every pair of VRs has a logical connection between them, a physical connection is only established when there is actual data being transmitted. The source code for our implementation of the connection pool concept is presented in Appendix B.7.

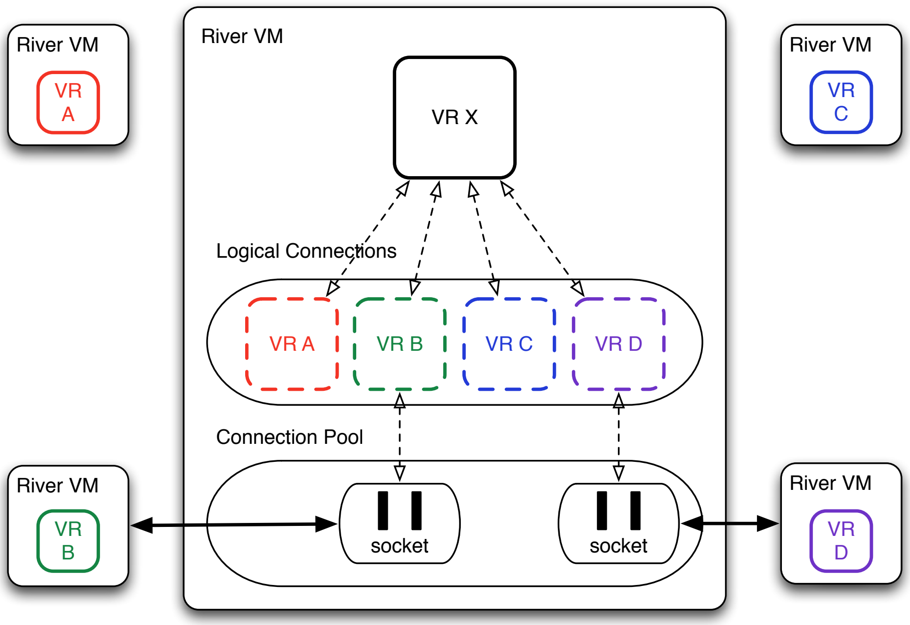

Figure 4.6: Connection Model

4.3.1 Name Resolution

From the programmer's perspective River VRs are identified only by UUIDs. This is done to facilitate VR migration by allowing a VR to move between VMs without disrupting any inter-VR communication. Since a VR keeps its UUID when moving to another VM, communication carries on unaffected. To translate a UUID into a VM address (IP address and port number) we use a mechanism similar to ARP [53] for name resolution.

The VM maintains a cache that matches UUIDs to an (IP, port) tuple. If no entry is found for the requested UUID, the VM broadcasts a UUID resolution request over UDP multicast, which is a special system message to locate the VM on which the VR named by that UUID currently executes. When the target VM receives the request and confirms that it owns the specified VR, it sends out a broadcast reply (also over UDP multicast to avoid infinite name resolution loops) that contains its IP address and port number tuple. The source VM receives the reply message and adds a new entry to the UUID-to-IP cache. When an existing entry is detected to have expired (after 60 seconds by default), it is expunged from the cache and the resolution process is repeated as needed.

Since the request message is sent out within the VR thread, whereas the reply packet is received within the network thread, the former blocks for a minimal amount of time to wait for a reply. If no reply is received, the operation is retried four more times, after which the `send()` operation is aborted and a `RiverError` exception is raised if a UUID does not resolve. See Appendix B.2 for the relevant source code.

4.4 Discovery, Allocation, and Deployment

A special system VR, called the Control VR, is created on every VM during initialization and exists until the VM terminates. Its purpose is to receive and respond to system requests, such as those for discovery, allocation, and deployment. What differentiates the Control VR from other VRs is that the River VM is made aware of it as the VR to whose queue the system messages will be delivered (see Figure 4.7). Our implementation of Control VR is shown in Appendix B.3.

However, other than this special property, in its structure and behavior the Control VR is identical to any other VR. It is responsible for responding to discovery, allocation, and deployment requests as well as creation of new VRs and cleanup of terminated ones. Additionally, the Control VR maintains a map of VM attributes, such as River version, Python version, CPU speed, amount of memory, and others. (See Table 3.3 for a complete list.) Recall from Chapter 3 that before distributed execution can begin, the initiator needs to follow the discover, allocate, deploy procedure. See Appendix B.1 for the source code of our implementation of this procedure.

Discovery is a procedure by which an initiator locates other VMs on the network that are available

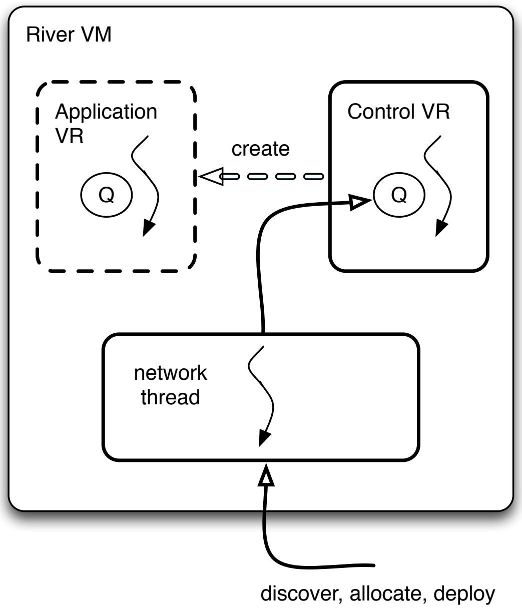

Figure 4.7: Control VR

for execution of River applications. Discovery is typically done at application startup, but it can also be used at any point during application execution to dynamically discover additional VMs. We elected to implement discovery using UDP multicast, rather than relying on a centralized registry, to keep River usage as simple as possible. We also did not want to introduce the added burden on the user to run another daemon that maintains the centralized registry.

To discover available VMs on the network, the initiating VR sends out a **discover** request, which is an SFP that is multicast to all running VMs. Since it is a system message and does not contain a destination UUID, this SFP is added to the message queue of the Control VR on the receiving side. The Control VR will retrieve and process it.

The initiator can choose to discover all VMs or further constrain discovery based on attribute matching. For instance, the default behavior is to discover VMs whose status attribute indicates that the VM is available for use. Alternatively, one may want to only discover VMs with more than, say, one gigabyte of RAM. Discovery attribute matching adheres to the SFM conventions described in Chapter 3, so multiple attributes may be specified. The Control VR on the target VM performs matching and, if successful, replies to the initiator indicating that it is available. If matching is unsuccessful, the discover request is ignored.

Once the initiator obtains a list of VMs matching its search criteria, it needs to allocate them for execution. This is done by issuing an **allocate** request to the discovered VMs. Again, this is received by the Control VR on the target VM. By default the allocation request must come from the same user as the one who owns the target River VM, otherwise the request is ignored. The Control VR generates a new UUID for the VR about to be instantiated and replies to the initiator indicating a success.

Finally, after the initiator receives all the allocation replies, it issues a **deploy** request which contains the module source code, name of the module, function in which to begin execution, and any arguments that are passed to the designated function. The target Control VR verifies the allocation, imports the module, and instantiates a new VR. Since the VR runs in its own thread, it begins execution as soon as the Control VR calls its `start()` method.

The VR module can be imported in several ways. If the module source code was supplied with the deploy request, which is the default behavior, a module is created dynamically. Otherwise, if the initiator exported a local directory (see Section 5.2), the Control VR attempts to retrieve the source code by reading the specified file from the exported directory. Finally, if all else fails, the module is imported from the local file system. If no import is successful, deployment fails, and an error is propagated back to the initiator.

```
>>> class A: pass
>>> class B(A): pass
>>> class C(B): pass
>>> import inspect
>>> inspect.getmro(A)
(<class '__main__.A'>)
>>> inspect.getmro(B)
(<class '__main__.B'>, <class '__main__.A'>)
>>> inspect.getmro(C)
(<class '__main__.C'>, <class '__main__.B'>, <class '__main__.A'>)
>>> rv = [A, B, C]
>>> rv
[<class '__main__.A'>, <class '__main__.B'>, <class '__main__.C'>]
>>> rv.sort(key=lambda x: len(inspect.getmro(x)), reverse=True)
>>> rv
[<class '__main__.C'>, <class '__main__.B'>, <class '__main__.A'>]
```

Figure 4.8: Obtaining Base Class List

It is possible that a module contains more than one subclass of `VirtualResource`, and one of them may be a derivation of another. Thus, we need to select the correct class for instantiation, which will be the most specialized one. To properly make this determination we use the Python `inspect` module [57] to obtain the base classes of each candidate in *method resolution order*. For example, consider three classes: A, B, and C, where B is derived from A, and C is derived from B. Figure 4.8 shows usage of the `getmro()` function from the `inspect` module to obtain the list of base classes of each A, B, and C in the order of the depth of their class hierarchy. We also show how to sort a list of A, B, and C appropriately so that the most specialized class is the first element.

To determine the correct VR class to instantiate from a River module, we iterate through the specified module's dictionary and check each attribute. If the attribute is a subclass of `VirtualResource`, it is added to a list of potential candidates for instantiation. We then sort this list by the depth of each candidate's class hierarchy, given by the length of the `getmro()` function's return value. The list is then returned to the Control VR, which instantiates the most specialized class, given by the first element of the list. Note that if two classes have equal depth of the class hierarchy, the `sort()` function will place only one of them at the head of the list, and that class will be instantiated. In case of multiple inheritance, we rely on the `getmro()` function to provide the correct class hierarchy. In practice this ambiguity should not arise because each VR class is contained in its own file. Figure 4.9 shows the relevant source code.

```
def getVRclasses(module):
    rv = []
    for name in dir(module):
        ref = getattr(module, name)
        try:
            if ref != VirtualResource and issubclass(ref, VirtualResource):
                rv.append(ref)
            except TypeError:
                pass

    rv.sort(key=lambda x: len(inspect.getmro(x)), reverse=True)
    return rv
```

Figure 4.9: Determining the Class to Instantiate

4.5 Fault Tolerance

River was originally conceived with fault tolerance in mind. The Core API and underlying programming model supports execution transparency through soft names via UUIDs and decoupled message queues. This allows VRs to seamlessly migrate between VMs with only a brief pause in execution. The version of the River run-time system described in this thesis does not support checkpointing or migration, however, there exists a branch of the code where this work has been implemented. It utilizes a special version of Python, called Stackless Python [73, 69], to capture the arbitrary run-time state of VRs. Stackless Python removes the use of the C run-time stack from program execution in the Python interpreter. This feature combined with the notion of *tasklets*, allows dynamic program state to be pickled just as any other

Python object instance. This version of River is described in [6].

4.6 Experimental Results

We wrote several micro-benchmarks and applications to measure the performance of River. Our experimental platform consisted of a varying number of identical nodes of a cluster. Each node was equipped with dual Opteron 270 dual-core processors, 4 gigabytes of RAM, and Gigabit NIC, and all were connected to an isolated Gigabit network. For all the experiments we used Python 2.4.3 and executed tests with the -O option (optimization turned on). All the results shown represent an average over three runs.

4.6.1 SFM

To determine the cost of the SFM mechanism we constructed a simple ping-pong benchmark. Using this benchmark we compare performance of different types of mechanisms that could be used to send data between two Python processes running on two different hosts. Note that only the last version of this particular benchmark is a River application; all others are standalone Python programs.

	128 bytes		16 kb		64 kb	
mechanism	lat	bw	lat	bw	lat	bw
sockets	48	20	220	568	642	778
pickle	73	16	275	453	964	518
SFM	84	14	285	439	1005	497
SFM w/Q	97	13	305	410	1031	485
River	222	9	455	276	1080	463

Table 4.1: Ping-Pong Latency (lat, μs) and Bandwidth (bw, Mbps)

Table 4.1 shows the latency and bandwidth[1] of five different implementations of a ping-pong benchmark that transmits a character array of a given size back and forth between two processes. The first version, `sockets`, simply sends the array from sender to receiver and back using the Python interface to the socket library; no encoding of the data is done. The `pickle` version is an example of how a programmer could use the `pickle` module in Python to transfer structured data to a remote machine. It demonstrates the cost of pickling a Python object and sending it over the network. Because we do not know the size of pickled data a priori, we prepend a small header that contains this information to the data. This incurs a performance penalty due to an additional copy before sending. The `SFM` version uses `encode()` and `decode()` functions with the named argument notation provided by the SFM model to serialize the data. The

[1]We define latency as the total time elapsed for the message, and Mbps as 2^{20} bits per second.

`SFM w/Q` implementation adds trivial use of the queue mechanism provided by the SFM model by putting the received packet on the queue and then retrieving it. Because this involves operations on a condition variable within the queue, it results in an additional performance cost. Finally, the `River` version is a full River application that transmits data back and forth between two VRs. Figures 4.10 and 4.11 show graphical representations of our bandwidth and latency results, respectively.

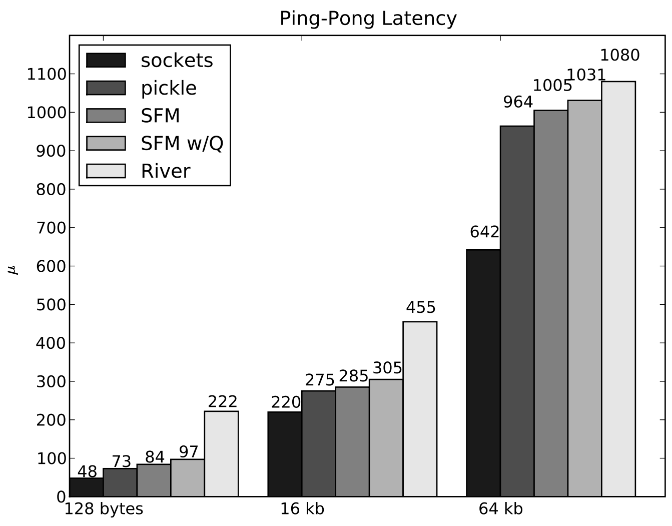

Figure 4.10: Ping-Pong Latency

Both `SFM` versions introduce overhead that results in lower bandwidth and higher latencies but significantly simplify the code that a programmer would normally have to write to send structured data across the network. As discussed in Chapter 3, the SFM model also allows powerful packet matching based on attributes and their values. These results represent the worst-case scenario since a typical distributed program would consist of both communication and computation. Additionally, our queue implementation has not been optimized yet. Finally, the `River` version adds yet more overhead in the form of additional threads. Recall from Section 4.2 that a typical River application involves three threads: the VR thread, the network receive thread, and the Control VR thread. Data is received by the network receive thread but processed by the VR thread. Synchronization between these threads in the form of a condition variable (see Section 4.2) results in additional performance penalty, approximately 30%, compared to `SFM` versions. Using 128 bytes and 16 kb for the buffer size is not sufficient to achieve optimal performance; we get better results with a 64 kb buffer. These findings indicate that any context switches due to a separate network thread and additional Python processing in the critical path of network communication results in significant overhead. For instance, both encoding and decoding functions incur overhead due to copying of buffer data.

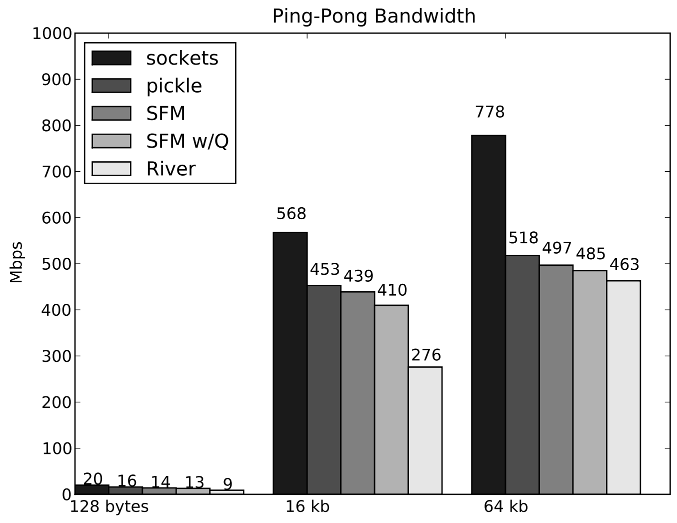

Figure 4.11: Ping-Pong Bandwidth

Also, Python libraries do not provide any scatter/gather I/O operations, which would be beneficial in this case.

4.6.2 River Core

To measure network performance of the River Core, we wrote a River version of the `ttcp` [74] benchmark and compared its results against the standard C version. Additionally, we used an implementation of the Conjugate Gradient (CG) solver to obtain application performance measurements. This section presents our experimental results of these two applications.

4.6.2.1 TTCP

We tested River network performance using the standard `ttcp` [74] benchmark. The C version comes with most Linux distributions and is freely available on the Internet. Additionally, we wrote a River version of the benchmark and a straight Python version. Table 4.2 presents our bandwidth measurements obtained by running each version with various buffer sizes and Figure 4.12 graphs the results. Note that unlike the ping-pong benchmark, which measures the round-trip time of a message, `ttcp` only measures how fast data can be sent out — there is no acknowledgement from the receiver.

It is worthwhile to note that if we use a buffer size of 64 kb, River is able to achieve the same bandwidth as straight Python and even C versions. With smaller buffer size, River performance suffers, again, due to synchronization between multiple threads and extra processing overhead in the critical path of

version	128 bytes	16 kb	64 kb
C (original)	643	896	896
Python	394	896	896
River	21	654	896

Table 4.2: TTCP Bandwidth (Mbps)

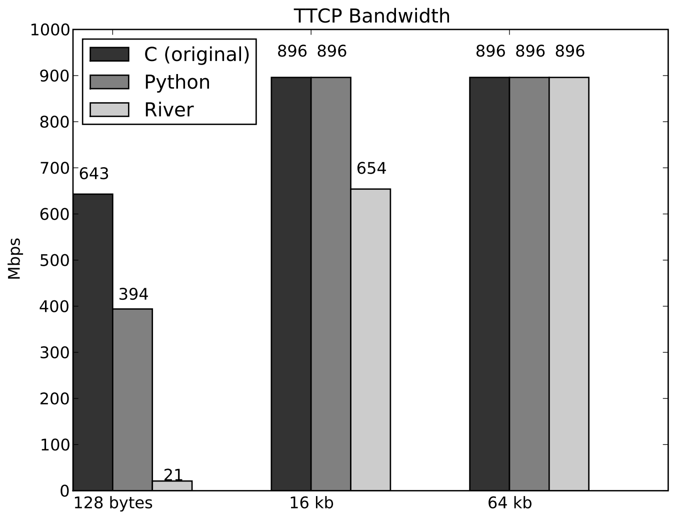

Figure 4.12: TTCP Bandwidth

network communication. With smaller buffer sizes, the cost of processing and synchronization dominates the cost of data transmission.

4.6.2.2 Conjugate Gradient Solver

We also used the conjugate gradient (CG) solver, shown in Appendix A.3, to benchmark River application performance. The CG solver is a fine-grain parallel program and involves substantial communication between consecutive computation steps. We have used a 2048×2048 matrix of floats for input. Our experimental platform was a small cluster of 8 nodes, each with dual Opteron 270 processors and 4 GB of RAM. To obtain the results for 32 VMs we ran four River VMs on each of the eight nodes, and for 16 VMs, two on each of the eight nodes.

The results, given in Table 4.3, show that River can be used to parallelize and speed up straight Python code. Obtaining the best possible performance for an application is not one of our goals, and, needless to say, one would not normally use Python for this purpose. However, we demonstrate that existing Python code can take advantage of River and obtain significant speedup.

Additionally, we wrote a version of the CG solver that uses the Numeric Python library to calculate the most computation-intensive steps: matrix multiplication and dot product. While this dramatically speeds up the overall computation, the communication is still done in Python and thus does not scale well. Performance quickly deteriorates as more VMs are added because communication, done in Python, begins to dominate computation, done in C. Also, this version of CG solver uses our own inefficient collective operations which further inhibit performance. The C MPI version, included for comparison, was compiled with -O2 (optimizations turned on), and used the LAM MPI implementation. All the results are shown in Table 4.3 and graphed in Figure 4.13.

VMs	1	2	4	8	16	32
River time	6375	3389	1704	917	505	443
River speedup	1.00	1.88	3.74	6.95	12.62	14.39
Numeric time	99.60	61.10	53.60	48.19	67.75	120.80
Numeric speedup	1.00	1.63	1.85	2.07	1.47	0.68
C/MPI time	81.95	42.04	24.63	25.32	16.22	11.41
C/MPI speedup	1.00	1.95	3.32	3.23	5.05	7.18

Table 4.3: Conjugate Gradients Solver Results (seconds)

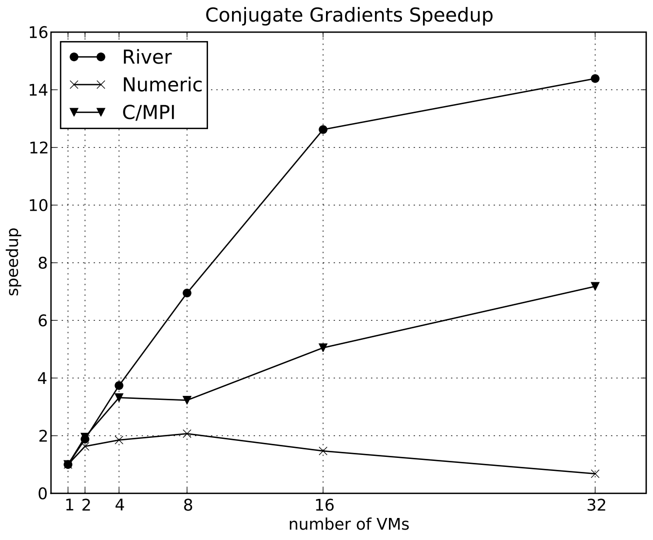

Figure 4.13: Conjugate Gradients Solver Speedup

Chapter 5

Remote Access and Invocation

The River Core interface described in Chapter 3 can be used to not only develop parallel and distributed Python programs but also to implement alternate parallel programming models. We find Python's syntax rich enough so that we can explore different parallel constructs; to demonstrate this we have developed several extensions to River. In this chapter we present the Remote Access and Invocation (RAI) extension to the River Core, which is a generalized RPC/RMI mechanism. In addition to supporting remote procedure call and remote objects, the RAI mechanism also supports remote access to global data. The RAI mechanism is built using the River Core API. We have used the RAI mechanism to support local directory export, a simple form of network file sharing described in Section 5.2, as well as the Trickle task farming extension, described in detail in Chapter 6. Trickle is built directly on top of the RAI mechanism, demonstrating that extensions are stackable and support reuse.

5.1 Programming Model

RAI is a unified interface to establish remote interfaces to the functions and classes from a designated Python module. RAI provides remote data access, remote procedure call, remote object creation, and remote method invocation. The implementation contains two components: a server and a client. The RAI server takes a Python module and serves its contents to the clients. The module can contain arbitrary Python code (classes, functions, data, etc.), rather than a specific River application. The classes and functions provided inside the module will be available for remote instantiation and invocation by the clients. Furthermore, functions and classes may be injected from the clients into the server where they can execute. Figure 5.1 illustrates the RAI concept and capabilities. For example, RAI allows us to (1) invoke a method defined in a remote object, (2) inject locally-defined code into the remote Python interpreter, and (3) use the injected code.

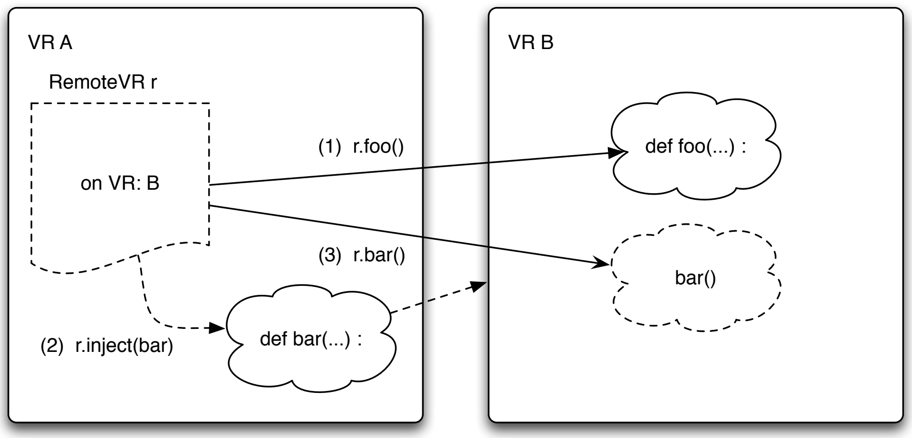

Figure 5.1: Remote Access and Invocation

To execute an RAI server on a machine, it is simply started with the standard River launcher, giving the name of the module, `rai.py`, on the command line as the River program to run. The RAI clients are written as standard River applications and are launched the same way.

Figure 5.2 gives an example of a complete RAI module that can be served. It consists of two functions that we want to make available for remote invocation. When the RAI server starts up, it will automatically set two custom VM attributes: `RAI_module`, the value of which will be the file name of the Python module being served, and `RAI_server`, the value of which is the UUID of the VR running the RAI server.

```
def add(*args):
  return reduce(lambda a,b: a+b, args)

def mult(*args):
  return reduce(lambda a,b: a*b, args)
```

Figure 5.2: Sample RAI Server Module

To access an RAI server's data and methods remotely it is necessary to obtain a handle to the RAI server, represented by the `RemoteVR` object (a special proxy object for remote access). To instantiate such an object, the client first needs to obtain the server's UUID, which appears as a value of the `RAI_server` attribute and thus can be obtained using the standard River discovery mechanism. Once a `RemoteVR` object is instantiated, the data and methods exported by the referenced RAI server can be accessed using the member dot notation.

Figure 5.3 presents the client-side code and shows how to invoke a remote procedure. Note that we only need to perform discovery to find an RAI server; allocation or deployment steps are not needed in this case. Also, we do not use the `vr_init()` method here, but instead perform discovery within `main()`. We discover our specific RAI server by matching on the module name being served (line 7). We then obtain that RAI server's UUID from the VM descriptor on line 12 and create a `RemoteVR` object to represent it on line 13. When creating a `RemoteVR` instance, it is necessary to pass it a reference to `self`, that is, a `VirtualResource` object, to allow remote objects access to the VR's `send()` and `recv()` methods for communication. Finally, we invoke methods through this remote object on lines 16 and 17.

```
 1 from river.core.vr import VirtualResource
 2 from river.extensions.remote import RemoteVR
 3
 4 class RAITester (VirtualResource):
 5     def main(self):
 6         try:
 7             vm = self.discover(RAI_module='samplerai')[0]
 8         except IndexError:
 9             print 'RAI server not found'
10             return -1
11
12         server = vm['RAI_server']
13         r = RemoteVR(server, self)
14
15         args = range(1,10)
16         print '  remote call add', args, r.add(*args)
17         print '  remote call mult', args, r.mult(*args)
```

Figure 5.3: Sample RAI Client

By default the RAI server executes in *restricted* mode, which allows the clients to execute the code and access the data that were specifically provided in the server module, but not anything else. However, it is also possible to execute the RAI server in *unrestricted* mode, which gives clients access to the entire remote Python interpreter and all of its contents. As such, the clients can remotely direct the server-side interpreter to import any standard library module or call any built-in function, which would not be allowed in the default restricted mode.

Unrestricted mode can be invoked by passing a `--unrestricted` command-line argument to the RAI server. Note that no server code is necessary in this case, as all we need on the remote side is a Python interpreter and standard library modules.

Another feature of unrestricted mode is the ability to *inject* code and data into the remote Python interpreter. The injected function can then be invoked in a standard way[1]. Figure 5.4 shows an example of how to access a remote RAI server running in unrestricted mode and how remote and local invocations and arguments to these invocations can be mixed seamlessly.

```
 1 from river.core.vr import VirtualResource
 2 from river.extensions.remote import RemoteVR
 3 import operator
 4
 6 def factorial(x):
 7     if x == 1: return 1
 8     return x * factorial(x-1)
 9
10 class RPCTester (VirtualResource):
11     def main(self):
12         try:
13             vm = self.discover(RAI_module='<unrestricted>')[0]
14         except IndexError:
15             print 'RAI server not found'
16             return -1
17
18         server = vm['RAI_server']
19         r = RemoteVR(server, self)
20
21         r.inject(factorial)
22         print '  remote sum of local range', r.sum(range(10))
23         print '  local sum of remote range', sum(r.range(5))
24         print '  remote factorial', r.factorial(r.sum(r.range(5)))
```

Figure 5.4: RAI Client for an Unrestricted Server

In unrestricted mode the module name in the `RAI` VM attribute is set to `<unrestricted>`. We match on this attribute when performing discovery on line 13. As before, we obtain a reference to the remote VR on line 19. At this point we can access any built-in Python methods and variables by using the dot notation on this remote VR object. On line 21 we inject the local `factorial()` function into the remote interpreter.

Beginning on line 22 we show two examples of mixing remote and local invocations. First, we invoke the built-in `sum()` method remotely and pass it a *local* argument: a list locally generated by the built-in `range()` function. Next, on line 23 we invoke the `sum()` function locally, passing a return value from a remote invocation of `range()` as an argument.

[1]In this prototype of RAI we assume an open network and do not employ strong authentication or provide encryption. However, River and RAI could be deployed on top of a virtual private network (VPN) to achieve stronger security.

Finally, on line 24 we perform several nested remote method calls. First, we remotely call the `range()` function, which returns a list. We then pass this list to the remote `sum()` function, which calculates and returns the sum of all elements. Finally, this value is given as an argument to the injected `factorial()` function that is also invoked remotely.

5.1.1 Passing Remote Object References

With RAI it is possible to pass a reference to a `RemoteObject` to the RAI server or even another VR. For instance, consider the following code that opens a file on the remote side:

```
r = RemoteVR(server, self)
f = r.open('/etc/fstab')

print f.read()
```

In this case a remote reference to a file object is returned by the `open()` function and thus the invocation of `read()` is also remote.

