J Grid Computing
https://doi.org/10.1007/s10723-020-09525-z

Remote Method Delegation: a Platform for Grid Computing

Bradley Wood · Brock Watling · Zachary Winn · Daniel Messiha · Qusay H. Mahmoud · Akramul Azim

Received: 7 August 2019 / Accepted: 18 June 2020

Abstract While many cluster and grid computing frameworks are available, the task of building secure distributed systems or implementing distributed algorithms continue to be a challenging task due to the inherent distributed nature of such systems with multiple failure modes and security issues. In this paper, we present the design and development of remote method delegation (RMD), which is a secure lightweight grid computing platform with load balancing and code migration. RMD is focused on improving the usability issues that plague related industry solutions. The platform is implemented on the JVM (Java Virtual Machine) and supports the Java and Kotlin programming languages, however, the platform should theoretically work with other JVM languages. RMD was designed to simplify the implementation of distributed algorithms by providing a Kotlin DSL (domain specific language) that allows the programmer to define jobs within dedicated code blocks. Users from around the world can donate their own computing resources by hosting their own job server. RMD is secured by placing all untrusted code within a sandbox environment that prevents potentially malicious actions from taking place. To demonstrate the feasibility of the proposed model, a proof of concept implementation has been constructed with real examples demonstrating the usefulness of the proposed solution.

B. Wood (✉) · B. Watling · Z. Winn · D. Messiha · Q. H. Mahmoud · A. Azim
Department of Electrical and Computer Engineering, Ontario Tech University, 2000 Simcoe St N, Oshawa, ON L1G 0C5, USA
e-mail: bradley.wood@uoit.net

B. Watling
e-mail: brock.watling@uoit.net

Z. Winn
e-mail: zachary.winn@uoit.net

D. Messiha
e-mail: daniel.messiha@uoit.net

Q. H. Mahmoud
e-mail: qusay.mahmoud@uoit.ca

A. Azim
e-mail: akramul.azim@uoit.ca

Keywords Java virtual machine · Message passing interface · Load balancing · Code migration · Cluster computing · Grid computing · Remote procedure calls · Remote method delegation · High performance computing · Parallel computing · Domain specific language · Java · Kotlin

1 Introduction

It is often a daunting task to solve problems that require high performance computing systems due to the inherent distributed nature of cluster and grid computers. Researchers commonly encounter parallelizable problems that are too tough to be solved by a single machine. It is often problematic that many researchers are not experts on high performance computing which could pose a large impediment to their research. It is our goal to create a platform that reduces boilerplate and works out of the box with little configuration and minimal developer training.

High Performance Computing: High Performance Computing (HPC) typically refers to the combination of several computing machines to achieve a greater performance output than any of the individual machines that the system is composed of [20].

Grid Computing: Grid computing is the combination of many loosely coupled, geographically dispersed machines that are often used for a variety of jobs. Grid computers are usually made up of heterogeneous computers systems [13].

Remote Evaluation: Remote evaluation refers to the act of migrating code from client to server for subsequent evaluation on the server [28]. The results of the executed jobs are returned back to the client. Remote evaluation is the core mechanism used to facilitate the RMD platform.

Load Balancing: Load balancing is an important part of many distributed systems and is used to improve the distribution of computational workloads and prevent any single resource in the system from becoming overloaded [19].

Internal DSL: An internal DSL (domain specific language) is a language tailored to a specific set of use-cases and is built on top of another programming language called the host language. This differs from an external DSL which will require its own compiler and runtime environment [16].

Java Virtual Machine: The Java Virtual Machine (JVM) is the platform of choice for our proposed solution. The JVM operates across many architectures and operating systems, executing cross-platform JVM bytecode, which enables developers to write platform independent software [21]. The "Write once, run anywhere" model of the JVM is particularly advantageous to RMD because it supports the requirement of executing code in a heterogeneous environment which could have varying hardware and operating systems across the grid. The use of bytecode also facilitates the use of various programming languages, as long as they compile to JVM bytecode. Kotlin, Scala, and Groovy are examples of alternative JVM languages.

RMD provides several contributions to high performance computing (HPC) and grid computing. RMD tackles many of the usability concerns of researchers by providing a higher level of abstraction to the programmer and allows for existing applications to be easily ported with little effort. A high level of abstraction is achieved with a combination of a Kotlin DSL and automated code migration followed by remote procedure calls. The design philosophy of increased abstraction enables the reduction of boilerplate code, allowing researchers and developers to spend more time focusing on the issue at hand. RMD does not require the use of the DSL, allowing the developer to use any JVM language, such as Java. RMD is a platform which also provides a DSL as a support mechanism to those who want it. In the context of RMD, a 'delegate' refers to the remote evaluation of locally defined methods and functions which can be executed both synchronously and asynchronously. The programmer can easily delegate a call to the grid by using a lambda expression or method reference or make use of our Kotlin DSL which allows developers to define jobs in dedicated code blocks. We provide a load balancing mechanism to ensure that none of the machines in the system will be over or under utilized. People from around the world can donate their computing resources to help increase the computational power of the grid. An implementation for RMD is provided on Github [30].

The remainder of this paper is structured in the following way: Section 2 will provide a brief overview of related works. In Section 3, we discuss the features, architecture, and implementation details of the proposed solution. Section 4 contains a performance evaluation of our system with experimental results. In Section 5, you will find advice on practical use and best practices for the proposed solution.

The contributions of this work are as follows:

- RMD is interoperable with other JVM languages
- provides a higher level of abstraction to the programmer, reducing boilerplate, and developer training

– we provide a publicly available implementation of the proposed platform on Github

2 Background and Related Works

There are many HCP and distributed computing frameworks that have been developed over the past few decades. To provide an understanding of the motivations for this project, this section describes related infrastructure.

