Elastic Remote Methods

K. R. Jayaram*
HP Labs, Palo Alto, CA. jayaramkr@hp.com

Abstract. For distributed applications to take full advantage of cloud computing systems, we need middleware systems that allow developers to build elasticity management components right into the applications. This paper describes the design and implementation of ElasticRMI, a middleware system that (1) enables application developers to dynamically change the number of (server) objects available to handle remote method invocations with respect to the application's workload, without requiring major changes to clients (invokers) of remote methods, (2) enables flexible elastic scaling by allowing developers to use a combination of resource utilization metrics and fine-grained application-specific information like the properties of internal data structures to drive scaling decisions, (3) provides a high-level programming framework that handles elasticity at the level of classes and objects, masking low-level platform specific tasks (like creating and provisioning virtual machine (VM) images) from the developer, and (4) increases the portability of ElasticRMI applications across different private data centers/IaaS clouds and enables efficient cluster utilization through Apache Mesos [5].

1 Introduction

Elasticity, the key driver of cloud computing, is the ability of a distributed application to dynamically increase or decrease its use of computing resources, to preserve its performance in response to varying workloads. Elasticity can either be *explicit* or *implicit*.

Implicit vs. Explicit Elasticity. Implicit elasticity is typically associated with a specific programming framework or a Platform-as-a-Service (PaaS) cloud. Examples of frameworks providing implicit elasticity in the domain of "big data analytics" are map-reduce (Hadoop [15] and its PaaS counterpart Amazon Elastic Map Reduce [11]), Apache Pig [3], Giraph [14], etc. Implicit elasticity is handled by the PaaS implementation and is not the responsibility of the programmer. Despite being unable to support a wide variety of applications and computations, each of these systems simplifies application development and deployment, and employs distributed algorithms for elastic scaling that are optimized for its programming framework and application domain.

Explicit Elasticity, on the other hand, is typically associated with Infrastructure-as-a-Service (IaaS) clouds and/or private data centers, which typically provide elasticity at the granularity of virtualized compute nodes (e.g., Amazon EC2) or virtualized storage (e.g., Amazon Elastic Block Store (EBS)) in a way that is agnostic of the application using these resources. It is the application developer's or the system administrator's responsibility to implement robust mechanisms to monitor the application's performance at runtime, request the addition or removal of resources, and perform *load-balancing*, i.e., redistribute the application's workload among the new set of resources.

* Thanks to Patrick Eugster and Hans Boehm for helpful feedback.

Programmable Elasticity. Existing *distributed* applications cannot take advantage of the cloud without elasticity being *retrofitted* to them. In fact, to *optimize* the performance of *new or existing* distributed applications while *deploying or moving* them to the cloud, engineering robust elasticity management components is essential. This is especially vital for applications that do not fit the programming model of (implicit) elastic frameworks like Hadoop, Pig, etc., but require high performance (high throughput and low latency), scalability and elasticity – the best example is the class of *datacenter infrastructure applications* like key-value stores (e.g., memcached [26], Hyperdex [28]), consensus protocols (e.g., Paxos [18]), distributed lock managers (e.g., Chubby [7]) and message queues. Elasticity frameworks which rely on externally observable resource utilization metrics (CPU, RAM, etc.) are insufficient for such applications (as we demonstrate empirically in Section 5). A distributed key value store, for example, may have high CPU utilization when there is high contention to update a certain set of "hot keys". Relying on CPU utilization to simply add additional compute nodes will only degrade its performance further. Hence, there is an emerging need for a elasticity framework that bridges the gap between implicit and explicit elasticity, allowing the use *fine grained* application specific metrics (e.g., size of a queue/heap, number of aborted transactions or average number of attempts to acquire certain locks) to build an elasticity management component right into the application without compromising (1) security, by revealing application-level information to the cloud service provider, and (2) portability across different cloud vendors.

Why RMI? Remote Procedure Call (RPC) [1] and its object-oriented incarnation Remote Method Invocation (RMI) are often criticized because their underlying concept of remote object references can blur address space boundaries, thus covertly introducing direct dependencies across nodes. Nonetheless, they remain a popular paradigm [12] for distributed programming, because of their simplicity – typical interaction in OO settings, remote or local, will result in invocations/executions of methods of some sorts, whether they take place on a proxy or a special shared space abstraction. Their popularity has led to the development of Apache Thrift [12] by Facebook with support for RPC across *different* languages, and the design of cloud computing paradigms like RAMCloud [21] with support for low-latency RPC. However, high-level support for elasticity is limited in existing RPC-like frameworks; and such support is vital to engineering efficient distributed applications and migrating existing applications to the cloud.

