Adaptive Heterogeneous Language Support within a Cloud Runtime

Kathleen Ericson and Shrideep Pallickara
Department of Computer Science
Colorado State University
Fort Collins, US
{ericson, shrideep}@cs.colostate.edu

***Abstract*— Cloud runtimes are an effective method of distributing computations, but can force developers to use the runtime's native language for all computations. We have extended the Granules cloud runtime with a bridge framework that allows computations to be written in C, C++, C#, Python, and R. We have additionally developed a diagnostics system which is capable of gathering information on system state, as well as modifying the underlying bridge framework in response to system load. Given the dynamic nature of Granules computations, which can be characterized as long-running with intermittent CPU bursts that allow state to build up during successive rounds of execution, these bridges need to be bidirectional and the underlying communication mechanisms decoupled, robust and configurable. Granules bridges handle a number of different programming languages and support multiple methods of communication such as named pipes, unnamed pipes, and sockets. This choice of underlying communication mechanisms allows limited resources, such as sockets, to remain available for use by the runtime.**

Keywords-Adaptive behavior, Cloud runtime, Granules, Language bridges

I. INTRODUCTION

Cloud runtimes with their support for orchestrating computations (some of which are based on the MapReduce framework [1]) have gained significant traction in the past few years. Though the cloud runtime may be developed in a specific programming language, a need often arises to orchestrate computations that have been developed in other programming languages. Here we describe our support for heterogeneous languages within Granules.

Granules [2, 3] is a lightweight runtime for cloud computing and is designed to schedule a large number of computations across a set of available machines. Granules has support for both MapReduce and dataflow graphs [4]. Granules computations 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 further data to process. This allows Granules to move away from the run-once semantics of frameworks such as Hadoop [5].

In Granules, computations specify a scheduling strategy, which govern their lifetimes. Computations scheduling strategies are defined across three dimensions: number of executions, data driven, or periodically. The number of executions limits the maximum amount of times a computation can be run. The data driven axis schedules computations as data becomes available on any input streams the computation has registered. The user can also specify that a computation should be executed on a specific interval. It is also possible to specify a custom scheduling strategy that is a combination along these three dimensions. A computation can change its scheduling strategy during execution, and Granules enforces the newly established scheduling strategy during the next round of execution.

Computations in Granules build state over successive rounds of execution. Though the typical CPU burst time for computations during a given execution is short (seconds to a few minutes), these computations can be long-running with computations toggling between activations and dormancy. Domains that Granules is being deployed in include earthquake science, epidemiological simulations, and brain-computer interfaces [6].

While the Granules Bridge framework has been developed to work within Granules, we have found our design flexible enough to work in a basic Java environment as well as with Hadoop with millisecond overheads.

Here we describe our framework to incorporate support for computations developed in diverse programming languages within Granules. There are three important challenges that we address.

Challenge 1: *Semantics of communications between bridged computations should be independent of the mechanism to implement them.*

Bridges can be implemented in different ways. Computations in different languages can use named pipes, unnamed pipes, sockets, or shared memory for communications with each other. We must abstract the content from the channel used to implement these bridges. 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 semantics of communications to be lightweight and bidirectional while introducing acceptable overheads. The overheads introduced should not preclude the possibility for developing real-time applications in other languages.

Challenge 2: *Account for resource usage at individual machines.*

Granules is designed to support data driven computations. Computations are scheduled for 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 be in the order of 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 (BCI) 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 (up to 10,000) of such computations concurrently on the same resource to maximize resource utilizations while processing streams.

Over time the availability of system 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 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 without requiring rewrites. Existence of a bridge to a language should imply that development of 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 would then be extended by computations developed in that language.

A. Paper Contributions

The Granules Bridge framework provides a mechanism for developers to bridge computations developed 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++, Python and R can additionally communicate using TCP (C# socket support is still in development); support for datagram sockets and shared memory is ongoing. This paper 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. For example, we have incorporated support for this framework within Hadoop, and our measured overheads here are similar to what we see in Granules. Finally, Granules is open-source and this framework has been released as part of the runtime.

Suitability to data driven computations: By dynamically adapting the communication mechanisms to changing system conditions over a period of time, computations may use a different mechanism during different rounds of execution.

Support for multiple languages: We have incorporated support for different bridging mechanisms to languages such as C, C++, C#, Python and R. This allows the runtime to orchestrate computations developed in different languages. This could also be used to create bridges that span multiple languages: for example, one may use Java as an intermediary for communications between R and Python. We have not yet benchmarked the costs incurred in doing so.