2.1 Message Passing Interface (MPI)

MPI is a specification for a communication protocol that is widely used for high performance computing applications. It maintains a focus on high performance and scalability [14]. MPI programs are usually designed to follow the Single Program Multiple Data (SPMD) programming paradigm [27]. Each node in the cluster will run the same program and communicate with other nodes using the MPI API. In an MPI based application, each process is assigned a ‘rank’ which is used for the identification of each process. A Process rank ranges from 0 (master node) upto the number of processes (exclusive) in the cluster. Message passing is a low level interface with little abstraction, as such, the programmer is explicitly responsible for all inter-process communications whilst RMD handles communications between nodes from behind the scenes with a high level of abstraction. Utilization of method references, lambda expressions, or the Kotlin DSL help RMD achieve a high degree of abstraction and to minimize boilerplate code. Explicit control over message passing does provide some significant advantages, such as the ability to create a ring topology by targeting specific processes by their rank. For example, in a three node cluster, the master node (rank 0) could pass a message to the process with a rank of 1. The rank 1 process could forward the message to the process of rank 2, and finally, to complete the ring, the rank 2 process could forward the message back to the master node. In the name of simplicity, RMD does not allow for manual communication management. It is possible to provide support for ring topologies in the future.

There are several implementations of the MPI specification that are currently available including MPICH (the reference implementation), OpenMPI, and a few others [14, 15].

2.2 Remote Procedure Calls

Remote Procedure Calls (RPC) is a programming paradigm that provides the programmer with a higher level of abstraction than the MPI and are usually highly transparent [29]. An RPC system provides the developer with a handle to a function that is typically located on another machine for the purposes of invocation. The RPC framework is responsible for managing communications between machines, including sending invocation requests, transmitting and marshalling arguments, unmarshalling and returning results. While RPC does increase abstraction, significant overhead is added to the system.

RPC systems can also be object-oriented, such systems are often called Remote Method Invocation (RMI). An RMI client communicates with a server through the use of a stub. The programmer invokes method calls through the stub object which is responsible for marshalling arguments, sending the invocation request as well as unmarshalling and returning results.

Java RMI is one of many implementations of this technique and operates using a proxy-based architecture [25]. The use of an adapted version of Java RMI in grid computing environments has been proposed in [1], however other Java-based grid programming environments have been shown to outperform Java RMI [26]. Java RMI also lacks many useful features that could make up for its large overhead. It does not support RPC’s for static methods, nor does it support code migration or load balancing. A common interface is required to define methods or services that can be invoked remotely. Classes declared on the server side implement these interfaces and bind instances of their type to the RMI registry. The RMI registry is responsible for declaring and invoking exported services and communicating with clients. The client will perform a lookup request on the remote server to find the appropriate object and acquire the stub that implements the common interface by means of a proxy to the remote object instance.

2.3 BOINC

BOINC (Berkeley Open Infrastructure for Network Computing) is a popular platform for public-resource computing [2]. Public-resource computing (also known as volunteer computing), is a type of distributed computing whereby members of the public can donate their computational resources or data storage, typically for the purposes of scientific supercomputing. BOINC is low-level and lacks the abstraction provided by RMD and requires extensive project configuration. There are many past and present research projects that have utilized the BOINC project, including FightAIDS@Home [11] which hopes to find drugs which are effective in disabling the HIV-1 Protease. Some popular projects include SETI@home [3] a project aimed at searching for extraterrestrial intelligence which utilizes over 5,000,000 processing units. Other projects include WEATHER@home [22] which aims to perform a set of regional climate modeling and to help better understand how climate change will affect weather patterns.

2.4 Related Works

In recent works, domain specific languages have been proposed for high-performance computing. ANTAREX [24] is a recently developed DSL for HPC, however it follows the Aspect Oriented Programming (AOP) paradigm whilst RMD is functional. Chapel [6] is a portable, parallel programming language that can also run in distributed systems such as in a cluster. RMD does have a significant advantage over these DSL's because our infrastructure does not require the use of our DSL. This is possible because the RMD framework is written in Java and provides the DSL as a support mechanism. The DSL works by inserting calls to the RMD framework that would otherwise be written by the developer. Due to the interoperability of JVM languages, developers can work in the languages that they already know and utilize code that was written in another language for their RMD projects. Additionally, utilizing method references and lambda expressions to define jobs in Java provides significant syntactic sugar over many library based approaches to HPC. Other DSLs such as Liszt [10] and DWARF [18] have been proposed to solve narrow HPC problems such as clustering data. Vivaldi [7] is a transcompiled DSL that targets python with a focus on volume processing and visualization.

Other popular HCP frameworks such as Apache Spark [31], Storm [17], and Hadoop [4] are geared towards large-scale data processing. The Hadoop project implements the MapReduce [9] programming model and provides a distributed file system to manage data.

OpenMP [8] is a popular shared-memory HPC framework that uses compiler directives to achieve parallelization. This simplistic approach allows for easy modification to parallelize existing code. OpenMP implementations are available for a variety of programming languages.

3 RMD Framework

3.1 Features

RMD shines in the field grid computing because of its usability features and support for multiple JVM languages. In fact, RMD maintains static typing, supports both synchronous and asynchronous jobs, is configurable, and provides a simple Kotlin DSL to aid the cleanliness of development.

3.1.1 Static Type Safety

The programmer is able to maintain static type safety by using both method references and lambda expressions to denote the job that you wish to delegate. This technique allows the compiler to infer the type information through a functional interface. A functional interface is an interface with only one abstract method and its implementation can be represented by lambda expression or method reference.

3.1.2 Synchronous and Asynchronous Jobs

Both synchronous and asynchronous jobs are supported by the client. Asynchronous jobs execute in a non-blocking manner, allowing the machine to perform useful work while the job is executing remotely. The use of synchronous jobs is also important and is highly applicable to divide-and-conquer algorithms. Such a problem could be divided into several sub-problems, which execute asynchronously on the job servers. Next, a synchronous job could be used to combine the sub-problems into a single result.

3.1.3 Distributed Transparency

RMD is designed to be highly transparent whereby one cannot tell the difference between jobs that are executed locally or by one of many external job servers. Behind the scenes, RMD handles the code migration and distribution of the application workload. The job server will also catch exceptions thrown by a job and return a sanitized stack trace back to the client where it can be re-thrown and analyzed by a developer. This creates the appearance that the task has been executed locally.