Contributions. This paper makes the following technical contributions:

1. ElasticRMI – A framework for elastic distributed objects in Java:
   (a) with the same simplicity and ease of use of the Java RMI, handling elasticity at the level of classes and objects, while supporting implicit and explicit elasticity. (Section 2)
   (b) that enables application developers to dynamically change the number of (server) objects available to handle remote method invocations with respect to the application's workload, without requiring any change to clients (invokers) of remote methods. (Section 3)

(c) enables flexible elastic scaling by allowing developers to use a combination of resource utilization metrics and fine-grained application-specific information to drive scaling decisions. (Section 3)

2. A runtime system that handles all the low-level mechanics of instantiating elastic objects, monitoring their workload, adding/removing additional objects as necessary and load balancing among them. (Section 4)
3. Performance evaluation of ElasticRMI using elasticity metrics defined by SPEC [29] and using four *existing* real world applications – Marketcetera financial order processing, Paxos consenus protocol, Hedwig publish/subscribe system and a distributed coordination service. (Section 5)

2 ElasticRMI – Overview

ElasticRMI is a framework for engineering elastic object-oriented distributed applications. Figure 1 illustrates the architecture of ElasticRMI. In designing ElasticRMI, our goals are (1) g-highlevel: to retain the simple programming model of Java RMI, and mask the low level details of implementing elasticity like workload monitoring, load balancing, and adding/removing objects from the application developer. (2) g-client: require minimal changes, if any, to the clients of an object (3) g-portable: make ElasticRMI applications as portable as possible across different IaaS cloud implementations.

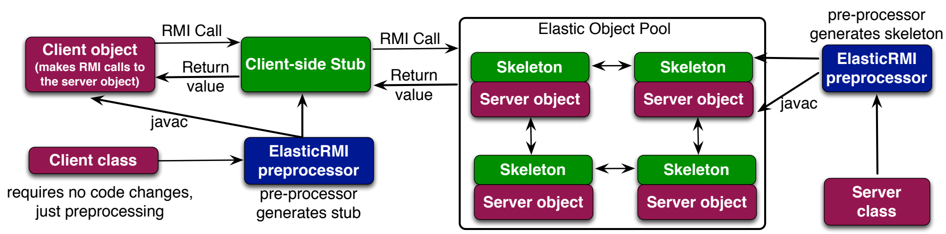


Fig. 1: ElasticRMI – Overview

2.1 Elastic Classes and Object Pools

Like any Java application, an ElasticRMI application consists of multiple components (implemented as classes) interacting with each other. The basic elasticity abstraction in ElasticRMI is an elastic class. An elastic class is also a remote class, and (some) of its methods may be invoked remotely from another JVM. The key difference between an elastic and a regular remote class is that an elastic class is instantiated into a pool of objects (referred to as elastic object pool), with each object executing on a separate JVM. But, the presence of multiple objects in an elastic object pool is transparent to its clients, because the pool behaves as a single remote object. Clients can only interact with the entire object pool, by invoking its remote methods. The interaction between a client and an elastic object pool is *unicast* interaction, similar to Java RMI. The processing of the method invocation, i.e., the method execution happens at a single object in the elastic object pool chosen by the ElasticRMI runtime, and not the client. ElasticRMIs runtime can redirect incoming method invocations to one of the objects in the pool, depending on various factors and performance metrics (Section 4 describes load balancing in detail). The runtime automatically changes the size of the pool depending on the pool's "workload". ElasticRMI is different from other frameworks which replicate a Java RMI server object [30],[20], where the same method invocation is multicast and consequently executes on multiple (replicated) objects.

2.2 Shared State and Consistency

In Java RMI, the *state* of a remote (server) object is a simple concept, because there is a single JVM on which the object resides, and there is exactly one copy of all its fields. Clients of the remote object can set the value of instance fields by calling a remote method, and access the values in subsequent method calls. In ElasticRMI, on the other hand, the entire object pool should appear to the client as a single remote object – thereby necessitating coordination between the objects in the pool to consistently update the values of instance fields. For consistency, we employ an *shared* in-memory key-value store (HyperDex [28] in our implementation) to store the state (i.e., `public`, `private`, `protected` and `static` fields) of the elastic remote object pool. The key-value store is *not* used to store local variables in methods/blocks of code and parameters in method declarations. Local variables are instantiated on the JVM in which the object resides. The key-value store is shared between the objects in the pool, and executes on a separate JVM.