However, if an injected function needs to make use of remote objects, it has no way of accessing them directly, as they do not share the same namespace. (The actual objects on the remote side are kept in the RAI server's namespace to which injected code does not have access.) The solution is to pass a `RemoteObject` reference from the initiator:

```
r = RemoteVR(server, self)
f = r.open('/etc/fstab')

def myread(f):
  return f.read()

r.inject(myread)
print r.myread(f)
```

Note the distinction: in the previous example `read()` is invoked *remotely* by the local code, while in this case it is invoked *locally* by the injected function. The reference to a remote file object gets translated into a local reference on the server.

Additionally, we can send a `RemoteObject` reference to another client VR:

```
self.send(dest=self.peer, file=f)
```

The destination VR can access the received `RemoteObject` in the standard RAI way:

```
p = self.recv(src=self.parent)
f = p.file
print f.read()
```

5.2 Local Directory Export

Local Directory Export (LDE) is a simple RAI module that provides an export of a local directory, allowing remote VRs to access its files. It is very useful in cases where all machines running a River application do not share the same network filesystem but need to access common data. In such cases, the data may be placed on the initiator machine, which exports the data directory using LDE by specifying `--export-dir` option to the River launcher. The River launcher automatically starts the RAI server running the LDE module before starting the main program.

The LDE module provides two functions: `listdir()`, which is a wrapper for the function of the same name in the `os` module, and `open()`, which is a wrapper for the Python built-in function. This enables LDE clients to list the contents of the directory being exported and access remote files for reading and writing.

As before, the VR on the client side needs only to get a `RemoteVR` reference using the standard discover calls, as shown in Figure 5.3, and then proceeds to access the files it needs through that reference. Figure 5.5 demonstrates access to remote files that are exported with LDE. The `discover()` calls are omitted.

```
r = RemoteVR(server, self)

files = r.listdir()
f = r.open(files[0])
data = f.read()
```

Figure 5.5: Using Local Directory Export

5.3 Design and Implementation

The server in execution simply waits for incoming requests from the client, which could be any of the following:

- **inject** Inject the specified source into the server.
- **call** Invoke the specified function with given arguments on the server.
- **getattr** Access a remote object attribute using dot notation.
- **delete** 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. 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. These are usually generated implicitly by the Python interpreter when a local reference is deleted on the client side and allows for objects to be reclaimed by the remote server's garbage collector. The source code to the RAI server is presented in Appendix C.2.

On the client end a special object of type `RemoteObject` is used as a proxy for both the remote VM handles and remote objects. 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 an attribute on the remote object and a request to access it is sent to the server. In this way, RAI can transparently propagate local attribute access and invocations to remote VRs.

All replies to attribute lookups are processed according to their type: scalars are returned directly, objects are saved in a lookup table on the server side and an object id is returned, and any exceptions are propagated back and raised on the client side. We save the object references in a dedicated environment to protect them from the garbage collector, as no other reference to the object will exist on the server otherwise. Functions are treated in the same manner as objects, except that they are not stored in a lookup table on the server.

The RAI client support code creates an instance of the `RemoteObject` class when a reference to an object is returned. The instance contains the object name, its id on the server, and UUID of the remote VR. Functions are returned as callable remote object instances. When invoked as a function, the remote object sends a message to the server, specifying the function name to call, its arguments, and awaits a reply with the return value. Return values from functions are processed in exactly the same way as replies to attribute lookups. Section 5.4 discusses in depth our various implementation attempts of remote functions. See Appendix C.1 for the RAI client source code.

Recall that to send a message from one VR to another, all attributes of the message need to be serialized. To allow for serialization, and thus for transmission, `RemoteObject` proxy implements special

Python methods `__getstate__()` and `__setstate__()`. The former method is invoked at serialization and is responsible for removing any references to a `VirtualResource` object that encapsulates it. Normally, a `RemoteObject` has a reference to its parent VR and the VR's `send()` and `recv()` methods, which it uses for communication. Before serialization, all these references are removed from the object and it is marked as *orphan*. In fact, the only data kept are the object name, object id, and the server UUID, on which the object resides.

After a remote object is transferred and the first time this *orphan* object is used for an attribute lookup[2], RAI code will detect that it is not attached to a parent VR and will locate the parent automatically. This is done with Python introspection using the `inspect` [57] module. First, we obtain the current stack frame with `inspect`. Then, we check the module name of the code in the current stack frame, which is given in the frame's global variable space. As long as the stack frame refers to the code from the RAI module, we travel back up the call stack and obtain the previous stack frame. Finally, the first stack frame outside the RAI code will belong to a `VirtualResource` object, and a local `self` reference will point to the VR itself. We then create a `RemoteVR` instance with this reference and *bind* it to the *orphan* object, which associates the VR's communication methods with the remote object. Figure 5.6 gives the relevant implementation.

```
def __getattr__(self, name):
    if self.orphan:
        frame = inspect.currentframe()
        while frame.f_globals['__name__'] == 'river.extensions.remote':
            frame = frame.f_back

        parent = frame.f_locals['self']

        r = RemoteVR(self.server, parent)
        self.bind(r)
```

Figure 5.6: Locating a Parent for Orphan Remote Objects

Since the actual objects exist on the RAI server, it does not need to follow the same procedure, but instead walks the specified argument list and converts any `RemoteObject` references to its local object references, looked up by object id.

Code injection from the initiator to remote VMs is also achieved with Python introspection. It is possible to obtain the source code for any function or class given a reference using the `inspect` module. 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

[2]In Python every invocation and variable access is preceded by an attribute lookup.

introduce the source into the running VM.

The RAI request invocation handler on the client side divides each request into two parts: sending and receiving. By default the two halves are invoked in sequence. However, such division makes it possible to support asynchronous invocation with the `fork()` method. Figure 5.7 illustrates this idea.

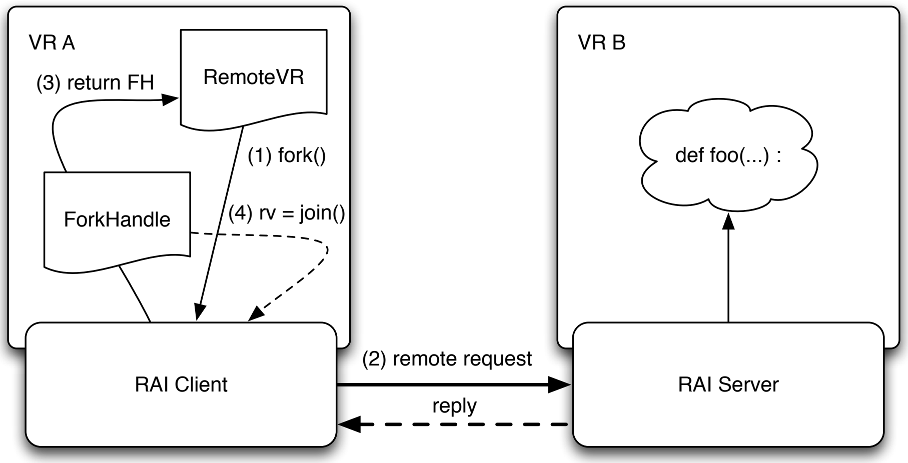

Figure 5.7: Asynchronous Invocation in RAI

When `fork()` is invoked to indicate an asynchronous method invocation, the RAI client sends a remote request to the server and then returns *immediately* without waiting for a reply. A special kind of a remote object is returned: a `ForkHandle` instance. This proxy object keeps track of outstanding remote invocations, and later, the `ForkHandle` instance can be used to join with an outstanding request. Note that the RAI server invokes the method synchronously as soon as the request is received, and sends a reply when the method completes. Since each VR has its own receive queue, the reply from the server can arrive at any time and it will be queued on the requesting VR. The client can then invoke the `join` method of the `ForkHandle` instance, which will wait for a reply and return the result, blocking if necessary. Our Trickle programming model, described in Chapter 6, makes extensive use of this feature. However, all invocations in the base RAI API are synchronous. That is, at every invocation a send is immediately followed by a blocking receive.

5.4 Discussion

Our implementation of remote function calls merits further discussion. In Python a function invocation is preceded by an attribute lookup with the function name as a key. This is translated into a remote request, and the server sends a reply indicating that the given attribute is a function. However, our representation of that function reference on the client side went through several iterations.

Our first initial implementation simply saved the last attribute name looked up. If the result of the lookup was a function, we returned a reference to a special `stub()` method that handled all remote function invocations. Our assumption was that if an attribute lookup is followed by a function call, it is the function just looked up being invoked.

However, this assumption proved flawed in the case of nested function invocations (line 22 of Figure 5.4). In such cases the Python interpreter performs *all* lookups first, and only then invokes the functions. Thus, in our scheme the `range()` function is looked up last, and so all three invocations will be treated as invocations of `range()`.

Our next approach was to still keep our `stub()` method but employ a partial function application using a closure to keep track of function names. Figure 5.8 shows our implementation of this technique.

```
def curry(func, fname, *args0, **kwargs0):
    def callit(*args, **kwargs):
        kwargs.update(kwargs0)
        return func(fname, *(args0+args), **kwargs)
    return callit
```

Figure 5.8: Partial Function Application in RAI

When a remote attribute lookup results in a reference to a remote function, we return a reference to the `stub()` function, embedding the method name as the first argument.

```
return curry(self.stub, fname, *args, **kwargs)
```

This approach solves our nested invocation problem. While it is an acceptable solution, it is needlessly complicated, and requires special treatment of remote function references when they are passed as arguments to other remote methods, for instance, creating a special wrapper object to represent remote references.

Our final solution came from a realization that remote functions are very similar to remote objects: both have a name, an id of the object they belong to, and the UUID of the server on which they reside. Therefore, in the final version of the implementation we return a `RemoteObject` instance both as references to objects *and* functions. A flag, indicating that an object is callable, is added to the `RemoteObject` instance if it represents a function or a method. The `stub()` method within the `RemoteObject` class has been renamed to `__call__()` to allow client code to invoke the object like a regular function[3]. Like the currying

[3]Python uses the existence of such a method within a class as an indication that instances of the class are callable.

solution, this approach complies with Python's nested invocation resolution semantics, but does so in a more Pythonic way.

5.5 Comparison with PYRO

PYRO [20], short for Python Remote Objects, is an advanced and powerful Distributed Object system that provides an object-oriented form of RPC, somewhat similar to Java RMI [34]. PYRO claims to be very easy to use but we feel that River's RAI syntax is more natural and superior in its flexibility and simplicity. To compare the two, we took a simple example from the PYRO documentation and wrote an RAI version of it.

First we define a module to be served to remote clients and name it `testmod.py`. This file is identical, regardless of whether it is served by PYRO or RAI. It provides a class definition with four simple arithmetic methods.

```
class testclass:
    def mul(s, arg1, arg2): return arg1*arg2
    def add(s, arg1, arg2): return arg1+arg2
    def sub(s, arg1, arg2): return arg1-arg2
    def div(s, arg1, arg2): return arg1/arg2
```

PYRO requires a server that will provide access to the module. Note that `testclass` is redefined here to inherit from PYRO's base object. The server registers itself with PYRO's name server that must be running on the network and then enters an infinite loop waiting for client requests.

```
import Pyro.naming
import Pyro.core
from Pyro.errors import PyroError, NamingError

import testmod

class testclass(Pyro.core.ObjBase, testmod.testclass):
    pass

def main():
    Pyro.core.initServer()
    daemon = Pyro.core.Daemon()
    locator = Pyro.naming.NameServerLocator()
    print 'searching for Name Server...'
    ns = locator.getNS()
    daemon.useNameServer(ns)

    try:
        ns.unregister('test')
    except NamingError:
        pass

    daemon.connect(testclass(), 'test')

    print 'Server object "test" ready.'
    daemon.requestLoop()

if __name__=="__main__":
        main()
```

Finally, we need a PYRO client that can make use of the remote object we serve. To find our object we contact the PYRO name server that must be running on the network and obtain the object's URI. Then we get a proxy reference to the remote object and can begin using it.

```
import Pyro.naming, Pyro.core
from Pyro.errors import NamingError

locator = Pyro.naming.NameServerLocator()
print 'Searching Name Server...',
ns = locator.getNS()

print 'finding object'
try:
    URI=ns.resolve('test')
    print 'URI:',URI
except NamingError,x:
    print "Couldn't find object, nameserver says:",x
    raise SystemExit

test = Pyro.core.getProxyForURI(URI)

print test.mul(111,9)
print test.add(100,222)
print test.sub(222,100)
print test.div(2.0,9.0)
print test.mul('*',10)
print test.add('String1','String2')
```

RAI does not require a separate server module; all the necessary code is built into the River framework. A client needs only to locate the necessary server using the standard River discovery mechanism; to do so, we can simply look for the matching name of the module being served. Once we get a handle to the remote VR, we instantiate our `testclass` object as we would in any Python program. Note that this step is somewhat obscure in PYRO: a proxy handle is obtained instead. After that, we call the object's methods just like in PYRO.

```
from river.core.vr import VirtualResource
from river.extensions.remote import RemoteVR

class RAITester (VirtualResource):
    def main(self):
        try:
            vm = self.discover(RAI_module='testmod')[0]
        except IndexError:
            print 'RAI server not found'
            return -1

        server = vm['RAI_server']
        print 'Found RAI server', server

        vr = RemoteVR(server, self)
        test = vr.testclass()

        print test.mul(111,9)
        print test.add(100,222)
        print test.sub(222,100)
        print test.div(2.0,9.0)
        print test.mul('*',10)
        print test.add('String1','String2')
```

In total PYRO requires 3 source files (module, server, client) and 3 processes (name server, server, client). RAI, on the other hand, requires only 2 files (module and client) and runs 2 processes (server and client). No name server is required is used for RAI. We also feel that the RAI syntax is much simpler and natural, providing more flexibility of use.

PYRO	213
RAI	941

Table 5.1: Remote Invocation Cost (μs)

Finally, we measured remote function invocation cost with both PYRO and RAI, shown in Table 5.1. Here PYRO outperforms RAI because it is a single-threaded library, whereas RAI is part of a multi-threaded framework. RAI performance is low due to synchronization between multiple threads (e.g., the network receive thread) and extra processing overhead in the critical path of network communication. Also, recall that at any given time, a River process involves three separate threads: network thread, Control VR thread, and application VR (in this case, RAI server) thread. These results are in line with the ping-pong benchmark results, described in Section 4.6, when run with the smallest buffer size of 128 bytes. The size of data transmitted by RAI in this case is minimal and thus the synchronization overhead dominates. The percent difference between RAI and PYRO performance is approximately equal to the percent difference between the performance of `River` and `sockets` versions of the ping-pong benchmark, as shown in Table 4.1. In a production River system the performance of small messages would need to be addressed.

Chapter 6

The Trickle Programming Model

Trickle is an extension to River that provides a simple programming model to write distributed scripts and programs and brings heterogeneous distributed programming to the Python programming language. The Trickle interface is explicit, but simple; a programmer can easily express parallel execution of coarse-grained work. Following the Python philosophy, Trickle provides a few simple mechanisms that can be learned quickly. At its core, Trickle is a task farming extension to Python, built on top of the RAI mechanism described in Chapter 5, but it also provides a rich distributed object system. Unlike other distributed object systems for Python, such as PYRO [20], Trickle provides a simple namespace that does not require a dedicated name server or an object broker. Trickle can be used to distribute work statically or dynamically. As such, it is ideal for extending an existing Python program to take advantage of unused cycles in a network of machines.

6.1 Programming Model

The Trickle programming model provides a MPMD (multiple program, multiple data) model. It incorporates three basic concepts: injection, remote access, and asynchronous invocation. An initiating Trickle program can inject local functions and classes into remote Trickle VMs, as illustrated in Figure 6.1. Once injected, remote objects can be instantiated, 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 programming model extends the standard Python execution environment by allowing multiple Python interpreters running River VMs to interact in a coordinated manner. A Trickle *initiator* takes care of deploying appropriate code to all the workers. Note that the worker VMs run no specific code; they only run the standard `river` executable. Just like with the River Core, a user will start blank River VMs on each machine that can be involved in a distributed computation. Each Trickle worker VM has access to all resources available to the owner of the River VM process. This means that a Trickle worker can access the local file system or any network-mounted file systems available to the user of the remote VM.

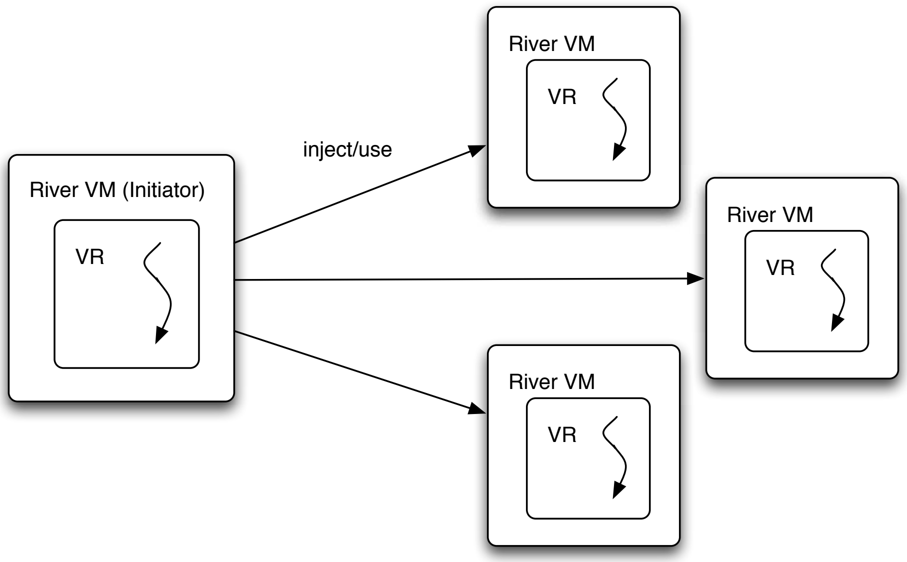

Figure 6.1: Trickle Model

The Trickle initiator, invoked by the `trickle` executable, discovers available VMs and executes the Trickle program passed to it as a command-line argument. The initiator can issue a command to *connect* to remote River VMs. Once connected, the initiator can *inject* data, functions, or classes into remote VMs. Access to remote data or code is achieved through the remote access mechanism, described in Chapter 5. Remote access is seamless, so a remote request appears as a local request. Consider the simple Trickle program presented in Figure 6.2.

This code connects to the available River VMs using the `connect()` function. 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

```
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]
```

Figure 6.2: A Simple Trickle Program

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 RAI (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 [45, 48], in which distributed processes differentiate themselves via a rank value. Also, unlike distributed object systems, Trickle does not require separate client and server code.

6.1.1 Connecting and Injecting

As described previously, a Trickle initiator uses `connect()` to discover remote worker 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] ...)
```

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 Figure 6.3 for different forms of 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)
```

Figure 6.3: Trickle Injection

6.1.2 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.

Figure 6.4 shows how to access remotely injected data. In this case, we have injected a list into

```
1 vmlist = connect(4)
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*']
```

Figure 6.4: Synchronous Data Access

four remote VMs; each VM gets its own copy of the list. We can transparently perform updates on the remote lists (line 5). This also works on remote dictionaries and any remote object that supports the item assignment interface [56].

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). Figure 6.5 shows how to call an injected `factorial()` function and the `range()` built-in function.

Remote object creation and invocation is demonstrated in Figure 6.6. In this case, we are only creating a remote object on a single Trickle VM in line 13. We invoke the remote object just as we would a local object on lines 14-16. A remote object will exist on a remote VM as long as there is a local reference to the proxy object.

```
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
```

Figure 6.5: Synchronous Function 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']
```

Figure 6.6: Synchronous Object Invocation

6.1.3 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. The 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:

```
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. As with remote invocation, any Python data type that can be pickled is a valid argument. When lists are used, the `funclist` and `arglist` must be of equal length; otherwise an exception is generated.

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. Figure 6.7 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.

The `fork()` function does not use local threads to achieve asynchronous invocation, rather, a non-blocking send issues the remote invocation request, as explained in Chapter 5. Thus, each Trickle VM needs only one computation thread.

```
def foo(x):
    return x + 10

def bar(x):
    return x + 20

vmlist = connect()
inject(vmlist, foo, bar)

h0 = fork(vmlist[0], foo, 1); h1 = fork(vmlist[1], bar, 2)
r0 = join(h0); r1 = join(h1)
print r0, r1

hlist = fork(vmlist[0:2], [foo, bar], [1, 2])
rlist = join(hlist)
print rlist

hlist = fork(vmlist, foo, range(len(vmlist)))
print join(hlist)

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

Figure 6.7: Asynchronous Invocation

6.1.4 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 `forkgen()` and `forkwork()` functions to accomplish this task:

```
rlist =  forkgen(vmlist, func, worklist [, chunksize=n])
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 `worklist` parameter may also be a work generator function, provided by the user. The `forkgen()` and `forkwork()` functions will issue work to remote VMs in a dynamic fashion until all the work is completed. 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. The `forkgen()` function is a Python generator and should be invoked in a `for` loop to allow work to be scheduled dynamically. It will return partial results from remote function invocations as they become available. The `forkwork()` function will wait for all invocations to complete and then return a list of results.

Figure 6.8 shows an example of using `forkgen()` to calculate a running sum of an integer array. The array is broken into multiple chunks of work that will be consumed by participating VMs; it is calculated on line 8. Note that the chunk size is independent of the number of VMs. Any remaining work will be assigned to one of the VMs. We invoke `forkgen()` in a `for` loop on line 11, and add up return values as they become available. Two VRs were used in this example and thus two partial sums were returned.

```
 1 def rsum(nums):
 2   return sum(nums)
 3
 4 vmlist = connect()
 5 inject(vmlist, rsum)
 6
 7 a = range(0,1000)
 8 c = len(a) / len(vmlist)
 9
10 total = 0
11 for rv in forkgen(vmlist, rsum, a, c):
12     print 'partial sum =', rv
13     total += rv
14
15 print 'final sum =', total

$ trickle exforkgen.py
partial sum =   124750
partial sum =   374750
final sum =   499500
```

Figure 6.8: Dynamic Work Scheduling

6.2 Word Frequency Example

In this section we present a complete Trickle program. Figure 6.9 is a distributed word frequency counter called WFreq[1]. 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.

```
import sys, 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] + '*')

vmlist = connect(n)
inject(vmlist, wordcount)
rlist = forkwork(vmlist, wordcount, files, chunksize=cs)
final = mergecounts(rlist)
```

Figure 6.9: Document Word Frequency Counter

[1]Refer to Figure 3.10 for this same example written using River Core API.

The `WFreq` program consists of three parts: a worker function, a merge function, and the Trickle code to distribute the work. The worker function `wordcount()` 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.

6.3 Implementation

Trickle is implemented on top of the River Core and uses the RAI mechanism. Remote object access, code injection and asynchronous invocation support is provided by RAI (see Chapter 5). Thus, most Trickle methods are simply wrappers of their respective RAI counterparts. The only component implemented specifically within Trickle is support for dynamic scheduling. The complete source code to Trickle is shown in Appendix C.3.

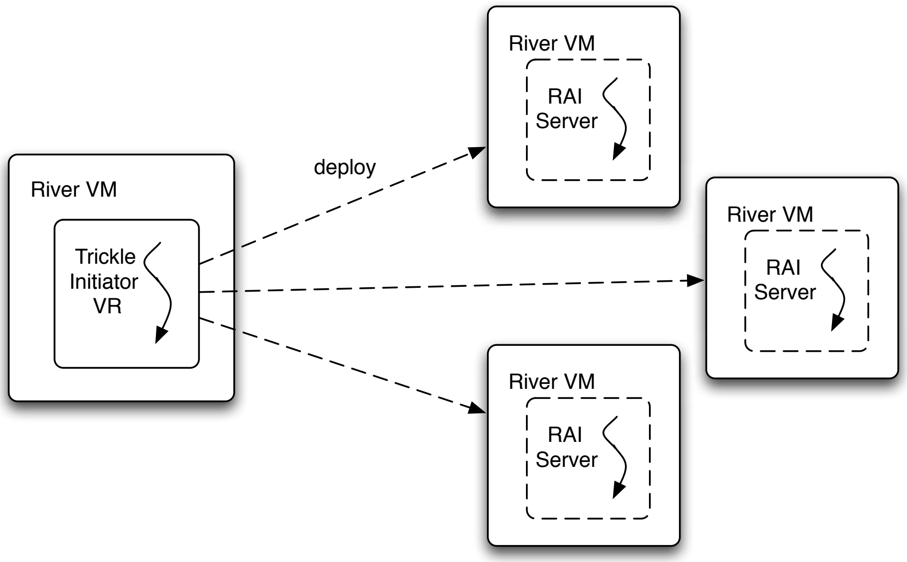

Figure 6.10: Trickle Deployment

6.3.1 Deployment

All Trickle servers start out as blank River VMs, available for use. The Trickle initiator discovers available VMs and deploys the RAI server VR onto them, as shown in Figure 6.10. Trickle configures the

RAI server to expose the standard Python environment, that is, to run in *unrestricted* mode. In this way, once Trickle client code connects to the remote VM, it can access anything in the exported Python environment. To protect the River run-time data, the RAI server creates a separate `globals` and `locals` dictionary for use by injected code. At startup all Python built-in functions are copied to the `globals` dictionary to allow remote access to them.

The Trickle initiator class is a subclass of `VirtualResource`. After deploying the RAI server to available VMs, it saves a list of them internally. This list is used by the `connect()` function to create `RemoteVR` objects corresponding to all participating RAI servers. This list of `RemoveVR` objects is returned by the `connect()` function to the user. The Trickle user code is not imported, since it is not part of any object. Instead we use the Python `execfile()` function to execute it within the context of the Trickle initiator VR.

As described in Section 5.3, `RemoteVR` and its parent class, `RemoteObject`, are special proxy objects used for both the remote VM handles and remote objects. In this way, Trickle can transparently propagate local invocations to remote VMs. The Trickle client code acquires VM proxy handles with the `connect()` function, and then injects or remotely invokes code on them.

Aside from `forkgen()` and `forkwork()`, all other Trickle functions are simply wrappers for their RAI counterparts with special argument handling to allow lists of VM handles and functions to be passed in.

6.3.2 Dynamic Scheduling

The Trickle dynamic scheduling mechanism, `forkwork()` is built using basic asynchronous invocation and Python generators. Internally, `forkwork()` calls a generator called `forkgen()`, as shown in Figure 6.11.

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

Figure 6.11: Implementation of `forkwork()`

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. An abbreviated version, without the support for `chunksize`, is given in Figure 6.12.

```
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
```

Figure 6.12: Implementation of `forkgen()`

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.

6.4 Discussion

Trickle requires injection to be performed explicitly by the user. For instance, a function cannot be passed to `fork()` and invoked remotely unless it is explicitly injected first. There are advantages and disadvantages to this approach. On one hand, a programmer would have to write less code and not worry about injecting if it was done implicitly by Trickle. On the other hand, our programming model is very explicit, and we require the programmer to perform that step for consistency.

The RAI implementation currently does not support propagating remote object references. For example, the Trickle initiator code may pass a remote object reference from one server to a function invoked on another server. While we allow remote objects to be passed as arguments to functions invoked on the *same* server, there is no code to handle external remote objects on the server side.

Likewise, we do not allow callbacks to the initiator — in case a remotely invoked function needs to make use of objects located on the initiator. The reason behind this is that we maintain a strict client-server separation: the Trickle initiator (RAI client) executes client code only, and the Trickle VMs (RAI servers) execute server code only. To support such a mechanism, we would need to allow the Trickle VMs to function as RAI clients, and the Trickle initiator, as an RAI server. This would introduce a synchronization requirement to limit concurrent access to shared data and thus complicate the programming model.

6.5 Comparison with IPython

IPython [50] attempts to create a comprehensive environment for interactive and exploratory computing. It also provides a sophisticated and powerful architecture for distributed computing that abstracts out parallelism in a very general way enabling many paradigms, such as SPMD, MPMD, or task farming. As such, we think it worthwhile to compare IPython with its River counterpart, Trickle.

To compare Trickle with IPython we have written an IPython version of the Word Frequency benchmark, described in Section 6.2. The Trickle version is shown in Appendix A.6 and the IPython version, in Appendix A.7.

The IPython interface works through an object of type `MultiEngineClient`. First, we need to import the IPython module and instantiate a `MultiEngineClient` object. This takes care of discovering and allocating the available hosts, in River terminology. Then we use the `get_ids()` function, which serves as an equivalent of Trickle's `connect()` method: it returns a list of IDs of machines that can be used for parallel computation.

```
from IPython.kernel import client

mec = client.MultiEngineClient()

ids = mec.get_ids()
```

With Trickle we can obtain the same list by calling the `connect()` function. No import is necessary because the Trickle initiator already populates the global namespace with the appropriate method references.

```
vrlist = connect()
```

IPython uses the ”push” functions to put data on the remote side. Unlike Trickle, different functions are used depending on the type of data. For instance, adding a function to the remote side requires the invocation of the `push_function()` method. In this case, we are pushing a function called `wordcount()` to the remote clients. Note that the syntax is reminiscent of the SFM syntax: a dictionary of key-value pairs.

```
mec.push_function(dict(wordcount=wordcount))
```

This concept is called injection in Trickle. However, Trickle does not require a special method for function injection; any type of data can be passed to `inject()`.

```
[vr.inject(wordcount) for vr in vrlist]
```

To put data on the remote hosts with IPython we use the `push()` function.

```
[mec.push(dict(files=slices[i]), targets=i) for i in ids]
```

With Trickle we do not need to explicitly inject data if it will be passed in as a function argument; we simply use it, as we would in a normal Python program.