Responsiveness to varying system conditions: The framework is lightweight and relies on diagnostics to autonomously tune communication mechanisms based on specified directives.

B. Organization

In section II, we describe the related work in this area. In section III we provide details about the Granules Bridge framework as well as the Diagnostics and Adaptive Systems. We describe our experiments and report on our benchmarks in section IV. Finally, we provide our conclusions and discuss future work in section IV.G.

II. Related Work

Purpose-specific wrappers are the general approach to solving communication problems between languages on a machine. While this means that wrappers can be written specifically for a particular application, and can take advantage of this lack of generality, it also means that this code is not generally reusable, and can easily become difficult to maintain when extending the original program.

The Java Native Interface (JNI) [7] is a framework that allows Java programs to interact with machine specific languages and programs from inside the JVM (Java Virtual Machine). It is designed so that C/C++ or assembly programs and Java programs can interact 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 that it only supports C/C++ and assembly programs – there is no support for other languages such as Python or C#.

Closely related to JNI is Java Native Access (JNA) (https://jna.dev.java.net/). JNA allows a developer to access system level code written in C, Windows dlls, as well as Jython [8] and JRuby (http://jruby.org/). While JNA claims to have a simpler interface, it has been reported to run approximately 100 times slower than equivalent JNI code.

CORBA [9] has been developed to handle communication between languages. While it has been primarily designed to work across a network, 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 communications. Additionally, communications are no longer lightweight with all the information needed to appropriately run a command.

MPI is a tool designed for developing parallel programs, and uses shared memory for communications.

An alternative approach for communications is through the use of XML [10]. XML allows for the development of a complex, extensible and self-descriptive language for communication, and would allow us to build a communication framework between programs similar to that provided by SOAP [11] for communications between distributed services. On the other hand, XML parsing adds a considerable overhead cost to all communication, and we would still either need to write XML to a pipe or send it across a network as packets. While XML does have the advantage of human readability, particularly useful when debugging, it would add to the general bulk of a message.

The Simplified Wrapper and Interface Generator (SWIG) [12], is generally used to connect C/C++ code to other languages. It is used by Hadoop 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.

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 this is a fully functional method of bridging between Java and R, it does not extend to any other language – it is built exclusively for R and Java. While preliminary tests have shown that JRI can outperform our current R bridge [15] on pure communications speed, our bridge is far more flexible and outperforms JRI for compute intensive operations.

Granules bridging defines a protocol for designing communication links between programs. It does not generate code, and developers are expected to implement their own wrappers. While this does require some more work for developers, there is no need to learn a new language to handle code generation. The Granules Bridge should be safer and more reliable as it provides stronger typing than SWIG, as well as defining the protocol for communication with any language – not simply C and C++.

The major difference between Granules Bridges and other methods of communication bridging is that Granules supports the definition of multiple bridging mechanisms, as well as the ability to switch between these methods of communication based on system directives and changing resource utilizations.

III. GRANULES BRIDGES

Granules bridges are designed to provide a conduit between computations in different programming languages on a single machine. The bridging framework is built to accommodate communications through byte arrays, though it is possible to use strings for communication across pipes.

Granules has been designed primarily to handle data from sensors, so it is not expected that the Granules bridges will need to carry large amounts of data. Environmental data sensors generally produce data in a stream of 2-8 KB packets. Brain Computer Interfaces (BCI) [16] 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 very large. In summary, Granules bridges are neither designed nor expected to handle transmissions of data in the order of GBs of data.

Granules bridges are designed to be bidirectional. Not only will a Java computation steer a non-Java computation through the bridge, but the non-Java computation can steer the Java computation through the bridge as well. This means that both sides of a bridge need to be actively listening for communications. All basic classes needed to set up this communication have been developed as a part of Granules, and can be extended as needed to handle even more complex communications.

A. Bridge Design

There are several classes involved in creating a Granules bridge to computations developed in other languages. Granules provides: `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`. These classes simplify development of wrappers to handle communications with computations written in another language.