2.3 Stubs and Skeletons

ElasticRMI modifies the standard mechanisms used to implement RPCs and Java RMI, as illustrated by Figure 1. The ElasticRMI *pre-processor* analyzes elastic classes to generate *stubs* and *skeletons* for client-server communication. As in Java RMI, a stub for a remote object acts as a client's local representative or proxy for the remote object. The caller invokes a method on the local stub which is responsible for carrying out the method call on the remote object. However, ElasticRMI's stubs are also capable of performing load balancing among the objects in the elastic pool. When the stub's method is invoked, it initiates a connection with the remote JVM, serializes and marshals parameters, waits for the result of the method invocation and unmarshals the return value/exception before returning it to the sender. It is important to note that the stub is generated by the ElasticRMI preprocessor and is different from the client application, to which the entire object pool appears as a single object, i.e., the existence of a pool of objects is known to the stub but not to the client application. In the remote JVM, each object in the pool has a corresponding skeleton, which in addition to the duties performed in regular Java RMI, can also perform dynamic load balancing based on the CPU utilization of the object and redirect all further method invocations to other objects in the pool after ElasticRMI decides to shut it down in response to decreasing workload.

2.4 Instantiation of Object Pools in a Cluster

An elastic class can only be instantiated by providing a minimum and maximum number of objects that constitute its elastic object pool. Obviously, instantiating all objects in the pool on separate JVMs on the *same* physical machine may degrade performance. Hence, ElasticRMI attempts to instantiate each object in a virtual node in a compute cluster. Virtual nodes can be obtained either (1) from IaaS clouds by provisioning and instantiating virtual machines, or (2) from a cluster management/resource sharing system like Apache Mesos [5]. Our implementation of ElasticRMI uses Apache Mesos [5] because it supports both clusters of physical nodes (in private data centers) or virtual nodes (from IaaS clouds). Mesos can also be viewed as a thin resource-sharing layer that manages a cluster of physical nodes/virtual machines. It divides these nodes into "slices" (called *resource offers* or slave nodes [5]), with each resource offer containing a configurable reservation of CPU power (e.g., 2 CPUs at 2GHz), memory (e.g., 2GB RAM), etc. on one of the nodes being managed. Mesos implements the "slice" abstraction by using Linux Containers (http://lxc.sourceforge.net) to implement lightweight virtualization, process isolation and resource guarantees (e.g., 2CPUs at 2GHz) [5]. While instantiating an elastic class, the ElasticRMI runtime requests Mesos for a specified number of slave nodes, instantiating an object on each slave node.

Mesos aids in portability of ElasticRMI applications, just like the JVM aids the portability of Java applications. Mesos can be installed on private data centers and many public cloud offerings (like Google Compute Engine, Amazon EC2, etc.). As long as Mesos is available, ElasticRMI applications can be executed, making them portable across different cloud vendors (Recall goal g-portable).

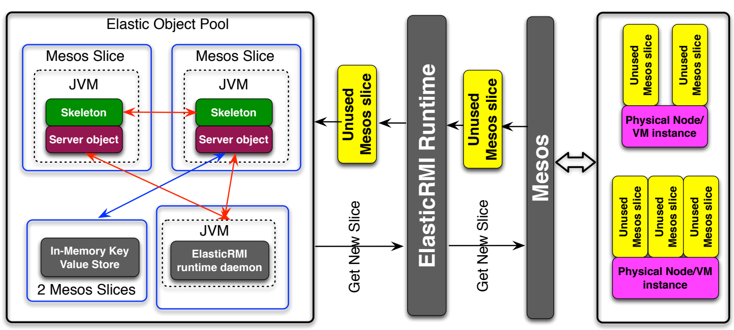


Fig. 2: ElasticRMI – Server Side

2.5 Automatic Elastic Scaling

The key objective of ElasticRMI is to change the number of objects in the elastic pool based on its workload. The "workload" of an elastic object can have several application-specific definitions, and consequently, ElasticRMI allows programmers to define the workload of an elastic class and specify the conditions under which objects should be added or removed from the elastic pool. This can be done by overriding select methods in the ElasticRMI framework, as discussed in detail in (the next) Section 3.1. During the lifetime of the elastic object, the ElasticRMI runtime monitors a elastic object pool's workload and decides whether to change its size either based on default heuristics or by applying the programmer's logic by invoking the overridden methods discussed above.

If a decision has been made to increase the size of the pool, the ElasticRMI runtime interacts with the Mesos master node to request additional compute resources. If the request is granted, ElasticRMI runtime instantiates the additional object, and adds it to the pool (See Figure 2 for an illustration). If the decision is to remove an object, ElasticRMI communicates with its skeleton to redirect subsequent remote method calls to other objects in the pool. Once redirection starts, ElasticRMI sends a SHUTDOWN message to the object. The object acknowledges the message, and waits for all pending remote method invocations to finish execution or throw exceptions indicating abnormal termination. Then the object notifies the ElasticRMI runtime that it is ready to be shutdown. ElasticRMI terminates the object and relinquishes its slice to Mesos. This slice is then available to other elastic objects in the cluster, or for subsequent use by the same elastic object if a decision is made in the future to increase the size of its pool.

