THESIS

ON THE SUPPORT FOR HETEROGENEOUS LANGUAGES IN CLOUD RUNTIMES

Submitted by

Kathleen Ericson

Department of Computer Science

In partial fulfillment of the requirements

For the degree of Master of Science

Colorado State University

Fort Collins, Colorado

Fall 2010

Masters Committee:

Department Chair: L. Darrell Whitley

Advisor: Shrideep Pallickara

A. P. Willem Bohm
David Randall

ABSTRACT

ON THE SUPPORT FOR HETEROGENEOUS LANGUAGES IN CLOUD RUNTIMES

Cloud runtimes are an effective method of distributing computations, but often have little support for computations written in diverse languages. We have extended the Granules cloud runtime with a bridge framework that allows computations to be written in a number of languages. Granules computations are dynamic and can be characterized as long-running with intermittent CPU bursts, allowing state to build during successive rounds of execution. Our goal is to develop a framework that supports real-time processing in long-running computations that maintain state across multiple runs of the computation. Due to the nature of Granules computations, we need the bridges to be bidirectional – both Granules and the bridged computation should be able to steer the program flow as needed. In order to conserve resources, and maintain communications during heavy loads, the framework needs to allow communication over multiple channels, and be able to switch the bridging mechanism in a transparent manner. Different communication methods should be available to a computation at all times, without requiring rewrites of an original computation. Our current implementation supports bridging in C, C++, C#, Python, and R. We have also designed a diagnostics system, which gathers information on system state and is able to modify the underlying bridge framework in response to system load. This diagnostics system is capable of initiating a switch of communications methods transparently, which allows the system to free up limited resources as necessary.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1 INTRODUCTION

Cloud runtimes with their support for orchestrating computations across multiple machines have gained significant traction in the past few years. Though the cloud runtime may be developed in a specific programming language, developers often wish to run computations already developed in a different language in the cloud. In this thesis, we describe our support for heterogeneous languages within the Granules cloud runtime.

Granules [1, 2] is a lightweight runtime for cloud computing and is designed to orchestrate millions of computations on a set of available machines. Granules supports both the MapReduce paradigm [3] and graph-based program flow [4]. Granules allows computations build state across multiple rounds of computation. Each computation has a finite state machine associated with them – computations may change state depending on the availability of data on any of their input datasets or as a result of external triggers. When the processing is complete, computations can become dormant waiting for more data.

In Granules, computations specify a scheduling strategy, which governs their lifetimes. A developer schedules computations along three dimensions: number of runs, data availability and time. A computation can be limited to running a certain number of times, as data is available, or periodically at a set interval. While any one of these axes can be extended into infinity, one can also specify a custom scheduling strategy that is a combination along these three dimensions. For example, a developer may specify that a computation should run in response to incoming data from a sensor – but should also be able to take action if it does not get sensor data as expected. Additionally, it would make sense to limit the maximum number of times it tries to work without data before deciding the sensor is dead. For example, the scheduling strategy could state that the

computation should run as data is available, or every 5 minutes. The computation should go through no more than 100 iterations, about 8 hours in this case, without receiving data from a sensor.

A computation can change its scheduling strategy during execution, and the new strategy will take effect the next round of execution. This scheduling change can be significant: for example, switching from one axis to another such as changing from running periodically to running as data is available; or the scheduling change may be minor: such as updating the number of times to run, or changing the frequency of a computation scheduled to run periodically. By allowing computations to enter a dormant state when scheduling conditions have not been met, Granules can successfully interleave thousands of computations on a single resource.

Computations in Granules are able to build state over successive executions of computations. While CPU usage for various computations may occur in short bursts – lasting between seconds and minutes – Granules computations can remain active for months, retaining state across thousands of runs.

Granules is currently being deployed in a variety of fields, including earthquake science, epidemiological simulations, and brain-computer interfaces [5]. The profiles of these applications range from requiring real-time support, to very long-running computations.

The objective of this work is to allow Granules, which is Java based, to interact with computations developed in other languages. This would allow Granules to be deployed in many other fields, specifically areas which have a large base of computations already written in different languages. This satisfies the occasion where there is either no time to convert the base code as well as situations where Java is not the ideal language for computations. There were four main design goals in developing Granules Bridges.

- **Optimize the utilization of resources on a given machine.** Granules often interleaves up to 10000 computations on a single resource concurrently. Using only socket-based communications would tie up limited resources – there can be up to maximum of 64K TCP

sockets on a given host. Alternatively, relying extensively on files can introduce an I/O bottleneck that may impact the performance of computations that operate on datasets stored on disks.

- **Minimize the overheads for communications with computations written in other languages.** The communications overhead of the bridge should not be high enough to prevent real-time applications written in other languages.
- **Support for bidirectional steering**. Computations in Granules are generally long running, and will need to stay in communication throughout their lifetime. It is important that both Granules and the Bridged computation can take the initiative to start communications.
- **Communication mechanisms should be able to evolve over time**. Applications are long-running, and it can be expected that the resources available to an application will change over the lifetime of a typical Granules computation. We want to make sure that a computation can change its underlying communication mechanisms in response to these changes.

The Granules Bridge framework provides a mechanism for developers to bridge computations designed in diverse languages. Computations use the bridge to transfer information about state transitions, input datasets, results of the processing, and any errors/exceptions that occurred during the processing. We support incorporation of different communication mechanisms across bridges. Currently, our bridges to C, C++, C# and Python can communicate via pipes. C, C++, C#, Python and R can additionally communicate using TCP. Support for datagram sockets and shared memory is ongoing. The rationale for supporting this feature is that Granules can collect information from each round of execution for a computation with bridging, and if multiple communication mechanisms are supplied, Granules can use this information to determine the most effective medium for communications with a bridged computation.

1.1 Challenges

There are three challenges addressed in this work:

Challenge 1: Bridge communication semantics should be independent of the mechanism to implement them

Computations in different languages can use named pipes, unnamed pipes, sockets, or shared memory for communications with each other. We need to keep the communication protocol general enough to use across all communications mediums. The protocol needs to be abstract enough to be applicable to all communication formats, yet strict enough to allow meaningful messages to be exchanged. This allows us to introduce additional channels without having to recode the semantics of the data exchanged over the channel. An additional requirement here is for these communications protocols to be lightweight while introducing acceptable overheads.

Challenge 2: Account for resource usage at individual machines.

Granules is designed to support data driven computations. These are computations where the scheduling strategy dictates execution when data is available on one of their input streams, and held dormant otherwise. Although the CPU bound processing time for individual packets of a data stream may only be milliseconds, the computations are long running in the sense they are scheduled for multiple rounds of execution when these packets are generated over a prolonged duration by a data source such as a sensor. For example, one of our benchmarks involves a Brain Computer Interface application where the user's EEG data streams could be produced continually. Since all computations are not active at all times, Granules interleaves a large number of such computations on the same resource to maximize resource utilizations while processing streams.