3.1.4 Configurability

The implementation provides a few configuration options to developers. In the configuration file, users will define a list of hosts, which could be either Job Servers or Load Balancers. Due to the distributed transparency, both types of entities are acceptable because the client, job server, and load balancer do not distinguish between each other. Sometimes failures may occur in the grid, which is why users can specify the use of one of three protocols to handle failures, such as a lack of external computing resources. Users can choose between a retry protocol, an exception-based model, and finally, local job execution. In an exception-based model, the application will simply throw an exception if there is no available Job Server to process the request. Alternatively, the job request can be sent again, or the job can be executed locally.

3.1.5 Kotlin DSL

In an attempt to provide platform support to other JVM languages, we implemented a Kotlin DSL (domain specific language). The domain specific language in this context is an internal DSL, which means that it is built on top of the Kotlin, and the existing JVM infrastructure to provide domain specific enhancements i.e., grid computing. This approach allows the user to define their jobs within blocks that are specific to their computing application. For the Kotlin DSL, we provide three types of blocks: 'delegate,' 'async,' and 'callback'. The delegate block is a block of code that represents a job that is to be executed synchronously on the remote job server. The last statement in the block is the result to be returned to the programmer. The async and callback blocks are designed to go together because a callback is required to retrieve the result from an asynchronous job. The result of the last statement in the async block is the result sent to the callback. The Kotlin DSL serves to enhance the features of RMD by providing a cleaner way to define jobs with a higher level of abstraction and no need to implement any special interface.

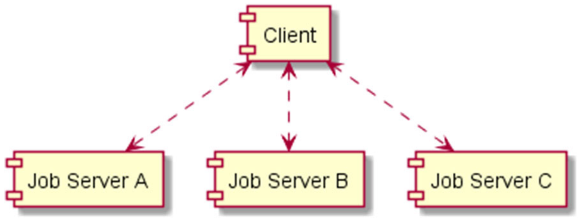


Fig. 1 2-Tier

3.2 Distributed RMD Architecture

The distributed system is designed with a multi-tier architecture and should be used with either a 2-tier (Fig. 1) or 3-tier setup (Fig. 2).

In the 3-tier architecture, the client communicates with the load balancer, which acts as a middleman between the client and job servers. Any scheduling algorithm could be used with the load balancer via dependency injection. It should be noted that in the 2-tier setup, the client can still make use of several

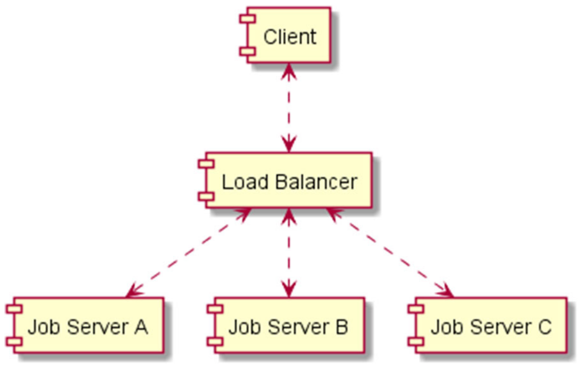


Fig. 2 3-Tier

job servers because it has its own built-in load balancer. This approach requires the client to have prior knowledge about the existence of each job server. This approach will have the added benefit of reducing the amount of communication required to send a job request to the grid because requests will not need to pass through an external load balancer first.

3.3 Implementation

We have provided our own implementation of this project on the Java Virtual Machine with a modular design allowing for a separation of concerns. In total, there are 5 modules including the load balancer, job server, the main client, the Kotlin client (Kotlin DSL support), and the communication module. Each module is responsible for one specific job, and they collectively work together. Since the project is implemented on JVM, it can theoretically support any language that executes on top of the JVM, and not just Java. Our implementation officially supports both the Java and Kotlin programming languages.

3.3.1 Communications

Network communications is an important part of the distributed system. The communication module has the job of guaranteeing that messages are passed from client to server and are handled by the correct handler. Our implementation communicates with the TCP protocol although it is possible to implement the communication module using the UDP protocol. The UDP protocol would add extra challenges to the communications because UDP does not guarantee the order of messages upon arrival.

The communication module implements a very simple request-reply protocol. The protocol (described in Table 1) requires that each message contains the following information: message length, request Id, and serialized message contents. The length of the message indicates the number of bytes to be read, whilst the request Id is used to pair a request with its corresponding reply. The request Id is generated by a counter that is incremented after each additional request is made. The contents of the message are represented by an object that must inherit from either the 'Request,' or 'Response' classes used to provide contextual information to determine whether a message is a request, or a response, and to indicate whether the request has succeeded. Table 2 describes the structure of the messages that will be transmitted back and forth. The content of each message is serialized and deserialized using a framework called Fast-Serialization, which has a greater performance than the native JVM serialization algorithms [23]. This is important because marshalling and unmarshalling of objects can cause significant overhead.

Table 1 Communication protocol

Number of bytes	Content
4	Message Length (N)
4	Request Id
N	Serialized Message Content (See Table 2)

3.3.2 Client

The client module is responsible for providing the developer with the infrastructure required to deploy their applications. The following features are implemented by the client module:

1. Provides the high level RMD API
2. Finds declaring class and method signature (call-site) at runtime from a method reference (to the job)
3. Determines the set of job dependencies
4. Performs code migration
5. Sending job requests

This module is dependent on both the communication module, and the load balancing module. The load balancing module is also used on the client side to allow for use of multiple job servers without requiring an external load balancer to act as the middleman. This allows for the removal of an extra layer of communication from the system but requires the client to have prior knowledge of external job servers.