3 Programming With ElasticRMI

This section illustrates the use of ElasticRMI for both implicit and explicit elasticity through examples, along with an overview of how to make such decisions with a *global* view of the entire application. Our implementation also includes a preprocessor similar to `rmic` which in addition to generating stubs and skeletons, converts ElasticRMI programs into plain Java programs that can be compiled with the `javac` compiler.

3.1 ElasticRMI Class Hierarchy

The ElasticRMI API follows a class hierarchy similar to Java RMI, which is intentional – to facilitate its adoption among RMI developers. A distributed application built using ElasticRMI consists of interfaces declaring methods and classes implementing them. The interfaces and classes used for specifying the remote behavior of the ElasticRMI system are defined in the `java.elasticrmi` package hierarchy, shown in Figure 3. An elastic interface is one that declares the set of methods that can be invoked from a remote JVM (client). All elastic interfaces must extend ElasticRMI's marker interface – `java.elasticrmi.Elastic`, either directly or indirectly. The `java.elasticrmi` package also provides the `ElasticObject` class, which extends `java.rmi.UnicastRemoteObject` and overrides all its methods. All the basic functionalities of ElasticRMI are actually implemented in this class. When any application-defined class extends `ElasticObject`, all the ElasticRMI code gets inherited and becomes available. A class becomes elastic by implementing one or more elastic interfaces and by extending `java.elasticrmi.ElasticObject`. `ElasticObject` also implements the `Serializable` interface; so any of its application-defined descendants also automatically implements `Serializable`, thus supporting marshalling for application-defined types. `ElasticObject` also prevents the methods of Java's `Object` class from being directly inherited into the elastic class, thereby, preventing

```java
interface Elastic extends Remote { } //marker interface used by preprocessor
//Optionally implemented by the application
abstract class Decider extends UnicastRemoteObject implements Elastic {
  abstract int getDesiredPoolSize(ElasticObject o);
}
abstract class ElasticObject extends UnicastRemoteObject {
  ElasticObject() //Default constructor
  //Constructor to notify the runtime to consult a Decider object for scaling
       decisions
  ElasticObject(Decider d) {...}

  void setMinPoolSize(int s) //Set min pool size
  void setMaxPoolSize(int s) //Set max pool size
  void setBurstInterval(float ival) //Make scaling decisions every 'ival' ms
  void setCPUIncrThreshold(float t) //Add objects when CPU util > t
  void setCPUDecrThreshold(float t) //Remove objects when CPU until < t
  void setRAMIncrThreshold(float t) //Add objects when RAM util > t
  void setRAMDecrThreshold(float t) //Remove objects when RAM until < t

  float getAvgCPUUsage() //Get CPU util averaged over burst interval
  public float getAvgRAMUsage() //Get RAM util averaged over burst interval
  int getPoolSize() //Get pool size
  //Returns average # of calls to each remote method over the burst interval
  HashMap<String,float> getMethodCallStats() {...}

  //Called by the runtime to poll each object about changes to the size of
  //the pool. Can return positive or negative integers
  abstract int changePoolSize();
}
```

Fig. 3: A snapshot of the ElasticRMI server-side API. Client code does not need any modifications. Other details omitted for brevity.

methods of `Object` class from being invoked remotely – methods like `hashcode()` need different implementations for elastic remote objects.

3.2 Programming With Implicit Elasticity

```java
class CacheImplicit extends
  ElasticObject {
  CacheImplicit() {
    setMinPoolSize(5);
    setMaxPoolSize(50);
  }
  ...
}
```

Fig. 4: Example of a distributed cache class which relies on ElasticRMI's implicit elasticity.

ElasticRMI uses coarse-grained performance metrics for implicit elastic scaling, which it supports by default. The default coarse-grained metric used by ElasticRMI is average CPU utilization across the objects in an elastic pool. ElasticRMI measures the average CPU utilization of each object in the elastic pool every 60s – objects are added (in increments of 1 object) when the average utilization exceeds 90% and removed when average utilization falls below 60%. ElasticRMI uses a time interval, referred to as the *burst interval* (the default is 60s as mentioned above) to decide whether to change the size of elastic object pool.

Figure 4 shows an example of a distributed cache class (e.g., a web cache, content/object cache) that relies on ElasticRMI's implicit elasticity mechanisms. The programmer simply implements a cache store based on some well-known algorithm without worrying about adapting to new resources and load balancing. The programmer only has to specify the minimum and maximum size of the elastic object pool, and ElasticRMI uses its default burst interval and utilization metrics to make scaling decisions.