Finally, to execute the function with the correct arguments in IPython, we need to pass the appropriate Python statement *as a string* to the `execute()` function. All variables and methods referenced by the statement need to exist on the remote side. In this case, we have previously pushed the `wordcount()` function and the `files` list.

```
mec.execute('freqs = wordfreq(files)')
```

With Trickle, we invoke remote functions in a standard Python way and pass in local arguments; they will be automatically transmitted to the remote side. Note that we use asynchronous execution with `fork()` here.

```
handles = [vr.fork(wordcount, slices[vr]) for vr in vrlist]
```

Finally, we collect the results in the form of partial dictionaries from each node. With IPython, we need to pull the data from the remote side, again, by supplying the variable name as a string.

```
rvlist = mec.pull('freqs')
```

With Trickle, we just use the return value as returned by the function. Note that since we use asynchronous execution here, we collect the return values with the `join()` function.

```
rvlist = [h.join() for h in handles]
```

6.6 Experimental Results

We used a version of the word frequency program, presented in Section 6.2 as a benchmark to evaluate Trickle performance with a partial Enron Email Dataset [22] given as input. The dataset consisted of 55,366 files for a total of 277 MB in size. Three versions of the benchmark were used: a River Core version, listed in Appendix A.2, a Trickle version, listed in Appendix A.6, and an IPython version, listed in Appendix A.7.

We ran all versions of the `wfreq` program on a small cluster to see what kind of speedup is possible. Our experimental platform consisted of a varying number of identical nodes of a cluster. Each node was equipped with dual Opteron 270 dual-core processors, 4 gigabytes of RAM, and Gigabit NIC, and all were connected to an isolated Gigabit network. For all the experiments we used Python 2.4.3 and executed tests with the -O option (optimization turned on). To obtain results for 32 VMs we ran four River VMs on each of the eight nodes. Each node was given a complete copy of the data set.

We have compared the performance of the IPython version and the River/Trickle versions using the same dataset. The River/Trickle versions performed significantly faster, regardless of the number of VMs participating. Table 6.1 shows the results of running on 1, 2, 4, 8, 16, and 32 VMs. Figures 6.13 and 6.14 show the graphs representing both absolute performance and speedup of the `wfreq` benchmark. The results shown are an average over three runs. As can be seen in the results, the serial portions of the code limit the speedup in the case of River and Trickle. Furthermore, the Trickle version is slightly slower compared to River, as it incurs additional overhead from the RAI layer. However, these results show that it is relatively easy to leverage multiple machines with Trickle. Additionally, both River and Trickle versions significantly outperform IPython.

VMs	1	2	4	8	16	32
River Core time	33.88	18.82	11.83	9.12	8.38	8.84
River Core speedup	1.00	1.80	2.86	3.71	4.04	3.83
Trickle time	33.61	19.54	12.54	9.52	9.35	10.67
Trickle speedup	1.00	1.72	2.68	3.53	3.59	3.15
IPython time	46.70	36.51	37.14	42.13	57.83	83.60
IPython speedup	1.00	1.28	1.25	1.11	0.80	0.56

Table 6.1: Word Frequency Results (seconds)

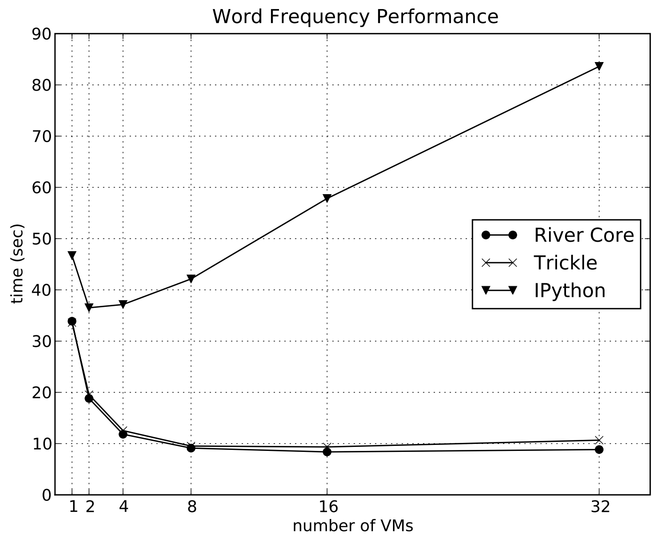

Figure 6.13: Word Frequency Performance

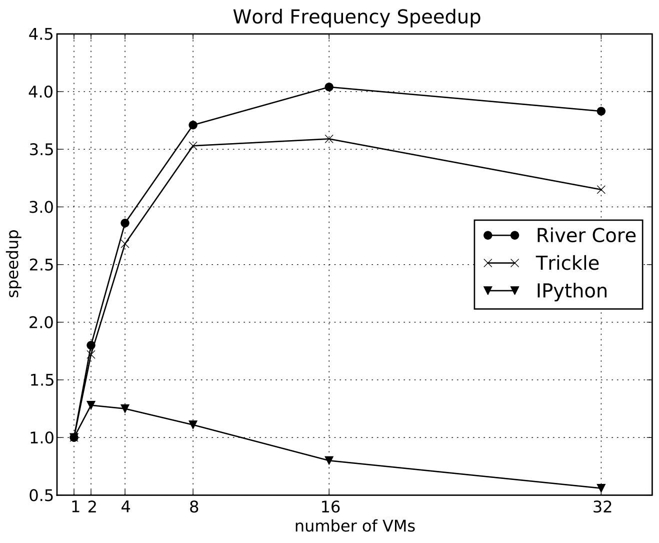

Figure 6.14: Word Frequency Speedup

6.7 Trickle Quick Reference

Trickle extends River with a small number of global built-in functions. A new VM object is used as a proxy to access remote elements. A quick reference for the Trickle interface is given in Figure 6.15.

```
# Starting a Trickle program
$ trickle foo.py

# Connecting to Trickle VMs
vmlist = connect([n])

# Injecting objects into a vm or vmlist
# obj can be any type: immutable, mutable, function, class
inject(<vm>|<vmlist>, <obj0> [, <obj1>] ...])

# Synchronous remote access
y = vm.x                   # get remote value of x
vm.d[x] = y                # update a remote list/dict
y = vm.foo(*args)          # invoke remote function
f = vm.FooClass(*args)     # instantiate a remote object
y = f.bar(x)               # invoke remote method

# Asynchronous remote access (parallel invocation)
<h>|<hlist> = fork(<vm>|<vmlist>, <func|funclist>,
                  (<arg0> [, <arg1> ] ...]) | <arglist>)
<r>|<rlist> = join(<h>|<hlist>)
<r>         = joinany(<h>|<hlist>)
<r>|<rlist> = forkjoin(<vm>|<vmlist>, <func|funclist>,
                      (<arg0> [, <arg1>,] ...]) | <arglist>)
<rlist>     = forkwork(<vm>|<vmlist>, <func>, <worklist>
                       [, chunksize=<n> ])
<r>         = forkgen(<vmlist>, <func>, <worklist>
                       [, chunksize=<n> ])
```

Figure 6.15: Trickle Quick Reference

Chapter 7

River MPI Implementation

We have developed a River-based MPI [45, 48] implementation, called rMPI, for the rapid prototyping and analysis of MPI functionality. Our base implementation of the MPI 1.2 standard is an extension to the River Core and lends itself to quick understanding and modification of point-to-point communication, collective communication, and derived data types. The rMPI interface conforms closely to the C MPI interface and can be used to learn about MPI implementation and explore techniques that would otherwise be too cumbersome in existing production systems. The rMPI extension is a good example of using River for understanding the design details of an existing interface and to explore alternative implementations.

7.1 Design

To further demonstrate the capabilities of the River framework and to support rapid prototyping and easier understanding of MPI implementations we have developed a River extension called rMPI. Unlike previous approaches to provide a Python interface to MPI libraries [36, 42, 46], rMPI is an implementation of MPI written *entirely* in Python; it does not depend on an underlying C MPI implementation.

We have focused on making rMPI as useful as possible by following two important guidelines: (1) provide a Python MPI interface that strictly conforms to the C MPI interface; and (2) create a small, modular implementation that supports selective inclusion of MPI functionality as needed (e.g., a derived data type engine or non-blocking communication). These guidelines have several desirable consequences.

Close conformance to the C MPI interface enables programmers to easily get started with rMPI by following documentation for the C bindings. Thus, translating MPI programs from C to Python and vice versa is easy and straightforward. This fluidity makes it possible to leverage Python for rapid application development and debugging, then migrate to C for performance. Interestingly, our Python MPI interface conforms more strictly to the MPI standard than popular Python wrappers for the C MPI implementations [36, 42, 46]. Unlike existing Python wrappers for MPI, we do not try to make MPI more Python-like. Instead, we make Python more C-like.

The modularity found in rMPI facilitates rapid development and comparison of multiple algorithms for the same functionality. Therefore, rMPI can be used to quantitatively and qualitatively evaluate implementation techniques. Such evaluation can ultimately be used to guide the evolution of C MPI implementations or aid the development of application-specific MPI libraries. Because our implementation is quite small (the base implementation is 810 lines of Python code), it does not take much effort to begin making changes and experimenting with new ideas. We allow additional functionality, such as derived data types and optimized collective communication, to be added as needed.

Our experience with using Python and rMPI for systems development has been quite refreshing when compared to the conventional approach using C and standard C tools. In contrast to a C MPI 1.2 implementation, such as USFMPI [12], the developer does not become tied to the design decisions – in the case of C, making core changes would often require a significant amount of software engineering. With rMPI, it is relatively easy to go from an idea to a working implementation. Furthermore, the simplicity of the rMPI framework allows one to try several different implementation strategies.

7.2 Programming Model

The rMPI interface requires that all names from the `mpi.py` are imported into the target application. Also, each rMPI program must define a new class that subclasses `MPI`. By convention, execution of each rMPI process begins at `main()`. This mimics C convention, including command-line argument passing. Figure 7.1 shows code for a simple rMPI program.

In C MPI many arguments are supplied as pointers so that variables can be updated within the MPI calls (e.g., `MPI_Comm_rank(MPI_COMM_WORLD, &rank)`). Python does not have an ”address of” (&) operator. Therefore, in rMPI we introduce a wrapper type called `MPI_Int`. New variables can be instantiated as `x = MPI_Int()`. Such references can be used to allow MPI functions to modify the object value. Access to the object value is achieved via the `value` member: `x.value`. This simple Python class allows us to simulate pass by pointer semantics, which allows us to better adhere to the C MPI interface.

An important design decision in the rMPI interface is how to treat send and receive buffers. In C, a pointer to a region of memory is passed to the MPI communication routines. To retain the order and type of parameters from the C MPI interface, we decided to require that send and receive buffers be simple Python lists. We chose to use lists, rather than arrays [55], because of their simpler and more natural syntax. In Python, lists are passed by reference, so a list reference can serve as a ”pointer” to a buffer. One consequence of this scheme is that receive buffers must be preallocated. The rMPI routines do not dynamically allocate

```
from mpi import *

class Hello(MPI):
    def main(self, args):
        MPI_Init(args)
        rank = MPI_Int()
        np = MPI_Int()
        MPI_Comm_rank(MPI_COMM_WORLD, rank)
        MPI_Comm_size(MPI_COMM_WORLD, np)
        status = MPI_Status()

        recvbuf = [ 0.0 ] * 4
        sendbuf = [ rank.value * 100.0 ] * 4

        if rank.value == 0:
            for i in xrange(1, np.value):
                MPI_Recv(recvbuf, 4, MPI_FLOAT, i, 0, MPI_COMM_WORLD, status)
                print 'From rank %d:' % (i), recvbuf
        else:
            MPI_Send(sendbuf, 4, MPI_FLOAT, 0, 0, MPI_COMM_WORLD)
        MPI_Finalize()
```

Figure 7.1: A Simple rMPI Program

space for incoming data. Again, this mimics the C MPI interface and also helps avoid excess copying. The `Hello` example in Figure 7.1 sends a list of 4 elements from each non-rank 0 process to rank 0. The receive buffer in rank 0 is reused for each message.

Using a simple Python list reference works in many situations. However, a common idiom in C MPI is to pass a pointer that addresses a location within an allocated memory region. This is especially useful in conjunction with derived data types in order to relocate data, e.g., transposing a row to a column in a matrix. To support this idiom in Python, we allow buffer references to be tuples of a list reference and an offset into the list: `(buf, offset)`. This notation, while not strictly C MPI notation, allows us to retain the number and type of the parameters to the MPI functions while supporting C-like semantics. For example, say you have a list with 100 doubles. You can receive 10 doubles starting at offset 50 like this:

```
recvbuf = [ 0.0 ] * 100
MPI_Recv((recvbuf, 50), 10, MPI_DOUBLE, src, tag, comm, status)
```

The base rMPI implementation supports blocking send (`MPI_Send()`) and receive (`MPI_Recv()`).

The base can be extended to include non-blocking communication and corresponding support functions (e.g., `MPI_Isend()`, `MPI_Irecv()`, `MPI_Wait()`, etc.). In addition, the base implementation includes naive implementations of all the major collective communication functions. Most of the implementations have linear time complexity in the number of processes. Such implementations keep the base relatively small and easier to understand initially. However, we provide an extension built on top of the non-blocking extension that provides optimized implementations of all of the collectives. We have implemented most of the algorithms presented by Thakur *et al* in [72].

We provide a fairly complete derived data type engine for rMPI programs as an extension to the base implementation. For example, the following code transposes an $n \times n$ matrix using point-to-point communication and derived data types. Note that the matrix A is simply a 1-dimensional Python list that is used to store a 2-dimensional array.

```
A = [0.0] * (n * n)

column_t = MPI_Datatype()
MPI_Type_vector(n, 1, n, MPI_DOUBLE, column_t)
MPI_Type_commit(column_t)

# send columns
for i in xrange(n):
    MPI_Send((A,i), 1, column_t, rank, tag, MPI_COMM_WORLD)

# receive rows
for i in xrange(n):
    MPI_Recv((A,i*n), n, MPI_DOUBLE, rank, tag, MPI_COMM_WORLD, None)
```

7.3 Implementation

Rather than create a single monolithic implementation of the rMPI Python interface, we provide a modular framework in which MPI functionality can be included selectively. We provide three main building blocks: a derived data type engine, non-blocking communication support, and optimized collective communication operations. The purpose of structuring rMPI in this manner is two-fold: to simplify the code for the novice and to encourage experimentation in the major submodules.

These modules are provided as derivations of the `MPI` base class. User programs, in turn, are inherited from these derived classes. Since Python supports multiple inheritance, a user program may specify two different submodules as its base classes to include support for, say, both derived datatypes and non-blocking communication.

The complete rMPI source code is included in Appendix C.

7.3.1 Base Implementation

The base rMPI implementation consists of blocking point-to-point operations and non-optimized collectives. Using River, the base supports process discovery and the mapping of River UUIDs to MPI ranks. It provides basic support for MPI communicators and groups and can redirect console output from remote rMPI processes to the initiating process. In addition, it can display Python exceptions on the initiator so that errors in remote processes can be detected. Finally, the base implementation supports both eager and synchronized sends. The source code of the base implementation is shown in Appendix C.4.

Using Super Flexible Messaging (SFM) as the basic communication mechanism facilitates rapid development. Since messages are dynamically typed, there is no need to specify separate message types (as would be done in a C implementation); everything is specified at the call site of a send. Figures 7.2 and 7.3 show how SFM is used to implement flow control in `mpi_send_buf()` and `mpi_recv_buf()`, respectively, which are the basic building blocks for all rMPI communication routines. The code fragment in Figure 7.2 allows large messages to be sent in chunks so that a complete copy of a data buffer is not needed. The SFM notation makes it relatively easy to implement a chunk protocol and ensure that data is transferred in the proper order.

All the major MPI peer-to-peer and collective functions are implemented using `mpi_send_buf()` and `mpi_recv_buf()`. They allow for the sending of small messages in an eager fashion or large messages in a sequence of chunks. The parameters include: `buf` (a list of simple values), `offset` (index into `buf`), `count` (the number of values to send), `datatype` (the data type of the values in `buf`), `src` (the rank of the sender), `dst` (the rank of the receiver), `tag` (an MPI idiom for naming transfers), and `comm` (the communication namespace).

The small message code sends the entire `buf` in a single message. By setting up the proper values for `start`, `end`, and `size`, as well as setting `last` to `True`, the receiver is given all the information needed to process the send request. The large message code walks through the `buf` in `chunksize` increments. On each iteration, the values for `start`, `end`, and `size` are updated appropriately. Once the last chunk is identified, `last` is set to `True`.

```
def mpi_send_buf(self, buf, offset, count, datatype, dest, tag, comm):
    destUuid = comm.rankToUuid(dest)
    chunksize = self.mpi_getchunksize(datatype, count)
    offset = 0

    if isinstance(buf, tuple):
        buf, offset = buf

    sendsize = count
    if sendsize <= chunksize:  # small send (eager)
        self.send(dest=destUuid, tag=tag, buf=buf[offset:offset+count],
                  start=0, end=count, size=count, mtype='small', last=True)

    else:                      # send large buf
        last = False; start = offset; end = start + chunksize

        while True:
            self.send(dest=destUuid, tag=tag, buf=buf[start:end],
                      start=start-offset, end=end-offset, size=chunksize,
                      mtype='large', last=last)
            m = self.recv(src=destUuid, tag=tag, ack=True)

            if last:
                break

            start += chunksize
            end = start + chunksize

            if (start + chunksize) >= (offset + count):
                end = offset + count
                chunksize = end - start
                last = True

    return MPI_SUCCESS
```

Figure 7.2: Sending Data in rMPI

```
def mpi_recv_buf(self, buf, offset, count, datatype, src, tag, comm):
    srcUuid = comm.rankToUuid(src)

    while True:
        m = self.recv(src=srcUuid, tag=tag)
        if m.mtype == 'large':
            self.send(dest=srcUuid, tag=tag, ack=True)

        buf[m.start+offset:m.end+offset] = m.buf[0:m.size]
        if m.last:
            break

    return MPI_SUCCESS
```

Figure 7.3: Receiving Data in rMPI

7.3.2 Derived Datatypes

Our derived datatype engine supports the most common MPI types, including `contiguous`, `vector`, and `indexed`. The implementation is a seamless extension to the base provided as another class, called `MPI_DD`, derived from the `MPI` base class. User programs that need to make use of derived datatypes inherit their class from `MPI_DD` instead of `MPI` to include support for derived datatypes. The source code of the derived datatypes support is presented in Appendix C.5.

Upon new datatype initialization, we create a map representing the memory regions occupied by the new type and store it as a collection of *chunks*. For instance, a `contiguous` type would be composed of a single chunk of size *n*. By contrast, a simple `vector` type would be composed of *n* chunks of size 1. The send mechanism transparently extracts these chunks from a derived datatype instance at transmission. The current implementation uses a *packing* scheme to construct on-the-wire data. However, the engine could be modified to support more sophisticated on-the-wire formats (e.g., to avoid copying).

Figure 7.4 shows our implementation of the `vector` datatype. If the old type is a base type, we simply create a chunk for every element. However, if the old type is derived and we are creating a recursively derived datatype, we need to copy the chunks from the old type, making sure to calculate proper offsets. For instance, if we have a matrix, we can create a *column* type, and then recursively another type representing *every other column* of the matrix.

As we add a new chunk to a datatype map, we go through the list of existing chunks and check if the new chunk can be combined with an existing one. This is done to minimize the total number of chunks. Moreover, at *commit* time (in the `MPI_Type_commit()` function), we iterate through the list again and attempt to combine contiguous chunks. This is useful when, say, a matrix type is derived from a column type, and results in a single chunk of memory describing the matrix.

```
def MPI_Type_vector(self, count, block_length, stride, old_type, new_type):
    # old type must exist (either basic or derived)
    if not mpi_types.has_key(old_type):
        return MPI_ERR_ARG

    # derived type: each element may have multiple chunks
    if isinstance(old_type, MPI_Datatype):
        n = block_length * old_type.size
        w = block_length * old_type.width
        for i in xrange(count):
            offset = i * stride * old_type.width
            for j in xrange(block_length):
                new_type.copy_chunks(old_type, offset + j)

    # base type: each element is its own chunk
    else:
        n = block_length
        w = block_length
        for i in xrange(count):
            offset = i * stride
            new_type.add_chunk(offset, block_length)

    new_type.count = len(new_type.chunks)
    new_type.size = n * count
    new_type.width = w

    # "register" the new type
    mpi_types[new_type] = mpi_robj.next()

    return MPI_SUCCESS
```

Figure 7.4: Implementation of the Vector Derived Datatype

7.3.3 Non-blocking Communication

Simple blocking sends and receives are relatively easy to understand. However, production MPI implementations support non-blocking communication. Such support can significantly complicate matters. We provide it as an extension (another derivation of the `MPI` base class) so that it can be studied and evaluated independently. Our code turns `MPI_Isend()` and `MPI_Irecv()` calls into `TransferRequests`. Only a *post* is transmitted, but the actual communication takes place when a transfer request is processed: at any point a blocking call is made (e.g., `MPI_Wait()`). We create an instance of a *transfer manager* class, which keeps track of outstanding requests and processes them as needed. This is similar to how non-blocking communication is implemented in production single-threaded MPI systems. To achieve true parallelism, a communication thread may be introduced, as done in USFMPI [12]. Figure 7.5 shows our request processing function. The complete source code of non-blocking communication support is included in Appendix C.6.

7.3.4 Optimized Collectives

Due to the requirements of optimized collectives, they are built on top of non-blocking communication. Thus, it is derived from the `MPI_NB` class, rather than the base `MPI` class. We have implemented many of the algorithms from [72] as well as dissemination allgather [7]. Exploring and evaluating optimized collectives in rMPI is quite easy since Python provides a more compact notation that is more pseudo-code-like when compared to C. For example, we have implemented the Bruck allgather algorithm, shown in Figure 7.6. The source code of the optimized collectives implementation is shown in Appendix C.7.

```
def trans_process_request(self, request):
    if request.trans_status == TRANS_STAT_DONE:
        self.transManager.unregister(request)
        return

    # a confirmed send request and the first chunk is not sent yet
    if request.type == MPI_REQ_SEND and \
            request.trans_status == TRANS_STAT_WORK and \
            request.trans_chunknum == 0:
        self.trans_send_chunk(request)

    while (request.trans_status != TRANS_STAT_DONE):
        p = self.recv(dest=self.uuid,
                      mtype=(lambda x: x==’sendpost’ or x==’recvpost’ or \
                                       x==’send’ or x==’recv’))
        srcRank = self.uuidToRank[p.src]
        if p.mtype == "sendpost" :
            recvreq = self.transManager.recvready(srcRank, p.tag, p.sender)

        elif p.mtype == "recvpost":
            sendreq = self.transManager.sendready(srcRank, p.tag, p.recver)
            # if sendreq is not registered yet,
            # this post will be save in transManager.recvqueue
            if sendreq != None:
                self.trans_send_chunk(sendreq)

        elif p.mtype == "send":
            req = self.transManager.getreq(p.recver)
            self.trans_recv_packet(p, req)

        elif p.mtype == "recv":
            sendreq = self.transManager.getreq(p.sender)
            self.trans_send_chunk(sendreq)

    self.transManager.unregister(request)
```

Figure 7.5: Non-blocking Request Processing Function in rMPI

```
def MPI_AG_Bruck(self, sendbuf, sendcount, sendtype, recvbuf, recvcount,
                 recvtype, comm):
    msgUid = self.getMsgUid()
    size = comm.size()
    power2 = math.floor(math.log(size, 2))
    steps = int(math.ceil(math.log(size, 2)))
    status = [MPI_Status(), MPI_Status()]
    recvbuf[0:sendcount] = sendbuf
    sendIndex = 0

    for step in xrange(steps):
        recvIndex = 2**step * sendcount

        if (step == power2):
            sendcnt= (size - 2**step) * sendcount
        else:
            sendcnt = 2**step * sendcount

        destNode = (self.rank.value - 2**step) % size
        srcNode = (self.rank.value + 2**step) % size
        send_req = MPI_Request(); recv_req = MPI_Request()

        MPI_Isend((recvbuf,sendIndex), sendcnt, sendtype, destNode, msgUid,
                   comm, send_req)
        MPI_Irecv((recvbuf, recvIndex), sendcnt, recvtype, srcNode, msgUid,
                   comm, recv_req)
        MPI_Waitall(2, [send_req, recv_req], status)

    if self.rank.value > 0:  # reorder
        for i in xrange(size - self.rank.value):
            recvbuf.extend(recvbuf[:sendcount])
            del recvbuf[:sendcount]

    return MPI_SUCCESS
```

Figure 7.6: The Bruck Allgather Algorithm in rMPI

7.4 Comparison with other Python MPI packages

There are several MPI packages available for Python; all of them are Python bindings to a C/MPI library, rather than an implementation from scratch like rMPI. The most popular ones are pyMPI [42], MYMPI [36], and PyPar [46]. We examine all three packages and compare them to rMPI. Of the three, MYMPI adheres most closely to the MPI standard, however, we believe that rMPI is superior in this regard. We found all three packages lacking in the functionality they provide: MYMPI is again the most complete, implementing 30 of the 120+ MPI calls, but still lacks such features as collective communication operations. For comparison, rMPI implements 48 MPI calls, including non-blocking and collective communication operations.

The pyMPI package is a special version of the Python interpreter along with a module. Programs written in pyMPI do not call `MPI_Init()`. Instead, it is called implicitly when the interpreter is launched, somewhat breaking the MPI standard semantics. In contrast, MYMPI and PyPar are only modules that can be used with a standard Python interpreter. In MYMPI, programs explicitly call `MPI_Init()`. PyPar resembles the C MPI interface the least and also invokes `MPI_Init()` implicitly.

There is a fundamental difference in the semantics of pyMPI and the other two packages. While pyMPI as the interpreter is a parallel application, with MYMPI and PyPar, the code executed by the standard interpreter is the parallel application. MYMPI and PyPar are much smaller since they are composed of only one Python module, not a full interpreter.

MYMPI claims to match more closely the syntax of C and Fortran than pyMPI and PyPar. MYMPI arguments are explicit but many conventions are broken. For example, buffer references are returned by `MPI_Recv()` and other communication functions instead of error codes. With pyMPI many arguments, such as communicator and status, are implicit and omitted. We believe that rMPI is superior in adherence to the true MPI semantics and syntax, when compared to the other MPI wrappers.

To compare, let us examine the invocation of `MPI_Recv` in all the packages. The C prototype and invocations in pyMPI, PyPar, MYMPI, and rMPI are shown in Table 7.1.

With pyMPI most arguments are implicit and are, in fact, omitted from the function calls. For instance, the communicator is always assumed to be `MPI_COMM_WORLD` and it is not possible to specify a different one. As such, the only parameter passed in is the rank of source from which to receive. The return value is the buffer where the received data is stored. There is no support for specifying tags or custom communicators or datatypes. Thus, there is no support for derived datatypes either. Likewise, PyPar lacks the support for all three. Again, it is only possible to specify the source rank and the return value is the buffer where data is stored.

MYMPI allows the programmer to pass in a datatype as well as a tag and a communicator. It

Package	Semantics
C MPI	`int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status);`
pyMPI	`buf = mpi.recv(source)`
PyPar	`buf = pypar.receive(source)`
MYMPI	`buffer = mpi.mpi_recv(count, datatype, source, tag, communicator)`
rMPI	`rv = self.MPI_Recv(buf, count, datatype, source, tag communicator, status)`

Table 7.1: `MPI_Recv()` Semantics in Various MPI Implementations

should be noted, however, that only basic types, such as an integer or a double, are supported. The return value is once again a buffer reference, not an error code. This changes the order of arguments, since `buffer` needs to be specified as the first argument. Also, the last argument, `status`, is completely omitted.

On the other hand, rMPI preserves all arguments and their order to simplify porting existing code to and from C. An object of type `MPI_Status()` is used in places where a pointer to `status` is passed in, as described in Section 7.2, or `None` can be specified in place of `NULL`. Finally, rMPI provides considerable support for derived datatypes, as presented in Section 7.3.2.

7.5 Experimental Results

We have compared the performance of rMPI with MYMPI, as well as C/MPI, using our Conjugate Gradient (CG) benchmark. We ran all versions of the CG solver on a small cluster to see what kind of speedup is possible. Our experimental platform consisted of a varying number of identical nodes of a cluster. Each node was equipped with dual Opteron 270 dual-core processors, 4 gigabytes of RAM, and Gigabit NIC, and all were connected to an isolated Gigabit network. For all the experiments we used Python 2.4.3 and executed tests with the `-O` option (optimization turned on). The C/MPI version was compiled with `-O2`. To obtain results for 32 VMs we ran four instances of the program on each of the eight nodes.

We have implemented a parallel conjugate gradient (CG) solver in both rMPI and MYMPI, compiled to use the LAM MPI library. We chose to use MYMPI in our testing, rather than pyMPI or PyPar, because it was the most complete and stable implementation, and most closely followed the MPI standard. Additionally, we provide the performance of a C/MPI CG solver and the River Core version for reference. A matrix size of 2048×2048 doubles was used as input. Table 7.2 presents the performance and speedup of all four versions. Figure 7.7 shows a graph of the absolute performance, and Figure 7.8, of the speedups. The results shown are an average over three runs.

It is worth noting that the absolute performance of MYMPI and rMPI is approximately the same.

VMs	1	2	4	8	16	32
rMPI time	6371	3433	1757	959	569	491
rMPI speedup	1.00	1.86	3.63	6.64	11.20	12.98
River time	6375	3389	1704	917	505	443
River speedup	1.00	1.88	3.74	6.95	12.62	14.39
MYMPI time	6322	3776	1860	962	498	273
MYMPI speedup	1.00	1.67	3.40	6.57	12.69	23.16
C/MPI time	81.95	42.04	24.63	25.32	16.22	11.41
C/MPI speedup	1.00	1.95	3.32	3.23	5.05	7.18

Table 7.2: rMPI Conjugate Gradients Solver Results (seconds)

This is to be expected, since the computation is done in Python, even though MYMPI uses the C MPI implementation for communication, whereas rMPI does everything in Python. As the number of VMs increases, so does the ratio of communication to computation. Thus, we see decreased performance of rMPI compared to MYMPI in the case of 32 VMs. When compared with the C version, the rMPI implementation appears to exhibit better speedup. However, this is because in rMPI computation in Python dominates the cost of communication. In C, the cost of communication begins to dominate much sooner. The absolute performance of rMPI and MYMPI for this problem is about 75 times slower than the C MPI implementation on average.