To allow the developer with a high degree of abstraction, and to maintain static type safety, method references and lambda expressions are used to specify the job that will be executed remotely. When delegating a job, a method reference can be used to refer to existing code, including external libraries. Method references also support the use of static methods, which happens to be one of limitations of Java

Table 2 Messages and their contents

Message name	Contents	Descriptions
Job Request	String	Class containing delegate
	int	Method Index
	byte[]	Serialized arguments
Job Response	boolean	Whether job was successful
	Object	Result produced by job
	Throwable	If exceptions occur
Migration Request	Map	Class names mapped to byte code
Migration Response	boolean	Whether migration was successful

RMI. Alternatively, the developer can use a lambda expression for added syntactic sugar. Since method references are a type of lambda expression, there is no need for RMD to differentiate between the two because they are both represented in the same way after compilation. At runtime, RMD needs to decipher the following information from the job: its method name, the declaring class file, and its method signature. The callsite information can be determined at runtime to avoid any sort of compiler bootstrapping. This is possible because calls to the RMD API's 'delegate' functions only accept our custom functional interfaces which are serializable. Since the lambda expressions can be serialized this information must be present at runtime. This information is required to understand which code needs to be migrated. In addition, a function to be delegated may make use of any dependency, whether it be locally defined, part of a library, or the Java standard library. In Fig. 4 we provide an example program to show the various ways in which class files can be dependent upon each other. To determine the set of dependencies required to execute, we rely on the ASM bytecode manipulation and generation framework [5] which provides the capability to analyze all aspects of a class file. Specifically, we are looking at the following things in each class file:

1. **Inheritance:** parent class, interfaces
2. **Annotations:** of classes, fields, methods, interfaces, and other annotations
3. **Fields:** check the type of field
4. **Methods:** method signature including arguments and return type
5. **Methods:** check the type of local variables
6. **Methods:** instructions referring to other class files such as retrieving a field

It is important to note that jobs may have dependencies which also have their own dependencies, so RMD solves this problem by recursively analyzing each class and ignoring classes that belong to the standard library (don't require migration), and class that have already been analyzed. At this point, information pertaining to the job and it's dependencies are cached, and do not need to be looked up again. However, since the client may be in contact with multiple job servers, we use a set to keep track of which dependencies have been migrated to each job server and only send the required dependencies when they are needed to execute a job. The entire migration process is repeated after each subsequent execution of an RMD application. For this reason, versioning issues will not be apparent as each job server will discard old class files. Figure 3 describes the flow of the migration process (Fig. 4).

3.3.3 Synchronous Java Example

In this example, the function 'factorial' executed with an input of 6 in a blocking manner. A method reference is used to tell the system what code is to be executed.

```
a = delegate(this::factorial, 6)
println("6! = " + a);
```

3.3.4 Asynchronous Java Example

In this code snippet, the main calculation is done asynchronously by the server and the client does not block while waiting for the reply. The client is then free to do other tasks while the server completes the job.

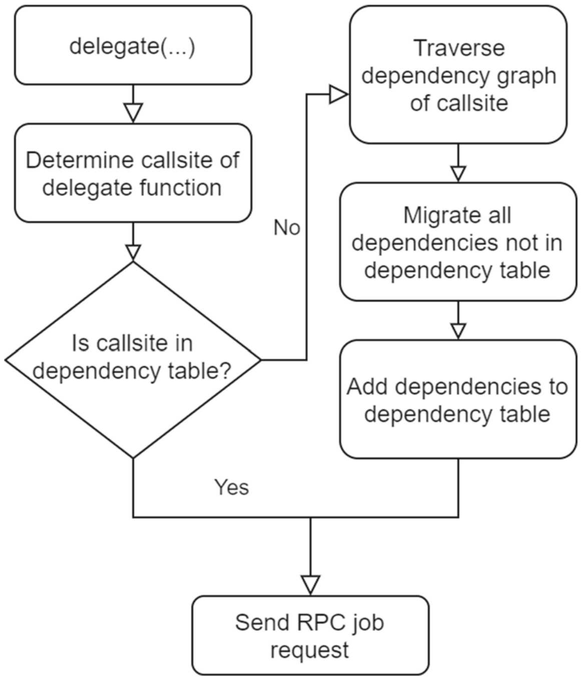

delegate(...)
Determine callsite of delegate function
Is callsite in dependency table?
No
Yes
Traverse dependency graph of callsite
Migrate all dependencies not in dependency table
Add dependencies to dependency table
Send RPC job request

Fig. 3 RMD job delegation process

Both the job and callback functions are defined via lambda expression, with the function parameters in the middle.

```
delegate((a, b) -> a * b,
   5, 9,
   n -> {
   // callback
   println("5 * 9 = " + n);
});
```

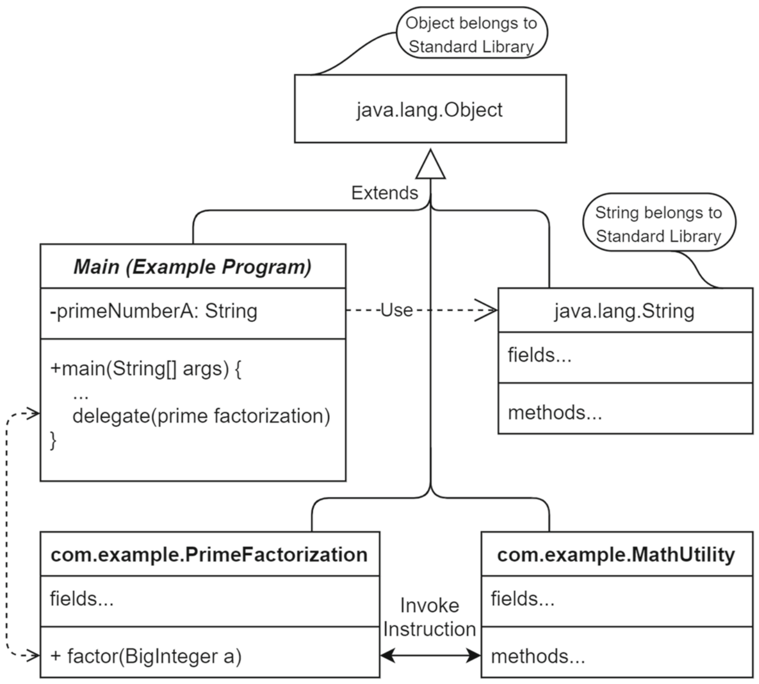
3.3.5 Kotlin Client

The Kotlin client module serves to provide extended support to the Kotlin programming language. This module defines an expressive Kotlin DSL (domain specific language) which serves to make use of the expressive grammar defined by the Kotlin programming language. This module is dependent upon the main Client module to handle any code migration and job requests. The DSL is designed to be extremely simple for users to understand.