3.3 Programming With Explicit Elasticity

ElasticRMI also allows programmers to *explicitly* define the workload of an elastic class and specify the conditions under which objects should be added or removed from the elastic pool. Workload definitions can either be coarse-grained or fine-grained.

Coarse-grained Metrics. A programmer can override the default burst interval, and the average CPU utilization thresholds used for changing the number of objects in the pool by calling the appropriate methods (`setBurstInterval(...)`, `setCPUIncrThreshold(...)` and `setCPUDecrThreshold(...)`) in `java.elasticrmi.ElasticObject` which is available in all elastic classes since they extend `ElasticObject` (see Figure 3).

Figure 5 shows an example of a cache class that changes the CPU and memory (RAM) utilization thresholds that trigger elastic scaling. The core ElasticRMI API includes specific methods to set CPU and memory thresholds because they are commonly used for elastic scaling – if both CPU and RAM thresholds are set, the runtime interprets them using a logical OR, i.e., in the example shown in Figure 5, the ElasticRMI runtime increases the size of the pool by in increments of 1 object every five minutes, either if average CPU utilization exceeds 85% or if average memory utilization exceeds 70% across the JVMs in the elastic object pool.

```
class CacheExplicit1 extends
  ElasticObject {
  CacheExplicit1() {
    setMinPoolSize(5);
    setMaxPoolSize(50);
    setBurstInterval(5*60*1000); //5
        mins
    setCPUIncrThreshold(85);
    setRAMIncrThreshold(70);
    setCPUDecrThreshold(50);
    setRAMDecrThreshold(40);
  }
  ...
}
```

Fig. 5: Example of a distributed cache class which relies on ElasticRMI's explicit elasticity support. This example uses coarse-grained metrics to make scaling decisions

Fine-grained Metrics. Using CPU and memory utilization is a coarse-grained approach to elastic scaling. ElasticRMI provides additional support, through the `changePoolSize` method (see Figure 3) which can be overridden by any elastic class. The runtime periodically invokes `changePoolSize` to poll each object in the elastic object pool, about desired changes to the size of the pool. The method returns an integer – positive or negative corresponding to increasing or decreasing the pool's size. The logic used to decide on elastic scaling is left to the developer, and it may be based on (1) parameters of the JVM on which each object resides, (2) properties of shared instance fields of the elastic object, or of data structures used by the object, e.g., number of pending client operations stored in a queue, and (3) metrics computed by the object like average response time, throughput, etc. ElasticRMI allows classes to use only a single decision mechanism for elastic scaling, i.e., if `changePoolSize` is overridden, then scaling based on CPU/Memory utilization is disabled. The ElasticRMI runtime periodically (determined by the burst interval) calls `changePoolSize` on each object in the elastic object pool, and averages the return values to determine the number of objects that have to be added/removed from the pool. This mechanism, however, makes scaling decisions *local to an elastic class*. Making application-level scaling decisions is described in (the next) Section 3.3.

```
public class CacheExplicit2 extends ElasticObject {
  float avgLockAcqFailure, avgLockAcqLatency
  public int changePoolSize() {
    HashMap<String, float> sMap;
    sMap = getMethodCallStats();

    float putLatency = sMap.get("put").getLatency();
    float getLatency = sMap.get("get").getLatency();
    if(putLatency > 100 || putLatency > 3*getLatency) {
      if(avgLockAcqFailure > 50) return 0;
      if(avgLockAcqLatency >= 0.8*putLatency) return 0; else return 2;
    }
 }
}
```

Fig. 6: Example of a distributed cache class which relies on ElasticRMI's explicit elasticity support. This example uses fine-grained application-specific metrics to make scaling decisions

Figure 6 illustrates the use of `changePoolSize` to make scaling decisions. The `CacheExplicit2` class is implemented to use metrics specific to distributed object caches, e.g., `avgLockAcqFailure` (which measures the failure rate of acquiring write locks to ensure consistency during a `put` operation on the cache and `avgLockAcqLatency` (which measures the average latency to acquire write locks) to make decisions about changing the size of the elastic object pool. In Figure 6, the `CacheExplicit2` class does not add new objects to the pool when there is a lot of contention; contention is detected by using the conditions shown in `changePoolSize`. When the failure rate for acquiring write locks (`avgLockAcqFailure` is greater than 50%) or when the predominant component of `putLatency` is `avgLockAcqLatency`, no additional objects are added to the pool because there is already high contention among objects serving client requests to acquire write locks. When these conditions are `false`, the size of the pool is increased by two – controlling the number of objects added is another feature of `changePoolSize`.