1) *JavaByteMessage*

The backbone for bridged communication is the `JavaByteMessage`. This is implemented in both Java and any language being bridged to. This communication protocol is rigid enough to make implementations relatively simple, yet flexible enough to handle 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.

```
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 Granules and a bridged program. `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`. 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 which is easily extendable to handle diverse types of computations

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. First, there is an integer determining 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 command types as needed.

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 is necessary to be able to properly reconstruct the arrays, but otherwise the array contents are completely customizable.

The `streamIdentifier` is necessary for the external program to be able to identify the Granules resource it has been working with. In some cases, several Granules resources may be interacting with the same base program, and this would be needed for identification of messages. In order to pass along extra information about a message the description field is included as well. This is useful for debugging or logging, 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.

2) `JavaGenericComputation`

A `JavaGenericComputation` is responsible for running the external computation, as well as setting up communication methods. This is the link between Java and the computation, and is how Java interacts though `JavaByteMessages`. A `JavaGenericComputation` is expected to know about various available methods of communication (named pipes, unnamed pipes, TCP, etc.), and be able to orchestrate the actual switching of bridging methods through a series of handshakes.

3) Readers and Writers

To handle a variety of potential methods of communication, all readers and writers developed to work with the bridge are built around generic `InputStreams` and `OutputStreams`. This means that the same backend is capable of handling any type of communication – it only needs to have a stream provided.

In order to handle `JavaByteMessages`, a `StreamReader` needs to be instantiated. A `StreamReader` needs not only an `InputStream`, but also a reference to an implementation of `JavaMessageHandler`. The `StreamReader` needs this reference in order to pass a `JavaByteMessage` after it has been read in fully. This means that the `StreamReader` can be reused across all bridge implementations – it is not tied in any way to the `JavaByteMessage`.

`StreamWriters` are responsible for taking a `JavaByteMessage` and sending it across its `OutputStream` as a byte array. It first needs to call the `JavaByteMessage` function to turn it into a byte array, and then needs to send the length of the array across the stream before sending the actual array.

These three utilities should be general enough to be used for 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.

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. This can be as complex or simple as the developer wishes to make it.

B. Diagnostics and Directives For Adaptive Communications

We have added the ability for basic diagnostics on a bridge method to be gathered. We can use these diagnostics to create a program profile which describes the processing footprint generated by a given program over a given method of communication. These profiles are in turn used when enforcing policies. While the diagnostics system is closely tied into the Granules environment, it is possible for a user to develop programs with the ability to change the underlying communications method for a computation without using the Granules diagnostics system.

In the current implementation, we are using developer generated policies in order to obtain adaptive behavior. The system should be able to modify communications based on the current system state as well as the requirements of the computations currently running. Future versions will incorporate a learning component which will be capable of evolving policies based on past behavior, as well as 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 describe behavior for a given state of a machine, or define any constraints of a computation. 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, one directive may state to move any currently running computation which has an I/O ratio of less than 50% to piped communication to make sure that the machine does not run out of sockets. In another scenario: if less than 10% of all sockets are being used, all new computations will be automatically started with a socket-based bridge.

User policies pertain to specific computations. 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 would state that it should take no longer than 200 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. Other directives that we have used in our examples include a priority rating which

states that computations with a high rate of I/O and a low processing overhead are given a higher priority

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

1) Resource and Program Diagnostics

The Granules diagnostics system is responsible for monitoring system state, as well as keeping track of program profiles. This data is gathered at the resource level, and can be used in subsequent runs to determine the most efficient method of bridging. 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. A policy dictates when the system should switch communication methods – for example, a system policy may state that if the socket usage is over 60% any new bridge communications should occur over pipes.

Granules 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 more 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.

2) Program Profiles

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