Over time the availability of resources and the accompanying performance overheads associated with using them change. Bridges to other languages need to account for such changes. For example, (1) when a large number of processes use disk I/O for communications contentions will result in reduced response times, (2) if the number of sockets being used increases

substantially configured OS thresholds would be breached resulting in errors, (3) if shared memory is being used exclusively for communications it would result in reduced memory for applications that need them too. Since system conditions change dynamically, the framework must respond to these changes autonomously and transparently to ensure sustained system throughput.

Challenge 3: Support reusability.

Once a bridge to a language has been developed, this bridge functionality should be accessible to all computations written in that language. The framework needs to be reusable across any computations in the bridged language without requiring rewrites. Existence of a bridge to a language should imply that developing bridged computations in that language should be just as simple as developing those in the runtime's native language. In object oriented terms, the bridge should be a base class that implements all functionality expected of computations in the native language. This base class could then be extended to meet the needs of computations as necessary.

1.2 Contributions

This thesis makes the following contributions.

- **Broad applicability:** Though this framework was developed for a specific runtime, there is nothing here that would preclude its applicability in systems that need to incorporate support for other languages. As a proof of concept, we have used Granules Bridges through Hadoop to bridge to Python. This work has been released as part of the Granules runtime, allowing users to adapt and modify the bridging framework to meet their needs.
- **Suitability to data driven computations:** Data driven computations execute in rounds, with periods of dormancy while waiting for more data. The adaptive behavior of Granules Bridges allows the underlying channel of communication to be changed in between these rounds of computations.
- **Support for multiple languages:** We have incorporated support for different bridging mechanisms to C, C++, C#, Python and R. Based on our success with these disparate

languages, we should be able to bridge to almost any language with the current bridging framework. Bridging across languages allows the runtime to orchestrate computations developed in different languages. It is possible to additionally chain computations with these bridges – such as communicating between Python and R through C. We have not yet benchmarked the costs of doing so.

- **Responsiveness to varying system conditions:** The framework is lightweight and relies on various diagnostics to autonomously tune communication mechanisms based on specified directives. The framework needs to be able to respond quickly to environment changes, even while a computation may be in a dormant state.

The remainder of this thesis is organized as follows: Chapter 2 describes related work and Chapter 3 discusses the bridging framework in detail. In Chapter 4 we introduce several experiments designed to gather baseline performance information, then move on to several case studies which include more complex computations in Chapter 5. Finally, we conclude and discuss directions for future work in Chapter 6.

Chapter 2 RELATED WORK

Purpose-specific wrappers are the general approach to allowing communication between programming languages on a single machine. While such wrappers can be optimized to take advantage of this specialization, it also means that the wrapper is not generally reusable. A slightly different application would need its own specific wrapper, and the usefulness of existing wrappers can be overwhelmed by the problems incurred when trying to update the program which uses the wrapper.

The Java Native Interface (JNI) [6] (http://java.sun.com/docs/books/jni/html/intro.html#994) is a framework that allows Java programs to interact with C/C++ or assembly programs at a basic level on a single machine. Downsides to this approach include (1) the introduction of instability to the JVM, (2) a loss of portability of Java code, and (3) the learning curve necessary to create stable code in JNI. This framework is also limited in what it supports – there is no support for interpreted languages such as Python or R, and it does not seem to support other compiled languages such as C#.

Closely related to JNI is Java Native Access (JNA) (https://jna.dev.java.net/). The JNA framework allows a developer to access system level code written in C, Windows dlls, as well as Jython [7] and JRuby (http://jruby.org/) programs. While JNA claims to be simpler, it has been reported to run approximately 100 times slower than equivalent JNI code.

CORBA [8] (http://www.omg.org/gettingstarted/corbafaq.htm) has been developed to handle communication between languages. While CORBA was designed to work among networked machines, it is possible to use it for intra-machine communication as well. A downside of using

CORBA, however, is the need to create stubs and skeletons that need to be traversed during all communications. Additionally, the CORBA network protocol can become bulky with all the needed information to appropriately run a command.

An alternative approach for communications is through the use of XML [9]. XML allows for the development of a complex, extensible and self-descriptive language for communication. With XML, we would be able to build a communication framework between programs in a manner similar to SOAP [10]. XML does, however, add a considerable overhead cost to all communication for parsing. We would also still either need to write XML to a pipe or send it across a network as packets – much as in our current solution. One advantage of XML over our current bridge implementation is that XML is human readable. While this could help with debugging, all our message packets would become considerably larger, impacting all communications overhead.

The Simplified Wrapper and Interface Generator (SWIG) [11], is generally used to connect C/C++ code to other languages. It is used by Hadoop [12] to handle communication between its own Java-based operations and functions written in different languages. SWIG is essentially a code generator for C/C++ programs which generates the code necessary to communicate between programs. It does not provide a protocol, or enforce coding guidelines – leading to potentially unsafe handling of communications.

The Java R Interface (JRI) [13] is used exclusively to handle communications between Java and R programs. Its sister project rJava [14] allows Java objects to be called and manipulated from R programs. While these packages provide a fully functional method of bridging between Java and R, neither is capable of extending to any other language – both JRI and rJava are built exclusively for R and Java.

Granules bridging defines a protocol for designing communication links between programs. It does not generate code, and developers are expected to implement computation specific wrappers. Basic wrappers have been included in the latest Granules release, which should make this

implementation task simpler. While this does require a developer to write code – unlike approaches such as SWIG which have code generators, there is no need for a developer to learn a new language to handle code generation. We expect the Granules Bridge to be safer and more reliable than approaches such as SWIG, as it provides stronger typing, as well as defining the protocol for communication with any language – not simply C and C++.

The major difference between Granules Bridges and the other methods of communication bridging discussed in this section is that Granules supports the definition of multiple bridging mechanisms, as well as offering the ability to switch between these methods of communication. When Granules bridges are used within the Granules framework, the diagnostics system can be leveraged. This system not only monitors system state to keep track of limited resources such as memory and sockets, but also can gather statistics on computations, and store information about the performance of different types of bridges. This allows the system to adapt to changes in the system state based off of what it knows about the behavior of a computation.

Chapter 3 THE BRIDGE FRAMEWORK

Granules bridges are designed to provide a link between computations written in different programming languages on a single machine. The bridging framework has been built to accommodate communications through messages encoded as byte arrays. This allows for more flexible communications, as different languages can have very different methods of encoding data such as strings.

Granules has been designed primarily to handle streaming data from sensors, and Granules Bridges are designed around this assumption. Environmental data sensors generally produce data in a stream of 2-8 KB packets. Brain Computer Interfaces (BCI) data packets can be double this size (depending upon the time window being sent as well as the resolution of the recording), but are still not particularly large. In short, Granules bridges are neither designed nor expected to handle transmissions of data in the order of GBs of data.