Making Application-Level Scaling Decisions. While being more flexible than using coarse-grained metrics like CPU utilization, overriding `changePoolSize` may not be optimal for applications using multiple elastic classes (where the application contains *tiers* of elastic pools). ElasticRMI also supports decision making at the level of the application using the `Decider` class. It is the developer's responsibility to ensure that elastic objects being monitored communicate with the monitoring components, either by using remote method invocations or through message passing. The ElasticRMI runtime assumes responsibility for calling `getDesiredPoolSize` method of the monitoring class to get the desired size of each elastic object pool, and determines whether objects have to be added or removed. More details are available in Appendix C.

4 The ElasticRMI Runtime

The runtime (1) handles shared state among the objects in an elastic object pool, (2) instantiates each object of a pool on a separate JVM with each JVM executing on a separate virtual compute node (Mesos slice in our implementation), (3) performs load balancing, and (4) is responsible for fault-tolerance.

4.1 Shared State and Consistency

As described in Section 2.1, the need for an elastic object pool to appear to clients as a "single object" necessitates that the objects in the pool coordinate to update the state of its instance fields (`public`, `private`, `protected`) and `static` fields.

For consistency, we use HyperDex [28], a distributed in-memory key-value store (with strong consistency). Using an in-memory store provides the same data durability guarantees as Java RMI which stores the state of instance fields in RAM in a single Java virtual machine heap. HyperDex is different from the distributed cache in the examples of Section 3, which is used for illustration purposes only. The ElasticRMI preprocessor translates reads and writes of instance and static fields into `get(...)` and `put(...)` method calls of HyperDex. Figure 7 demonstrates how the preprocessor translates accesses to shared state, through a simple elastic class `C1`.

```
class C1                          |class C1Processed {
 extends ElasticObject {          | void foo() {
  int x, z;                       |  int x=Store.get("C1$x");
                                  |  if(x==5)
  void foo() {                    |   Store.put("C1$z", 10);
    if(x = 5) z = 10;             | }
  }                               | void bar() {
  synchronized                    |  while(!ERMI.lock("C1")) ;
    void bar() {                  |  ...
    ...                           |  ERMI.unlock("C1");
  }                               | }
}                                 |}
```

Fig. 7: Handling shared state through an in-memory store (HyperDex [28] here).

Figure 7 shows `C1` and how it is transformed by the ElasticRMI preprocessor to insert calls to HyperDex (abstracted by `Store`) and ElasticRMI's runtime (abstracted by `ERMI`). For variable `x`, `C1$x` is used as the key in `Store`. For synchronized methods, ElasticRMI uses a lock per class named after the class – the example in Figure 7 uses a lock called `"C1"`.

If an elastic class has an instance or static field f, a method call on f is handled as follows:

– If f is a remote or elastic object, the method invocation is simply serialized and dispatched to the remote object or pool as the case may be.
– *Else*, if m is `synchronized`. the method call is handled as in Figure 7, but the runtime acquires a lock on f through HyperDex. In this case, ElasticRMI guarantees mutual exclusion for the execution of $f.m(...)$ with respect to other methods of f.
– *Else*, (i.e., m is neither remote, elastic nor synchronized), $f.m(...)$ involves retrieving f from HyperDex and executing $m(...)$ locally on an object in the elastic object pool, and storing f back into HyperDex after $m(...)$ has completed executing.

The key idea behind ElasticRMI is to increase parallelism and hence the number of remote method executions per second when there is limited or no shared state. Increasing shared state increases latency due to the network delays involved in accessing HyperDex. Having shared state and mutual exclusion through locks or synchronized methods further decreases parallelism. However, we note that this is not a consequence of ElasticRMI, but rather dependent on the needs of the distributed application. If the developer manually implements all aspects of elasticity by using plan Java RMI and an existing tool like Amazon CloudWatch+Autoscaling, he still has to use something like a key-value store to handle shared state. ElasticRMI cannot and does not attempt to eliminate this problem – it is up to the programmer to reduce shared state.

Note that ElasticRMI does not guarantee a transactional (ACID) execution of $m(...)$ with respect to other objects in the pool, and using `synchronized` does not provide ACID guarantees *either in RMI or in* ElasticRMI.

4.2 Instantiation of Elastic Objects

An elastic class can only be instantiated by providing a minimum (≥ 2) and maximum number of objects that constitute its elastic object pool (see Figure 3). During instantiation, if the minimum number of objects is k, ElasticRMI's runtime creates k objects on k new JVM instances on k virtual nodes (Mesos slices), if k virtual nodes are available from Mesos [5]. If only $l < k$ are available, then only l objects are created. Under no circumstance does ElasticRMI create two or more JVM instances on the same slice obtained from Mesos. Then, ElasticRMI instantiates the HyperDex on one additional Mesos slice, and continues to monitor the performance of the HyperDex over the lifetime of the elastic object. ElasticRMI may add additional nodes to HyperDex as necessary. ElasticRMI also provides the flexibility to the application developer to notify system (cluster) administrators when there is no availability of "slices" or "resources" in Mesos, so that additional computing resources can be added dynamically to it. ElasticRMI also enables administrators to be notified if the utilization of the Mesos cluster exceeds or falls below (configurable) thresholds, so that additional computing resources can be added to Mesos pro-actively before the cluster runs out of slices.