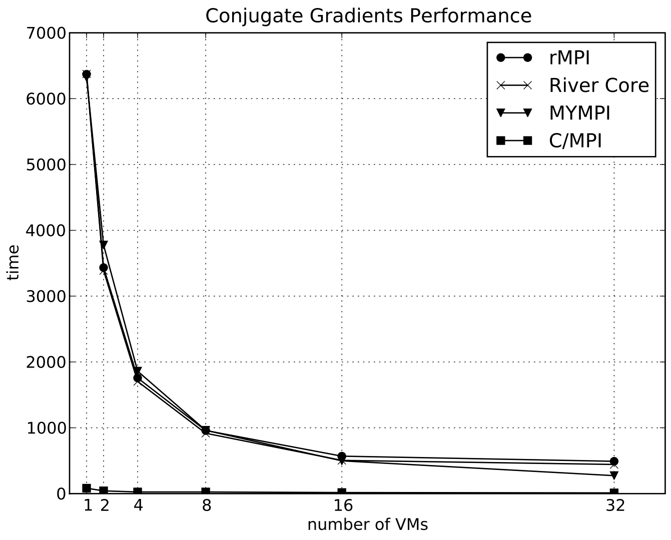

Figure 7.7: rMPI Conjugate Gradients Solver Performance

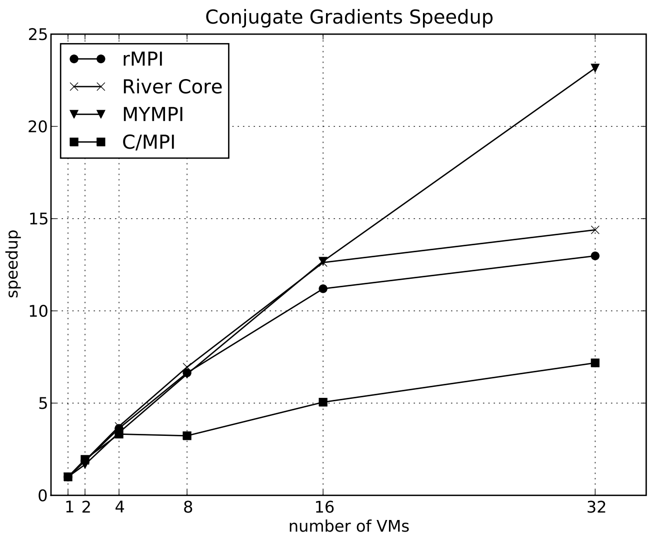

Figure 7.8: rMPI Conjugate Gradients Solver Speedup

Chapter 8

Conclusions and Future Work

This chapter presents some concluding remarks, reflections on our experience with River development, and offers directions for future work.

8.1 Experience with River Development

River has been a very enjoyable project to work on. It is our sincere hope that it will continue its existence beyond this thesis. It has already been used successfully in graduate parallel computing classes at USF. Trickle, for example, together with the Python Imaging Library [52], has been used for distributed image processing and movie creation. We would like to open up the whole River framework and its components to further developments by releasing it as an open source project. To that extent, we have made the distribution freely available on our website [66] and several people have already expressed interest in integrating it with their projects.

Compared to previous parallel and distributed systems, River is unique in its ability to allow a programmer to quickly move from a new idea to a working prototype. A negative consequence of prototyping with conventional languages is that the programmer becomes tied to the design decisions at an early stage because making changes to the core of a run-time system can be a substantial engineering effort. River, by contrast, was specifically designed to allow considerable changes and additions with a minimal amount of work. The simplicity and elegance of the River core combined with Python's dynamic typing and concise notation make it easy to rapidly develop a variety of parallel run-time systems. Largely, this is a testament to Python's flexibility and ease of use.

The choice of using Python as the implementation language was probably most instrumental to River's success. Certainly, Python has its own disadvantages for systems development, most notable of which is performance. Yet, in our opinion, the flexibility and ease of modification that Python allows far outweigh the performance drawbacks. On several occasions, we were able to move from a discussion of a feature or modification to its implementation within minutes, practically modifying River ”on the fly!”

The single most important piece of the River framework is undoubtedly the Super Flexible Messaging (SFM) mechanism. It allowed us to transmit arbitrary data without spending any time on protocol design. Again, this would not be possible without language support. When designing the SFM framework itself, we wanted an efficient send/receive mechanism so we decided to build our underlying protocol on top of sockets. Prior experience in implementing MPI from scratch [12] led us to focus on developing a fixed message structure at first. However, we did not yet know what we wanted in the message structure and concluded that a lazy approach would be best to encourage rapid development, so we decided that a message could hold anything. Python’s flexibility and ease of use has allowed to us to experiment with different approaches in a short amount of time. River attempts to conform to the same ease-of-use principles. For example, the SFM concept has been extremely successful in facilitating rapid development of both distributed run-time systems (RAI, Trickle, rMPI) and applications.

When running our experiments, we noticed that our simple discovery mechanism was not robust enough in its original implementation to reliably locate more than 12-16 VMs. Python’s and River’s flexibility allowed us to easily modify our code within minutes to make discovery more robust by sending out more than one discover request. In the end, we were able to reliably discover 96 River VMs running on 24 nodes on a gigabit network with just five discover requests.

Such flexibility of the underlying run-time system certainly lends itself to River’s usage for education in parallel programming classes, where students can learn about implementing a parallel system and quickly move from an idea to experimentation with an existing system without having to design something from scratch. River’s design has been an ongoing process that happened in parallel with its implementation and we believe it is well thought-out and easy to understand.

Additionally, the code base is relatively small; the River core is under 3000 lines of code and under 6000 lines with all the extensions, including rMPI. Table 8.1 shows the lines of code counts of River components. A similar project done in C would consist of several thousand lines of code.

River Core	2744
RAI	535
Trickle	352
rMPI	1987

Table 8.1: Lines of Code of River Components

8.2 Future Work

While River is certainly usable in its present state, it would greatly benefit from additional features. To that extent, we think there are several directions worth exploring.

While we were able to make our discovery mechanism more robust by sending out multiple requests, this solution will not scale well. A better solution would be some sort of a tree broadcast, or the election of a ”master” VM to keep track of VMs coming and leaving and maintain a list that would then be provided to initiators upon request, such as the *group membership protocol*, described in [9]. This would avoid the delay currently present during the discovery time and potentially scale to hundreds of nodes.

The SFM model is an excellent approach to specifying and transmitting arbitrary data but additional constructs to express various conditions would be beneficial. For example, there is no way to apply logical constraints across multiple message fields. Recall from Chapter 3 that SFM allows us to match a message based on multiple values of an attribute using a `lambda` function.

```
m = recv(tag=lambda x: x == 23 or x == 42)
```

However, this only allows matching on a subset of *one* attribute; there is currently no way to specify that one *or* another attribute should be matched. For example, if we want to match a message based on its source *or* its tag, the current semantics do not give us a way to accomplish that. One possibility would be to augment the SFM semantics and introduce a reserved keyword argument, such as `OR`, whose value would be a tuple of possible attribute matches.

```
m = recv(OR=(tag=42, src=A))
```

This would provide logical OR semantics across multiple message attributes. Specifying additional attributes in the `recv()` function call would continue serving as a logical AND condition, as it does currently.

River would also benefit from additional support for fault tolerance. As discussed in Section 4.5, there exists a version of the framework built on top of Stackless Python [73, 69] and described in [6]. However, that branch of the code needs to be completed and rigorously tested. Once completed, it will provide state management for River applications, allowing them to be checkpointed or even seamlessly moved to another VM.

To improve performance of River applications, there could be a River/Python to C translator. It could be more specific and target, for instance, MPI applications to facilitate going from rMPI to C/MPI.

Another idea would be to rewrite the River Core in RPython [54] (a restricted subset of Python that can be easily translated to C). A River Core engine with SFM support, written in C, would also be beneficial. Implementing SFM in C would require additional support for our extended syntax, however.

To better support multi-core and multi-processor machines, River could be implemented on top of the new `multiprocessing` Python module [58]. The `multiprocessing` module is a package that supports spawning processes using an API similar to the `threading` module, currently used in River. It effectively uses subprocesses, instead of threads, thereby avoiding performance drawbacks associated with the Python thread model. Queues or pipes are used for synchronization and communication between processes. Since the data will be located on the same machine, no network communication is necessary and this will simplify the River code and improve performance. Furthermore, another possibility is to use the `multiprocessing` module to completely eliminate threads from River and replace them with subprocesses. Thus, there would be a separate subprocess running the network receive function and yet another one running the Control VR. Communication between them can be achieved using shared memory, queues, and pipes. In many cases (refer to the source code in Appendix B.2), the threads currently synchronize using a local socket, so the change to subprocesses will not affect the code much. The biggest modification would involve rewriting our `Queue` implementation to support subprocesses instead of threads.

Additional performance improvements can come from using the new Python Buffer API [60] for communication. This API can be used by an object to expose its data in a raw, byte-oriented format. Clients of the object can use the buffer interface to access the object data directly, without needing to copy it first. Avoiding the extra copy would speed up the communication framework in River and all its extensions.

As discussed in Chapter 4, we assume a secure network and thus do very little to guarantee data security and integrity. River would certainly benefit from a more stringent security model, perhaps with underlying support for secure transport mechanisms, such as SSL.

Finally, River can be used as the underlying engine in a cloud system. Projects like Google's App Engine [2] present a very restricted Python environment for serving web pages and web applications. River can be adapted to serve the purpose of virtualizing clusters in the Cloud and presenting a platform for distributed cloud computing.

Bibliography

[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] Google App Engine. `http://code.google.com/appengine`.

[3] 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.

[4] 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.

[5] David M. Beazley. *Python Essential Reference*. Sams Publishing, third edition, 2006.

[6] G. D. Benson. State Management for Distributed Python Applications. In *Proceedings of the 13th IEEE Workshop on Dependable Parallel Distributed and Network-Centric Systems (DPDNS08)*, Miami, Florida, USA, April 2008.

[7] G. D. Benson, C. Chu, Q. Huang, and S. G. Caglar. A comparison of MPICH allgather algorithms on switched networks. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 10th European PVM/MPI Users' Group Meeting, Venice, Italy, September 29 - October 2, 2003, Proceedings*, volume 2840 of *Lecture Notes in Computer Science*, pages 335–343. Springer, 2003.

[8] G. D. Benson and A. S. Fedosov. Python-based distributed programming with Trickle. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, USA, June 2007. CSREA Press.

[9] Kenneth P. Birman. *Reliable Distributed Systems: Technologies, Web Services, and Applications*. Springer, first edition, 2005.

[10] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984.

[11] Ralph Butler, William Gropp, and Ewing Lusk. Components and interfaces of a process management system for parallel programs. *Parallel Computing*, 27(11):1417–1429, October 2001.

[12] S. G. Caglar, G. D. Benson, Q. Huang, and C. Chu. USFMPI: A multi-threaded implementation of MPI for linux clusters. In *Proceedings of the 15th IASTED International Conference on Parallel and Distributed Computing and Systems*, Marina del Rey, CA, December 2003.

[13] David Callahan, Bradford L. Chamberlain, and Hans P. Zima. The cascade high productivity language. In *of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments, HIPS, Santa Fe, NM, USA*, pages 52–60. IEEE Computer Society, April 2005.

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

[15] Nicholas Carriero, David Gelernter, and Jerrold Leichter. Distributed data structures in linda. In *POPL '86: Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 236–242, New York, NY, USA, 1986. ACM.

[16] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, 2007.

[17] The Cascade high-productivity language. `http://chapel.cs.washington.edu/`.

[18] Philippe Charles, Christian Grothoff, Vijay A. Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, San Diego, CA, USA*, pages 519–538. ACM, October 2005.

[19] Common Object Request Broker Architecture. `http://en.wikipedia.org/wiki/CORBA`.

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

[21] HPCwire Editor. The search for a new HPC language. *HPCwire, August 25, 2006*. `http://www.hpcwire.com/hpc/837711.html`.

[22] Enron Email Dataset. `http://www.cs.cmu.edu/~enron/`.

[23] 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.

[24] Flashmob Computing. `http://www.flashmobcomputing.org`.

[25] The Fortress programming language. `http://research.sun.com/projects/plrg/`.

[26] GM: A message-passing system for Myrinet networks. `http://www.myri.com/scs/GM-2/doc/html/`.

[27] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[28] Per Brinch Hansen. Distributed processes: a concurrent programming concept. *Commun. ACM*, 21(11):934–941, 1978.

[29] Gerald Hilderink, Jan Broenink, Wiek Vervoort, and Andre Bakkers. Communicating Java Threads. In A. Bakkers, editor, *Parallel Programming and Java, Proceedings of WoTUG 20*, volume 50, pages 48–76, University of Twente, Netherlands, 1997. IOS Press, Netherlands.

[30] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[31] Cay S. Horstmann and Gary Cornell. *Core Java 2, Volume II: Advanced Features*. Prentice-Hall PTR, seventh edition, 2004.

[32] Matthew Izatt, Patrick Chan, and Tim Brecht. Ajents: towards an environment for parallel, distributed and mobile java applications. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 15–24, New York, NY, USA, 1999. ACM.

[33] Java Message Service Documentation. `http://java.sun.com/products/jms/docs.html`.

[34] Java remote method invocation. `http://java.sun.com/j2se/1.5.0/docs/guide/rmi/index.html`.

[35] 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.

[36] 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.

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

[38] Aaron W. Keen, Tingjian Ge, Justin T. Maris, and Ronald A. Olsson. Jr: Flexible distributed programming in an extended java. *ACM Trans. Program. Lang. Syst.*, 26(3):578–608, 2004.

[39] Paul J. Leach, Michael Mealling, and Richard Salz. A universally unique IDentifier (UUID) URN namespace. Internet proposed standard RFC 4122, July 2005.

[40] Ewing Lusk and HPCwire Editor. HPCS languages move forward. *HPCwire, August 25, 2006*. `http://www.hpcwire.com/hpc/827250.html`.

[41] David May. Occam. *SIGPLAN Not.*, 18(4):69–79, 1983.

[42] Patrick Miller. PyMPI: Putting the py in MPI, 2007. `http://pympi.sourceforge.net/`.

[43] A Conversation with Gordon Moore. `ftp://download.intel.com/museum/Moores_Law/Video-Transcripts/Excepts_A_Conversation_with_Gordon_Moore.pdf`.

[44] Gordon E. Moore. Cramming more components onto integrated circuits. pages 56–59, 2000.

[45] Message Passing Interface (MPI) Forum home page. `http://mpi-forum.org/`.

[46] Ole Nielsen. Pypar: Parallel programming in the spirit of Python, 2007. `http://datamining.anu.edu.au/~ole/pypar/`.

[47] NumPy. `http://numpy.scipy.org`.

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

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

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

[51] Michael Philippsen and Matthias Zenger. JavaParty: Transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997.

[52] Python Imaging Library. `http://www.pythonware.com/products/pil/`.

[53] D. C. Plummer. RFC 826: Ethernet Address Resolution Protocol: Or converting network protocol addresses to 48.bit Ethernet address for transmission on Ethernet hardware, November 1982. Status: STANDARD.

[54] PyPy, a flexible and fast Python implementation. `http://codespeak.net/pypy`.

[55] array – Efficient arrays of numeric values. `http://www.python.org/doc/2.4.3/lib/module-array.html`.

[56] Emulating container types. `http://www.python.org/doc/2.4.3/ref/sequence-types.html`.

[57] inspect module – Inspect live objects. `http://www.python.org/doc/2.4.3/lib/module-inspect.html`.

[58] multiprocessing – Process-based "threading" interface. `http://docs.python.org/library/multiprocessing.html`.

[59] pickle module – Python object serialization. `http://www.python.org/doc/2.4.3/lib/module-pickle.html`.

[60] Python Buffer Objects. `http://docs.python.org/c-api/buffer.html`.

[61] Python data model. `http://docs.python.org/reference/datamodel.html`.

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

[63] Python Tutorial: More control flow tools. `http://docs.python.org/tutorial/controlflow.html`.

[64] socket – Low-level networking interface. `http://www.python.org/doc/2.4.3/lib/module-socket.html`.

[65] xmlrpclib – XML-RPC client access. `http://www.python.org/doc/2.4.3/lib/module-xmlrpclib.html`.

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

[67] Vijay A. Saraswat and Radha Jagadeesan. Concurrent clustered programming. In *Concurrency Theory, 16th International Conference, CONCUR, San Francisco, CA, USA*, volume 3653 of *Lecture Notes in Computer Science*, pages 353–367. Springer, August 2005.

[68] Abraham Silberschatz. On the synchronization mechanism of the ada language. *SIGPLAN Not.*, 16(2):96–103, 1981.

[69] Stackless Python. `http://www.stackless.com`.

[70] W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff. *UNIX network programming: The Sockets Networking API*, volume 1. Prentice-Hall PTR, third edition, 2004.

[71] Robert E. Strom and Shaula Yemini. Nil: An integrated language and system for distributed programming. In *SIGPLAN '83: Proceedings of the 1983 ACM SIGPLAN symposium on Programming language issues in software systems*, pages 73–82, New York, NY, USA, 1983. ACM.

[72] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in MPICH. *International Journal on High Performance Computing Applications*, 19(1):49–66, 1999.

[73] Christian Tismer. Continuations and Stackless Python. In *Proceedings of the Eighth International Python Conference*, 2000. `http://www.python.org/workshops/2000-01/proceedings/papers/tismers/spcpaper.htm`.

[74] Test TCP. `http://sd.wareonearth.com/~phil/net/ttcp/`.

[75] Mellanox IB-Verbs API (VAPI). `http://www-linux.gsi.de/~mbs/fair/docs/MellanoxVerbsAPI.pdf`.

[76] Andrew Wilkinson. PyLinda: Distributed computing made easy, 2007. `http://www-users.cs.york.ac.uk/~aw/pylinda`.

[77] The X10 programming language. `http://www.research.ibm.com/x10`.

Appendix A

Examples Source Code

This appendix presents the complete source code to sample River programs from Chapter 3.

A.1 Monte Carlo Pi Calculator [mcpi.py]

```
import math, random, time, socket, sys
from river.core.vr import VirtualResource

class MCpi (VirtualResource):
    def __init__(self, **kwargs):
        VirtualResource.__init__(self, **kwargs)
        self.host = socket.gethostname()

    def vr_init(self):
        if len(self.args) < 2:
            print 'usage: %s <n>' % self.args[0]
            return False

        print '%s > Deploying MCpi' % self.host
        VMs = self.allocate(self.discover())
        for vm in VMs:
            print '  %s: %s %s %s %s %s %s' % (vm['host'], vm['version'],
                vm['os'], vm['arch'], vm['pyver'], vm['status'], vm['user'])

        VMs = self.deploy(VMs=VMs, module=self.__module__, func='slave')

        self.slaves = [vm['uuid'] for vm in VMs]

        return True

    def master(self, n):
        print '%s > master' % self.host, self.uuid
        print '%s > slaves:' % self.host, self.slaves
        numslaves = len(self.slaves)

        print '%s > sending work to %d slaves' % (self.host, numslaves)
        for uuid in self.slaves:
            self.send(dest=uuid, tag='n', n=n/numslaves)

        in_circle = 0
        print '%s > receiving %d results' % (self.host, numslaves)
        while numslaves > 0:
            p = self.recv(tag='result')
            in_circle += p.result
            numslaves -= 1

        pi = 4.0 * in_circle / n
        PI = 4.0 * math.atan(1.0)
        print 'PI = %f, our approximation = %f' % (PI, pi)

    def slave(self):
        print '%s > slave' % self.host, self.uuid
        master = self.parent

        print '%s > waiting for work' % self.host
        p = self.recv(src=master, tag='n')
        print '%s > got work: N=%d' % (self.host, p.n)

        in_circle = 0
        for i in range(0, p.n):
            x = 2.0 * random.random() - 1.0
            y = 2.0 * random.random() - 1.0
            length_sqr = x * x +  y * y
            if length_sqr <= 1.0: in_circle += 1

        print '%s > sending results' % self.host
        self.send(dest=master, result=in_circle, tag='result')

    def main(self):
        print 'MCpi master/slave version starting'
        print '  my UUID:', self.uuid
        print '  args:   ', self.args

        if len(self.slaves) < 1:
            print 'ERROR: need at least one slave'
            return -1

        stime = time.time()
        self.master(int(self.args[1]))
        ftime = time.time()
        print '%s>   %s seconds' % (self.host, ftime-stime)
```

A.2 Word Frequency Counter [wfreq.py]

```
import time, socket
from river.core.vr import VirtualResource

class WFreq (VirtualResource):
    def __init__(self, **kwargs):
        VirtualResource.__init__(self, **kwargs)
        self.host = socket.gethostname()

    def vr_init(self):
        VMs = self.allocate(self.discover())
        VMs = self.deploy(VMs=VMs, module=self.__module__, func='worker')

        self.workers = [vm['uuid'] for vm in VMs]

        return True

    def worker(self):
        p = self.recv(src=self.parent, tag='files')

        m = {}
        for file in p.files:
            self.wordcount(file, m)

        count = 0
        for w in m: count += m[w]

        print '%s> local count %d' % (self.host, count)

        self.send(dest=self.parent, tag='result', result=m)

    def wordcount(self, filename, m):
        print '%s> %s' % (self.host, filename)
        f = open(filename)
        for l in f:
            tokens = l.split()
            for t in tokens:
                    m[t] = m.get(t, 0) + 1
        f.close()

    def main(self):
        if len(self.args) < 2:
            print 'usage: %s <files>' % self.args[0]
            return

        files = self.args[1:]
        slices = {}
        size = len(self.workers)
        slice = len(files) / size
        extras = len(files) % size
        end = 0
        for worker in self.workers:
            beg = end
            end = beg+slice
            if extras:
                end += 1
                extras -= 1
            slices[worker] = files[beg:end]

        stime = time.time()

        for worker in self.workers:
            self.send(dest=worker, tag='files', files=slices[worker])

        m = {}
        for worker in self.workers:
            p = self.recv(tag='result')
            rm = p.result
            for w in rm:
                    m[w] = m.get(w, 0) + rm.get(w, 0)

        count = 0
        for w in m: count += m[w]

        print '%s> global count %d' % (self.host, count)
        ftime = time.time()
        print '%s> %s seconds' % (self.host, (ftime - stime))
```

A.3 Conjugate Gradients Solver [rcg.py]

```
import sys, time, getopt, operator
from river.core.vr import VirtualResource

class rConGrad (VirtualResource):
    def vr_init(self):
        vmlist = self.deploy(self.allocate(self.discover()),
            module=self.__module__, func='worker')

        print 'deployed on %d VMs' % len(vmlist)
        for vm in vmlist:
            print '  %s' % vm['host']

        self.workers = [vm['uuid'] for vm in vmlist]

        return True

    def worker(self):
        # receive the parameters from parent / master / process 0
        p = self.recv(src=self.parent, tag='init')

        verbose = p.verbose
        self.peers = p.peers
        globaln = p.globaln
        t = p.t
        iters = p.iters
        np = len(self.peers)
        n = globaln / np
        if verbose:
            print globaln, t, iters
            print self.peers

        # remove self from peerlist
        # this really only matters to master (process 0)
        self.peers.remove(self.uuid)

        # receive a piece of matrix A
        p = self.recv(src=self.parent, tag='A')
        A = p.A
        if verbose:
            print_matrix(A)

        # receive a piece of vector b
        p = self.recv(src=self.parent, tag='b')
        b = p.b
        if verbose:
            print_vector(b)

        k = 0
        x = [0.0] * n
        r = b

        dp = self.pdot(r, r)
        dp0 = dp
        while dp > t and k < iters:
            k += 1
            if k == 1:
                p = r[:]
            else:
                beta = dp / dp0
                p = daxpy(beta, p, r)

            s = self.pmatmul(A, p)
            alpha = dp / self.pdot(p, s)
            x = daypx(alpha, x, p)
            r = daypx(0 - alpha, r, s)

            dp0 = dp
            dp = self.pdot(r, r)

        # send results to parent
        self.send(dest=self.parent, tag='done', k=k, x=x)

    def pdot(self, x, y):
        local = dot(x, y)

        if self.uuid != self.parent:
            self.send(dest=self.parent, local=local, tag='pdot')
            p = self.recv(src=self.parent, tag='allreduce')
            return p.result

        else:
            for w in self.peers:
                p = self.recv(tag='pdot')
                local += p.local

            for p in self.peers:
                self.send(dest=p, result=local, tag='allreduce')
            return local

    def pmatmul(self, A, x):
        # we need to gather the global x here

        if self.uuid != self.parent:
            self.send(dest=self.parent, x=x, tag='pmatmul')
            p = self.recv(src=self.parent, tag='allgather')
            globalx = p.globalx

        else:
            globalx = x[:]
            for w in self.peers:
                p = self.recv(src=w, tag='pmatmul')
                globalx += p.x

            for w in self.peers:
                self.send(dest=w, globalx=globalx, tag='allgather')

        return matmul(A, globalx)


    def main(self):
        verbose = False

        opts,args = getopt.getopt(self.args[1:], 'v')
        for o,a in opts:
            if o == '-v': verbose = True

        if len(args) < 1:
            print 'usage: %s [-v] <input file>' % self.args[0]
            return

        # since initiator / process 0 will be doing work too,
        # we'll make it seem like he is no different from
        #  other VRs... MPI-style :)
        self.parent = self.uuid
        self.workers.insert(0, self.uuid)
        np = len(self.workers)

        # read parameters
        f = open(args[0])
        globaln = int(f.readline())
        t = float(f.readline())
        iters = int(f.readline())

        if globaln % np != 0:
            print 'error: matrix order should be evenly divisible',
            print 'by number of processors'
            return

        n = globaln / np
        # send parameters and piece of matrix A to each VR, including
        #  ourselves
        for w in self.workers:
            A = [read_vector(f, globaln) for i in xrange(n)]
            self.send(dest=w, tag='init', peers=self.workers, globaln=globaln,
                t=t, iters=iters, verbose=verbose)
            self.send(dest=w, tag='A', A=A)

        # read in vector b
        b = read_vector(f, globaln)

        # send a piece of vector b to each VR
        for i, w in enumerate(self.workers):
            self.send(dest=w, tag='b', b=b[i*n:(i+1)*n])

        f.close()

        stime = time.time()

        # we are no different than anyone else now, let's
        #  do some work!
        self.worker()

        # receive piece of the answer from each worker
        x = []
        for w in self.workers:
            p = self.recv(src=w, tag='done')
            x += p.x
            k = p.k

        ftime = time.time()

        print 'Solution found in %d iterations (%.2f s)' % (k, ftime-stime)
        if verbose:
            print_vector(x)

def print_vector(v):
    for i in v:
        print '%.2f' % i

def print_matrix(m):
    for row in m:
        for i in row:
            print '%.2f ' % i,
        print

def read_matrix(f, n):
    return [read_vector(f, n) for i in xrange(n)]

def read_vector(f, n):
    v = []

    while len(v) < n:
        line = ''
        while not line:
            line = f.readline().strip()

        v += [float(x) for x in line.split()]

    return v

def dot(x, y):
    return reduce(operator.add, map(operator.mul, x, y))

def daxpy(alpha, x, y):
    return map(lambda x,y: alpha * x + y, x, y)

def daypx(alpha, x, y):
    return map(lambda x,y: alpha * y + x, x, y)

def matmul(A, x):
    return [reduce(operator.add, map(operator.mul, row, x)) for row in A]

def dot0(x, y):
    sum = 0.0

    for i in xrange(len(x)):
        sum += x[i] * y[i]

    return sum

def daxpy0(alpha, x, y):
    for i in xrange(len(x)):
        x[i] = alpha * x[i] + y[i]

    return x

def matmul0(A, x, y):
    n = len(A)

    for i in xrange(n):
        y[i] = 0.0
        for j in xrange(n):
            y[i] += A[i][j] * x[j]

    return y

if __name__ == '__main__':
    main()
```

A.4 Remote Dictionary Extension [remdict.py]

```
from river.core.vr import VirtualResource

class DistributedDictionary (VirtualResource):
    def __getitem__(self, key):
        self.send(dest=self.parent, action='get', key=key)
        p = self.recv(src=self.parent, key=key)
        return p.value

    def __setitem__(self, key, value):
        self.send(dest=self.parent, action='set', key=key, value=value)

    def vr_init(self):
        self.m = {} ; deployed = [] ; end = 0

        VMs = self.discover()
        if len(VMs) == 0:
            print 'ERROR: Need at least one worker VM'
            return False

        VMs = self.allocate(VMs)

        files = self.args[1:]
        slice = len(files) / len(VMs) ; extras = len(files) % len(VMs)

        for worker in VMs:
            beg = end ; end = beg+slice
            if extras:
                end += 1 ; extras -= 1

            deployed += self.deploy(VMs=[worker],
              module=self.__module__, func='ddmain', args=files[beg:end])

        self.peers = [vm['uuid'] for vm in deployed]
        self.startfunc = self.serve

        return True

    def ddmain(self):
        self.main()
        self.send(dest=self.parent, action='done')

    def serve(self):
        done = 0
        while done < len(self.peers):
            p = self.recv()
            if p.action == 'get':
                v = self.m.get(p.key, 0)
                self.send(dest=p.src, key=p.key, value=v)
            elif p.action == 'set':
                self.m[p.key] = p.value
            elif p.action == 'done':
                done += 1

        for k, v in self.m.items():
            print '%016s : %d' % (k, v)
```