Granules bridges have a need for an extra layer of complexity above a simple call out from Java to an external application. Our bridges need to allow either side to steer computation: both sides need to be able to initialize communications and steer the process over time. This means that both sides of a bridge need to be actively listening for communications in order to respond promptly. All basic communication classes needed to set this up have been developed as a part of Granules, and can be extended as needed for more complex communications.

3.1 Bridge Design

There are several classes involved in creating a new Granules bridge. Granules provides several classes to the developer for the Java side: `JavaByteMessage`, `StreamReader` and `StreamWriter` to handle binary communication, as well as a `StreamStringReader` and the `JavaMessageHandler` interface to manage incoming messages. Communication with a computation is managed by the extendable `JavaGenericComputation`, which is also generic enough to handle basic computations as is. These classes simplify development of wrappers to handle communications with computations written in another language, as well as provide a guideline of the basic tools necessary for full bridging capabilities in other languages.

3.1.1 `JavaByteMessage`

The backbone for bridged communication is the `JavaByteMessage`. This needs to be implemented in both Java and the languages the bridge is designed to span to. The `JavaByteMessage` communication protocol is rigid enough to make implementations relatively simple and straight forward, yet flexible enough to handle bridging to diverse languages and program types. This flexibility is most apparent in the input and output byte arrays – it is left up to the developer to decide the most appropriate method of using these fields. The only assumptions made about these fields is that either side of the bridge has a specialized piece of code to correctly interpret these fields.

```
private int controlMessageType;
private byte [] input;
private byte [] output;
private String streamIdentifier;
private String description;
private boolean isFromJava;
```

This simple communication protocol allows arbitrary messages to be sent back and forth between a Granules based application and a bridged computation. `JavaByteMessages` are converted to byte arrays to be sent across the bridge, and are always preceded with an integer describing the length of the converted `JavaByteMessage`, allowing multiple languages to correctly read the entire message from the bridge. While we have developed several sample programs to illustrate possible uses of the Granules Bridge, it has been designed to be easily extended as the need arises. In particular, a basic `JavaGenericComputation` class has been developed to handle diverse types of computations, while maintaining the ability to function correctly without extension.

The structure of a `JavaByteMessage` was designed to be both flexible, yet strongly defined so that messages could be marshaled/unmarshaled in languages without reflection. The first field is an integer describing the message type. This is a predefined enumeration of the different types of messages which might be sent. The current list includes: START_STOP, INITIALIZE, DATA, RESULTS, PAUSE_RESUME, STATUS, COMPLETE, ABORT and CHANGE_BRIDGE_TYPE. Again, this is extendable, so developers are able to add new message types as the need arises.

Both input and output are simple byte arrays, essentially configurable as needed – the only necessary constant is reserving the first 4 bytes of each to store the length of the rest of the array. This allows diverse languages to read this field correctly. While these four bytes are reserved in order to properly reconstruct the arrays, the rest of these arrays are free-form, and it is up to the developer to design an appropriate scheme for the use of these fields.

In our examples, we have followed a basic format for the input and output fields. After the initial 4 bytes for length, for the rest of the array the following pattern holds: 4 bytes declaring the type of the data stored – this is an enumerated value, followed by the actual data. For example

when a string is being sent, the data contains 4 bytes for length, then the actual string. The same approach is used when passing arrays: 4 bytes for length, followed by the array contents.

The `streamIdentifier` is necessary for the external program to be able to identify the Granules resource a message has come from. In some cases, several Granules resources may be interacting with the same base computation. The `description` field is used to pass along extra information about a message, making it a vital field for debugging and logging purposes, when a little extra information about the message can be used to monitor a computation's progress.

To help keep track of message direction, the `isFromJava` field was introduced. This flag is again primarily useful for logging or debugging program execution as it helps to keep track of message flow.

3.1.2 `JavaGenericComputation`

A `JavaGenericComputation` is responsible for starting up the connection to the external computation, as well as handling the various methods of communication. This is the link between Java and the computation, and how a Granules application actually transmits `JavaByteMessages`. A `JavaGenericComputation` is expected to know about various available methods of communication (named pipes, unnamed pipes, TCP sockets, etc.), and be able to orchestrate the actual switching of bridging channels through a series of handshakes in a transparent manner. The Granules application should not need to be aware that a communications switch is occurring.

3.1.3 Readers and Writers

To handle a variety of potential methods of communication, all readers and writers developed for Granules bridges are all designed to interact with the base Java `InputStreams` and `OutputStreams`. This means that the same backend is capable of handling communications across a variety of media – it only needs to have a stream provided.

For a Granules application to receive `JavaByteMessages`, a `StreamReader` needs to be instantiated. A `StreamReader` needs both an `InputStream` and a reference to an implementation of `JavaMessageHandler`. The `StreamReader` needs this reference to be able to pass off a `JavaByteMessage` appropriately after it has been read in fully. This allows `StreamReaders` to be flexible, and fully decoupled from `JavaByteMessage` decoding. This simple `StreamReader` is capable of being used across all bridging formats.

`StreamWriters` are responsible for accepting `JavaByteMessages` and sending them across its internal `OutputStream` as a byte array. It first needs to call the `JavaByteMessage toByteArray()` function to turn the message into a byte array, finds the length of the array, sends the length across the stream, and then finally sends the actual array.

In our experiments, we have found that these three utilities are general enough to be used for all computations we have tried. Theoretically, they should be able to handle any type of computation. From our own tests with C, C++, C#, Python and R, these classes did not need to be modified or extended to achieve fully functioning bridging behavior between Java and the computation.

3.1.4 `JavaMessageHandler`

Included in the Granules bridging code is the `JavaMessageHandler` interface. This interface has only a single method which needs to be implemented: `handleMessage(JavaByteMessage)`. It is up to the developer to properly implement this code for the computation they are working with, and is not inherently threaded. The developer is expected to add this functionality if needed and is strongly encouraged to do so. The *`JavaMessageHandler`* can be as simple or complex as the developer wishes it to be. At the most basic level, it needs to be able to accept a `JavaByteMessage`, and perform whatever action necessary with it.

3.2 Diagnostics and Directives For Adaptive Communications

For Granules implementations of the bridge, we have developed a suite of programs which allows basic diagnostics on a bridge method to be gathered. These diagnostics can then be used to create a program profile which describes the processing footprint generated by programs over all methods of communication it has used. These profiles can then in turn be used when enforcing policies. While this diagnostics system is closely tied into the Granules environment, it is still possible for a user to develop programs with the ability to switch underlying communication methods for a computation without using the Granules diagnostics system.

Currently, we are using developer generated policies in order to obtain adaptive behavior. The system should be able to modify the underlying communication channel based on the current system state as well as any requirements of the computations currently running. In future versions, we plan to incorporate a learning component which will be capable of developing new policies as well as evolving existing policies based on past behavior. We also plan to include an interface which allows developers to specify program directives at compilation.