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 to specify a policy that dictates when the communication policy should be changed. The system needs to be constantly aware of system state to initiate a communications switch as soon as necessary.

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++ and Python Bridges.

C. Implementation Challenges

A bridge can only be as complex as a developer is willing or able to write an effective wrapper. These wrappers need to be written for each language being bridged to. They need to be able to receive and send `JavaByteMessages`, as well as 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 needs of a computation.

Depending on which language a bridge is being built for, there were some difficulties to overcome. Languages such as Java and C# have built in byte-primitive transformations, and python has the Construct package available. For the C and C++ examples, we needed to develop our own methods to handle the conversion between byte arrays and primitives. In general, implementation ease or difficulty will hinge on the tools available to the developer in the language that is being bridged to.

In Java, conversion between byte arrays and primitives can be handled using `DataInput/OutputStreams`. C# has a directly analogous `BitReader/Writer` and contains the useful method `BitConverter`, which will go directly from an array to a primitive type. But C# strings are a little trickier to derive from a byte array than Java Strings, and it can be difficult to intercept the standard input/output with the `BitReader/Writer`.

Construct [17] is a python library for parsing and building data structures that are either binary or textual. It allows the developer to declare complex data structures based on 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.

One major 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 method of communication supported, but it also needs to be able to switch to any supported method on the fly. This involves a notable amount of up-front work, but once an original 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 for all our tests. While the different computations (Collage and Fibonacci) would 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`.

IV. Performance Evaluation

For testing purposes, we developed three different applications in four different languages. In C++ and C#, a simple recursive Fibonacci sequence program was called through Granules. We also developed a sample bridge for a Python program which builds image collages, as well as an R program which classifies EEG signals which are streamed to it. The bridging code necessary in all languages (C, C++, C#, Python, and R) are also included with the examples, which should help future implementations in these languages.

These applications are 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. Results from tests run based on these sample

programs should be able to give us a sound estimate of production environment runs.

With these experiments we can show that Granules bridges are viable solutions for orchestrating computations such as Python scripts for Bioinformatics, or R scripts to handle EEG classification in real-time. In our benchmarks, we are not only establishing language overheads, but also the effect of message size on these overheads. We additionally examine the overhead incurred for attempting to switch the underlying type of bridged communications.

With the exception of C#, all tests were staged on a 4 core machine running Fedora 12 with 4 GB of RAM. For the C# example we used a machine running Windows 7 with 2 cores and 2 GB of RAM.

A. *Fibonacci sequence Examples*

Both Fibonacci examples are simple and primarily used as proof of concept examples. They both rely on Unnamed Pipes for communication. The Fibonacci code is similar in either implementation:

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

To make this interact properly with Java through `JavaByteMessages`, basic wrappers needed to be developed to handle program interaction, as well as a program to listen to stdin (standard input) for incoming `JavaByteMessages`, and prepared to generate and return `JavaByteMessages` via stdout (standard output). In C++ the bulk of code which needed to be written was primarily code to handle conversion of variables to and from byte arrays. The C# implementation relied on utilities in the .NET framework.

On the Granules end, a simple load generating interface has been developed which allows a user to communicate with the attached Fibonacci program. The bridge will remain active until the user generates a kill command.

This simple example exemplifies a typical Granules computation. It is a data-driven computation, which stores state and is dormant awaiting data to be pushed to it. Once it receives data, it activates and performs the needed processing, pushes the results out, and returns to its dormant state awaiting activation.

B. *Python Collage Generator*

This example is based on a python application we developed to generate an image collage based on an initial image using the Google AJAX Search API [18]. To handle image processing, the Python Imaging Library (PIL) (http://www.pythonware.com/products/pil/) is used. For initial tests, the collage application was written sequentially. We then used Granules' bridging capabilities to distribute the problem, resulting in a significantly faster-running solution.

To distribute the problem, we use the Map-Reduce framework that is supported by Granules. In this model, a load of work is split up among many different processes, the Mappers, each of which will perform some function on the data provided to it. Each Mapper may perform the same work, or a variant of the same work. Once a Mapper has completed its work, it sends its results on to a Reducer. Reducers are responsible for gathering results from Mappers and aggregating the results.