A.5 Ping-pong Benchmark [pingpong.py]

```
import time, getopt
from river.core.vr import VirtualResource
from river.core.oneq import QueueError

class PingPong (VirtualResource):
    def pong(self):
        p = self.recv(src=self.parent, tag='init')
        iters = p.iters
        reply = p.reply

        bytes = 0
        stime = time.time()

        i = 0
        while i < iters:
            p = self.recv()
            bytes += len(p)
            if reply:
                bytes += self.send(dest=p.src, seq=p.seq, data=p.data)
            i += 1

        ftime = time.time()
        t = ftime - stime

        print '%d bytes processed' % (bytes)
        print 'total: %.4f s, per iteration: %.4f s' % (t, t/iters)
        print 'bandwidth: %.2f mbps' % (bytes/t/131072)

    def main(self):
        bufsize = 65536 ; iters = 4096 ; reply = True
        recv = self.recv

        try:
            opts, args = getopt.getopt(self.args[1:], 'n:l:vh',
              ['ttcp', 'async', 'help'])
        except getopt.GetoptError, e:
            print 'ERROR:', e
            return -1

        for o, a in opts:
            if o == '-n': iters = int(a)
            elif o == '-l': bufsize = int(a)
            elif o == '--ttcp': reply = False
            elif o == '--async': recv = self.recv_nb
            elif o == '-h' or o == '--help': return self.usage()

        discovered = self.discover()
        if len(discovered) < 1:
            print 'ERROR: need one other VM'
            return -1

        vm = discovered[0:1]
        deployed = self.deploy(self.allocate(vm), module=self.__module__,
          func='pong')

        if len(deployed) < 1:
            print 'ERROR: deployment failed'
            return -1
        vm = deployed.pop(0)
        child = vm[’uuid’]
        print ’Pinging’, vm

        self.send(dest=child, tag=’init’, iters=iters, reply=reply)
        buffer = ’R’ * bufsize

        bytes = 0 ; i = 0 ; j = 0

        stime = time.time()

        while i < iters:
            bytes += self.send(dest=child, data=buffer, seq=i)
            if reply:
                try:
                    p = recv()
                    bytes += len(p)
                    j += 1
                except QueueError:
                    pass

            i += 1

        if reply:
            print ’%d outstanding replies’ % (iters - j)
            while  j < iters:
                p = self.recv()
                bytes += len(p)
                j += 1

        ftime = time.time()
        t = ftime - stime
        rtt = t / iters * 1000000

        print ’%d bytes processed’ % (bytes)
        print ’buffer size: %d,  iterations: %d’ % (bufsize, iters)
        print ’total: %.4f s, per iteration: %.4f s’ % (t, t/iters)
        print ’bandwidth: %.2f mbps’ % (bytes/t/131072)
        print ’latency: %.5f us’ % (rtt/2)

    def usage(self):
        print ’usage: %s [options]’ % self.args[0]
        print
        print ’Options:’
        print ’  -h [--help]            : this message’
        print ’  -n <n>                 : number of iterations to run’
        print ’  -l <n>                 : buffer size to use’
        print ’  --ttcp                 : emulate ttcp behavior (no replies)’
        print ’  --async                : asynchronous replies’
        return -1
```

A.6 Word Frequency Counter with Trickle [wfreq-trickle.py]

```
import sys, time, glob, operator

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

    return m

if len(sys.argv) < 2:
    print ’usage: %s <files>’ % sys.argv[0]
    sys.exit(1)

print ’WFreq starting...’

stime = time.time()

vrlist = connect()

files = glob.glob(sys.argv[1] + ’*’)
slices = {}
np = len(vrlist)
slice = len(files) / np
extras = len(files) % np
end = 0

for vr in vrlist:
    beg = end
    end = beg+slice
    if extras:
        end += 1
        extras -= 1
    slices[vr] = files[beg:end]

[vr.inject(wordcount) for vr in vrlist]

handles = [vr.fork(wordcount, slices[vr]) for vr in vrlist]
rvlist = [h.join() for h in handles]

m = {}
for rm in rvlist:
    for w in rm:
            m[w] = m.get(w, 0) + rm.get(w, 0)

count = reduce(operator.add, m.values())
ftime = time.time()
```

```
print 'global count %d' % count
print '%s seconds' % (ftime - stime)
```

A.7 Word Frequency Counter with IPython [wfreq-ipy.py]

```
import sys, time, glob, operator
from IPython.kernel import client

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

    return m

if len(sys.argv) < 2:
    print 'usage: %s <dir>' % sys.argv[0]
    sys.exit(1)

print 'WFreq starting...'

stime = time.time()

mec = client.MultiEngineClient()

vrlist = mec.get_ids()
print 'total of %d clients' % len(vrlist)

files = glob.glob(sys.argv[1] + '*')
slices = {}
np = len(vrlist)
slice = len(files) / np
extras = len(files) % np
end = 0
for vr in vrlist:
        beg = end
        end = beg+slice
        if extras:
                end += 1
                extras -= 1
        slices[vr] = files[beg:end]

mec.push_function(dict(wordcount=wordcount))

[mec.push(dict(files=slices[vr]), targets=vr) for vr in vrlist]

mec.execute('freqs = wordcount(files)')
rvlist = mec.pull('freqs')

m = {}
for rm in rvlist:
    for w in rm:
        m[w] = m.get(w, 0) + rm.get(w, 0)

count = reduce(operator.add, m.values())
ftime = time.time()

print 'global count %d' % count
print '%s seconds' % (ftime - stime)
```

Appendix B

River Core Source Code

This appendix presents the source code to the River run-time system.

B.1 Virtual Resource Base Class [vr.py]

```
import time, getpass, inspect, traceback, types
from threading import Thread, Condition
from river.core.oneq import OneQueue
from river.core.uuid import uuidgen
from river.core.options import *
from river.core.errors import *
from river.core.debug import DBG

class VirtualResource (Thread):
    ANY = lambda x,y: True

    def __init__(self, **kwargs):
        Thread.__init__(self)
        self.vm = kwargs['vm']
        try:
            self.uuid = kwargs['uuid']
        except KeyError:
            self.uuid = uuidgen()
        try:
            self.setName(kwargs['name'])
        except KeyError:
            pass

        self.parent = kwargs.get('parent', None)
        self.args = kwargs.get('args', [self.getName()])
        self.label = kwargs.get('label')

        self.startfunc = self.main
        try:
            if kwargs['func'] != 'run':
                self.startfunc = getattr(self, kwargs['func'])
        except KeyError:
            pass

        self.user = getpass.getuser()

        self.q = OneQueue()

        # by default we use the 'quick' queue get
        self.recv = self.q.getnocb
        self.recv_nb = self.q.get_nb
        self.peek = self.q.peek

        # attached LDE, if any
        self.lde = None

        # by default we are an application VR
        self.sysvr = False

        self.vm.register(self.uuid, self)

    def send(self, **kwargs):
        if kwargs.has_key('src'):
            raise RiverError, 'src is a reserved keyword'

        if not kwargs.has_key('dest'):
            raise RiverError, 'dest keyword is required'

        kwargs['src'] = self.uuid

        return self.vm.send(**kwargs)

    def mcast(self, **kwargs):
        if kwargs.has_key('src'):
            raise RiverError, 'src is a reserved keyword'

        kwargs['src'] = self.uuid

        return self.vm.mcast(**kwargs)

    def regcb(self, **kwargs):
        # if we set a callback, we have to use the 'slow' get
        self.recv = self.q.get

        return self.q.regcb(**kwargs)

    def unregcb(self, **kwargs):
        return self.q.unregcb(**kwargs)

    def setVMattr(self, key, value):
        return self.vm.setVMattr(key, value)

    def discmatch(self, packet, attrs):
        for a in attrs:
            try:
                if isinstance(attrs[a], types.FunctionType) or \
                    isinstance(attrs[a], types.MethodType):
                    if not attrs[a](packet[a]):
                        return False
                elif packet[a] != attrs[a]:
                    return False
            except KeyError:
                return False

        return True


    def discover(self, **kwargs):
        tests = {}
        for k in kwargs.keys():
            if isinstance(kwargs[k], types.FunctionType) or \
                isinstance(kwargs[k], types.MethodType):
                tests[k] = kwargs[k]
                kwargs[k] = '*'

        if len(kwargs) == 0:
            kwargs['status'] = 'available'
            kwargs['user'] = self.user
            kwargs['label'] = self.label

        # when all VMs are returned, not everyone may be available
        # for deployment (busy, wrong user, etc.)
        self.discVMs = {}

        kwargs['type'] = '__control__'
        kwargs['command'] = 'discover'

        self.mcast(**kwargs)

        time.sleep(DISCWAIT)

        while self.peek(type='__control__', command='discover'):
            p = self.recv(type='__control__', command='discover')
            if self.discmatch(p.attributes, tests):
                self.discVMs[p.src] = VMdesc(p.attributes)

        return self.discVMs.values()

    def allocate(self, VMs):
        # we allocate where we are told. it is up to the user to
        # verify that all the nodes requested were indeed allocated
        self.allocVMs = {}

        for vm in VMs:
            self.send(type='__control__', command='allocate', user=self.user,
                label=self.label, dest=vm['cvr'])

        n = len(VMs)
        while n:
            p = self.recv(type='__control__', command='allocate')
            vmdesc = self.discVMs[p.src]
            if p.result == 'success':
                vmdesc['uuid'] = p.newuuid
                self.allocVMs[p.src] = vmdesc
            else:
                print 'allocate failed for', vmdesc['host'], p.result
            n -= 1

        if len(VMs) != len(self.allocVMs):
            raise RiverError, 'Unable to allocate requested VMs'

        return self.allocVMs.values()

    def deallocate(self, VMs):
        for vm in VMs:
            self.send(dest=vm['cvr'], type='__control__', command='deallocate')

    def deploy(self, VMs, module, func='run', args=[]):
        self.depVMs = {}

        # get the source code for the module
        mod = __import__(module)
        fname = mod.__file__
        if fname.endswith('.pyc') or fname.endswith('.pyo'):
            fname = fname[:-1]
        f = open(fname)
        src = f.read()
        f.close()

        if not args:
            args = self.args

        for vm in VMs:
            self.send(type='__control__', command='deploy', module=module,
                func=func, args=args, source=src, user=self.user,
                lde=self.lde, dest=vm['cvr'])

        n = len(VMs)
        while n:
            p = self.recv(type='__control__', command='deploy')

            vmdesc = self.allocVMs[p.src]
            if p.result == 'success':
                self.depVMs[p.src] = vmdesc
            else:
                print 'deploy failed for', vmdesc['host'], p.result

            n -= 1

        if len(VMs) != len(self.depVMs):
            raise RiverError, 'Unable to deploy to requested VMs'

        return self.depVMs.values()

    def run(self, *args):
        name = self.getName()

        DBG('general', '=== %s STARTING ===', name)

        try:
            self.startfunc(*args)
        except Exception, e:
            traceback.print_exc()
            DBG('general', '=== %s FAILED ===', name)
        else:
            DBG('general', '=== %s COMPLETED ===', name)

        try:
            from river.extensions.remote import RemoteObject
            # delete remote objects, if any
            for x in dir(self):
                if isinstance(getattr(self, x), RemoteObject):
                    del self.__dict__[x]
        except ImportError, e:
            print 'WARNING: RAI extensions not available:', e
            print '   NOT cleaning up remote objects'

        self.send(dest=self.vm.control.uuid, type='__control__',
            command='vr_exit')

    def main(self):
        print '[main] Override this method in subclass'

class VMdesc(object):
    def __init__(self, attributes):
        self.attrs = attributes

    def __getitem__(self, key):
        return self.attrs[key]

    def __setitem__(self, key, value):
        self.attrs[key] = value

    def __repr__(self):
        return '%s on %s' % (self.attrs['cvr'], self.attrs['host'])

    def has_key(self, key):
        return self.attrs.has_key(key)

    def keys(self):
        return self.attrs.keys()

    def values(self):
        return self.attrs.values()

    def items(self):
        return self.attrs.items()

def getVRclasses(module):
    rv = []
    for name in dir(module):
        ref = getattr(module, name)
        try:
            if ref != VirtualResource and issubclass(ref, VirtualResource):
                rv.append(ref)
        except TypeError:
            pass

    rv.sort(key=(lambda ref: (inspect.getmro(ref)).__len__()), reverse=True)
    return rv
```

B.2 River Virtual Machine [riverVM.py]

```
import sys, struct, os, time, operator
from copy import deepcopy
from threading import Thread, Lock, Condition
from select import select
from socket import *
import river.core.getmyip as getmyip
from river.core.packet import Packet, encode, decode
from river.core.oneq import OneQueue
from river.core.pool import ConnPool
from river.core.kbthread import KeyboardThread
from river.core.errors import *
from river.core.options import *
from river.core.debug import DBG
import profile

class InputThread(Thread):
    def __init__(self, vm):
        Thread.__init__(self)
        self.vm = vm

    def runp(self):
        globals()['profileme'] = self.run1
        profile.runctx('profileme()', globals(), locals())

    def run(self):
        while 1:
            self.vm.netread()

class ARPEntry(object):
    def __init__(self, uuid, addr):
        self.uuid = uuid
        self.addr = addr
        self.tstamp = time.time()

class RiverVM(object):
    def __init__(self, **kwargs):
        DBG('vm', 'Initializing River VM...')

        # ARP-style cache of uuid->addr
        self.arpcache = {}

        # list of VRs on this VM
```

```
self.VRs = {}

# control VR
self.control = None

# figure out our IP
# note that KeyError is raised if iface is None, which
#   forces IP lookup via ’broadcast’ method
try:
    if kwargs.get(’iface’) == None:
        raise KeyError

    from osdep import getifaddrs
    iflist = getifaddrs.getifaddrs()

    for iface,ip in iflist:
        if iface == kwargs[’iface’]:
            self.myip = ip
            break
    else:
        raise RiverError, ’invalid interface %s’ % kwargs[’iface’]

except ImportError:
    raise RiverError, ’your system does not support interface selection’
except KeyError:
    self.myip = getmyip.get_ip_bcast()

DBG(’vm’, ’I believe my IP is %s’, self.myip)

# create a multicast socket
self.usock = socket(AF_INET, SOCK_DGRAM)
try:
    self.usock.setsockopt(SOL_SOCKET, SO_REUSEPORT, 1)
except NameError:
    self.usock.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)

# bind to our port
self.usock.bind((’’, udp_port))
# join our multicast group on the specified interface
mreq = struct.pack(’4s4s’, inet_aton(mcast_addr), inet_aton(self.myip))
self.usock.setsockopt(IPPROTO_IP, IP_ADD_MEMBERSHIP, mreq)
# use the specified interface for outgoing mcast packets
self.usock.setsockopt(IPPROTO_IP, IP_MULTICAST_IF, inet_aton(self.myip))
# set TTL for mcast packets
self.usock.setsockopt(IPPROTO_IP, IP_MULTICAST_TTL, 32)

# create a socket for IPC
self.ipcsock = socket(AF_INET, SOCK_DGRAM)

global ipc_addr
ip,port = ipc_addr
i = port
while i < (port + MAXVMS):
    try:
        self.ipcsock.bind((ip, i))
        break
    except error:
        i += 1

else:
    raise RiverError, ’Unable to bind port for IPC; too many VMs?’
DBG(’vm’, ’  IPC socket bound to port %d’, i)
ipc_addr = (ip,i)

# create a TCP listening socket
self.lsock = socket(AF_INET, SOCK_STREAM)
self.lsock.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)

self.port = tcp_port
while self.port < (tcp_port + MAXVMS):
    try:
        self.lsock.bind((’’, self.port))
        break
    except error:
        self.port += 1
else:
    raise RiverError, ’Unable to bind port for TCP; too many VMs?’

DBG(’vm’, ’  TCP socket bound to port %d’, self.port)
self.lsock.listen(4)

# connection pool
self.pool = ConnPool(MAXCONNCACHE)
self.pool.sethello(encode(type=’__hello__’,
    srcaddr=(self.myip, self.port)))

# wrap mcast and ipc sockets inside ’net connections’
self.pool[(mcast_addr, udp_port)] = self.usock
self.mcastnc = self.pool[(mcast_addr, udp_port)]
self.mcastnc.expire(None)

self.pool[ipc_addr] = self.ipcsock
self.ipc = self.pool[ipc_addr]
self.ipc.expire(None)

self.spr,self.spw = os.pipe()
self.sendq = []

# create network input thread
self.nt = InputThread(self)
self.nt.setName(’Network Input Thread’)
self.nt.start()

# create keyboard input thread (will be started by ControlVR)
self.kt = None
daemon = kwargs.get(’daemon’)
if (sys.platform != ’win32’) and not daemon:
    self.kt = KeyboardThread()
    self.kt.setName(’Keyboard Input Thread’)

# pre-encode a ’tickle’ message
self.tickle = encode(type=’__ipc__’, command=’tickle’)
```

```
    def netread(self):
        rset = self.pool.values()
        rset.append(self.lsock)

        rset,wset,eset = select(rset, [], [])

        if self.lsock in rset:
            s,addr = self.lsock.accept()
            if __debug__:
                DBG('network', '[RVM:netread] new connection from %s', addr)
            self.pool[addr] = s
            rset.remove(self.lsock)

        if len(rset) == 0: return

        for nc in rset:
            try:
                if __debug__:
                    DBG('network', '[RVM:netread] reading %s', nc)
                packets = nc.recv()

            except ConnError:
                if __debug__:
                    DBG('network', '[RVM:netread] dropped %s', nc)
                del self.pool[nc.addr]
                continue

            for p in packets:
                if __debug__:
                    DBG('packet', 'recv %s', p)

                if nc == self.pool[ipc_addr]:
                    self.ipccmd(p)
                    continue

                if self.arpcheck(p):
                    continue

                if self.hellocheck(p):
                    if __debug__:
                        DBG('network', '[RVM:netread] remap %s to %s',
                            nc.addr, p.srcaddr)
                    self.pool.remap(nc.addr, p.srcaddr)
                    continue

                # add packet to the appropriate queue
                try:
                    vr = self.VRs[p.dest]
                    vr.q.put(p)
                    if __debug__:
                        DBG('network', '[RVM:netread] for %s', p.dest)

                # no destination specified
                except AttributeError:
                    if self.control:
                        self.control.q.put(p)

                # destination is not our VR
                except KeyError:
                    pass


    def register(self, uuid, vr):
        DBG('vm', '[RVM:register] new VR %s', uuid)
        self.VRs[uuid] = vr

    def setcontrol(self, uuid):
        if self.control is None:
            DBG('vm', '[RVM:setcontrol] %s', uuid)
            self.control = self.VRs[uuid]

    def unregister(self, uuid):
        DBG('vm', '[RVM:unregister] deleting VR %s', uuid)
        del self.VRs[uuid]

    def hasVR(self, uuid):
        return self.VRs.has_key(uuid)

    def VRcount(self, all=False):
        if all:
            return len(self.VRs)
        x = reduce(operator.add, [int(not vr.sysvr) for vr in self.VRs.values()])
        return x

    def setVMattr(self, key, value):
        return self.control.setattr(key, value)

    def arpcheck(self, p):
        type = p.get('type', None)
        if type != '__arp__':
            return False

        if p.command == 'request':
            DBG('arp', '[RVM:arp] request %s', p.target)

            # one of ours?
            if self.hasVR(p.target):
                self.mcast(type='__arp__', command='reply', src=p.target,
                    srcaddr=(self.myip, self.port))

        elif p.command == 'reply':
            DBG('arp', '[RVM:arp] reply %s -> %s', p.src, p.srcaddr)
            self.arpcache[p.src] = ARPEntry(p.src, p.srcaddr)

        return True

    def arpdup(self, uuid1, uuid2):
        self.arpcache[uuid2] = self.arpcache[uuid1]

    def hellocheck(self, p):
        type = p.get('type', None)
        if type != '__hello__':
            return False

        return True

    def ipccmd(self, p):
        command = p.get('command', None)

        if command == 'quit':
            DBG('vm', '[RVM:netread] exit')
            sys.exit(0)

        elif command == 'tickle':
            DBG('network', '[RVM:netread] tee hee, that tickles')

    def send(self, **kwargs):
        uuid = kwargs['dest']

        # are we sending to a local VR?
        # as an optimization, just add it to the right queue
        if self.hasVR(uuid):
            if __debug__:
                DBG('network', '[RVM:send] local optimized')
            p = Packet(deepcopy(kwargs))
            vr = self.VRs[uuid]
            vr.q.put(p)
            return len(p)

        addr = self.resolve(uuid)
        if not addr:
            raise RiverError, 'Unable to resolve %s' % uuid

        nc = self.pool[addr]

        # if this is a new connection, tickle network thread
        # so that it can be added to select() set
        if nc.new:
            if __debug__:
                DBG('network', '[RVM:send] tickle for %s', addr)
            self.ipc.send(self.tickle)

        if __debug__:
            DBG('packet', 'send %s', kwargs)
        return nc.send(encode(**kwargs))

    def resolve(self, uuid):
        try:
            entry = self.arpcache[uuid]
            if time.time() < entry.tstamp + ARPTIMEOUT:
                return entry.addr
        except KeyError:
            pass

        if __debug__:
            DBG('arp', '[RVM:resolve] %s not found in cache', uuid)
```

```
        # block until resolved (with timeout?)
        i = 5
        stime = time.time()
        while not self.arpcache.has_key(uuid) and i:
            DBG('arp', '[RVM:resolve] ARP request broadcast')
            self.mcast(type='__arp__', command='request', target=uuid)
            DBG('arp', '[RVM:resolve] waiting for reply')

            time.sleep(ARPSLEEP)
            i -= 1

        if not i:
            DBG('arp', '[RVM:resolve] timed out')
            return None

        if __debug__:
            DBG('arp', '[RVM:resolve] resolved to %s', self.arpcache[uuid].addr)

        return self.arpcache[uuid].addr

    def mcast(self, **kwargs):
        DBG('packet', 'mcast %s', kwargs)
        self.mcastnc.send(encode(**kwargs))

    def killVR(self):
        p = Packet()
        p.type = '__control__'
        p.command = 'reset'
        p.src = 'River VM'

        for vr in self.VRs.values():
            if vr is not self.control:
                vr.q.put(p)

    def close(self):
        # terminate network input thread
        x = encode(type='__ipc__', command='quit')
        self.ipc.send(x)

        self.nt.join()

        # terminate keyboard thread
        if self.kt is not None:
            self.kt.close()
            self.kt.join()

        self.lsock.close()

        for s in self.pool.values():
            s.close()

        p = Packet()
        p.type = '__control__'
        p.command = 'quit'
        p.src = 'River VM'
        for vr in self.VRs.values():
            vr.q.put(p)

        DBG('vm', '[RVM:close] River ran dry')
```

B.3 Control VR [control.py]

```
import os, sys, platform, getpass, new, inspect
from river.core.uuid import uuidgen
from river.core.vr import *
from river.core.console import *
from river.core.options import *
from river.core.errors import *
from river.core.debug import DBG

# recipe 18.2 from the Python Cookbook
def makemodule(name, source):
    module = new.module(name)
    sys.modules[name] = module
    exec source in module.__dict__
    return module

class ControlVR (VirtualResource):
    def __init__(self, **kwargs):
        VirtualResource.__init__(self, **kwargs)
        self.args[0] = 'ControlVR'

        self._vmstate = 'available'

        self.allocations = {}

        self.restrict = kwargs.get('restrict', True)

        try:
            self.console = kwargs['console']

            # add an RAI server for the console
            from river.extensions.rai import RAIServer
            self.rai = RAIServer(self.console, self)
        except KeyError:
            self.rai = None
        except ImportError, e:
            print 'WARNING: RAI extensions not available:', e
            print '   Console will NOT be remotely available'
            self.rai = None

        self.console_out = Console_Out(self, '-')
        self.console_in = Console_In(None)

        if self.vm.kt is not None:
            self.vm.kt.configure(self, self.console_in.saved_stdin)
            self.vm.kt.start()

        self.user = getpass.getuser()
```

```
        # we are a system VR
        self.sysvr = True

        # get python version (determined in base launcher)
        self.python_version = kwargs.get('python_version', 'Unknown')

        # fill in attributes that we'll send back on discover
        self.attrs = {
              'version'    : current_version,
              'cpu'        : 1,
              'arch'       : platform.machine(),
              'mem'        : 1,
              'os'         : platform.system(),
              'pyver'      : platform.python_version(),
              'pyver'      : self.python_version,
              'host'       : platform.node(),
              'cvr'        : self.uuid,
              'status'     : self.status,
              'user'       : self.user,
              'label'      : self.label,
              'userVRs'    : self.vm.VRcount(),
              'allVRs'     : self.vm.VRcount(True),
              'caps'       : 'con '
             }

        # keep track of running VRs
        self.VRs = {}

        # load control extensions
        DBG('control', '[CON:init] loading extensions')
        self.extensions = []
        globals()['control_extension'] = self.register_extension
        globals()['send'] = self.send
        rvmmod = __import__('river.core.riverVM')
        extdir,_ = os.path.split(rvmmod.__file__)
        extdir += '/extensions/control'
        try:
            for f in os.listdir(extdir):
                if not f.endswith('.py'): continue
                execfile('%s/%s' % (extdir, f), globals())
                DBG('control', '[CON:init] loaded extension %s', f)
        except OSError, e:
            pass

    def register_extension(self, types, commands, func):
        self.extensions.append((types,commands,func))

    def attributes(self):
        self.attrs['status'] = self.status
        self.attrs['userVRs'] = self.vm.VRcount()
        self.attrs['allVRs'] = self.vm.VRcount(True)

        return self.attrs

    def setattr(self, key, value):
        self.attrs[key] = value

        if key not in self.attrs['caps']:
            self.attrs['caps'] += '%s ' % key

    def setstatus(self, status):
        self._status = status

    def getstatus(self):
        count = self.vm.VRcount()
        if count >= MAXUSERVRS:
            self._status = 'busy'
        else:
            self._status = 'available'
        return self._status

    def discmatch(self, p):
        myattrs = self.attributes()
        pattrs = p._m_.copy()

        del pattrs['type']
        del pattrs['command']
        del pattrs['src']

        for k in pattrs:
            # special case: match any user in unrestricted mode
            if k == 'user' and not self.restrict:
                continue

            try:
                x = pattrs[k]
                y = myattrs[k]
            except KeyError:
                DBG('control', '[CON:match] attribute not found: %s', k)
                return False

            if x != y and x != '*':
                return False

        return True

    def discover(self, p):
        # if request is from ourselves, ignore it
        if self.vm.hasVR(p.src):
            DBG('control', '[CON:discover] ignoring from self')
            return

        if not self.discmatch(p):
            DBG('control', '[CON:discover] matching failed')
            return

        DBG('control', '[CON:discover] replying')

        try:
            self.send(type='__control__', command='discover',
                attributes=self.attributes(), dest=p.src)
        except ConnError:
            print p
            print 'ERROR: Unable to reply to', p.src
            print '       Perhaps blocked by a firewall?'

    def allocate(self, p):
        # for allocate to succeed the user must match or
        #  we must be running in unrestricted mode
        if p.get('user') == self.user or not self.restrict:
            newuuid = uuidgen()
            self.allocations[p.src] = newuuid
            result = 'success'
        else:
            newuuid = None
            result = 'ERROR: not available'

        DBG('control', '[CON:allocate] %s', result)
        self.send(type='__control__', command='allocate', result=result,
            newuuid=newuuid, dest=p.src)

    def deallocate(self, p):
        self.allocations.pop(p.src, None)

    def deploy(self, p):
        # make sure a VR has already been allocated
        allocated = False

        try:
            newuuid = self.allocations[p.src]
            del self.allocations[p.src]
            allocated = True

            module = self.rimport(p)
            classes = getVRclasses(module)
            klass = classes[0]

        except (ImportError, IndexError, KeyError, RiverError), e:
            # possible errors: not allocated, no such module,
            #  no VR subclasses in module
            # or remote import failed
            result = 'ERROR: %s' % e

            if not allocated:
                result = 'ERROR: not allocated'

            DBG('control', '[CON:deploy] %s', result)
            self.send(type='__control__', command='deploy', result=result,
                dest=p.src)

            return

        if len(classes) > 1:
            DBG('general', 'WARNING: extra VR subclasses in %s', p.module)
            DBG('general', classes)
            DBG('general', '  using %s', str(klass))
        DBG('control', '[CON:deploy] creating new VR %s', newuuid)

        obj = klass(vm=self.vm, name=p.module, uuid=newuuid, func=p.func,
            args=p.args, parent=p.src)

        self.console_in.setmvr(obj)

        obj.lde = p.get('lde')
        obj.start()

        self.send(type='__control__', command='deploy', result='success',
            dest=p.src)

        self.VRs[p.module] = obj

    def rimport(self, p):
        module = None

        try:
            DBG('control', '[CON:rimport] trying included source')
            if p.source != None:
                return makemodule(p.module, p.source)

            from river.extensions.remote import RemoteVR

            DBG('control', '[CON:rimport] trying LDE for %s', p.module)
            if p.get('lde') == None:
                raise RAIError, 'No directory export'

            lde = RemoteVR(p.lde, self)
            f = lde.open(p.module + '.py')
            code = f.read()
            f.close()

            return makemodule(p.module, code)

        except (RAIError, ImportError), e:
            DBG('control', '[CON:rimport] %s', e)
            DBG('control', '[CON:rimport] fallback to local')

        try:
            return __import__(p.module)

        except ImportError:
            DBG('control', '[CON:rimport] local import failed')

        raise RiverError, 'unable to import module %s' % p.module

    def vr_exit(self, p):
        # exit a VR on this VM
        self.vm.unregister(p.src)
        self.cleanup()

    def cleanup(self):
        tbd = set()
        for name in self.VRs.keys():
            vr = self.VRs[name]

            DBG('control', '[CON:cleanup] %s exited, joining', name)
            del self.VRs[name]
            vr.join()

            klasses = inspect.getmro(vr.__class__)
            for k in klasses:
                if k.__name__ is 'VirtualResource':
                    break
                tbd.add(k.__module__)

        for mod in tbd:
            DBG('control', '[CON:cleanup] deleting module %s', mod)
            try:
                del sys.modules[mod]
            except KeyError:
                pass

    def process_console(self, p):
        if p.subcommand == 'setout':
            self.console_out.setstream(p.src)
        if p.subcommand == 'unsetout':
            self.console_out.unsetstream(p.src)
        if p.subcommand == 'write':
            self.console_out.write_out(p.data)
        if p.subcommand == 'read':
            self.console_in.read_in(p.data, self)

    def extmatch(self, p):
        for (types, commands, func) in self.extensions:
            if p.type in types and p.command in commands:
                func(p)
                return True

    def main(self):
        DBG('control', 'Control VR starting')
        DBG('control', '  my UUID: %s', self.uuid)
        DBG('control', '  args:    %s', self.args)
        DBG('control', '  attrs:   %s', self.attributes())

        while 1:
            p = self.recv()

            type = p.get('type')

            if type == '__rai__' and self.rai is not None:
                self.rai.dispatch(p)
                continue

            elif type != '__control__':
                DBG('control', '[CON:main] unknown type %s', type)
                continue

            DBG('control', '[CON:main] command: %s', p.command)
            if p.command == 'vr_exit':
                self.vr_exit(p)
            elif p.command == 'discover':
                self.discover(p)
            elif p.command == 'allocate':
                self.allocate(p)
            elif p.command == 'deallocate':
                self.deallocate(p)
            elif p.command == 'deploy':
                self.deploy(p)
            elif p.command == 'quit':
                DBG('control', '[CON:main] terminated')
                break
            elif p.command == 'reset':
                user = p.get('user')
                if user == self.user:
                    self.vm.killVR()
                else:
                    DBG('control', '[CON:main] reset from wrong user: %s', user)
            elif p.command == 'console':
                self.process_console(p)
            elif self.extmatch(p):
                pass
            else:
                DBG('control', '[CON:main] unknown command %s', p.command)

    status = property(getstatus, setstatus)
```