Directives are the basic unit of a policy, which drives the adaptive behavior of the system. Directives can define behavior for a given system state, or instead place constraints on how a computation may be run. System directives are those that pertain specifically to machine state, without specific details about computations. For example: given a 90% usage of all available sockets, an effective system directive may be to reroute any currently running computation which has an I/O ratio of less than 60% to piped communication, helping ensure that the machine does not run out of sockets. Another system directive may state that if less than 10% of available sockets are being used, all new computations should be automatically started with a socket-based bridge. An important note about system directives is that the current system does not perform checking: it is possible for a user to declare conflicting directives, and force the system to constantly switch between communications methods.

User policies differ from system policies in that they refer only to computations. They can be designed specifically for an individual computation, or specify behavior for all computations on a given machine. For example, a user may be setting up a policy for a BCI application. These applications need to be able to respond to user input in real time. If Electroencephalogram (EEG) data is sent out every second, a reasonable directive could state that it should take no longer than 250 ms for the computation to return a result. The monitoring system is then responsible for ensuring that an appropriate method of communication is being used to meet this directive. A machine-wide directive may state that computations with a high rate of I/O and a low processing overhead should be given a higher priority when scheduling runs. We have implemented this policy for testing.

In order to test the capability and flexibility of this last policy, we have developed several benchmarks, described in Chapter 4 which have been designed to detect any complications in profile effectiveness as well as the ability of profiles to detect situations in which a directive needs to be enforced. There are four main components needed for adaptive communications:

3.2.1 Resource and Program Diagnostics

The Granules diagnostics system is responsible for monitoring system state, as well as keeping track of profiles for individual computations. This data is gathered at the resource or machine level, and can be used in subsequent runs to determine the most efficient method of bridging. When starting the computation, a user will have the option of either specifying a particular method of communication to use, or specifying a policy that should be used to determine the communication method. Policies dictate when the system should switch communication methods – for example, a system policy may state that if the socket usage is over a given percentage, any new bridge communications should be initialized over pipes.

A Granules resource is responsible for monitoring the state of the machine it is running on. This data will be used both for immediate consultation on job runs, as well as long-running

statistics gathering. Two main features we monitor are the number of open sockets, and the number of open file descriptors. Both these features represent limited resources, and are gathered by Granules at configurable intervals.

3.2.2 Program Profiles

Currently, program profiles are built by Granules when a program is first invoked. A program profile holds basic information about previous runs, which methods of communication are supported (e.g. TCP, UDP, named pipes, unnamed pipes, and shared memory), how to access each communication method available, as well as some general data about the behavior of the program. This collection of general data currently includes averages for: running time, input size, and size of the output.

3.2.3 Utilizing Diagnostics

We can gather diagnostics and return this information to the user. This allows users to choose which type of bridging to use, or alternatively specify a policy that dictates when the communication method should be changed. The system needs to be constantly aware of system state to initiate a communications switch as soon as necessary.

3.2.4 Evaluating Communications Switching

In order to evaluate the effectiveness of profiles, as well as explore the overhead incurred when switching communication types, we designed several tests in order to isolate these problem areas. For these tests, we used C,C++,C# and Python Bridges. These tests are fully discussed in section 4.4.

3.3 Implementation Challenges

A major challenge we described in this work is the wrapper. A bridge can only be as complex as a developer is willing or capable of writing an effective wrapper. A primary wrapper needs to

be written for each language being bridged to. This wrapper allows the bridged computation to receive and send `JavaByteMessages`, as well as actually orchestrate the computation based on the contents of these messages. This does not, however, preclude developers from writing wrappers for bridges of varying complexity based on the capabilities of each language and the needs of a computation.

Bridging across such disparate languages led to various difficulties. Languages such as Java and C# have built in byte-primitive transformations, Python has the Construct package and R has a basic function for binary reading and writing. This made reading and writing `JavaByteMessages` a relatively simple task for these languages. For the C and C++ examples, however, we needed to develop our own helper methods to handle the conversions from byte arrays and primitives and back. In general, we found that implementation ease or difficulty hinges on the tools available in the language that is being bridged to.

In Java, conversion between byte arrays and primitives can be handled with ease by using `DataInput/OutputStreams`. C# has the directly analogous `BitReader/Writers` and contains the useful method `BitConverter`, which can convert directly from an array of bytes to a specified primitive type. On the other hand, C# `strings` are more difficult to derive from a byte array than Java `Strings`, and it can be difficult to intercept the standard input/output streams with the `BitReader/Writer` in C# than with the Java counterparts.

Construct [15] is a python library designed for parsing and building data structures that are either binary or textual. It allows the developer to build complex data structures from simpler ones. With this library, the `JavaByteMessage` format can be fully declared in a more precise manner than either the Java or C# versions, and there is no need to implement marshalling/unmarshalling code as this is handled automatically by Construct.

Another challenge to this work is the need to make multiple methods of communication available in the target language. Not only does the program need to be able to start in any

supported method of communication, but it also needs to be able to switch to any supported method on the fly. While this does involve a notable amount of up-front work, once a basic communications wrapper has been constructed, it can be reused for any other computation in the language. From our own experiments, we found it possible to write a generic Python wrapper which handled all basic communications across all tests. While the different computations (Collage and Fibonacci) require separate computation-specific wrappers to handle the direct interface with the computation, the same class could be used to handle receiving and sending `JavaByteMessages`, and was able to orchestrate switching between supported communications methods.

Chapter 4 PERFORMANCE EVALUATION

In order to assess the feasibility of Granules Bridges, we first developed several tests to gather information about overheads, as well as how the communications framework scales as more data is passed across the bridge. These tests are described below.

4.1 Communications Overheads with the Fibonacci Sequence

For our initial tests, we first gathered basic information about overheads generated by Granules bridges. We wrote a simple recursive Fibonacci sequence in all target languages: C, C++, C#, Python, and R. To find these overheads, we set up a basic system where we connect to the computation through Granules and repeatedly query the computation for a number in the sequence (we generally found that 34 was a good number – long enough to get decent timing results, but short enough to run all tests in a reasonable amount of time). On the Granules side we started a timer, generated the `JavaByteMessage` to send to the bridged computation, and then stopped the timer after receiving a response from the computation. The computation is responsible for timing the actual calculation, and returning this value to Granules. By subtracting these two times; the total running time and the time needed for the computation; we can isolate the overhead introduced when `JavaByteMessages` are generated and decoded.