3.3.6 Synchronous Kotlin Example

Since Kotlin has extremely expressive syntax, a function that accepts a lambda expression can be written as a block with a body. For example, code in the del-

Fig. 4 Example RMD program with dependencies



egate block would be executed remotely in a blocking manner.

```
val a = 100
val b = 200

val result = delegate {
    // blocking remote execution
    a * b
}

println("$a * $b = $result")
```

3.3.7 Asynchronous Kotlin Example

To execute an asynchronous job, the programmer can define code within an ‘async’ block and optionally provide a ‘callback’ block to acquire the result.

```
val a = 10
val b = 20

async {
   a * b // executes remotely
} callback {
   // executes locally
   println("$a * $b = $it")
}
```

3.3.8 Job Server

The job server has several duties to perform as part of the RMD platform. Firstly, the job server facilitates resource donation for users who choose to run it on their own hardware. It must also handle incoming migration requests and to perform the execution of any job requests. Upon receiving a migration request, the job server attempts to load all classes specified in the request. This is easily accomplished by providing a custom class loader that searches the class map (key=class name, value=class file) instead of the local file system. In the unlikely event that a class fails to load, either because it is corrupted or its dependencies are missing, the job server will reply with a failure message.

After the migration action has been completed, clients can send a number of job requests in parallel to one or more servers. Each job request contains the following information: the method name, the method signature, the name of its declaring class, and the function arguments. This information is required to find the declaring class, and to invoke the method by using reflection. In the event that the class or method is not found, the server will send the failure response. The server executes job requests on a thread-pool which can also gracefully catch any exceptions. If a job throws an exception during its execution, the server will respond with the exception information including the detailed message and stack trace which will be rethrown by the client.

3.3.9 Enforcing Security Constraints

Since the job server is responsible for loading and executing untrusted code, a security manager was implemented to inhibit malicious actors. This feature is important because users would not donate their computing resources without any sort of security guarantee. The JVM provides a few ways to define security policies. Firstly, a developer could declare a security policy file for their program and specify it through the command line arguments. This solution is not acceptable for RMD because we need to simultaneously grant permissions to the RMD platform whilst denying them to all untrusted code. To achieve this level of flexibility we must implement a custom security manager. Whenever an application attempts to invoke privileged code such as network or file system access, the JVM will check the system security manager to determine whether the action is permitted.

The job server has a security policy that will deny all permissions to untrusted code and this policy will not interfere with any of the permissions required by the job server. The security manager operates on a per-thread basis so that it can allow the job server to communicate with clients while at the same time preventing untrusted code to access the internet or other system resources. The JVM allows applications to specify their own custom security manager at runtime which is implemented by inheriting from the **java.lang.SecurityManager** class. Every time a permission check is done by the JVM, RMD’s security manager checks the calling thread’s thread group, and compares it with the thread group that jobs belong to. When a new job arrives, it is dispatched onto the thread-pool where each of its threads belong to the same thread-group. This allows the security manager to differentiate between the RMD infrastructure and untrusted code that may run on the machine.

3.3.10 Load Balancer

The load balancer has the important job of managing the use of resources on the network. This module depends upon the communication module to handle all network IO between clients and job servers. The load balancer accepts connections from clients and forwards their job request based on resource availability. For the simplicity of the RMD load balancer, our prototype implementation only supports a round-robin scheduling algorithm. However, due to our open design, more sophisticated scheduling algorithms can be implemented by the developer. After a client initializes a connection with a load balancer, it must first send a migration request to the load balancer before sending any job requests. This protocol acts the same as direct communication between a client and server, therefore the client does not need to distinguish between a job server or load balancer. The load balancer must verify that a job server has the required code before forwarding a job request to the chosen server. This is a problem because of the one-to-many relationship between the load balancer and job servers. To solve this problem, the load balancer stores a map between a JVM class name, and its bytecode instructions. It also records a list of each class file that the job server has so it can prevent the migration of class files that already exist on the job server. The migration process is lazy and only occurs when a job request occurs to a server that has not previously seen the required class files.

3.3.11 Failure Handling

Failures are inevitable in a large-scale grid computing network. Such failures are most likely to be caused by network errors, which is why a few failure protocols have been designed and implemented. In general, failures during job execution are ignored, and the job request is passed on to the next available job server. However, this is not possible if no computing resources are found on the grid. Handling this type of error can be configured using a configuration file. Currently, there are three supported protocols to handle this type of failure. In the first protocol, jobs are to be executed locally when there are no external computing resources available (Fig. 5).

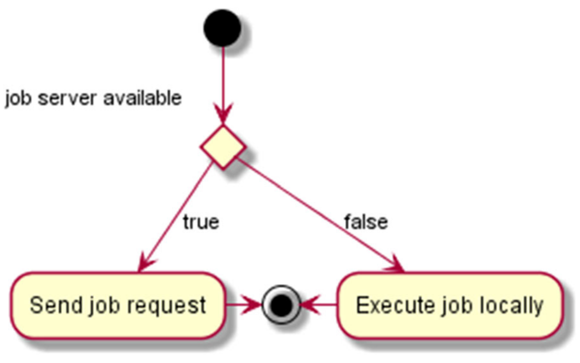


Fig. 5 Local execution on failure

Secondly, such events can be handled with a wait and retry protocol. If users decide to choose this approach, they will not be bogged down by the local execution of the jobs, but they must accept that any network issues may not resolve themselves and no error messages are displayed (Fig. 6).

Finally, the last protocol is the error protocol. If a lack of resources ever occurs, the client will throw an exception indicating that no external resources are available.