The modified python code is separated into a single Map-Reduce layer, and the tasks needed are separated into Mapper and Reducer work. First, an image is magnified, and broken into predetermined boxes. The Mappers are responsible for finding the predominant color in a selection of boxes. Meanwhile, the Reducer finds appropriate images with the Google AJAX Search API to fill in for 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 replacing the correct boxes with the needed image. Once all Mappers have finished, and the reducer has completed modifying the image, the reducer returns the path to the completed collage. Figure 1. and Figure 2. depict the original image and the collage generated by the program.

Figure 1. Original Image

Figure 2. Generated Collage

We developed two different wrappers – one for communication via unnamed pipes, while the other uses TCP over sockets. There are two separate main scripts required – one for the Mappers, the other for the Reducer. The Mapper and Reducer wrapper format was distinct enough that each needed its own specific wrapper.

C. *R EEG Classification*

This example involves classifying EEG data which is separated into 4 different tasks: imagined left leg movement, imagined right hand movement, 3D image manipulation, and math problems. We are currently working with data stored on file, using a pseudo-streaming application to simulate live EEG signals. This data has already been cleaned to remove noise related to muscle movements such as eye movement.

In this example, we take advantage of the capability to store state – each Mapper is responsible for training and retaining a neural network (in R) which can be used in any subsequent runs to test incoming data. As R is not designed to handle streaming data, we are stretching its capabilities. Here we are using strings to drive communications. An implementation which is capable of handling byte array communication is currently in development.

Once an EEG stream has been classified at a Mapper, it sends its prediction on to a Reducer. The Reducer is responsible for gathering predictions from all Mappers, and

then coming to a consensus prediction. By this method, we are able to have smaller neural networks that are faster to train on each Mapper, and still achieve a reasonable level of accuracy. Since the goal of this paper is to analyze communication overhead, we have not measured the classification accuracy of the neural network.

D. Marshalling/Unmarshalling Packet Overhead

In this benchmark, we are looking at specifically how much time is spent marshalling and unmarshalling `JavaByteMessages`. On the Java end, a timer is started just before a message is converted to a byte array and sent to the bridged computation. Once the byte array has been converted back to a `JavaByteMessage` on the computation end, it starts its own timer to keep track of how long the actual computation takes. The timer is stopped as soon as the computation is done, and this time is sent back with the computation results to Java. Once Java has the RESULTS message, it will stop its timer. The Java time and computation time are saved to file to be analyzed in R.

We measured the marshalling and unmarshalling overhead to be the difference between the round-trip Java time and the actual computation time. All times are recorded in milliseconds – the highest-resolution available on all target computation languages.

TABLE I. COMMUNICATIONS OVERHEAD ACROSS LANGUAGES (MILLISECONDS)

	Mean	***Max***	***Min***	***SD***	***Bytes***
C++ Pipes	0.62	4	0	0.753	57
Python Pipes	4.9165	9.1973	0.7411	2.51897	107
C# Pipes	19.879	86.549	2.757	21.36672	57
C++ TCP	1.5	42	0	5.89	57
Python TCP	5.5616	12.6916	1.3235	4.0915	107
R Pipes	330.861	369.184	316.329	11.588	1000+

In TABLE I. , the communication overheads for our Fibonacci and collage examples implemented in C++, C# and Python are shown. While the C# examples show a significant increase over the C++ implementation, there are several reasons why this may be occurring. One possible reason for this discrepancy is that we are attempting to use built in C# objects to help marshal and unmarshal messages. These discrepancies bear further research, and we are currently looking into rewriting the marshalling functions.

We have additionally included preliminary benchmarking from our R BCI application. There is again a higher overhead than the C++ results. As R is not designed to handle streaming data, we believe this to be a direct result of the current implementation.

There is a general overall trend where the compiled languages (C++ and C#) are outperforming the interpreted languages (Python and R). This is expected, as compiled languages generally run faster. Additionally, the different computations are also sending different amounts of data. While the compiled languages are running Fibonacci examples (generally sending a single integer as the input, and receiving a long as the result), Python and R are running the collage search and EEG classification respectively, which have different input sizes.

In general, it appears that the `JavaByteMessage` overhead generated from marshalling and unmarshalling is acceptable, and will not drastically effect overall runtime.

E. Data Communication Overhead

While the marshalling and unmarshalling tests gathered information about bridge performance with relatively small input sizes, we wanted to see how performance varied with increasing data sizes. To determine the effects of input size on communication costs, we ran several tests in which a logarithmically increasing amount of data is sent to and from a bridged program. In this test, inputs of random bytes are generated and sent to the bridged program. The bridged program decodes the message and generates a new message to return which includes the original input. This test is run in C# using unnamed pipes, and C++ and Python for both unnamed pipes and TCP. All delays are in milliseconds.