To ensure all implementations could interact properly with `JavaByteMessages`, a computation specific wrapper needed to be developed to handle direct interaction with the Fibonacci code. We also needed to write a language wrapper which would listen to an appropriate communication channel (standard input or a TCP socket) for incoming `JavaByteMessages`, then generate and return `JavaByteMessages` back through the appropriate

channel to Java. As mentioned above, for C and C++ the bulk of code which needed to be written were helper functions to handle the appropriate conversion from byte arrays to primitives and back. The C# and Python implementations were significantly easier, as such utilities already exist in the .NET framework, and through construct for Python. While R does have the ability to open up byte readers and writers on sockets, we have yet to implement an R version which uses `JavaByteMessages` across unnamed pipes – R does not allow programmers to connect to stdin or stdout with the binary readers and writers.

To keep times as close as possible, we used similar code for the Fibonacci sequence across all languages:

```
long fib(long n){
   if(n <=1)
      return n;
   else
     return fib(n-1) + fib(n-2);
}
```

While we expect to see some variation due to differences in languages, and what they have been designed to handle, we should still find comparable overheads since we are subtracting the actual computation time from overall running time. In Granules, a simple load generating interface has been developed which allows a user to communicate with an attached Fibonacci program. The bridge remains active until the user generates a kill command. The same Granules Fibonacci bridge program was used for these tests – this not only ensures that the baselines are standardized, but also demonstrates the wide applicability of Granules bridges: there was no need for language specific Granules interfaces.

This is a very simple example, yet still typical of Granules computations: work is done in short bursts, but the actual computations may be long-lived and continue to exist in the background waiting for more work.

In this experiment, we gathered these overheads across all bridge methods supported for each language: TCP and unnamed pipes for C, C++, C#, and Python, and TCP for R. Our results are shown below in Table 1.

Table 1 Communications Overhead Across Languages in Milliseconds

Method	**Mean**	**Min**	**Max**	**SD**
C pipes	0.468	0.366	1.119	0.139
C TCP	0.639	0.435	3.478	0.369
C++ pipes	0.439	0.348	0.964	0.096
C++ TCP	0.613	0.434	1.371	0.156
C# pipes	2.709	0.949	45.818	4.845
C# TCP	3.022	1.064	48.71	6.370
Python pipes	1.116	0.931	3.175	0.417
Python TCP	1.244	1.043	3.403	0.302
R TCP	195.403	13.323	6044.863	964.087

With the exception of R, all communications seem to be within an acceptable range – generally less than 4ms overhead, with several outliers found in the C# examples. As the C# tests were run on a separate machine, this may simply be a result of a slower machine – by looking at the standard deviation, the maximum overheads for C# appear to be outliers, and not a point for significant concern.

The R results are not very promising in these examples, as there was a vast range of results – the standard deviation is just short of a full second. There are several possible reasons for this, the most likely being that the polling mechanism to check for incoming data has too low of a resolution – so off-timed requests may result in the 6 second overhead we see in several cases.

4.2 Data Communication Scaling

In the first round of tests, the Fibonacci code only needed a single long passed as input, and then passed back as output. Only 16 bytes are being passed along the bridge in each direction. Sixteen bytes is a very small amount of data to be passing around, significantly smaller than typical sensor data. If the bridge framework cannot scale in an effective manner, the usefulness of Granules Bridges with more complex problems will be severely limited.

To evaluate the scalability of bridges, we ran several tests in which we sent an exponentially increasing amount of data to and from a bridged program. In this test, we generated a random array of bytes to pass as the `input` of a `JavaByteMessage` between the bridged computation and Java. Once the bridged computation decodes the message, it generates a new message to return which includes the original input. This test is run in C, C++, C# and Python for both unnamed pipes and TCP sockets, and in R over sockets. Once again, we measured all delays in milliseconds.

This approach should be able to empirically determine the penalty of increasing magnitude of data inherent with each communication channel for each language. This is the kind of data which should be directly usable by the diagnostics system when determining which communications methods to use under a heavy workload. We plan to fully investigate the effectiveness of this approach, as well as the system's ability to learn from this kind of data in future work.

This test was run with input sizes ranging from 20 bytes to 10MB. While we generally expect to see streaming data in the kilobyte range, we decided to do a true stress-test of our system by working with streaming data up to 10MB in size. The results can be seen below in Figure 1.

In C++, Python, and R over sockets, we see that there is a heavy cost for messages of 8KB and below in size. While we have yet to fully define this behavior, we believe this to be an artifact of the network. With the exception of R, all languages behave well with increasing message sizes,

generally remaining at or near a constant cost for communications across all languages.

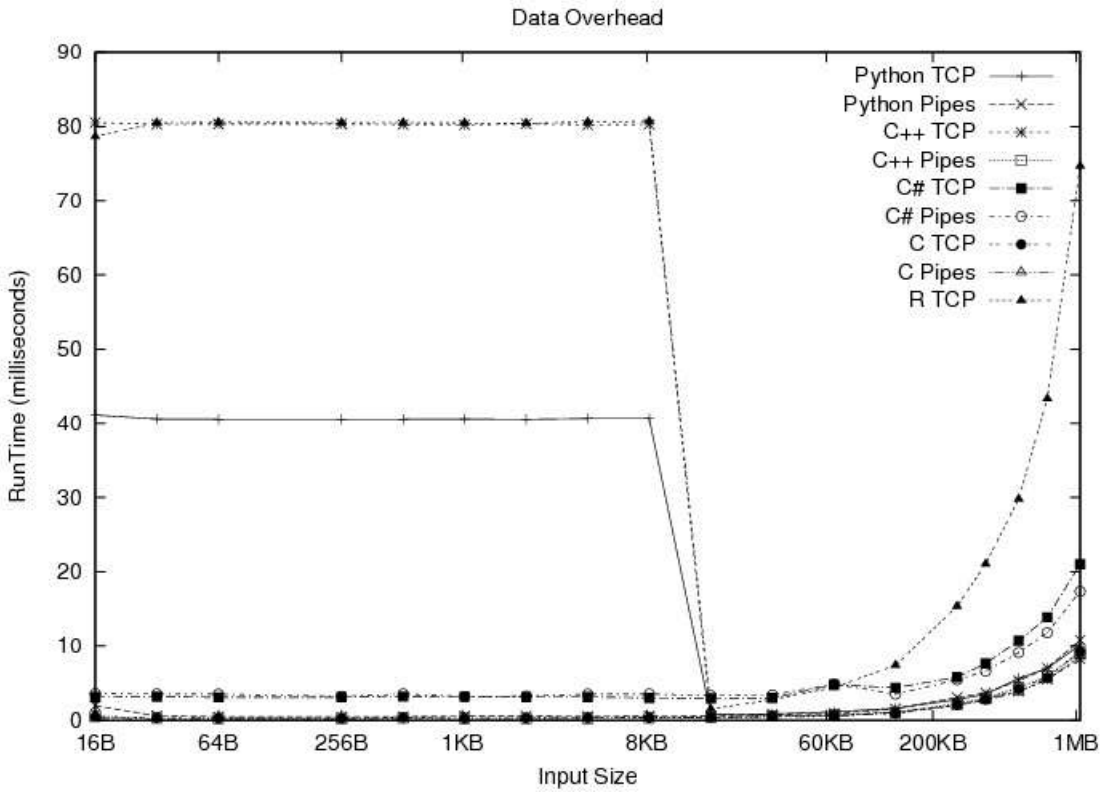

Figure 1 Communications overheads as message size increases – in C, C++, C#, Python and R

4.3 Switching Overheads

This test was designed to determine the overhead incurred when switching between communication types. The usability of the diagnostics system hinges on these results – if the cost of switching between communications channels is too high, any gain from switching to a less resource-intensive method of communication may be overshadowed.