B.4 Super Flexible Packet [packet.py]

```
import operator, struct, sys, cPickle as pickle
from river.core.options import current_version
from river.core.debug import DBG
from river.core.errors import SFMError

hdrfmt = '!%dsi' % len(current_version)

class Packet(object):
    def __init__(self, m={}, size=0):
        self.__dict__['_m_'] = m
        self.__dict__['_size_'] = size

        self.__dict__['items'] = m.items
        self.__dict__['iteritems'] = m.iteritems
        self.__dict__['keys'] = m.keys
        self.__dict__['iterkeys'] = m.iterkeys
        self.__dict__['values'] = m.values
        self.__dict__['itervalues'] = m.itervalues
        self.__dict__['has_key'] = m.has_key
        self.__dict__['get'] = m.get

    def __getattr__(self, key):
        try:
            return self._m_[key]
        except KeyError:
            raise AttributeError, 'Packet has no attribute: %s' % key

    def __setattr__(self, key, value):
        return self._m_.__setitem__(key, value)

    def __getitem__(self, key):
        return self._m_.__getitem__(key)

    def __setitem__(self, key, value):
        return self._m_.__setitem__(key, value)

    def __delitem__(self, key):
        return self._m_.__delitem__(key)

    def __str__(self):
        return 'Packet: ' + str(self._m_)

    def __len__(self):
        return self._size_

def encode(**kwargs):
    payload = pickle.dumps(kwargs, pickle.HIGHEST_PROTOCOL)
    hdr = struct.pack(hdrfmt, current_version, len(payload))
    return hdr+payload

def decode(data):
    left = len(data)
    hdrlen = struct.calcsize(hdrfmt)

    # incomplete header
    if left < hdrlen:
        return None,data

    version,size = struct.unpack(hdrfmt, data[:hdrlen])

    if version != current_version:
        DBG('packet', '[decode] discarding wrong version: %s',
          version)
        raise SFMError, 'Invalid input'

    # incomplete packet
    if left < (hdrlen + size):
        return None,data

    off = hdrlen
    m = pickle.loads(data[off:off+size])
    p = Packet(m, size=hdrlen+size)
    off += size

    return p, data[off:]
```

B.5 SFM Queue [oneq.py]

```
import time, types
from threading import Condition
from river.core.packet import Packet, encode, decode
from river.core.options import *
from river.core.errors import *
from river.core.debug import DBG

class OneQueue(object):
    def __init__(self):
        self.q = []
        self.callbacks = {}
        self.cv = Condition()
        self.waits = 0
        self.calls = 0

    def match(self, packet, attrs):
        if packet.get('src') == 'River VM':
            return True

        for a in attrs:
            try:
                pv = packet[a]
                av = attrs[a]
            except KeyError:
                return False

            if isinstance(av, types.FunctionType) or \
               isinstance(av, types.MethodType):
                if not av(pv):
                    return False
            elif pv != av:
                return False
        return True

    def checkcb(self, packet):
        for attrs in self.callbacks:
            func = self.callbacks[attrs]
            if self.match(packet, attrs):
                self.callstack.append((packet, func))
                return True
        return False

    def regcb(self, **kwargs):
        try:
            func = kwargs['callback']
            del kwargs['callback']
            self.callbacks[kwargs] = func
        except KeyError:
            pass

    def unregcb(self, **kwargs):
        try:
            func = kwargs['callback']
            del kwargs['callback']
            del self.callbacks[kwargs]
        except KeyError:
            pass

    def put(self, packet):
        self.cv.acquire()
        self.q.append(packet)
        self.cv.notify()
        self.cv.release()

        DBG('queue', '%s', packet)

        if __debug__:
            DBG('queue', '[Q:put] new total: %d packets', len(self.q))

    def get(self, **kwargs):
        self.cv.acquire()

        rv = None
        while rv is None:
            self.callstack = []
            for p in self.q:
                self.checkcb(p)

            for packet, func in self.callstack:
                self.q.remove(packet)
                func(packet)

            for p in self.q:
                if self.match(p, kwargs):
                    rv = p
                    break

            # if we got nothing, block until something comes in
            if rv is None:
                self.cv.wait()

        self.q.remove(rv)
        self.cv.release()

        return rv

    def getnocb(self, **kwargs):
        self.calls += 1
        self.cv.acquire()

        rv = None
        while rv is None:
            for p in self.q:
                if self.match(p, kwargs):
                    rv = p
                    break

            if rv is None:
                self.waits += 1
                self.cv.wait()
        self.q.remove(rv)
        self.cv.release()

        return rv

    def peek(self, **kwargs):
        for p in self.q:
            if self.match(p, kwargs):
                return True
        return False

    def get_nb(self, **kwargs):
        self.cv.acquire()

        rv = None
        for p in self.q:
            if self.match(p, kwargs):
                rv = p
                self.q.remove(p)
                break

        self.cv.release()

        if rv:
            return rv

        raise QueueError, ’would block’
```

B.6 Network Connection [nc.py]

```
import sys, time
from socket import *
from socket import _socketobject
from river.core.packet import Packet, encode, decode
from river.core.options import *
from river.core.errors import *
from river.core.debug import DBG

class NetConn(object):
    def __init__(self, addr):
        self.tstamp = time.time()
        self.buffer = ’’
        self.scount = 0
        self.rcount = 0

        if isinstance(addr, _socketobject):
            self.sock = addr
            self.fileno = self.sock.fileno
            # ghetto way of determining protocol
            try:
                self.addr = self.sock.getpeername()
                self.prot = ’TCP’
            except error:
                # caller (pool) will fill in self.addr
                self.addr = None
                self.prot = ’UDP’
                self.send = self.usend
                self.recv = self.urecv

            return

        s = socket(AF_INET, SOCK_STREAM)
        t = s.gettimeout()
        s.settimeout(1.0)
        n = 5
        connected = False
        DBG(’network’, ’[NC:init] connect to %s’, addr)
        while n:
            DBG(’network’, ’[NC:init] attempt %d’, n)
            n -= 1
            try:
                s.connect(addr)
                connected = True
                break
            except error, strerror:
                if isinstance(strerror, tuple):
                    errno,strerror = strerror

                DBG(’network’, ’[NC:init] ERROR: %s %s’, addr, strerror)

        if not connected:
            raise ConnError, ’Unable to connect’

        self.prot = ’TCP’
        s.settimeout(t)
        self.addr = s.getpeername()
        self.sock = s
        self.fileno = s.fileno

    def getpeername(self):
        return self.addr

    def recv(self):
        try:
            data = self.sock.recv(NCBUFSIZE)
        except error, (errno, strerror):
            raise ConnError, strerror

        if not data:
            raise ConnError, ’Dropped connection’

        self.tstamp = time.time()

        return self.process(data)

    def urecv(self):
        data,addr = self.sock.recvfrom(NCBUFSIZE)

        return self.process(data)
```

```
def process(self, data):
    packets = []
    rv = len(data)
    self.rcount += rv

    if __debug__:
        DBG(’network’, ’[NC:process] %d bytes from %s’, rv, self.addr)

    self.buffer += data

    try:
        while self.buffer:
            p,self.buffer = decode(self.buffer)
            if p is None: break
            packets.append(p)
    except SFMError:
        DBG(’network’, ’[NC:recv] SFM decode error; discarded’)

    return packets

def send(self, data):
    self.tstamp = time.time()

    if __debug__:
        DBG(’network’, ’[NC:send] %d bytes to %s’, len(data), self.addr)
    n = self.sock.send(data)
    self.scount += n
    return n

def usend(self, data):
    if __debug__:
        DBG(’network’, ’[NC:send] %d bytes to %s’, len(data), self.addr)
    n = self.sock.sendto(data, self.addr)
    self.scount += n
    return n

def close(self):
    DBG(’network’, ’[NC:close] %s’, self.addr)
    DBG(’network’, ’    %d bytes sent, %d bytes received’,
      self.scount, self.rcount)
    self.sock.close()

def expire(self, when):
    if when is None:
        self.tstamp = sys.maxint
    else:
        self.tstamp = time.time() + when

def __str__(self):
    return ’%s connection to %s’ % (self.prot, self.addr)
```

B.7 Connection Pool [pool.py]

```
from threading import Lock
from river.core.nc import NetConn
from river.core.errors import *
from river.core.debug import DBG

# maintain a pool of [open] connections. whenever the pool is full,
# expunge least recently used

class ConnPool(object):
    def __init__(self, maxsize):
        DBG(’network’, ’[POOL:init] max: %d’, maxsize)
        self.max = maxsize

        self.pool = {}
        self.lock = Lock()

        self.hello = None

    def __getitem__(self, addr):
        try:
            nc = self.pool[addr]
            # mark this as an existing connection
            nc.new = False
        except KeyError:
            self.check()
            nc = NetConn(addr)
            if self.hello: nc.send(self.hello)
            self.lock.acquire()
            self.pool[addr] = nc
            self.lock.release()
            # mark this as a new connection
            nc.new = True

        return nc

    def __setitem__(self, addr, sock):
        self.check()
        self.lock.acquire()
        nc = NetConn(sock)
        if nc.addr is None:
            nc.addr = addr
        self.pool[addr] = nc
        self.lock.release()

    def __delitem__(self, addr):
        DBG(’network’, ’[POOL:del] %s’, addr)

        try:
            nc = self.pool[addr]
            nc.close()

            del self.pool[addr]

        except KeyError:
                raise RiverError, 'Connection cache inconsistency'

        def has_key(self, key):
            return self.pool.has_key(key)

        def values(self):
            return self.pool.values()

        def close(self, addr):
            nc = self.pool[addr]
            nc.close()

            del self.pool[addr]

        def check(self):
            if len(self.pool) >= self.max:
                DBG('network', '[POOL:check] full, expunging LRU')
                self.expunge()

        def expunge(self):
            self.lock.acquire()
            x = self.pool.items()
            x.sort(sortfun)

            addr,nc = x[0]
            DBG('network', '[POOL:expunge] expunging %s', addr)
            nc.close()
            del self.pool[addr]
            self.lock.release()

        def remap(self, old, new):
            self.lock.acquire()
            nc = self.pool[old]
            del self.pool[old]
            nc.addr = new
            self.pool[new] = nc
            self.lock.release()

        def sethello(self, packet):
            self.hello = packet

    def sortfun(tup1, tup2):
        addr1,nc1 = tup1
        addr2,nc2 = tup2

        return int(nc2.tstamp - nc1.tstamp)
```

Appendix C

River Extensions Source Code

This appendix presents the source code to the River Extensions: RAI and Trickle.

C.1 RAI Client Code [remote.py]

```
import inspect, types
from river.core.errors import *
from river.core.debug import DBG

def curry(func, fname, *args0, **kwargs0):
    def callit(*args, **kwargs):
        kwargs.update(kwargs0)
        return func(fname, *(args0+args), **kwargs)
    return callit

class RemoteObject(object):
    def __init__(self, name, id, server, parent, type = ’object’):
        self.name = name
        self.id = id
        self.server = server
        self.type = type

        self.orphan = True
        self.parent = None
        self.send = None
        self.recv = None

        self.bind(parent)

    def bind(self, parent):
        DBG(’rai’, ’[Remote:bind] %s to %s’, self, parent)
        self.parent = parent
        self.send = parent.send
        self.recv = parent.recv
        self.orphan = False

    def __getnewargs__(self):
        return (self.name, self.id, self.server, None)

    def __getstate__(self):
        DBG(’rai’, ’[Remote:getstate] making %s an orphan’, self)
        d = self.__dict__.copy()
        d[’orphan’] = True
        d[’parent’] = None
        d[’send’] = None
        d[’recv’] = None

        return d

    def __setstate__(self, d):
        self.__dict__.update(d)
        DBG(’rai’, ’[Remote:setstate] orphan object %s’, self)

    def __del__(self):
        # orphans, functions, fork handles should not be deleted
        if self.orphan or self.id is None: return
        if self.type == ’callable’: return
        if isinstance(self, ForkHandle): return

        DBG(’rai’, ’[Remote:del] deleting remote object %s’, self.name)
        self.shalf(command=’delete’)

    def __str__(self):
        return ’<%s (%s) on %s>’ % (self.name, self.id, self.parent)

    def __getattr__(self, name):
        DBG(’rai’, ’[Remote:getattr] lookup %s in %s’, name, self.name)

        if self.orphan:
            DBG(’rai’, ’[Remote:getattr] orphan object %s’, self)

            # try to find a parent VR for this object
            frame = inspect.currentframe()
            # it is possible that the caller is one of our member
            #  functions, so we may need to look deeper
            while frame.f_globals[’__name__’] == ’river.extensions.remote’:
                frame = frame.f_back
            parent = frame.f_locals[’self’]

            DBG(’rai’, ’[Remote:getattr] new RemoteVR, parent %s’, parent)
            r = RemoteVR(self.server, parent)
            self.bind(r)

        return self.rai(command=’getattr’, name=name)

    def __call__(self, *args, **kwargs):
        DBG(’rai’, ’[Remote:__call__] calling %s %s in %s’, self.name,
            str(args), self.id)
        return self.rai(command='call', name=self.name, args=args, kwargs=kwargs)

    def fork(self, fname, *args, **kwargs):
        if not isinstance(fname, str):
            fname = fname.__name__

        DBG('rai', '[Remote:fork] forking %s %s in %s', fname, str(args),
            self.name)

        self.shalf(command='call', name=fname, args=args, kwargs=kwargs)

        return ForkHandle(fname, self.id, self.server, self)

    # this generic function does all the real work
    def rai(self, **kwargs):
        self.shalf(**kwargs)
        return self.rhalf(command=kwargs['command'])

    def shalf(self, **kwargs):
        kwargs['type'] = '__rai__'
        kwargs['dest'] = self.server
        kwargs['id'] = self.id

        self.send(**kwargs)

    def rhalf(self, **kwargs):
        kwargs['type'] = '__rai__'
        kwargs['src'] = self.server

        p = self.recv(**kwargs)

        return self.processrv(p)

    def processrv(self, p):
        if p.result == 'object':
            DBG('rai', '[Remote:processrv] %s is an object', p.name)
            return RemoteObject(p.name, p.id, p.src, self.getparent())

        elif p.result == 'function':
            DBG('rai', '[Remote:processrv] %s is a function', p.name)
            return RemoteObject(p.name, p.id, p.src, self.getparent(),
                type = 'callable')

        elif p.result == 'success':
            return p.rv

        elif p.result == 'StopIteration':
            raise StopIteration

        raise RAIError, 'RAI failed: %s' % p.result

    def getparent(self):
        if isinstance(self, RemoteVR):
            return self
        else:
            return self.parent


class RemoteVR(RemoteObject):
    def __init__(self, server, parent):
        RemoteObject.__init__(self, 'RemoteVR', None, server, parent)

    def __str__(self):
        return '<RemoteVR %s>' % self.server

    def inject(self, obj):
        frame = inspect.currentframe().f_back
        name = None

        # travel up the call stack in case we were called by the
        # global inject() function
        while frame.f_locals.get('self') is self.parent:
            frame = frame.f_back

        # find the appropriate object to inject
        for k in frame.f_locals:
            if frame.f_locals[k] is obj:
                name = k
                break

        DBG('rai', '[RemoveVR:inject] injecting %s %s', name, type(obj))

        # function
        if inspect.isfunction(obj):
            injtype = 'function'
            src = inspect.getsource(obj)
            filename = inspect.getsourcefile(obj)

        # class
        elif inspect.isclass(obj):
            injtype = 'class'
            # trickle hack: look inside parent to determine the
            # correct source file, since inspect is stupid
            try:
                DBG('rai', '[RemoteVR:inject] trying %s', self.parent.getName())
                filename = self.parent.getName()
                src = get_class_source(name, filename)
            except:
                raise RAIError, 'Unable to find code to inject'
            filename = inspect.getsourcefile(obj)

        # data
        else:
            injtype = 'data'
            src = obj
            filename = '<data>'

        DBG('rai', '[RemoteVR:inject] %s from %s', name, filename)

        self.rai(command='inject', name=name, source=src, injtype=injtype)
```

```
class ForkHandle(RemoteObject):
    def __str__(self):
        return '<ForkHandle to %s in %s>' % (self.name, self.parent)

    def join(self):
        return self.rhalf(command='call')

def get_class_source(classname, filename):
    f = open(filename)

    buf = ''
    ind0 = None
    copying = False
    save_next = True

    for l in f:
        ind = len(l) - len(l.lstrip())

        if save_next:
            ind0 = ind
            save_next = False

        if ind0 is not None and ind < ind0:
            break

        if l.find('class') >= 0 and l.find(classname) > 0:
            copying = True
            save_next = True

        if copying:
            buf += l

    f.close()

    if not buf:
        raise RiverError, 'Parsing failed'

    return buf
```

C.2 RAI Server Code [rai.py]

```
import new, getopt, random, sys, traceback
from river.core.vr import VirtualResource
from river.extensions.remote import *
from river.core.errors import *
from river.core.debug import DBG

def ERR_INVALID_OBJ(id): return 'Invalid object id %s' % str(id)

def isobject(x):
    return not (isinstance(x, int) or isinstance(x, float) or \
                isinstance(x, long) or isinstance(x, str) or \
                isinstance(x, list) or isinstance(x, dict) or \
                isinstance(x, tuple) or x is None)

class NSWrapper(object):
    def __init__(self, ns):
        self.ns = ns

    def __getattr__(self, name):
        if self.ns.has_key(name):
            return self.ns[name]
        else:
            raise AttributeError, "No such attribute '%s'" % name

class RAIServer (object):
    def __init__(self, mainobj, parent, restrict=True):
        self.restrict = restrict

        self.ns = globals().copy()
        self.ns.update(self.ns['__builtins__'])
        del self.ns['__builtins__']

        if mainobj is None:
            self.mainobj = NSWrapper(self.ns)
        else:
            self.mainobj = mainobj

        self.objcache = { None : self.mainobj }

        self.puuid = parent.parent
        self.send = parent.send
        self.recv = parent.recv

    def genID(self):
        i = None
        while self.objcache.has_key(i):
            i = random.randint(0,9999)
        return i

    def hideobject(self, obj):
        id = self.genID()
        self.objcache[id] = obj

        DBG('rai', '[RAI:hide] caching object, id %d', id)
        DBG('rai', str(obj))

        return id

    def unwrap_one(self, x):
        if isinstance(x, RemoteObject):
            if x.type == 'callable':
                DBG('rai', '[RAI:unwrap] func %s in %s', x.name, x.id)
                return getattr(self.objcache[x.id], x.name)
            else:
                DBG('rai', '[RAI:unwrap] obj %d', x.id)
                return self.objcache[x.id]

        return x
```

```
def unwrap(self, x):
    if len(x) == 0:
        return x

    # unwrap any references to remote objects (our local objects)
    if isinstance(x, tuple) or isinstance(x, list):
        x = tuple([self.unwrap_one(i) for i in x])
    elif isinstance(x, dict):
        for i in x:
            x[i] = self.unwrap_one(x[i])

    return x

def reply(self, **kwargs):
    kwargs['type'] = '__rai__'

    rv = kwargs['rv']
    name = kwargs['name']
    result = kwargs['result']

    if result != 'success':
        DBG('rai', '[RAI:reply] ERROR: %s', result)

    if callable(rv):
        kwargs['rv'] = None
        kwargs['result'] = 'function'
        DBG('rai', '[RAI:reply] %s is a function', name)

    elif isobject(rv):
        DBG('rai', 'name change from %s to %s', kwargs['name'],
            rv.__class__.__name__)
        kwargs['name'] = rv.__class__.__name__
        kwargs['id'] = self.hideobject(rv)
        kwargs['rv'] = None
        kwargs['result'] = 'object'

    self.send(**kwargs)

def megafunc(self, packet):
    rv = None
    id = packet.id
    name = packet.name
    cmd = packet.command
    result = 'success'

    DBG('rai', '[RAI:%s] %s in object %s', cmd, name, id)

    args = self.unwrap(packet.get('args', []))
    kwargs = self.unwrap(packet.get('kwargs', {}))

    try:
        obj = self.objcache[id]

        if cmd == 'getattr':
            rv = getattr(obj, name)
```

```
        elif cmd == 'call':
            DBG('rai', '    args: %s kwargs: %s', args, kwargs)
            func = getattr(obj, name)
            rv = func(*args, **kwargs)

    # object does not exist
    except KeyError:
        result = ERR_INVALID_OBJ(id)
    # no such attribute
    except (AttributeError, TypeError), e:
        result = str(e)
    except StopIteration:
        result = 'StopIteration'
    # any other error
    except Exception, e:
        result = str(e)

    self.reply(command=cmd, id=id, name=name, rv=rv, result=result,
        dest=packet.src)

def inject(self, packet):
    result = 'success'

    if self.restrict:
        result = 'Permission denied'

    else:
        DBG('rai', '[RAI:inject] %s (%s)', packet.name, packet.injtype)

        if packet.injtype == 'data':
            self.ns[packet.name] = packet.source
        else:
            exec packet.source in self.ns

    self.send(type='__rai__', command='inject', id=None, name=packet.name,
        result=result, dest=packet.src)

def delete(self, packet):
    id = packet.id

    # do not delete the main object
    if id is None:
        return

    try:
        del self.objcache[id]
        DBG('rai', '[RAI:del] deleted object %d (%d left)', id,
            len(self.objcache))
    except KeyError:
        DBG('rai', '[RAI:del] %s', ERR_INVALID_OBJ(id))

def dispatch(self, p):
    if p.type == '__control__' and p.command == 'quit':
        if p.src == 'River VM' or p.src == self.puuid:
            raise RiverExit, 'termination requested'
        else:
                DBG('rai', '[RAI:dispatch] unauthorized termination request')

        if p.type != '__rai__':
            DBG('rai', '[RAI:dispatch] not an RAI packet')
            return

        if p.command == 'call' or p.command == 'getattr':
            self.megafunc(p)

        elif p.command == 'inject':
            self.inject(p)

        elif p.command == 'delete':
            self.delete(p)

        else:
            DBG('rai', '[RAI:dispatch] unknown command %s', p.command)

class RAIServerVR (VirtualResource):
    def __init__(self, **kwargs):
        VirtualResource.__init__(self, **kwargs)

        try:
            self.module = kwargs['module']
        except KeyError:
            self.module = None

    def usage(self):
        print 'usage: river %s [--unrestricted] [module]' % self.args[0]

    def main(self):
        restrict = True

        try:
            opts,args = getopt.getopt(self.args[1:], 'u',
                ['help', 'unrestricted'])
        except getopt.GetoptError, e:
            print 'ERROR:', e
            return -1

        for o,a in opts:
            if o == '-h' or o == '--help':
                self.usage()
                return -1
            elif o == '-u' or o == '--unrestricted':
                restrict = False

        if restrict and len(args) == 0:
            self.usage()
            return -1

        if args and self.module:
            if args[0].endswith('.py'):
                filename = args[0][:-3]
            module = self.module
        elif args:
            filename = args[0]
            if args[0].endswith('.py'):
                filename = args[0][:-3]

            try:
                module = __import__(filename)
            except ImportError, e:
                print 'ERROR: RAI unable to import %s: %s' % (args[0], e)
                return -1

            print 'RAI serving module', args[0]

        elif not restrict:
            module = None
            filename = '<unrestricted>'

        else:
            print 'ERROR: no module specified in restricted mode'
            return -1

        if not restrict:
            DBG('rai', 'WARNING: running in unrestricted mode!')

        DBG('rai', '[RAI:main] %s serving %s', self.uuid, filename)
        rai = RAIServer(module, self, restrict)
        self.setVMattr('RAI_module', filename)
        self.setVMattr('RAI_server', self.uuid)

        while 1:
            p = self.recv()
            try:
                rai.dispatch(p)
            except RiverError, e:
                traceback.print_exc()
            except RiverExit:
                break
```

C.3 Trickle [trickle.py]

```
import sys, os, traceback, getopt, cmd, readline

from river.core.riverVM import RiverVM
from river.core.control import ControlVR
from river.core.vr import VirtualResource
from river.core.launcher import RiverLauncher
from river.extensions.remote import RemoteObject, RemoteVR
from river.extensions.rai import RAIServerVR
from river.core.errors import *
from river.core.options import *

class TrickleError (RiverError):
    pass
```

```
class Trickle (VirtualResource):
    def __init__(self, **kwargs):
        VirtualResource.__init__(self, **kwargs)

        g = globals()
        g['connect'] = self.connect
        g['inject'] = self.inject
        g['fork'] = self.fork
        g['join'] = self.join
        g['joinany'] = self.joinany
        g['forkjoin'] = self.forkjoin
        g['forkgen_old'] = self.forkgen_old
        g['forkwork_old'] = self.forkwork_old
        g['forkgen'] = self.forkgen
        g['forkwork'] = self.forkwork

    def vr_init(self):
        discovered = self.discover()
        allocated = self.allocate(discovered)
        deployed = self.deploy(allocated, module='rai',
            args=['rai.py', '--unrestricted'])

        self.vrlist = [vm['uuid'] for vm in deployed]

        return True

    def connect(self, np = 0):
        vrcount = len(self.vrlist)
        if vrcount < np:
            raise TrickleError, 'Not enough VMs found'

        if np == 0:
            np = vrcount

        rv = [RemoteVR(uuid, self) for uuid in self.vrlist[:np]]

        return rv

    def inject(self, vrs, *args):
        if type(vrs) != list:
            vrs = [vrs]
        for vr in vrs:
            for code in args:
                vr.inject(code)

    def fork_1(self, vr, fname, *args):
        return vr.fork(fname, *args)

    def fork(self, vrlist, flist, *args):
        rv = []
        flistp = False
        arglistp = False

        # Handle simple case: single vr and single func
        if not isinstance(vrlist, list) and not isinstance(flist, list):
            return vrlist.fork(flist, *args)

        if isinstance(flist, list):
            flistp = True
            if len(vrlist) != len(flist):
                raise TrickleError, 'Invalid arguments'

        else:
            f = flist

        if len(args) == 1:
            if isinstance(args[0], list):
                arglistp = True
                arglist = args[0]
                if len(vrlist) != len(arglist):
                    raise TrickleError, 'Invalid arguments'

        for i, vr in enumerate(vrlist):
            if arglistp:
                args = arglist[i]
                if not isinstance(args, tuple):
                    args = [args]
            if flistp:
                f = flist[i]

            rv.append(vr.fork(f, *args))

        return rv

    def join(self, hlist):
        if not isinstance(hlist, list):
            return hlist.join()

        if len(hlist) == 0:
            raise TrickleError, 'join on an empty list'

        rvlist = [h.join() for h in hlist]
        hlist = []
        return rvlist

    def joinany(self, hlist):
        if not isinstance(hlist, list):
            return hlist.join()

        if len(hlist) == 0:
            raise TrickleError, 'join on an empty list'

        m = {}
        for h in hlist:
            m[h.server] = h

        p = self.recv(type='__rai__', command='call')

        h = m[p.src]
        hlist.remove(h)
        return h, h.processrv(p)

    def forkjoin(self, vrlist, flist, *args):
        hlist = fork(vrlist, flist, *args)
        return join(hlist)

    def forkgen_old(self, vrlist, fname, work):
        hlist = []

        while True:
            while len(work) > 0 and len(vrlist) > 0:
                w = work.pop()
                v = vrlist.pop()
                print w
                hlist.append(fork(v, fname, w))

            if len(hlist) > 0:
                h, rv = joinany(hlist)
                vrlist.append(h.vr)
                yield(rv)
            else:
                break

    # Support sending chunks of work to each VM
    # Allow work to be a generator
    def forkgen(self, vrlist, fname, work, chunksize=1):
        hlist = []

        # Initialize generator
        if not isinstance(work, list):
            workgen = work()

        while True:
            while len(vrlist) > 0:
                if isinstance(work, list):
                    if len(work) > 0:
                        if len(work) >= chunksize:
                            w = work[0:chunksize]
                            work = work[chunksize:]
                        else:
                            w = work
                            work = []
                        v = vrlist.pop()
                        hlist.append(v.fork(fname, w))
                    else:
                        break
                else:
                    try:
                        w = workgen()
                    except StopIteration:
                        break
                    v = vrlist.pop()
                    hlist.append(fork(v, fname, *w))

            if len(hlist) > 0:
                h, rv = joinany(hlist)
                vrlist.append(h.getparent())
                yield(rv)
            else:
                break

    def forkwork_old(self, vrlist, fname, work):
        results = []

        for rv in forkgen_old(vrlist, fname, work):
            results.append(rv)

        return results

    def forkwork(self, vrlist, fname, work, chunksize=1):
        results = []

        for rv in forkgen(vrlist, fname, work, chunksize=chunksize):
            results.append(rv)

        return results

    def main(self):
        sys.argv = self.args[1:]

        if len(self.args) == 1:
            print 'usage: %s <module>' % self.args[0]
            return -1

        try:
            execfile(self.args[1], globals())
        except SystemExit:
            pass
        except Exception, e:
            traceback.print_exc()

        # tell all the RAI servers to exit
        for vr in self.vrlist:
            self.send(dest=vr, type='__control__', command='quit')

    def interpreter(self, interpreter):
        interpreter.cmdloop()

class TrickleLauncher (RiverLauncher):
    def __init__(self):
        RiverLauncher.__init__(self)

        self.dbgopts = ['rai']
        self.interactive = False

        self.register_cmdline_options('', ['interactive'], self.parse_cmdline_cb)

    def usage(self):
        RiverLauncher.usage(self)

        print 'Trickle options:'
        print '  --interactive          : run in interactive mode'
        print
        print 'Either a module to run or --interactive must be specified.'

    def parse_cmdline_cb(self, o, a):
        if o == '--interactive': self.interactive = True

    def launch(self):
        if not self.args and not self.interactive:
            self.usage()
            return False

        self.startRVM()

        self.args.insert(0, 'trickle.py')
        args = []

        if self.interactive:
            filename = '<interpreter>'
            vr = Trickle(vm=self.rvm, name=filename, args=self.args,
                func='interpreter')
            rv = vr.vr_init()

            interpreter = TrickleInterpreter()
            args = [interpreter]

            print '[Trickle Interpreter]'
        else:
            filename = self.args[1]
            vr = Trickle(vm=self.rvm, name=self.args[1], args=self.args)
            rv = vr.vr_init()

        vr.setVMattr('Trickle', filename)

        if not rv:
            print 'ERROR: vr_init() failed'
        else:
            self.cvr.console_in.setmvr(vr)
            rv = vr.run(*args)

        return rv

class TrickleInterpreter(cmd.Cmd):
    intro = """Running Trickle in interactive mode...
Type "help" for more information.
"""
    prompt = '>>> '
    helpmsg = """Trickle-specific functions:

connect
inject
fork
join
"""

    def preloop(self):
        self._history = []
        self._globals = globals()
        self._locals = {}

    def precmd(self, line):
        self._history.append(line.strip())

        return line

    def do_history(self, arg):
        for i in self._history:
            print i

    def do_help(self, arg):
        print self.helpmsg

    def do_EOF(self, arg):
        return True

    def emptyline(self):
        pass

    def default(self, line):
        try:
            exec line in self._locals, self._globals
        except Exception, e:
            traceback.print_exc()

    do_quit = do_EOF
    do_exit = do_EOF

if __name__ == '__main__':
    sys.setrecursionlimit(100)

    tl = TrickleLauncher()
    try:
        tl.parse_cmdline(sys.argv)
        tl.launch()
        tl.stop()
    except RiverError, e:
        print >> sys.stderr, 'ERROR:', e
    except RiverExit:
        pass
```