4.3 Load Balancing

Unlike websites or web services, where load balancing *has to be performed* on the server-side, ElasticRMI has the advantage that both client and server programs are pre-processed to generate stubs and skeletons respectively. Hence, we employ a *hybrid* load balancing model involving both stubs and skeletons – note that *all* load balancing code is generated by the pre-processor, and the programmer *does not have to handle any aspect of it explicitly.* Please also note that this section describes the simple load balancing techniques used in ElasticRMI, but we do not claim to have invented a new load balancing algorithm.

On the server side, the runtime, while instantiating skeletons in an elastic object pool, assigns monotonically increasing unique identifiers (*uid*) to each skeleton, and stores this information in HyperDex. The skeleton with the lowest *uid* is chosen by the runtime to be the leader of the elastic object pool, called the *sentinel*. This is similar to leader election algorithms that use a so-called "royal hierarchy" among processes in a distributed system. The sentinel, in addition to performing all the regular functions (forwarding remote method invocations) to *its* object in the pool, also helps in load balancing. The client stub created by the ElasticRMI preprocessor (see Section 2.4) has the ability to communicate with the sentinel to invoke remote methods. While contacting the sentinel for the first time, the stub on the client JVM requests the identities (IP address and port number) of the other skeletons in the pool from the sentinel.

For load-balancing on the client-side, the stub then re-directs subsequent method invocations to other objects in the object pool either randomly or in a round-robin fashion. If an object has been removed from the pool after its identity is sent to a stub, i.e., if the sending itself fails, the remote method invocation throws an exception which is intercepted by the client stub. The stub then retries the invocation on other objects including the sentinel. If all attempts to communicate with the elastic object pool fail, the exception is propagated to the client application.

For load-balancing on the server side, the sentinel is also responsible for collecting and periodically broadcasting the state of the pool – number of objects, their identities and the number of pending invocations – to the skeletons of all its members. We use the JGroups group communication system for broadcasts. If the sentinel notices that any skeleton is overloaded with respect to others, it instructs the skeleton to redirect a portion of invocations to a set of other skeletons. To decide the number of invocations that have to be redirected from each overloaded skeleton, our implementation of the sentinel uses the first-fit greedy bin-packing approximation algorithm (See http://en.wikipedia.org/wiki/Bin_packing_problem). As mentioned in the previous paragraph, client-side load balancing occurs at the stub while server-side load balancing involves skeletons and the sentinel which monitor the state of the JVM and that of the elastic object pool to redirect incoming method invocations.

4.4 Fault Tolerance

ElasticRMI has also mostly preserved the fault- and fault tolerance model of Java RMI, where objects typically reside in main memory, and can crash in the middle of a remote method invocation. Existing RMI applications implement fault tolerance protocols on top of this model, and we want to preserve it to make adoption of ElasticRMI easier. In short, ElasticRMI *does not hide/attempt to recover* from failures of client objects, key-value store (HyperDex) or the server-side runtime processes and propagates corresponding exceptions to the application. However, ElasticRMI *attempts to recover* from failures of the sentinel and from Mesos-related failures. In short, the failure of the sentinel triggers the leader election algorithm described in 4.3 to elect a new sentinel, and mesos-related failures affect the addition/removal of new objects until Mesos recovers. A detailed account of fault tolerance is presented in Appendix B

5 Evaluation

In this section, we evaluate the performance of ElasticRMI, using metrics relevant to elasticity.

5.1 Elasticity Metrics

Measuring elasticity is different from measuring scalability. (Recall that) Scalability is the ability of a distributed application to increase its "performance" proportionally (ideally linearly) with respect to the number of available resources, while elasticity is the ability of the application to adapt to increasing or decreasing workload; adding or removing resources to *maintain* a specific level of "performance" or "quality of service (QoS)"[29]. Performance/QoS is specific to the application – typically a combination of throughput and latency. A highly elastic system can scale to include newer compute nodes, as well as quickly provision those nodes. There are no standard *benchmarks* for elasticity, but the

Standard Performance Evaluation Corporation (SPEC) has recommended elasticity *metrics* for IaaS and PaaS clouds [29].