A switch between communications channels requires a "handshake" across the bridge, so we are not only looking at how long it takes the "handshake" to occur, but also how long it may take afterwards for communications to settle into the new method. In previous experiments, we noticed that Python particularly would need about 2 messages sent across a fresh bridge after initialization before communications were stable. We want to not only pin down the "handshake" effect, but also any extra time needed in order to stabilize the communications.

In order to test these theories, we used the same test for communication overhead described above in section 4.2, switching the channel of communications after hitting the 10MB mark. We timed not only the basic cost of a communication switch message and response, but also the continuing cost of sending messages of various sizes. In 0the direct costs of communications switching overhead in each language is shown.

Table 2 Switching Communications Overhead in C, C++, C# and Python (Milliseconds)

Method	Mean	Min	Max	SD
C	2.766	0.815	12.754	2.253
C++	2.086	0.501	5.843	1.300
C#	10.540	3.600	70.144	9.832
Python	3.028	1.131	7.785	1.709

While we are seeing much more variation in C#, this is again most likely a result of the difference in hardware found in the Windows machine used for testing. Despite this, we found the costs of switching underlying communications methods in all languages to be acceptable. We additionally found the communications overheads with increasing data sizes comparable to those described in the previous section.

4.4 Cross-runtime Applicability

While the diagnostics system is tightly tied to the Granules runtime, Granules bridges are not. In this experiment we show that our bridges function as well in the Hadoop environment as in Granules. In this example we again looked at a simple bridged Python Fibonacci computation.

Due to the differences in how Granules and Hadoop are designed – most notably Hadoop's run-once semantics, we needed to make several modifications to our previous code to run this test. For this example, we needed to have a Python instance acting as a TCP server, which a Hadoop Mapper can connect to for the lifetime of a single request.

Table 3 Hadoop and Granules communications overheads to Python over TCP

Method	**Mean (ms)**	**Min (ms)**	**Max (ms)**	**SD (ms)**
Hadoop	1.778	1.337	2.286	0.229
Granules	1.244	1.043	3.403	0.302

In Table 3, we show the data gathered when running the Fibonacci example in Hadoop directly compared with the results we found when running the same function in Granules from Table 1. This clearly shows that there are no extra communications costs incurred when using a different cloud computing runtime.

Chapter 5 CASE STUDIES

In the previous chapter, we looked at very simple examples using the bridge. In this chapter we look into two different applications which use the bridge. The first is the Python Collage Generator, which generates collage images using the Map-Reduce framework. We then move on to observe the functionality of Granules Bridges when analyzing EEG signals in real-time. Both applications require much more data to be sent across the bridges than previous examples and are representative of real-life applications of Granules Bridges.

5.1 Python Collage Generator

This example is based on a python application we developed to generate an image collage using the Google AJAX Search API (http://code.google.com/intl/en/apis/ajaxsearch/) [11]. The application takes an initial image, and then finds similar pictures of appropriate colors to generate the collage. To handle image processing, we used the Python Imaging Library (PIL) (http://www.pythonware.com/products/pil/). Initially, the collage application was written sequentially and took hours to return a finished collage of the quality seen in Figure 4. We used Granules' bridging capabilities to distribute the problem, resulting in a significantly faster-running solution – instead of hours, this collage can be generated in minutes.

To distribute the problem, we use the Map-Reduce framework [3] that is supported by Granules. In this model, work is split up among many different processes, referred to as Mappers. Each Mapper will perform some function on the data provided. They all may perform the same work, or a variant of the same work. Once a Mapper has finished, it sends its results to a

Reducer. Reducers are separate processes which are responsible for gathering results from Mappers and aggregating the results. A computation may have a single Reducer, or many. There may also be several layers of Mappers and Reducers.

We separated the sequential Python code into a single Map-Reduce layer, and the tasks were separated into Mapper and Reducer work. First, an image is magnified, and broken into predetermined boxes. The Mappers split the boxes, and each is responsible for finding the predominant color in a subset of the boxes. Meanwhile, the Reducer finds appropriate images with the Google AJAX Search API [16] that will be used to fill in the collage. The search finds images in each of the predetermined supported colors: Red, Orange, Yellow, Green, Teal, Blue, Purple, Pink, White, Gray, Black, and Brown. Once a Mapper completes and returns its list of predominant colors, the Reducer is responsible for placing the correct image into the boxes. Once all Mappers have finished, and the reducer has completed modifying the image, the reducer returns the path to the completed collage.