C.4 rMPI Base [mpi.py]

```
import sys, time, math, string, pickle, random, traceback, inspect
from river.core.vr import VirtualResource
from river.core.debug import DBGR as DBG
from river.core.console import *

mpi_chunksize_bytes = 2**16-1000
mpi_calibration_list_size = 1024
```

```
##
## MPI Datatypes and Helper Datatypes
##

# rMPI specific datatypes
class MPI_Int(object):
    def __init__(self, i=None):
        self.value = i

    def __repr__(self):
        return '%d' % self.value

class MPI_Status( object ):
    def __init__( self, *args ):
        self.MPI_SOURCE = MPI_Int(0)
        self.MPI_TAG = 0
        self.count = 0

class MPI_Comm( object ):

    def __init__( self, _rankToUuid, _uuidToRank ):
        self._rankToUuid = _rankToUuid
        self._uuidToRank = _uuidToRank

    #takes in a rank object, returns the uuid
    def rankToUuid( self, rank ):
        return self._rankToUuid[rank.value]

    #takes in a uuid, returns the rank object
    def uuidToRank( self, uuid ):
        return MPI_Int( self._uuidToRank[uuid] )

    # return a sorted list of available ranks in the communicator
    def ranks(self):
        ranklist = self._rankToUuid.keys()
        ranklist.sort()
        return ranklist

    # GDB: add rank()

    def size(self):
        return len(self._rankToUuid.keys())

    def has_uuid( self, uuid ):
        return self._uuidToRank.has_key( uuid )

    def has_rank( self, rank ):
        return self._rankToUuid.has_key( rank.value )


class MPI_Group( object ):
    #args = _rankToUuid, _uuidToRank
    def __init__( self, *args ):
        if len( args ) == 2:
            self._rankToUuid = args[0]
            self._uuidToRank = args[1]
        else:
            self._rankToUuid = None
            self._uuidToRank = None

    #sets a Null groups attributes, noop if called on a non null group
    def setmaps( self, _rankToUuid, _uuidToRank ):
            self._rankToUuid = _rankToUuid
            self._uuidToRank = _uuidToRank

    def rankToUuid( self, rank ):
        return self._rankToUuid[rank.value]

    def uuidToRank( self, uuid ):
        return self._uuidToRank[uuid]

    def has_uuid( self, uuid ):
        return self._uuidToRank.has_key( uuid )

    def has_rank( self, rank ):
        return self._rankToUuid.has_key( rank.value )


class Chunk(object):
    def __init__(self, offset, size):
        self.offset = offset
        self.size = size

class MPI_Datatype(object):
    def __init__(self):
        # list of chunks
        self.chunks = []

        # chunk count (may be less than len(chunks))
        self.count = 0

        # size of type as total number of elements
        self.size = 0

        # send method (pack, full send, or overlap)
        self.method = None

    def __repr__(self):
        return '<derived datatype>'

    def add_chunk(self, offset, size):
        for c in self.chunks:
            if c.offset + c.size == offset:
                c.size += size
                DBG('dd', '[dd:add_chunk] + %d at [%d]',
                  size, c.offset)
                return
            elif offset + size == c.offset:
                c.offset = offset
                c.size += size
                DBG('dd', '[dd:add_chunk] %d + at [%d]',
                  size, c.offset)
                return

        DBG('dd', '[dd:add_chunk] new size %d at offset %d',
          size, offset)
        self.chunks.append(Chunk(offset, size))

    def copy_chunks(self, old_type, base):
        for c in old_type.chunks:
            offset = base + c.offset
            self.add_chunk(offset, c.size)

# Default communicator
MPI_COMM_WORLD = MPI_Comm( None, None )

# Technique to get enumerated constants
# We can easily add a new constant in an arbitrary position

class mpi_range(object):
    def __init__(self, start):
        self.start = start
    def next(self):
        tmp = self.start
        self.start += 1
        return tmp

# Using larger ints to decrease chance of user error
mpi_robj = mpi_range(10007)

# MPI Datatypes

MPI_CHAR      = mpi_robj.next()
MPI_INT       = mpi_robj.next()
MPI_LONG      = mpi_robj.next()
MPI_FLOAT     = mpi_robj.next()
MPI_DOUBLE    = mpi_robj.next()

# Datatype dictionary
mpi_types = {}
mpi_types[MPI_CHAR]   = str
mpi_types[MPI_INT]    = int
mpi_types[MPI_LONG]   = long
mpi_types[MPI_FLOAT]  = float
mpi_types[MPI_DOUBLE] = float

mpi_zero_values = {}
mpi_zero_values[MPI_CHAR]   = ' '
mpi_zero_values[MPI_INT]    = 0
mpi_zero_values[MPI_LONG]   = 0L
mpi_zero_values[MPI_FLOAT]  = 0.0
mpi_zero_values[MPI_DOUBLE] = 0.0

# Datatype sizes (hardcoded for now, based on pickle highest protocol)
mpi_types_size = {}
mpi_types_size[MPI_CHAR]   = 2
mpi_types_size[MPI_INT]    = 5
mpi_types_size[MPI_LONG]   = 5
mpi_types_size[MPI_FLOAT]  = 9
mpi_types_size[MPI_DOUBLE] = 9


# MPI Operations
MPI_MAX       = mpi_robj.next()
MPI_MIN          = mpi_robj.next()
MPI_SUM       = mpi_robj.next()
MPI_PROD      = mpi_robj.next()
MPI_LAND      = mpi_robj.next()
MPI_BAND      = mpi_robj.next()
MPI_LOR       = mpi_robj.next()
MPI_BOR       = mpi_robj.next()
MPI_LXOR      = mpi_robj.next()
MPI_BXOR      = mpi_robj.next()
MPI_MAXLOC    = mpi_robj.next()
MPI_MINLOC    = mpi_robj.next()

# Operations dictionary
mpi_ops = {}

mpi_ops[MPI_MAX]    = max
mpi_ops[MPI_MIN]    = min
mpi_ops[MPI_SUM]    = lambda x, y: x + y
mpi_ops[MPI_PROD]   = lambda x, y: x * y
mpi_ops[MPI_LAND]   = lambda x, y: x and y
mpi_ops[MPI_BAND]   = lambda x, y: x & y
mpi_ops[MPI_LOR]    = lambda x, y: x or y
mpi_ops[MPI_BOR]    = lambda x, y: x | y
mpi_ops[MPI_LXOR]   = lambda x, y: (x or y) and not (x and y)
mpi_ops[MPI_BXOR]   = lambda x, y: x ˆ y
mpi_ops[MPI_MAXLOC] = None
mpi_ops[MPI_MINLOC] = None

# MPI Errors
MPI_SUCCESS   = mpi_robj.next()
MPI_ERR_COMM  = mpi_robj.next()
MPI_ERR_TYPE  = mpi_robj.next()
MPI_ERR_COUNT = mpi_robj.next()
MPI_ERR_TAG   = mpi_robj.next()
MPI_ERR_RANK  = mpi_robj.next()
MPI_ERR_ARG   = mpi_robj.next()
MPI_ERR_OTHER = mpi_robj.next()

# Null Process Rank
MPI_PROC_NULL = MPI_Int(-1)

# Helpers
RANK_OFFSET = 1

##
## Main mpi VirtualResource class
##

# ASF: chop and glue routines
# these need to be imported after the definition of MPI_Datatype
```

```
from chop import chop, Glue

class MPI( VirtualResource ):

    # All VR's init here
    def __init__( self, **kwargs ):
        VirtualResource.__init__( self, **kwargs )
        self.args = kwargs['args']
        self.mpi_inited = False
        self.msg_uid = 0

        # ASF: populate global namespace with MPI_* functions
        # extra special Python trickery because globals() gives us
        # the wrong global dictionary
        #g = globals()
        g = sys.modules[self.getName()].__dict__
        for k in dir(self):
            if k.startswith('MPI_'):
                g[k] = getattr(self, k)

    # Initiator only falls thru to here, then to main
    # GDB: fix -np
    # np: number of processes
    # nc: number of consoles to map
    def vr_init( self, np=1, nc=1 ):
        # Initiator is always rank 0
        self.rank = None

        # Find and get the number of available VMs requested
        vmlist = self.discover()
        # Allocate np VMs
        nodes = self.allocate( vmlist[0:np] )
        uuids = [vm['uuid'] for vm in nodes]
        self.control_vrs = [vm['cvr'] for vm in nodes]

        # Check number of VMs grabbed
        needed = np
        got = len( uuids )
        if needed != got:
            print 'needed %d VMs, but got %d' % (np, got)
            self.deallocate( VMs=nodes )
            return False

        # Control VR reverse map
        self.nconsoles = nc
        self.cvrToRank = {}
        for i, cvr in enumerate(self.control_vrs):
            self.cvrToRank[cvr] = i

        # Map their ranks (starting at 1. 0 is reserved for initiator)
        self.rankToUuid = {}
        for cur in uuids:
            self.rankToUuid[uuids.index( cur )] = cur

        # The reverse mapping
        self.uuidToRank = {}
        for cur in self.rankToUuid:
            self.uuidToRank[self.rankToUuid[cur]] = cur

        # For now, take it on faith that all deployments were succesful
        # TODO: check deploy list against discover list for verification
        VMs = self.deploy( VMs=nodes, module=self.__module__,
          func='nodeStart' )

        # Set remote console access
        for i in xrange(nc):
            ConsoleClient(self.control_vrs[i], self)


        # Tell everyone their ranks, the module name, and send the
        # list of ranks
        # for now assume everything goes ok
        # GDB: move this to a deploy argument
        for vr in uuids:
            self.send( dest=vr, tag='ranks', ranks=self.rankToUuid,
              uuids=self.uuidToRank, rank=uuids.index( vr ) )

        self.startfunc = self.initiatorStart
        return True

    def initiatorStart( self ):

        active_vrs = len(self.uuidToRank)

        # Wait for MPI VRs to complete and display remote console output
        while active_vrs:
            p = self.recv()

            if p.src in self.uuidToRank and p.tag == 'done':
                active_vrs -= 1
            elif p.src in self.control_vrs and p.command == 'console':
                print p.data,
            else:
                print 'Unrecognized packet: ', p


        # Set remote console access
        for cvr in self.control_vrs:
            self.send(dest=cvr, type='__control__',
              command='console', subcommand='unsetout')

        print 'All MPI processes have finished'

    def nodeStart( self ):
        master = self.parent
        p = self.recv( src=master, tag='ranks' )
        self.rank = MPI_Int( p.rank )
        self.rankToUuid = p.ranks
        self.uuidToRank = p.uuids

        # ASF: add exception handling and print to stdout, which
        # will be sent to the initiator
    try:
        self.main(self.args)
    except Exception, e:
        traceback.print_exc(None, sys.stdout)

    ## can we check if console_out is done

    self.send( dest = self.parent, tag='done')

# Helper Functions

def isValidRank( self, rank, comm ):
    return comm.has_rank( rank )

def isInited( self ):
    return self.mpi_inited

def getMsgUid( self ):
    id = self.msg_uid
    self.msg_uid = self.msg_uid + 1
    return id

# Convert int ranks into MPI_Int values
def normalize_rank(self, rank):
    if isinstance(rank, MPI_Int):
        # TODO: why doesn't this work???
        pass
    elif rank.__class__.__name__ == 'MPI_Int':
        pass
    elif isinstance(rank, int):
        rank = MPI_Int(rank)
    else:
        raise TypeError, 'rank must be MPI_Int or int, not %s' % type(rank)

    return rank

# Buffer/chunk calculations

def mpi_getchunksize(self, datatype):
    if isinstance(datatype, MPI_Datatype):
        # derived type
        # TODO this calculation depends on base types in
        # buffer
        csize = mpi_chunksize_bytes / \
          mpi_types_size[MPI_DOUBLE]
    else:
        # base types
        csize = mpi_chunksize_bytes / \
          mpi_types_size[datatype]

    DBG('mpi', '[getchunksize] chunk size for type %s: %d',
      datatype, csize)

    return csize

##
## Environment Management Routines
##

def MPI_Init(self, *args):
    if not self.isInited():
        global MPI_COMM_WORLD
        MPI_COMM_WORLD._rankToUuid = self.rankToUuid
        MPI_COMM_WORLD._uuidToRank = self.uuidToRank
        self.mpi_inited = True
        del self.rankToUuid
    else:
        raise ValueError, 'MPI_Init called previously'

# Return a group of the processes associated with comm
def MPI_Comm_group( self, comm, group ):
    group.setmaps( comm._rankToUuid, comm._uuidToRank )
    return MPI_SUCCESS

def MPI_Comm_size( self, comm, size ):
    size.value = len( comm._rankToUuid )
    return MPI_SUCCESS

def MPI_Comm_rank( self, comm, rank ):
    rank.value = comm.uuidToRank( self.uuid ).value
    return MPI_SUCCESS

def MPI_Comm_split( self, oldComm, color, key, newComm ):
    pass

# inclRanks is a list of ranks(ints) to include in the new group
# This version of MPI_Group_incl knows the size of the new group
# based on the number of processes in oldGroup
def MPI_Group_incl( self, oldGroup, num, inclRanks, newGroup ):
    rankToUuid = {}
    uuidToRank = {}

    for rank in inclRanks:
        uuid = oldGroup._rankToUuid[rank]
        rankToUuid[rank] = uuid
        uuidToRank[uuid] = rank

    newGroup.setmaps( rankToUuid, uuidToRank )

# Returns the number of processes in group
def MPI_Group_size( self, group, size ):
    size.value = len( group._rankToUuid )
    return MPI_SUCCESS

def MPI_Abort ( comm, errorcode ):
    raise NotImplementedError, 'MPI_Abort not implemented'

def MPI_Get_processor_name (name, resultlength ):
    raise NotImplementedError, 'MPI_Get_processor_name not implemented'
def MPI_Wtime( self ):
    return time.time()
```

```
def MPI_Wtick ():
    raise NotImplementedError, 'MPI_Wtick not implemented'

def MPI_Get_count( self, status, datatype, count ):
    count.value = status.count

def MPI_Finalize( self ):
    self.mpi_inited = False

#
# Send/Recv Engine
#
# mpi_send_buf()
# mpi_recv_buf()
#
# These are used to build all of the MPI communication functions,
# both point to point and collective communication.
#
# Note that mpi_recv_buf() requires an offset parameter.  This
# parameter serves as a "pointer" into the recvbuf, thus avoiding
# a temporary buffer.


# Send large bufs as a series of chunks to reduce memory footprint
def mpi_send_buf(self, buf, offset, count, datatype, dest, tag,
                   comm, typesig):
    destUuid = comm.rankToUuid( dest )
    chunksize = self.mpi_getchunksize(datatype)

    # GDB: fix: compute send size based on datatype and count
    if isinstance(datatype, MPI_Datatype):
        sendsize = count * datatype.size
    else:
        sendsize = count
    DBG('mpi', '[mpi_send_buf] sendsize: %d', sendsize)

    mtype = 'large'
    last = False

    if sendsize <= chunksize:
        mtype = 'small'

    chunkcount = 0
    total = 0
    for c in chop(buf, offset, count, datatype, chunksize):
        chunkcount += 1
        total += len(c)
        if total >= sendsize:
            last = True

        self.send( dest=destUuid, tag=tag, buf=c,
          mtype=mtype, last=last, typesig=typesig )

        if mtype == 'large':
            p = self.recv(src=destUuid, tag=tag, ack=True)

    DBG('mpi', '[mpi_send_buf] sent %d chunks', chunkcount)

# Receive large bufs as a series of chunks to reduce memory footprint
def mpi_recv_buf(self, buf, offset, count, datatype, src, tag,
                   comm, typesig):
    srcUuid = comm.rankToUuid( src )
    g = Glue(datatype, buf, offset)

    chunkcount = 0
    while True:
        p = self.recv(src=srcUuid, tag=tag, typesig=typesig)
        if p.mtype == 'large':
            self.send( dest=srcUuid, tag=tag, ack=True )

        g.glue(p.buf)
        chunkcount += 1
        if p.last:
            break

    DBG('mpi', '[mpi_recv_buf] received %d chunks', chunkcount)

##
## Point to Point Communication Routines
##

# Notes
# - Tags can be and int or mpi_Tag()
# - Perform sanity checks to help programmer

def MPI_Send( self, buf, count, datatype, dest, tag, comm ):
    # Parameter sanity checks
    dest = self.normalize_rank(dest)
    offset = 0

    if isinstance(buf, tuple):
        buf,offset = buf

    if not isinstance(buf, list):
        raise TypeError, 'buf must be a list'

    # derived datatype
    if not isinstance(datatype, MPI_Datatype) and \
       not isinstance(buf[0], mpi_types[datatype]):
        raise TypeError, 'buf type and datatype do not match'

    if len(buf) < count:
        raise ValueError, 'buf length < count'

    # compute type signature
    if isinstance(datatype, MPI_Datatype):
        # TODO: depends on entries in buffer
        typesig = (MPI_DOUBLE, datatype.size)
    else:
        typesig = (datatype, count)

    DBG('mpi', '[send] type signature: %s', typesig)
```

```
        if dest.value != MPI_PROC_NULL.value:
            self.mpi_send_buf(buf, offset, count, datatype, dest,
              tag, comm, typesig)

        return MPI_SUCCESS

    def MPI_Recv( self, buf, count, datatype, src, tag, comm, status ):
        # Parameter sanity checks
        src = self.normalize_rank(src)
        offset = 0

        if isinstance(buf, tuple):
            buf,offset = buf

        if not isinstance(buf, list):
            raise TypeError, 'buf must be a list'

        # derived datatype
        if not isinstance(datatype, MPI_Datatype) and \
           not isinstance(buf[0], mpi_types[datatype]):
            raise TypeError, 'buf type and datatype do not match'

        #if len(buf) < self.getbufsize(count):

        if len(buf) < count:
            print "[MPI_Recv]buf=%s/count=%d" % (buf, count)
            raise ValueError, 'buf length < count'

        # compute type signature
        if isinstance(datatype, MPI_Datatype):
            # TODO: depends on entries in buffer
            typesig = (MPI_DOUBLE, datatype.size)
        else:
            typesig = (datatype, count)

        DBG('mpi', '[recv] type signature: %s', typesig)

        if src.value != MPI_PROC_NULL.value:
            srcUuid = comm.rankToUuid( src )
            self.mpi_recv_buf(buf, offset, count, datatype, src,
              tag, comm, typesig)
            if status != None:
                status.MPI_SOURCE = src
                status.MPI_TAG = tag
                # GDB: change to real receive count
                status.count = len(buf)

        return MPI_SUCCESS

    def MPI_Sendrecv(self, sendbuf, sendcount, sendtype, dest, sendtag,
      recvbuf, recvcount, recvtype, src, recvtag, comm, status):

        dest = self.normalize_rank(dest)
        src  = self.normalize_rank(src)

        if (src.value != MPI_PROC_NULL.value) and \
           (dest.value != MPI_PROC_NULL.value):
            self.MPI_Send(sendbuf, sendcount, sendtype, dest,
              sendtag, comm)
            self.MPI_Recv(recvbuf, recvcount, recvtype, src,
              recvtag, comm, status)

        return MPI_SUCCESS

    ##
    ## Collective Communcation Routines
    ##

    def MPI_Barrier( self, comm ):
        root = MPI_Int(0)

        # Gather then Bcast
        self.MPI_Gather( [0], 1, MPI_INT, [0], 1, MPI_INT, root, comm )
        self.MPI_Bcast( [0], 1, MPI_INT, root, comm )

        return MPI_SUCCESS

    def MPI_Bcast( self, sendbuf, count, datatype, root, comm ):
        root = self.normalize_rank(root)
        msgUid = self.getMsgUid()

        # Root sends to all processes
        if self.uuid == comm.rankToUuid( root ):
            for r in comm.ranks():
                if r != root.value:
                    self.MPI_Send(sendbuf, count, datatype,
                      MPI_Int(r), msgUid, comm)

        # Others receive from root
        else:
            status = MPI_Status()
            self.MPI_Recv(sendbuf, count, datatype, root, msgUid,
              comm, status)

        return MPI_SUCCESS

    # Distribute the contents of root  sendbuf into recvbufs of all the
    # processes
    def MPI_Scatter( self, sendbuf, sendcount, sendtype, recvbuf,
      recvcount, recvtype, root, comm ):
        root = self.normalize_rank(root)

        msgUid = self.getMsgUid()

        # Root sends to everyone
        if self.uuid == comm.rankToUuid( root ):
            for cur in comm.ranks():
                start = cur * sendcount
                end = start + sendcount
                if cur != root.value:
                    self.MPI_Send((sendbuf, start),
                sendcount, sendtype, MPI_Int(cur),
                msgUid, comm)
          else:
              recvbuf[0:recvcount] = \
                sendbuf[start:end]

    # Others receive from root
    else:
        self.MPI_Recv(recvbuf, recvcount, recvtype,
          root, msgUid, comm, MPI_Status())

    return MPI_SUCCESS

# Collect the contents of each procs sendbuf into recvbuf on the proc
# with rank root
def MPI_Gather( self, sendbuf, sendcount, sendtype, recvbuf, recvcount,
  recvtype, root, comm ):
    root = self.normalize_rank(root)
    msgUid = self.getMsgUid()
    destUuid = comm.rankToUuid(root)

    # Everyone sends to root
    if self.uuid != destUuid:
        self.MPI_Send(sendbuf, sendcount, sendtype,
          root, msgUid, comm)
    # Root recvs from everyone in comm
    else:
        for i in comm.ranks():
            start = i * recvcount
            end = start + recvcount
            if i != root.value:
                self.MPI_Recv((recvbuf, start),
                  recvcount, recvtype, MPI_Int(i),
                  msgUid, comm, MPI_Status())
            else:
                recvbuf[start:end] = sendbuf

    return MPI_SUCCESS

def MPI_Allgather( self, sendbuf, sendcount, sendtype, recvbuf,
  recvcount, recvtype, comm ):
    root = MPI_Int( 0 )

    self.MPI_Gather( sendbuf, sendcount, sendtype, recvbuf,
      recvcount, recvtype, root, comm )
    self.MPI_Bcast( recvbuf, recvcount * comm.size(), recvtype,
      root, comm )
    return MPI_SUCCESS

def MPI_Reduce( self, sendbuf, recvbuf, count, datatype, operator,
  root, comm ):
    root = self.normalize_rank(root)

    if not operator in mpi_ops:
        raise TypeError, 'invalid operator'
    op = mpi_ops[operator]
```

```
    size = comm.size()
    rank = comm.uuidToRank(self.uuid)

    # Temporary list buffer for the operands
    temp = [ mpi_zero_values[datatype] ] * (count * size)

    self.MPI_Gather( sendbuf, count, datatype, temp, count,
      datatype, root, comm )


    if rank.value == root.value:
        for i in xrange(count):
            recvbuf[i] = temp[i]
            for j in xrange(1, size):
                recvbuf[i] = op(recvbuf[i],
                  temp[j*count])

    return MPI_SUCCESS

def MPI_Allreduce( self, sendbuf, recvbuf, count, datatype, operator,
  comm ):
    root = MPI_Int( 0 )
    self.MPI_Reduce( sendbuf, recvbuf, count, datatype, operator,
      root, comm )
    self.MPI_Bcast( recvbuf, count, datatype, root, comm )

    return MPI_SUCCESS

def MPI_Reduce_scatter( self, sendbuf, recvbuf, count, datatype,
  operator, comm ):
    root = MPI_Int( 0 )

    reduceTemp = recvbuf * comm.size()
    reduceSize = count * comm.size()

    self.MPI_Reduce( sendbuf, reduceTemp, reduceSize, datatype,
      operator, root, comm )
    self.MPI_Scatter( reduceTemp, count, datatype, recvbuf, count,
      datatype, root, comm )

    return MPI_SUCCESS


def MPI_Alltoall( self, sendbuf, sendcount, sendtype, recvbuf,
  recvcount, recvtype, comm ):

    for i in xrange(comm.size()):
        root = MPI_Int( i )
        tempbuf = [ mpi_zero_values[recvtype] ] * sendcount
        self.MPI_Scatter( sendbuf, sendcount, sendtype,
          tempbuf, recvcount, recvtype, root, comm )
        recvbuf[i*sendcount : (i+1)*sendcount] = tempbuf

    return MPI_SUCCESS
    def MPI_Scan( self, sendbuf, recvbuf, count, datatype, operator, comm ):
        if not operator in mpi_ops:
            raise TypeError, 'invalid operator'
        op = mpi_ops[operator]

        size = comm.size()
        rank = comm.uuidToRank(self.uuid)

        # Temporary list buffer for the operands
        temp = [ mpi_zero_values[datatype] ] * (count * size)

        self.MPI_Allgather( sendbuf, count, datatype, temp, count,
          datatype, comm )

        for i in xrange(count):
            recvbuf[i] = temp[i]
            for j in xrange(1, self.rank.value + 1):
                recvbuf[i] = op(recvbuf[i],temp[j*count])

        return MPI_SUCCESS
```