Sometimes failure can occur after a job request has been issued. In this event, it is preferred to cut losses

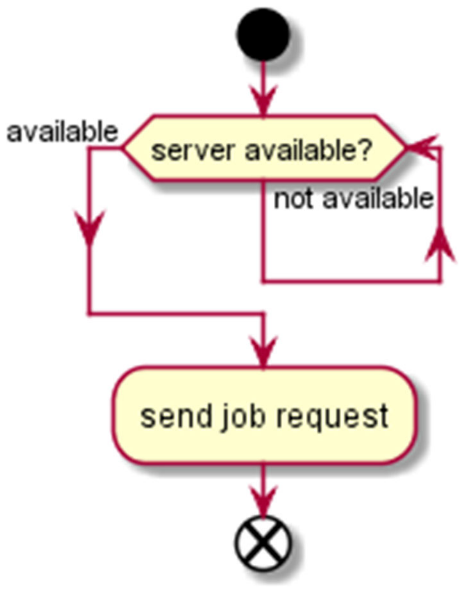


Fig. 6 Wait and retry

and pass on the request to the next available job server. This design decision was chosen because of the uncertainty surrounding the other nodes in the network. A cache-based system could have been implemented, but given that the duration of each job execution is likely to be relatively small, it is unlikely that the issue would be resolved within that time frame. This would require the client to attempt to reconnect to a server to retrieve the result with no guarantees of success (Fig. 7).

4 System Evaluation

The RMD platform provides two forms of significant overhead, initialization overhead and added overhead for each job request. Users are advised to follow the best practices that are discussed in Section 5 to limit the effects of overhead on their job.

4.1 RMD Overhead

RMD jobs will observe a one-time initialization overhead plus the overhead required to send each subsequent job request. Initialization overhead occurs during the first request to delegate a job. Firstly, RMD must locate and connect to external load balancers and/or job servers. At runtime, RMD must also determine the callsite of your job and determine all of its dependencies. There are two determinants of this upfront overhead, including the number of dependencies that your job depends on and the network throughput. In short, the greater the number of dependencies used by your job, the more time it takes to traverse the dependency graph and to send the class files over the network. However, since the code migration process only needs to occur once, the overhead applies a one-time upfront delay to the execution of the program. Classes that are part of the Java or Kotlin standard library are not migrated to the job server, as they already exist on the server side.

Each job request has three forms of overhead after the migration process has been completed. Firstly, the job must be looked up in the cache to verify that the code has been previously migrated to the external job server. Next, the arguments of the job must be serialized, which does provide a significant amount of overhead when compared to low level platforms like MPI. If you intend to write jobs that depend on large amounts of data, the cost of marshalling/unmarshalling will dominate the other forms of overhead.

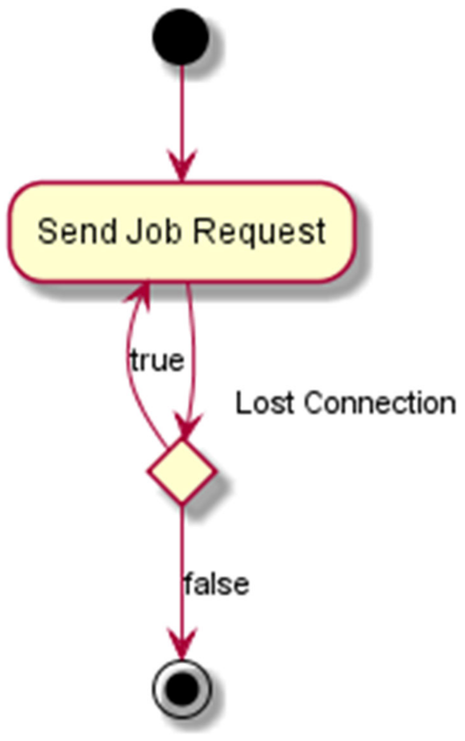


Fig. 7 Disconnect protocol

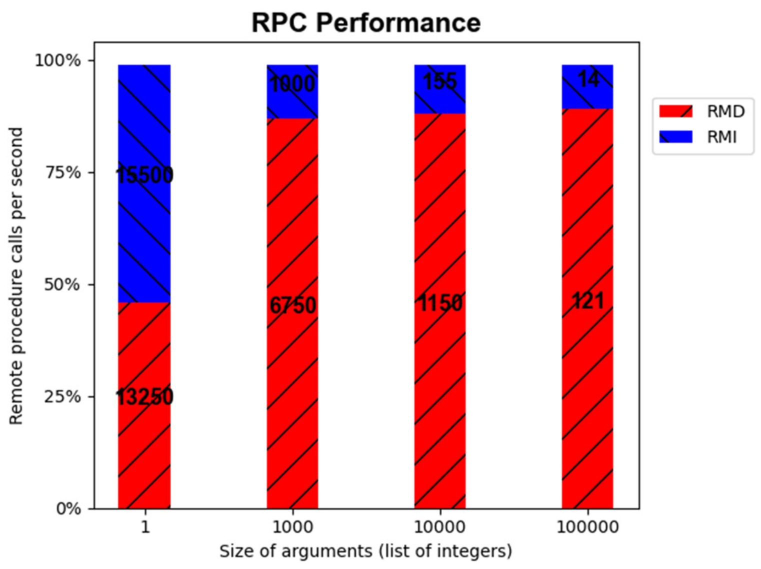
4.2 Experimental Results and Analysis

Behind the scenes, RMD is essentially an RPC system with load balancing and code migration. To test the remote evaluation performance of RMD, we measured a number of remote procedure calls that could be invoked one after the other on a single machine. We then compared the performance of RMD to Java RMI using the same benchmark environment (Table 3). To compare each platform, we counted the number of invocations per second of two no-op benchmark functions. The first benchmark function, shown in Fig. 8, takes a list of integers (non primitive) as input and sends them back to the client. The second benchmark function, shown in Fig. 9, takes an array of primitive

Table 3 Benchmark environment

OS	Ubuntu 19.04
JDK	Open-JDK 8
CPU	Intel i7 6700K

Fig. 8 RMD vs RMI: RPCs per second



integers as input and sends them back to the client. This benchmark serves to show how the performance scales as the size and complexity of the input to the benchmark function changes.