Figure 2 Original Image

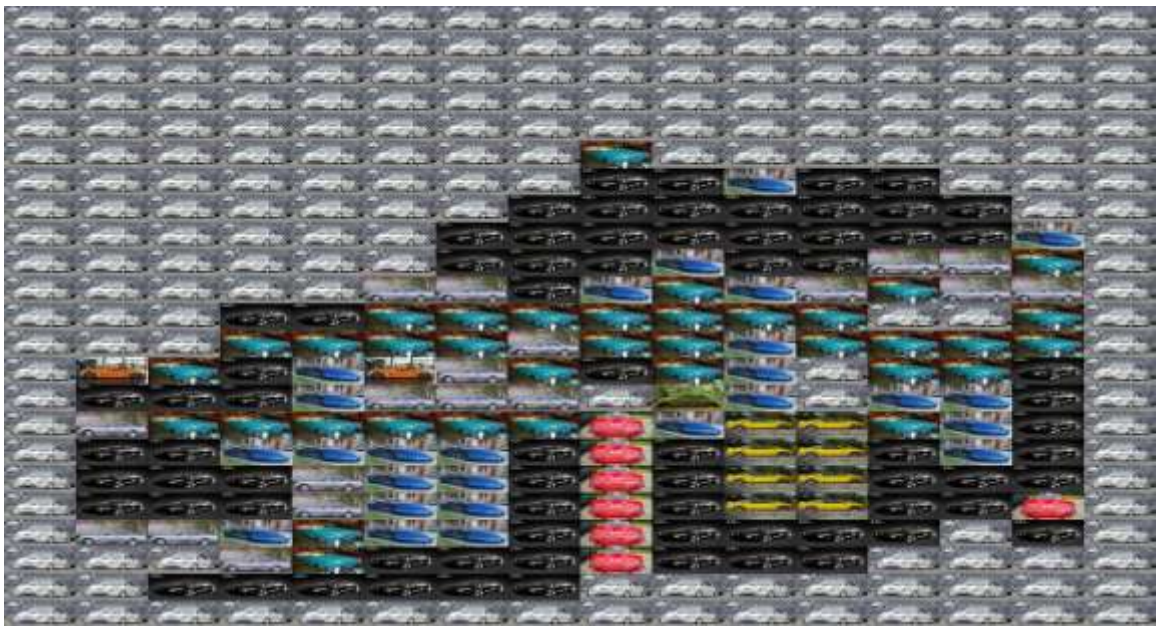

Figure 3 Collage Generated using boxes of 80x30 pixels, zoom factor of 1

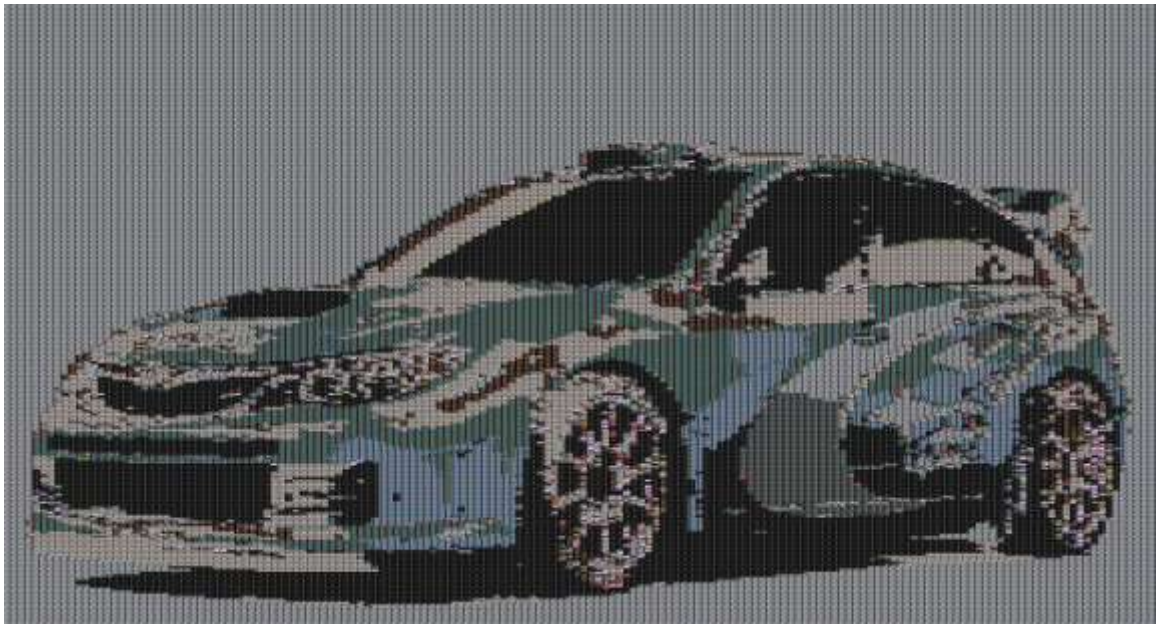

Figure 4 Collage generated using boxes of 80x30, zoom factor 10

As mentioned above, we used the Python construct package to handle reading and creating `JavaByteMessages`. For this example, two separate main scripts were needed – one for

Mappers, the other for the Reducer. The Mapper and Reducer backend code were different enough that separate wrappers needed to be generated.

In this example, the collage generation is a repeatable task. The user deploys the application at startup, setting the Mapper and Reducer Python functions. The user can then send information about collage generation to the cloud. This process can be repeated as often as is needed, by simply providing an image, information the quality, and a search topic to use to find images.

For testing the communications overheads with this example, we generated a collage from the above image seen in Figure 2. When running our benchmark tests we used boxes of 80x30 pixels, with a zoom factor of 1 – this generated image can be seen in Figure 3. We have also included a full high-definition collage in Figure 4 – boxes are 80x30 pixels, with a zoom factor of 10.

Our test collage (80x30, zoom 1) resulted in messages of about 4800 bytes (4.8KB) on average being sent between Java and Python. The results from these tests can be found below in Table 4. We found that as we increased the resolution of the resulting collage, the communications overhead would also increase – this follows the results of our experiments in section 4.2, where we looked at communications overhead as message size increased.

Table 4 Overheads for generating Python Collages

Mean	Min	Max	SD
17.395	15	21	1.001

5.2 R EEG Classification

This section combines the work found in [5] and [17]. In this example we compare specialized Granules code which uses JRI [13] to communicate with a backend R function for classifying EEG data. We extend upon [5] by moving on to compare the JRI based communication overheads with overheads found when using our own R bridges for communication, much as in [17].

The EEG we are classifying contains sets of four tests performed by the user: imagined right hand movement, imagined left leg movement, counting backwards from 100 by threes, and rotating a 3-D image of a computer. These tasks were performed by a user familiar with the tasks, so the data was clearer than the average untrained user. For these tests, we worked with two datasets of 10 five-second sequences of each task. We trained on one of the datasets, and reserved the second dataset for testing. As a note, the purpose of this work is to look into the feasibility of using Granules bridges for real-time processing of EEG signals, so we focus on communication overheads instead of accuracy.

In this example, we take advantage of Granules' ability to store state between runs – each Mapper is responsible for training and retaining a neural network (in R) which can then be used in subsequent runs to classify streaming data. This means we can skip the long process of training and simply classify input streams in subsequent runs. A cloud runtime with run-once semantics would be unable to handle this task.

This test additionally stresses the capabilities of R – it is not designed to handle streaming data. While, Granules has been designed to handle streaming data, it expects the data to be smaller by an order of magnitude than what we are seeing here. Our EEG data contains 19 channels, and the machine samples at 256 Hz – generating 38KB/second.

We are testing two methods of communication between R and Java: JRI and Granules Bridges. The JRI implementation uses `Strings` to drive communications across unnamed pipes. The Granules bridge version uses `JavaByteMessages` across a TCP socket. As we cannot currently compare the overheads across the same channel, we expect to see some variation in the results simply from channel specific overheads.

The basic setup of Mappers and Reducers is identical across implementations: An EEG stream is sent to all Mappers for classification. Once a Mapper has classified the stream, it sends its prediction on to the Reducer. The Reducer is then responsible for gathering predictions from all Mappers, and returning a consensus prediction back to the user. Using this method, we are able to

have smaller, easier to train neural networks on each Mapper, yet still potentially achieve a useful level of accuracy.

For these tests, we used only a single dataset for training, and a separate one for classification. Our previous work [5] shows that training times scale with the size of training sets – we will observe the same patterns with more 4 training sets as with one.