C.5 rMPI Derived Datatypes [dd.py]

```
# deprived datatypes for rMPI
from mpi import *
from river.core.debug import DBGR as DBG

DDT_FULL_SEND = mpi_robj.next()
DDT_PACK_SEND = mpi_robj.next()
DDT_OVERLAP_SEND = mpi_robj.next()
DDT_SMALL = 4096

class MPI_DD (MPI):
    def MPI_Type_contiguous(self, count, old_type, new_type):
        # old type must exist (either basic or derived)
        # at the same time, calculate number of chunks required
        if isinstance(old_type, MPI_Datatype):
            k = old_type.count * count
        elif mpi_types.has_key(old_type):
            k = 1
        else:
            return MPI_ERR_ARG

        DBG('dd', '[dd:contiguous] at most %d chunks', k)

        # copy chunks from old_type count times
        if isinstance(old_type, MPI_Datatype):
            n = old_type.size
            w = old_type.width
            for i in xrange(count):
                new_type.copy_chunks(old_type, i)

        # one contiguous chunk
        else:
            n = 1
            w = 1
            new_type.add_chunk(0, count)

        new_type.count = len(new_type.chunks)
        new_type.size = n * count
        new_type.width = w * count

        # "register" the new type
        mpi_types[new_type] = mpi_robj.next()

        DBG('dd', '[dd:contiguous] size: %d count: %d width: %d',
          new_type.size, new_type.count, new_type.width)

        return MPI_SUCCESS

    def MPI_Type_struct():
        pass

    # count: number of elements in the type
    # block_length: number of entries in each element
    # stride: num of elements of type old_type between successive elements
    def MPI_Type_vector(self, count, block_length, stride, old_type, new_type):
        # old type must exist (either basic or derived)
        # at the same time, calculate number of chunks required
        if isinstance(old_type, MPI_Datatype):
            k = old_type.count * count * block_length
        elif mpi_types.has_key(old_type):
            k = count * block_length
        else:
            return MPI_ERR_ARG

        DBG('dd', '[dd:vector] at most %d chunks', k)

        if isinstance(old_type, MPI_Datatype):
            n = block_length * old_type.size
            w = block_length * old_type.width
            for i in xrange(count):
                offset = i * stride * old_type.width
                DBG('dd', ' %d %d %d = %d', i, stride, w, offset)
                DBG('dd', '[dd:vector] %d * %d chunks from old',
                  block_length, old_type.count)
                for j in xrange(block_length):
                    new_type.copy_chunks(old_type,
                      offset + j)

        # base type: each element is its own chunk
        else:
            n = block_length
            w = block_length
            for i in xrange(count):
                offset = i * stride
                new_type.add_chunk(offset, block_length)

        new_type.count = len(new_type.chunks)
    new_type.size = n * count
    new_type.width = w

    # "register" the new type
    mpi_types[new_type] = mpi_robj.next()

    DBG('dd', '[dd:vector] size: %d count: %d width: %d',
      new_type.size, new_type.count, new_type.width)

    return MPI_SUCCESS

def MPI_Type_indexed():
    pass

def MPI_Type_commit(self, ddt):
    if not isinstance(ddt, MPI_Datatype):
        return MPI_ERR_ARG

    DBG('dd', '[dd:commit] original size: %d count: %d',
      ddt.size, ddt.count)

    newchunks = []
    for c in ddt.chunks:
        DBG('dd', '%3d [%3d]', c.offset, c.size)

    p = ddt.chunks.pop(0)
    for c in ddt.chunks:
        if p.offset + p.size == c.offset:
            p.size += c.size
        elif c.offset + c.size == p.offset:
            p.offset = c.offset
            p.size += c.size
        else:
            newchunks.append(p)
            p = c
    newchunks.append(p)

    ddt.chunks = newchunks
    ddt.count = len(newchunks)

    DBG('dd', '[dd:commit] new size: %d count: %d',
      ddt.size, ddt.count)

    for c in ddt.chunks:
        DBG('dd', '%3d [%3d]', c.offset, c.size)

    if ddt.count == 1:
        ddt.method = DDT_FULL_SEND
    else:
        if ddt.size <= DDT_SMALL:
            ddt.method = DDT_PACK_SEND
        else:
            ddt.method = DDT_OVERLAP_SEND

def MPI_Pack(self, inbuf, incount, datatype, outbuf, outsize,
  position, comm):
    if isinstance(inbuf, tuple):
        inbuf,base = inbuf
    else:
        base = 0

    if isinstance(outbuf, tuple):
        outbuf,out_off = outbuf
    else:
        out_off = 0

    if not isinstance(inbuf, list) or \
       not isinstance(outbuf, list) or \
       not isinstance(position, MPI_Int):
        return MPI_ERR_ARG

    pos = out_off + position.value
    if not isinstance(datatype, MPI_Datatype):
        outbuf[pos:pos+incount] = inbuf[:incount]
        position.value += incount
        return MPI_SUCCESS

    for i in xrange(incount):
        for c in datatype.chunks:
            if c.size == 0: continue
            offset = base + c.offset
            n = c.size
            outbuf[pos:pos+n] = inbuf[offset:offset+n]
            pos += n
        base += datatype.size

    position.value = pos - out_off

    return MPI_SUCCESS

def MPI_Unpack(self, inbuf, insize, position, outbuf, outcount,
  datatype, comm):
    if isinstance(inbuf, tuple):
        inbuf,in_off = inbuf
    else:
        in_off = 0

    if isinstance(outbuf, tuple):
        outbuf,base= outbuf
    else:
        base = 0

    if not isinstance(inbuf, list) or \
       not isinstance(outbuf, list) or \
       not isinstance(position, MPI_Int):
        return MPI_ERR_ARG

    pos = in_off + position.value
    if not isinstance(datatype, MPI_Datatype):
        outbuf[base:base+outcount] = inbuf[pos:pos+outcount]
        position.value += outcount
        return MPI_SUCCESS
        for i in xrange(outcount):
            for c in datatype.chunks:
                offset = base + c.offset
                n = c.size
                outbuf[offset:offset+n] = inbuf[pos:pos+n]
                pos += n
            base += datatype.size

        position.value = pos - in_off

        return MPI_SUCCESS

    def MPI_Type_hvector():
        pass

    def MPI_Type_hindexed():
        pass

    def MPI_Type_free(self, datatype):
        pass

    def MPI_Type_extent():
        pass

    def MPI_Type_size():
        pass

    def MPI_Type_lb():
        pass

    def MPI_Type_ub():
        pass

    def MPI_Address():
        pass
```

C.6 rMPI Non-blocking [nbmpi.py]

```
import sys, time, math, string, pickle, random
from river.core.vr import VirtualResource
from river.core.debug import *
from river.core.console import *
from mpi import *
from chop import chop, Glue

## Request Type
MPI_REQ_SEND = mpi_robj.next()
MPI_REQ_RECV = mpi_robj.next()
MPI_REQUEST_NULL = mpi_robj.next()

# stage 1. registered
trans_stat_new = "new"
# stage 2. confirmed with post
trans_stat_work = "work"
# stage 3. done
trans_stat_done = "done"

class MPI_Request( object ):
    def __init__( self, *args ):
        self.type = None
        self.source = None
        self.dest = None
        self.tag = 0
        self.count = 0
        self.comm = None
        self.buffer = None
        self.datatype = None
        self.offset = 0
        ## type signature for derived datatype
        self.typesig = None
        self.chops = None
        self.sendsize = 0
        self.gluebuf = None
        self.recvsize = 0
        ## the attrs for TransManager
        self.trans_sender = 0
        self.trans_recver = 0
        self.trans_id = 0
        self.trans_status = trans_stat_new
        self.trans_chunknum = 0

    def __repr__(self):
        output = None
        if self.type == MPI_REQ_SEND:
            output = "[SendRequest] destRank=%s, " % self.dest.value
        elif self.type == MPI_REQ_RECV:
            output = "[RecvRequest] srcRank=%s, " % self.source.value
        return output + "tag=%s, sender=%s, recver=%s, status=%s, chunknum=%d" \
               % (self.tag, self.trans_sender, self.trans_recver,
                  self.trans_status, self.trans_chunknum)


class MPI_NB( MPI ):

    ## override MPI_Init to initiate TransManager
    def MPI_Init( self ):
        if not self.isInited():
            global MPI_COMM_WORLD
            MPI_COMM_WORLD._rankToUuid = self.rankToUuid
            MPI_COMM_WORLD._uuidToRank = self.uuidToRank
            self.mpi_inited = True
            del self.rankToUuid
            self.transManager = TransManager()
        else:
            raise ValueError, 'MPI_Init called previously'

    ## To make blocking operations interact with nonblocking operations
    ## override MPI_Send and MPI_Recv to use TransManager
```

```
def MPI_Send( self, buf, count, datatype, dest, tag, comm ):
    send_request = MPI_Request()

    self.MPI_Isend(buf, count, datatype, dest, tag, comm, send_request)
    self.trans_process_request(send_request)

    return MPI_SUCCESS

def MPI_Recv( self, buf, count, datatype, src, tag, comm, status ):
    recv_request = MPI_Request()

    self.MPI_Irecv(buf, count, datatype, src, tag, comm, recv_request)
    self.trans_process_request(recv_request)

    return MPI_SUCCESS

## Nonblocking P2P operations
def MPI_Isend( self, buf, count, datatype, dest, tag, comm, request):
    offset = 0
    if isinstance(buf, tuple):
        buf,offset = buf

    if not isinstance(buf, list):
        raise TypeError, 'buf must be a list'

    # derived datatype
    if not isinstance(datatype, MPI_Datatype) and \
       not isinstance(buf[0], mpi_types[datatype]):
        raise TypeError, 'buf type and datatype do not match'

    if len(buf) < count:
        raise ValueError, 'buf length < count'

    # compute type signature
    if isinstance(datatype, MPI_Datatype):
        # TODO: depends on entries in buffer
        request.typesig = (MPI_DOUBLE, datatype.size)
    else:
        request.typesig = (datatype, count)

    DBG('mpi', '[send] type signature: %s', request.typesig)


     # Parameter sanity checks
    dest = self.normalize_rank(dest)
    destUuid = comm.rankToUuid( dest )

    if dest.value == MPI_PROC_NULL.value:
        return MPI_ERR_RANK

    chunksize = self.mpi_getchunksize(datatype, count)
    request.type = MPI_REQ_SEND
    # save the rank obj
    request.dest = dest
    request.tag = tag
    request.comm = comm
    request.count = count
    request.buffer = buf
    request.datatype = datatype
    request.offset = offset
    request.chops = chop(buf, offset, count, datatype, chunksize)

    if isinstance(datatype, MPI_Datatype):
        request.sendsize = count * datatype.size
    else:
        request.sendsize = count

    self.transManager.register(request)
    ## send the sendpost
    self.send( dest=destUuid, tag=tag, mtype="sendpost",
      sender=request.trans_sender)

    return MPI_SUCCESS

# MPI_Irecv (&buf,count,datatype,source,tag,comm,&request)
def MPI_Irecv( self, buf, count, datatype, src, tag, comm, request ):
    offset = 0
    if isinstance(buf, tuple):
        buf,offset = buf

    if not isinstance(buf, list):
        raise TypeError, 'buf must be a list'

    # derived datatype
    if not isinstance(datatype, MPI_Datatype) and \
       not isinstance(buf[0], mpi_types[datatype]):
        raise TypeError, 'buf type and datatype do not match'

    if len(buf) < count:
        raise ValueError, 'buf length < count'

    # compute type signature
    if isinstance(datatype, MPI_Datatype):
        # TODO: depends on entries in buffer
        request.typesig = (MPI_DOUBLE, datatype.size)
    else:
        request.typesig = (datatype, count)

    # Parameter sanity checks
    src = self.normalize_rank(src)
    srcUuid = comm.rankToUuid( src )

    if src.value == MPI_PROC_NULL.value:
        return MPI_ERR_RANK

    request.type = MPI_REQ_RECV
    # save the rank obj
    request.source = src
    request.tag = tag
    request.comm = comm
    request.count = count
    request.buffer = buf
    request.datatype = datatype
    request.offset = offset
    request.gluebuf = Glue(datatype, buf, offset)
    self.transManager.register(request)
    ## send the recvpost
    self.send(dest=srcUuid, tag=tag, mtype="recvpost",
      recver=request.trans_recver)

    return MPI_SUCCESS


def MPI_Wait(self, request, status):
    DBG('trans', 'MPI_Wait request= %s', request)

    if request == MPI_REQUEST_NULL:
        return MPI_SUCCESS

    self.trans_process_request(request)

    ## set up status
    if request.type == MPI_REQ_RECV:
        status.MPI_SOURCE = request.source
        status.MPI_TAG = request.tag
        status.count = request.recvsize
    request = MPI_REQUEST_NULL
    return MPI_SUCCESS


def MPI_Waitall(self, count, requests, statuses):
    if len(requests) < count:
        raise ValueError, 'request size < count'
    if len(requests) != len(statuses):
        raise ValueError, 'request size is not equal to status size'

    for i in xrange(count):
        self.MPI_Wait(requests[i], statuses[i])

    return MPI_SUCCESS


# Send chunks on demand
def trans_send_chunk(self, request):
    destUuid = request.comm.rankToUuid( request.dest )

    sendchunk = request.chops.next()

    request.sendsize -= len(sendchunk)
    if (request.sendsize <= 0):
        last = True
        request.trans_status = trans_stat_done
    else:
        last = False

    self.send(dest=destUuid, tag=request.tag, buf=sendchunk, mtype="send",
      last=last, typesig=request.typesig, recver=request.trans_recver)
    request.trans_chunknum += 1

    DBG('trans', 'trans_send_chunk after = %s', request)


## receive buffer from packet and ask for next buffer if there is one
def trans_recv_packet(self, p, request):
    offset = request.offset
    request.gluebuf.glue(p.buf)
    request.recvsize += len(p.buf)

    if p.last == False:
        srcUuid = request.comm.rankToUuid( request.source )
        ## ask for next chunk
        self.send(dest=srcUuid, mtype="recv", tag=request.tag,
          sender=request.trans_sender )
    else:
        request.trans_status = trans_stat_done

    request.trans_chunknum += 1
    DBG('trans', 'trans_recv_packet after = %s', request)


def trans_process_request(self, request):
    if request.trans_status == trans_stat_done:
        ## unregister the request
        self.transManager.unregister(request)
        return

    ## A confirmed send request and the first chunk is not sent yet
    if request.type==MPI_REQ_SEND \
            and request.trans_status == trans_stat_work \
            and request.trans_chunknum == 0 :
        self.trans_send_chunk(request)

    while (request.trans_status != trans_stat_done):
        DBG('trans', 'TransManager = %s', self.transManager)
        p = self.recv(dest=self.uuid,
          mtype=(lambda x: x=='sendpost' or x=='recvpost' \
                         or x=='send' or x=='recv'))
        DBG('trans', 'recv package = %s', p)
        ## source's rank value
        srcRankValue = self.uuidToRank[p.src]
        if p.mtype == "sendpost" :
            ## receive send post
            recvreq = self.transManager.recvready(srcRankValue, p.tag,
              p.sender)

        elif p.mtype == "recvpost":
            ## receive recv post
            sendreq = self.transManager.sendready(srcRankValue,
              p.tag, p.recver)
            ## if sendreq is not registered yet,
            ## this post will be save in transManager.recvqueue
            if sendreq != None:
                # start to send
```

```
                    self.trans_send_chunk(sendreq)

            elif p.mtype == "send":
                req = self.transManager.getreq(p.recver)
                self.trans_recv_packet(p, req)

            elif p.mtype == "recv":
                sendreq = self.transManager.getreq(p.sender)
                self.trans_send_chunk(sendreq)

            else:
                print "trans: unexpected packet=", p

        ## unregister the request
        self.transManager.unregister(request)
        return


class TransManager( object ):
    def __init__( self, *args ):
        self.transmap = {}
        self.rcount = 0
        ## request wait for ACK
        self.newsend = []
        self.newrecv = []
        ## might receive ACK for future requests
        self.sendqueue = []
        self.recvqueue = []

    def __str__(self):
        output = "Requests=%s/ new send=%s, new recv=%s / sendq=%s, recvq=%s " \
          % (self.transmap, self.newsend, self.newrecv, self.sendqueue,
             self.recvqueue)
        return output

    def register(self, request):
        self.rcount += 1
        request.trans_id = self.rcount
        if request.type == MPI_REQ_SEND:
            self.registerSend(request)
        elif request.type == MPI_REQ_RECV:
            self.registerRecv(request)

        self.transmap[request.trans_id] = request
        return

    def registerSend(self, sendreq):
        sendreq.trans_sender = sendreq.trans_id
        DBG('trans', 'registerSend= %s', sendreq.trans_id)

        ## check if vr received ACK already
        for ack in self.sendqueue :
            ## ack = [dest,tag,recver]
            if ack[0] == sendreq.dest.value and ack[1] == sendreq.tag:
                 sendreq.trans_recver = ack[2]
                 sendreq.trans_status = trans_stat_work
                 self.sendqueue.remove(ack)
                 DBG('trans', 'send request is confirmed= %s', sendreq)
                 break

        ## no ack yet, it is a new send request
        if sendreq.trans_status == trans_stat_new :
             self.newsend.append(sendreq.trans_id)

    def registerRecv(self, recvreq):
        recvreq.trans_recver = recvreq.trans_id
        DBG('trans', 'registerRecv= %s', recvreq.trans_id)

        ## check if vr received ACK already
        for ack in self.recvqueue:
            ## ack = [src,tag,sender]
            if ack[0] == recvreq.source.value and ack[1] == recvreq.tag:
                 recvreq.trans_sender = ack[2]
                 recvreq.trans_status = trans_stat_work
                 self.recvqueue.remove(ack)
                 DBG('trans', 'recv request is confirmed= %s', recvreq)
                 break

        ## no ack yet, it is a new recv request
        if recvreq.trans_status == trans_stat_new :
            self.newrecv.append(recvreq.trans_id)

    def sendready(self, dest, tag, recver):
        sendreq = None
        found = False
        ## match the first new send request
        for ns in self.newsend:
            sendreq = self.transmap[ns]
            if sendreq.dest.value == dest and sendreq.tag == tag:
                sendreq.trans_recver = recver
                sendreq.trans_status = trans_stat_work
                found = True
                break

        if not found:
            ## save the post for future send requests
            DBG('trans', 'unknown recvpost: dest=%s, tag=%s', dest, tag)
            self.sendqueue.append([dest,tag,recver])
            sendreq = None
        else:
            DBG('trans', 'send request is confirmed= %s', sendreq)
            self.newsend.remove(sendreq.trans_id)

        return sendreq


    def recvready(self, src, tag, sender):
        recvreq = None
        found = False
        for ns in self.newrecv:
            recvreq = self.transmap[ns]
            ## match the first new recv request
            if recvreq.source.value == src and recvreq.tag == tag:
                recvreq.trans_sender = sender
                recvreq.trans_status = trans_stat_work
                found = True
                break

        if not found:
            ## save the post for future recv requests
            DBG('trans', 'unknown sendpost: src=%s, tag=%s', src, tag)
            self.recvqueue.append([src,tag,sender])
            recvreq = None
        else:
            DBG('trans', 'recv request is confirmed= %s', recvreq)
            self.newrecv.remove(recvreq.trans_id)

        return recvreq


    def getreq(self, trans_id):
        return self.transmap[trans_id]

    def unregister(self, request):
        try:
            del self.transmap[request.trans_id]
        except KeyError:
            pass
```

C.7 rMPI Optimized Collectives [optmpi.py]

```
import sys, time, math, string, pickle, random
from river.core.vr import VirtualResource
from river.core.debug import *
from river.core.console import *
from nbmpi import *

## inherit from nonblocking mpi
## Optimized Collective operations
class MPI_OPTCC( MPI_NB ):

    """ the dissemination algorithm
    It uses ceiling(lgp) steps. In step k, 0 <= k <= (ceiling(lgp)-1),
    process i sends to process (i + 2ˆk) % p and receives from process
    (i - 2ˆk + p) % p.
    """
    ## override MPI_Barrier
    def MPI_Barrier( self, comm ):
        sendbuf = [1]
        recvbuf = [0]

        msgUid = self.getMsgUid()
        status = MPI_Status()

        i = self.rank.value
        p = comm.size()
        steps = int(math.ceil(math.log(p, 2)))
        for k in xrange(steps):
            dest = (i + 2**k) %p
            src = (i - 2**k + p) %p
            recv_request = MPI_Request()
            self.MPI_Irecv(recvbuf, 1, MPI_INT, MPI_Int(src), msgUid, comm,
              recv_request)

            self.MPI_Send(sendbuf, 1, MPI_INT, MPI_Int(dest), msgUid, comm)

            self.MPI_Wait(recv_request, status)

        return MPI_SUCCESS

    def MPI_Reduce_Rabens( self, sendbuf, recvbuf, count, datatype, operator,
                           root, comm ):
        root = self.normalize_rank(root)
        # use Recursive Halving Reduce Scatter
        self.MPI_Reduce_scatter_RecHalv( sendbuf, recvbuf, count, datatype,
          operator, comm )
        # binomial gather
        MPI_Gather_Binomial(sendbuf, sendcount, sendtype, recvbuf, recvcount,
          recvtype, root, comm)


    def MPI_Gather_Binomial( self, sendbuf, sendcount, sendtype,
                             recvbuf, recvcount, recvtype, root, comm ):
        root = self.normalize_rank(root)
        size = comm.size()
        rank = comm.uuidToRank(self.uuid)

        status = MPI_Status()
        msgUid = self.getMsgUid()

        # number of steps
        steps = int(math.ceil(math.log(size, 2)))
        # in case root is not 0
        taskRankValue = (rank.value-root.value)%size

        recvbuf[rank.value*sendcount:(rank.value+1)*sendcount] = sendbuf

        for step in xrange(steps):
            taskPair = taskRankValue ˆ (2 ** step)
            # real rank of the pair node
            pair = (taskPair+root.value)%size

            if taskPair >= size:
                # pair doesn't exist
                continue

            count = 2**step * sendcount
            if taskRankValue > taskPair:
                sendnode = rank.value
            else:
                sendnode = pair
        if (sendnode*sendcount + count) > len(recvbuf):
            count = len(recvbuf) - sendnode*sendcount
        sendIndex = sendnode * sendcount

        if rank.value == sendnode:
            ## send data
            self.MPI_Send((recvbuf, sendIndex), count, sendtype,
              MPI_Int(pair), msgUid, comm)
            ## once data is sent, the node is not in computation anymore
            break;
        else:
            ## recv data
            self.MPI_Recv((recvbuf, sendIndex), count, sendtype,
                          MPI_Int(pair), msgUid, comm, MPI_Status())

    return MPI_SUCCESS


def MPI_Reduce_Binomial( self, sendbuf, recvbuf, count, datatype, operator,
                         root, comm ):
    root = self.normalize_rank(root)

    if not operator in mpi_ops:
        raise TypeError, 'invalid operator'
    op = mpi_ops[operator]

    size = comm.size()
    rank = comm.uuidToRank(self.uuid)

    #copy sendbuf to recvbuf
    recvbuf[:] = sendbuf[:]
    # Temporary list buffer for the operands
    temp = [ mpi_zero_values[datatype] ] * count
    status = MPI_Status()
    msgUid = self.getMsgUid()

    # number of steps
    steps = int(math.ceil(math.log(size, 2)))
    # in case root is not 0
    taskRankValue = (rank.value-root.value)%size

    for step in xrange(steps):
        taskPair = taskRankValue ^ (2 ** step)
        # real rank of the pair node
        pair = (taskPair+root.value)%size
        if taskPair >= size:
            # pair doesn't exist
            continue
        elif taskRankValue > taskPair:
            ## send data
            self.MPI_Send(recvbuf, count, datatype, MPI_Int(pair),
              msgUid, comm)
            ## once data is sent, the node is not in computation anymore
            break;
        else:
            ## recv data
            self.MPI_Recv(temp, count, datatype, MPI_Int(pair), msgUid,
              comm, status)
            ## do the operation
            for i in xrange(count):
                recvbuf[i] = op(recvbuf[i],temp[i])

    del temp
    return MPI_SUCCESS


## ToDo: use nonblocking p2p instead
def MPI_Reduce_scatter_RecHalv( self, sendbuf, recvbuf, count, datatype,
                                operator, comm ):
    ## recursive halving
    ## this only works with power-of-2 processors
    size = comm.size()
    power2 = math.floor(math.log(size, 2))
    steps = int(math.ceil(math.log(size, 2)))
    if power2 != steps :
        return MPI_ERR_OTHER

    if not operator in mpi_ops:
        raise TypeError, 'invalid operator'
    op = mpi_ops[operator]

    msgUid = self.getMsgUid()
    status = MPI_Status()

    ## only allocate tempbuf once
    tempbuf = [ mpi_zero_values[datatype] ] * (size/2 * count)
    for step in xrange(steps):
        ## halving the distance
        distance = size/2**(step+1)
        # the node to exchange data
        # use XOR to find the pair node
        pairNode = self.rank.value ^ distance
        # sendIndex depends on the pair node's "group"
        sendIndex = (pairNode/distance) * count
        sendSize = distance * count

        if self.rank.value < pairNode :
            self.MPI_Send((sendbuf, sendIndex), sendSize, datatype,
              MPI_Int(pairNode), msgUid, comm)
            self.MPI_Recv(tempbuf, sendSize, datatype, MPI_Int(pairNode),
              msgUid, comm, status)
        else:
            self.MPI_Recv(tempbuf, sendSize, datatype, MPI_Int(pairNode),
              msgUid, comm, status)
            self.MPI_Send((sendbuf, sendIndex), sendSize, datatype,
              MPI_Int(pairNode), msgUid, comm)
        # print "tempbuf=", tempbuf
        # perform the predefined operation
        # operation index also depends on the node's "group"
        optIndex = (self.rank.value/distance) * distance * count
        for i in xrange(sendSize):
            sendbuf[optIndex+i] = op(sendbuf[optIndex+i], tempbuf[i])
        ## halving the tempory buffer
        del tempbuf[(sendSize/2):]

    ## recvbuf is one chunk of sendbuf
    recvbuf[:] = sendbuf[self.rank.value*count : (self.rank.value+1)*count]

    return MPI_SUCCESS


def MPI_Alltoall_Bruck( self, sendbuf, sendcount, sendtype,
                        recvbuf, recvcount, recvtype, comm ):
    size = comm.size()
    rank = comm.uuidToRank(self.uuid)

    status = MPI_Status()
    msgUid = self.getMsgUid()

    ## local ratation
    recvbuf[:] = sendbuf[rank.value*sendcount:] + \
                 sendbuf[:rank.value*sendcount]

    steps = int(math.ceil(math.log(size, 2)))
    for step in xrange(steps):
        destNode = (rank.value + 2 ** step) % size
        srcNode = (rank.value - 2 ** step) % size

        sendIndex = 2**step * sendcount
        sendSize = 2**step * sendcount
        sendchunk = []
        while (sendIndex<len(sendbuf)):
            if (sendIndex+sendSize > len(sendbuf)):
                sendSize = sendIndex+sendSize - len(sendbuf)
            send_request = MPI_Request()
            recv_request = MPI_Request()
            ## it is not safe to use recvbuf for both Isend and Irecv
            self.MPI_Isend((recvbuf, sendIndex), sendSize, sendtype,
              MPI_Int(destNode), msgUid, comm, send_request)
            self.MPI_Irecv((recvbuf, sendIndex), sendSize, recvtype,
              MPI_Int(srcNode), msgUid, comm, recv_request)

            self.MPI_Wait(send_request, status)
            self.MPI_Wait(recv_request, status)

            sendIndex = sendIndex + 2**(step+1) * sendcount

    ## inverse rotation
    recvbuf.extend(recvbuf[:(rank.value+1)*sendcount])
    del recvbuf[:(rank.value+1)*sendcount]
    recvbuf.reverse()

    return MPI_SUCCESS


def MPI_Allgather_Dissem( self, sendbuf, sendcount, sendtype, recvbuf,
                          recvcount, recvtype, comm ):
    msgUid = self.getMsgUid()
    size = comm.size()
    power2 = math.floor(math.log(size, 2))
    steps = int(math.ceil(math.log(size, 2)))
    statuses = [MPI_Status(), MPI_Status()]

    ## copy self sendbuf to end of recvbuf
    ## recvbuf is filled upward
    recvbuf[(size-1)*sendcount:] = sendbuf
    for step in xrange(steps):
        if (step == power2):
            ## the last stage for non-power of 2
            sendIndex = 2**step * sendcount
            recvIndex = 0
            sendcnt= (size - 2**step) * sendcount
        else:
            sendIndex = (size - 2**step) * sendcount
            recvIndex = (size - 2**(step+1)) * sendcount
            sendcnt = (2**step) * sendcount

        destNode = (self.rank.value + 2**step) % size
        srcNode = (self.rank.value - 2**step) % size

        send_request = MPI_Request()
        recv_request = MPI_Request()

        ## send continguous
        self.MPI_Isend((recvbuf, sendIndex), sendcnt, sendtype,
          MPI_Int(destNode), msgUid, comm, send_request)
        self.MPI_Irecv((recvbuf, recvIndex), sendcnt, recvtype,
          MPI_Int(srcNode), msgUid, comm, recv_request)

        self.MPI_Waitall(2, [send_request, recv_request], statuses)

    ## change the order
    ## last rank will be in order
    if self.rank.value < (size-1):
        ## move one chunk each time
        for i in xrange(size - self.rank.value - 1):
            recvbuf.extend(recvbuf[:sendcount])
            del recvbuf[:sendcount]

    return MPI_SUCCESS


def MPI_Allgather_Bruck( self, sendbuf, sendcount, sendtype, recvbuf,
                         recvcount, recvtype, comm ):
    msgUid = self.getMsgUid()
    size = comm.size()
    power2 = math.floor(math.log(size, 2))
    steps = int(math.ceil(math.log(size, 2)))
    statuses = [MPI_Status(), MPI_Status()]
    ## copy self sendbuf to recvbuf
    recvbuf[0:sendcount] = sendbuf
    ## always send from first one
    sendIndex = 0
```

```
for step in xrange(steps):
    recvIndex = 2**step * sendcount
    if (step == power2):
        ## the last stage for non-power of 2
        sendcnt= (size - 2**step) * sendcount
    else:
        sendcnt = 2**step * sendcount
    destNode = (self.rank.value - 2**step) % size
    srcNode = (self.rank.value + 2**step) % size
    send_request = MPI_Request()
    recv_request = MPI_Request()

    self.MPI_Isend((recvbuf, sendIndex), sendcnt, sendtype,
      MPI_Int(destNode), msgUid, comm, send_request)
    self.MPI_Irecv((recvbuf, recvIndex), sendcnt, recvtype,
      MPI_Int(srcNode), msgUid, comm, recv_request)

    self.MPI_Waitall(2, [send_request, recv_request], statuses)

## change the order
if self.rank.value > 0:
    for i in xrange(size - self.rank.value):
        recvbuf.extend(recvbuf[:sendcount])
        del recvbuf[:sendcount]

return MPI_SUCCESS
```