From the results above you can see that when marshalling and unmarshalling has a greater effect on the performance (due to increased data size or complexity), RMD makes significant gains in RPC performance vs Java RMI. However, if the data being transferred is primitive, the performance cost for marshalling/unmarshalling is much less and the overhead of RMD becomes much more apparent.

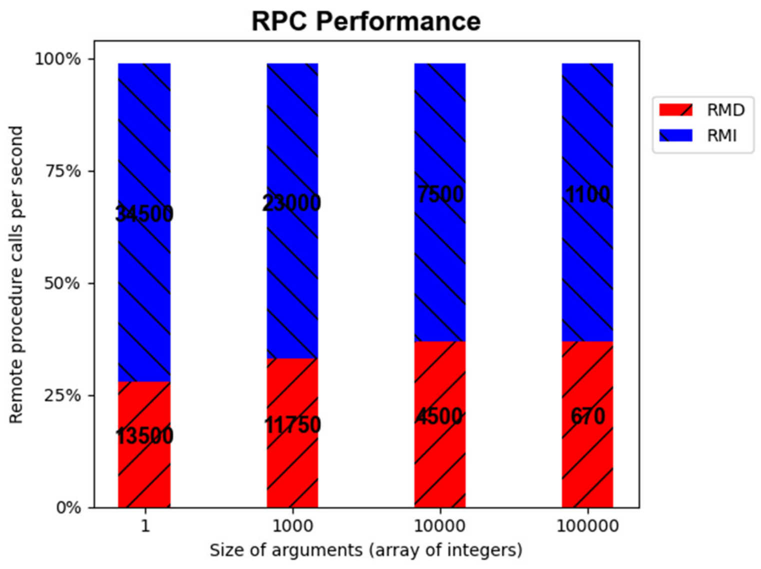
5 Practical use

Although RMD is simple to set up and configure, best practices should be followed to improve the efficiency of your system.

5.1 Writing Jobs

Before you begin implementing your jobs, check for existing algorithms because you may not need to modify their implementation to delegate their workload. You can delegate the jobs of existing code by

Fig. 9 RMD vs RMI: RPCs per second



making use of a method reference. If you detect an unreasonable one-time delay prior to the delegation of your job, try to limit the number of dependencies used by your job. More dependencies will affect the initialization overhead cost as this will require the client to detect, locate, and migrate more code at runtime. Jobs should be designed to take longer than the network delay however, if they are too long, a network or server-side failure could result in a large delay because the job will need to be redeployed. If you suspect the delays are due to network throughput issues, try using 2-tier setup as shown in Fig. 1.

If you plan to execute more than one asynchronous job at the same time, pay attention to common concurrency pitfalls. Your callback methods are not guaranteed to run on the same thread, and should synchronize the modification of variables, either by using a synchronized block or atomic references. Failure to do so may result in undesirable behaviour. For example, consider a program that executes 1000 asynchronous jobs and their callback function increments a global counter. It is extremely likely that your counter may not be equal to 1000 after completion.

Finally, pay attention to the context in which your job executes. Your job should not attempt to modify the state of any field outside the scope of the job. Instead, a result should always be returned by the job. You may however accept the state of local and global variables as input, so long as they are constant and immutable. Modification of the state of any object or reference outside the job will not transfer back to the master program.

5.2 Setting up Servers

Both the load balancer and job server modules act as standalone applications. You simply need to download and execute the build artifacts. If you are planning on hosting a job server, it should work out of the box, without any configuration. To host a load balancer, you will need a very simple configuration file to specify the location of external job servers.

5.2.1 Load Balancer

The configuration file of the load balancer should point to the known job server in the grid. By making use of the load balancer, clients do not need to know about all the job servers on the network. This allows administrators to easily scale up/down the size of the grid without taking the clients offline for maintenance. The geographic location of the load balancer should be taken into account because the added networking delay may not be worth the advantages of providing an external load balancer.

For advanced usage, the load balancer's behaviour can be programmatically modified to exhibit behaviour that is more beneficial to your use case. For example, you may want to implement your own scheduling algorithm that prioritizes job servers by lowest CPU utilization, or lowest network latency.

6 Conclusion and Future Work

We have created an intuitive interface for users to efficiently perform jobs securely on the grid. We implemented a load balancer to prevent servers from becoming over or underutilized, which keeps our servers running as efficiently as possible. RMD also supports resource donation, in which a client could opt-in to receive job requests and complete them for other users. This will allow the computational power of the grid to expand and run more jobs concurrently. RMD makes it easier for researchers to solve parallelizable problems by providing a highly abstract and simple way to define and execute different tasks on a cluster or grid computer.

If a researcher is in need of an easy to set up cluster or grid computing platform, they may benefit from the usability features provided by RMD. However, RMD is not without limitations. For example, it is not the tool for processing large amounts of data.

In the future, we would like to address some of the shortcomings of the RMD framework. Sometimes we wish to develop jobs that can be provided with continuous updates between the client and server during their execution. We aim to support communication channels such that objects could be transferred between the master program and its remote jobs. Unlike popular HPC platforms such as MPI, RMD favors a high level of abstraction for usability reasons. This prevents the programmer from having explicit control over the destination of each message. Without low level messaging capabilities, the programmer cannot create a ring topology. We envision further development and focus on the Kotlin DSL to accomplish the creation of ring topologies in a highly abstract way. Finally, RMD does not address optimization concerns that are raised with a high abstract design that does not provide the developer with low level control. We aim to provide a network-based grid optimizer such as the techniques defined in [12].