Table 5 Training a single neural network with one dataset

Method	**Mean**	**Min**	**Max**	**SD**
Binary Bridge	203648.1	130637	223541	15814.17
JRI Bridge	474928.5	415718	524825	32502.27

As can be clearly seen above in Table 5, our binary bridge can train more quickly than the JRI bridge. Both implementations are using the same backend code for training and reading in the files to be trained. This means we are incurring less of an overhead per message with the Binary bridges than with JRI. Our next test in this environment looks at the ability of each approach to classify data in a real-time setting.

For this test we again trained a single neural network on the first data, then used the second dataset for classification tests. In order to simulate real-time usage, we broke each 5 second stream from our test dataset into 250ms intervals – this would allow a user to make a decision in a BCI application 4 times every second. The results of this test are shown below in Table 6.

Table 6 Time to Classify 250 ms of EEG data (in ms)

Method	**Mean**	**Min**	**Max**	**SD**
Binary Bridge	87.896	87.086	90.092	0.467
JRI Bridge	87.432	85.711	90.471	1.010

As we can see, the results are very similar across both approaches. Based off of our previous work in [5], it seems clear that the Granules Bridges are just as effective for EEG analysis as the JRI Bridge.

Chapter 6 CONCLUSIONS AND FUTURE WORK

The Granules bridging framework allows developers to run non-Java computations through Granules, or other Java-based runtimes, in a simple and robust manner. This has the potential to bring a number of new applications into the cloud.

We have additionally shown that Granules Bridges are robust to various computational requirements: (1)The overheads incurred in using these bridges do not prohibit real-time analysis of streaming data, (2)Computations which execute multiple times are capable of building state across rounds of executions, (3)The bridges do not result in resource leaks so long-running computations can be supported indefinitely, (4)Bridges are capable of switching between underlying communications methods in a transparent manner, and (5)The bridges are bidirectional – both the Java and non-Java end of the bridge can steer the computation as necessary.

Based on our benchmarks in Chapter 4, we found that the Granules Bridge is a viable solution for intra-machine inter-language communication. In the baseline tests, a single Java program was needed in Granules to handle communication across all languages. There is no language-specific code on the Granules side of a computation. Our basic `JavaByteMessage` could be used unmodified for all tests – including the case studies performed in Chapter 4. The communications format can communicate as effectively with interpreted languages, such as Python, as with strongly-typed languages such as C. The framework provided is general enough to be all purpose, yet rigid enough to enforce coding guidelines.

From our results discussed in section 4.2 we found that our bridge performs as well as a JRI-based bridge when communicating with neural networks in R – the language with the worst overheads reported in Table 1. Given the performance of the R bridge in a real-time application, it

is safe to conclude that bridges to other languages will also be able to handle real-time computations. Additionally, we found the results shown in Figure 1 to be promising – with the exception of R, we found only a slight increase in communications overheads as we increased the amount of data sent across exponentially.

We have further shown that Granules Bridges are an effective method of communication outside the Granules framework. While our Hadoop example in section 4.4 is simplistic, it clearly shows that our bridges are fully functional within the Hadoop runtime. There is no extra overhead incurred outside Granules – any Java based bridge should have similar, if not identical, overheads.

Not only are our bridges an effective method of intra-machine, inter-language communication, but our work with communications switching (section 4.3) shows that we have found an approach to conserving system resources which can be effective in a live situation. This is a method which no other current bridging approach offers.

Areas for growth with the basic bridges include adding support for additional languages, as well as adding more methods of communication. While we currently support bridging over pipes and TCP, in the future we wish to add support for UDP and shared memory bridges. We are also looking into the possibility of chaining languages – for example; using Java as an intermediary between Python and R.

On the diagnostics end, we are looking at several avenues of growth. First, users should be able to build program profiles prior to any runs. While this is also a potential point of tampering, which may hurt future machine predictions, it can be a great help in bootstrapping a program profile. The next iteration of development also includes plans for storing program profiles to disk, thus allowing data to be used between resource sessions, as well as opening up the possibly of transferring or sharing profiles between similar machines.

We would additionally like to develop a robust learning system on top of the diagnostics gathering. With a learning system, we hope to aggregate data from separate machines which should allow the system to make generalizations about the behavior of new programs given previous performances.

The learning package should also be able to generate new system directives. With the diagnostics system, it should be able to build a profile for load patterns across a day, given the history of load patterns, and be able to generate directives based on time of day. For example, if there is generally a time of low or heavy resource use, we want the system to be able to predict when these changes will occur, and adjust in preparation so that the system does not grind to a halt during busy portions of the day.

BIBLIOGRAPHY

[1] S. Pallickara, *et al.*, "Granules: A Lightweight, Streaming Runtime for Cloud Computing With Support for Map-Reduce," in *IEEE International Conference on Cluster Computing*, New Orleans, LA., 2009.

[2] S. Pallickara, *et al.*, "An Overview of the Granules Runtime for Cloud Computing," in *IEEE International Conference on e-Science*, Indianapolis, 2008.

[3] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *ACM Commun.,* vol. 51, pp. 107-113, Jan. 2008 2008.

[4] M. Isard, *et al.*, "Dryad: distributed data-parallel programs from sequential building blocks," in *2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, Lisbon, Poutugal, 2007.

[5] K. Ericson, *et al.*, "Analyzing Electroencephalograms Using Cloud Computing Techniques," in *IEEE Conference on Cloud Computing Technology and Science*, Indianopolis, USA, (To appear) 2010.

[6] M. Chen, *et al.*, "Java JNI Bridge: A Framework for Mixed Native ISA Execution," in *International Symposium on Code Generation and Optimization*, 2006, pp. 65-75.

[7] D. Juneau, *et al.*, *The Definitive Guide to Jython: Python for the Java Platform*: Apress, 2010.

[8] S. Vinoski, "CORBA: integrating diverse applications within distributed heterogeneous environments," *Communications Magazine, IEEE,* vol. 35, pp. 46-55, 1997.

[9] E. Harold, *XML: Extensible Markup Language: Structuring Complex Content for the Web*. Foster City: IDG Books Worldwide, Inc., 1998.

[10] "SOAP Version 1.2 Part 1: Messaging Framework," 2001.

[11] D. M. Beazley, "SWIG: an easy to use tool for integrating scripting languages with C and C++," in *USENIX Tcl/Tk Conference*, Monterey, California, 1996, pp. 15-15.

[12] T. White, *Hadoop: The Definitive Guide*, 1 ed.: O'Reilly Media, 2009.

[13] "JRI - Java/R Interface," 0.5-0 ed, 2009.

[14] "rJava - Low-level R to Java interface," 0.8-3 ed, 2010.

[15] T. Filiba, "Construct," 2.00 ed, 2007, p. python parser.

[16] R. Hanson and A. Tacy, *GWT in Action: Easy Ajax with the Google Web Toolkit*: Manning Publications Co., 2007.

[17] K. Ericson*, et al.*, "Handwriting Recognition using a Cloud Runtime," in *Colorado Celebration of Women in Computing*, Golden, 2010.