In theory, this approach will be able to empirically determine the penalty inherent with each communication method, as well as pick up information about possible deviations from this base cost. Using this information a resource should be able to pick the method which incurs the least overhead, while using the least amount of limited resources, and eventually taking into account the general behavior of the program being run. We will investigate this aspect as part of our future work.

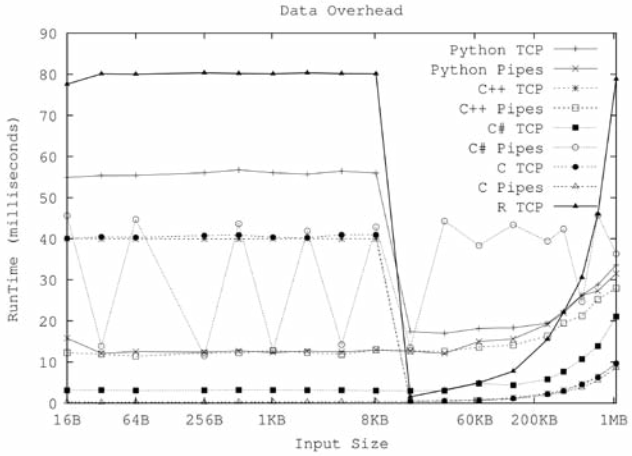

Figure 3. Data Overhead in C, C++, C#, Python and R

This test was run with input sizes ranging from 16 bytes to 1MB. 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 1MB.

As can be seen in Figure 3. , there is a heavy overhead on message sizes 8KB and under, particularly over TCP. This trend does not hold with C#, but that may be a result of running with a different operating system. After the 8KB overhead, all further communications in C++ and Python are below 10 milliseconds. Even the worst overhead (C++ over TCP with 80ms) is less than 0.1 seconds.

F. Switching Overhead

This test was designed to gather the overhead for switching communication types. As this requires a "handshake" across the bridge, 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. In previous experiments, we noticed that Python in particular would need about 2 messages on average sent across a fresh bridge 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, adding in a communications switching call once for every set of tested data sizes. 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. Comparing the message size times to the results displayed in Figure 3. , there was no noticeable difference in overhead between communications immediately following a switch, and those happening in a stable communication environment.

In TABLE II. , the direct cost of switching communications in each language is shown.

TABLE II. SWITCHING COMMUNICATION OVERHEAD (MILLISECONDS)

	Mean (ms)	Min (ms)	Max (ms)	SD (ms)
C++	3.663298	1.326986	8.686024	1.962471
Python	4.071485	2.064579	9.815149	2.082320

G. Benchmarks for using Granules Bridges within Hadoop

We used our bridge framework to provide support for Python computations within the Hadoop runtime. In this test we gathered overheads for bridging to a Python-based Fibonacci computation in Hadoop using TCP sockets for communication. We then directly compared the bridge overheads (1.77 ms) in Hadoop with the overheads in Granules (1.02 ms) for the same setup: Granules was faster by about 0.75 ms with a standard deviation of 0.248 ms. This minor difference between the bridging overheads, demonstrates that the Granules Bridges framework performs just as effectively in Hadoop as it does in Granules.

V. CONCLUSIONS AND FUTURE WORK

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

Our preliminary benchmarks indicate that the Granules Bridge is a viable solution for intra-machine inter-language communication. The framework we provide is general enough to be all purpose, yet rigid enough to enforce coding guidelines. The overhead produced by marshalling and unmarshalling messages is acceptable, as is the growth of overhead as message size increases. As we currently only support bridging over pipes and TCP, in the future we wish to add support for UDP and shared memory bridges. We believe that our findings will hold true across these other bridge methods.

With respect to the diagnostics, there are several avenues of growth. First, program profiles should be able to be built by users prior to any runs. While this can be a great help in bootstrapping a program profile, this is also a potential point of tampering, which may hurt future predictions. The next iteration of development also includes plans for storing program profiles – allowing data to be used between resource sessions, as well as possibly transferring or sharing profiles between similar machines.

Our goal is to build a robust learning system on top of the diagnostics gathering software. With this learning package, we hope to aggregate data from separate machines allowing the system to make generalizations about the behavior of new programs given previous performances. This learning package should be able to build new system directives. It should also be able to build a history of load pattern across a day, 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 a busy part of the day.

ACKNOWLEDGMENT

This research has been supported by funding (HSHQDC-10-C-130) from the US Department of Homeland Security's Long Range program.

REFERENCES

[1] J. Dean, et al., "Mapreduce: Simplified data processing on large clusters," ACM Commun., vol. 51, pp. 107-113, Jan. 2008.

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

[3] S. Pallickara, et al., "An Overview of the Granules Runtime for Cloud Computing," in IEEE Int'l Conference on e-Science, USA, 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] T. White, Hadoop: The Definitive Guide, 1 ed.: O'Reilly, 2009.

[6] K. Ericson, et al., "Analyzing Electroencephalograms Using Cloud Computing Techniques," in IEEE CloudCom, Indianopolis, 2010.

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

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

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

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

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

[12] 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.

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

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

[15] K. Ericson, et al., "Handwriting Recognition in a Cloud Runtime," in CCWIC, Golden, 2010.

[16] F. Galan, et al., "A brain-actuated wheelchair: Asynchronous and non-invasive Brain-computer interfaces for continuous control of robots," Clinical Neurophysiology, vol. 119, pp. 2159-2169, 2008.

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

[18] R. Hanson , et al., GWT in Action: Easy Ajax with the Google Web Toolkit: Manning Publications Co., 2007.