References

1. Alt, M., Gorlatch, S.: Adapting java rmi for grid computing. Fut. Gener. Comp. Syst. **21**, 699–707 (2005). https://doi.org/10.1016/j.future.2004.05.010
2. Anderson, D.P.: Boinc: a system for public-resource computing and storage. In: Fifth IEEE/ACM International Workshop on Grid Computing, pp. 4–10. https://doi.org/10.1109/GRID.2004.14 (2004)
3. Anderson, D.P., Cobb, J., Korpela, E., Lebofsky, M., Werthimer, D.: Seti@ home: an experiment in public-resource computing. Commun. ACM **45**(11), 56–61 (2002)
4. Borthakur, D.: The hadoop distributed file system: Architecture and design. Hadoop Proj. Website **11**(2007), 21 (2007)
5. Bruneton, E.: ASM 4.0 A Java bytecode engineering library (2011)
6. Chamberlain, B.L., Callahan, D., Zima, H.P.: Parallel programmability and the chapel language. Int. J. High Perform. Comput. Appl. **21**(3), 291–312 (2007)
7. Choi, H., Choi, W., Quan, T.M., Hildebrand, D.G., Pfister, H., Jeong, W.K.: Vivaldi: a domain-specific language for volume processing and visualization on distributed heterogeneous systems. IEEE Trans. Visual. Comput. Graph. **20**(12), 2407–2416 (2014)
8. Dagum, L., Menon, R.: Openmp: an industry standard api for shared-memory programming. IEEE Comput. Sci. Eng. **5**(1), 46–55 (1998)
9. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
10. DeVito, Z., Joubert, N., Palacios, F., Oakley, S., Medina, M., Barrientos, M., Elsen, E., Ham, F., Aiken, A., Duraisamy, K., Darve, E., Alonso, J., Hanrahan, P.: Liszt: A domain specific language for building portable mesh-based pde solvers. In: SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–12 (2011)
11. Estrada, T., Taufer, M., Anderson, D.P.: Performance prediction and analysis of boinc projects: an empirical study with emboinc. J. Grid Comput. **7**(4), 537 (2009). https://doi.org/10.1007/s10723-009-9126-3
12. Ferrari, T., Giacomini, F.: Network monitoring for grid performance optimization. Comput. Commun. **27**(14), 1357–1363 (2004)
13. Foster, I., Zhao, Y., Raicu, I., Lu, S.: Cloud computing and grid computing 360-degree compared. Cloud Computing and Grid Computing 360-Degree Compared, 5. https://doi.org/10.1109/GCE.2008.4738445 (2009)
14. Graham, R.L., Woodall, T.S., Squyres, J.M.: Open mpi: A flexible high performance mpi. In: Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics, Springer, Berlin, PPAM'05, pp 228–239. https://doi.org/10.1007/11752578_29 (2006)
15. Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A high-performance, portable implementation of the mpi message passing interface standard. Parallel Comput. 22(6):789–828, https://doi.org/10.1016/0167-8191(96)00024-5, http://www.sciencedirect.com/science/article/pii/0167819196000245 (1996)
16. Günther, S., Cleenewerck, T.: Design principles for internal domain-specific languages: A pattern catalog illustrated by ruby. In: Proceedings of the 17th Conference on Pattern Languages of Programs, PLOP'10, pp 3:1–3:35. ACM, New York. https://doi.org/10.1145/2493288.2493291 (2010)
17. Iqbal, M.H., Soomro, T.R.: Big data analysis: Apache storm perspective. Int. J Comput. Trends Technol. **19**(1), 9–14 (2015)
18. Islam, S., Balasubramaniam, S., Goyal, P., Sati, M., Goyal, N.: A Domain Specific Language for Clustering. In: Krishnan, P., Radha Krishna, P., Parida, L. (eds.) Distributed Computing and Internet Technology, pp. 231–234. Springer International Publishing, Cham (2017)
19. Kong, L., Mapetu, J.P.B., Chen, Z.: Heuristic load balancing based zero imbalance mechanism in cloud computing. J Grid Comput, 1–26 (2019)
20. Krašovec, B., Filipčič, A.: Enhancing the grid with cloud computing. J. Grid Comput. **17**(1), 119–135 (2019)
21. Lindholm, T., Yellin, F., Bracha, G., Buckley, A.: The Java Virtual Machine Specification, Java SE 8 Edition, 1st edn. Addison-Wesley Professional (2014)
22. Massey N, Jones R, Otto F, Aina T, Wilson S, Murphy J, Hassell D, Yamazaki Y, Allen M: weather@ home—development and validation of a very large ensemble modelling system for probabilistic event attribution. Quart. J. R. Meteorol. Soc. **141**(690), 1528–1545 (2015)
23. Moeller, R., et al: Fast-serialization. https://github.com/RuedigerMoeller/fast-serialization (2014)
24. Silvano, C., Agosta, G., Bartolini, A., Beccari, A.R., Benini, L., Besnard, L., Bispo, J., Cmar, R., Cardoso, J.M., Cavazzoni, C., et al.: The antarex domain specific language for high performance computing. Microprocess. Microsyst. **68**, 58–73 (2019)
25. Stevenson A, MacDonald S: Smart proxies in java rmi with dynamic aspect-oriented programming. In: 2008 IEEE International Symposium on Parallel and Distributed Processing, pp. 1–6. IEEE (2008)

26. Van Nieuwpoort, R., Maassen, J., Wrzesinska, G., Hofman, R., Jacobs, C., Kielmann, T., Bal, H.: Ibis: a flexible and efficient java-based grid programming environment. Concurr. Comput. Pract. Exper. **17**, 1079–1107 (2005). https://doi.org/10.1002/cpe.860
27. Vinter, B., Bjørndalen, J., Anshus, O., Larsen, T.: A Comparison of Three MPI implementations., IOS Press, Netherlands, pp. 127–136 (2004)
28. W Stamos, J., Gifford, D.: Implementing remote evaluation. IEEE Trans Softw. Eng. **16**, 710–722 (1990). https://doi.org/10.1109/32.56097
29. Wilbur, S.R., Bacarisse, B.: Building distributed systems with remote procedure call. Softw. Eng. J. **2**, 148–159 (1987)
30. Wood, B., Watling, B., Winn, Z., Messiha, D., H Mahmoud, Q., Azim, A.: Source code for the rmd project. https://github.com/BradleyWood/RMD (2019)
31. Zaharia, M., Xin, R.S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M.J., Ghodsi, A., Gonzalez, J., Shenker, S., Stoica, I.: Apache spark: a unified engine for big data processing. Commun ACM **59**(11), 56–65 (2016). https://doi.org/10.1145/2934664