Agility. This metric characterizes the ability of a system provisioned to be as close to the needs of the workload as possible [29]. Assuming a time interval $[t, t']$, which is divided into N sub-intervals, Agility maintained over $[t, t']$ can be defined as:

$$\frac{1}{N}(\sum_{i=0}^{N} Excess(i) + \sum_{i=0}^{N} Shortage(i))$$

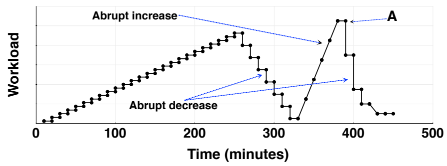
where (1)$Excess(i)$ is the excess capacity for interval i as determined by $Cap\_prov(i) - Req\_min(i)$, when $Cap\_prov(i) > Req\_min(i)$ and zero otherwise. (2) $Shortage(i)$ is the shortage capacity for interval i as determined by $Req\_min(i) - Cap\_prov(i)$, when $Cap\_prov(i) < Req\_min(i)$ and zero otherwise. (3) $Req\_min(i)$ is the minimum capacity needed to meet an application's quality of service (QoS) at a given workload level for an interval i. (4) $Cap\_prov(i)$ is the recorded capacity provisioned for interval i, and (5) N is the total number of data samples collected over a measurement period $[t, t']$, i.e., one sample of both $Excess(i)$ and $Shortage(i)$ is collected per sub-interval of $[t, t']$.

Meaning of Agility. Elasticity measures the shortage *and* excess of computing resources over a time period. For example, a value of elasticity of 2 over $[t, t']$ when there is no excess means that there is a mean shortage of 2 "compute nodes" over $[t, t']$. For an ideal system, agility should be as close to zero as possible – meaning that there is neither a shortage nor excess. Agility is a measurement of the ability to scale up and down while maintaining a specified QoS. The above definition of agility will not be valid in a context where the QoS is not met. Also, a controversial aspect of the above definition is that it gives equal weightage to both shortage and excess while averaging. *Shortage* typically results in lost business opportunities for a distributed application (e.g., an ad server, an e-commerce website, etc.), but *Excess* results in additional cost for the additional resources provisioned. Since virtual compute nodes are cheap (about $0.50 per hour even for an extra-large instance), there is ongoing debate over whether shortage should be weighted more than excess, because compute nodes (e.g. EC2 nodes) are cheap, but according to [29], there is disagreement on what the weights should be (70%/30% or 80%/20% for example). So we retain the existing definition of Agility because our purpose is to only evaluate the performance of these applications for research purposes.

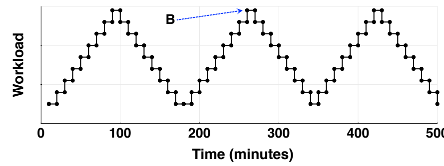
Provisioning Interval. Provisioning Interval is defined as the time needed to bring up or drop a resource. This is the time between initiating the request to bring up a new resource, and when the resource serves the first request.

5.2 ElasticRMI Applications for Evaluation and Workloads

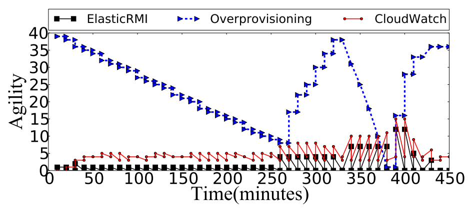
We have re-implemented four *existing* applications using ElasticRMI, to gauge its efficacy – (1) Marketcetera, a financial market order routing system [13], (2) Hedwig, a durable and reliable topic-based publish/subscribe sys-



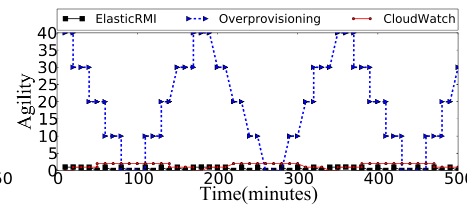
(a) Pattern for abruptly changing workload for all four systems. The workload pattern remains the same, but the meaning and magnitude are different for the four systems. For Marketcetera, the workload is orders/s but for DSC it is the number of updates/s. Point A is 20,000 orders/s for Marketcetera and 13,500 updates/s for DCS.



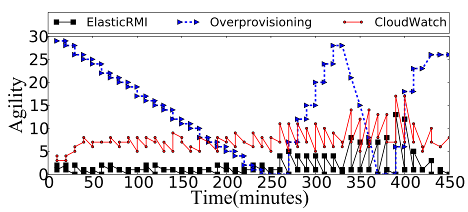
(b) Cyclical workload example for all four systems. As in Figure 8a, the pattern remains the same for all four systems but the meaning and magnitude are different. Point B is 18,000 orders/s for Marketcetera and 9,000 updates/s for DCS.



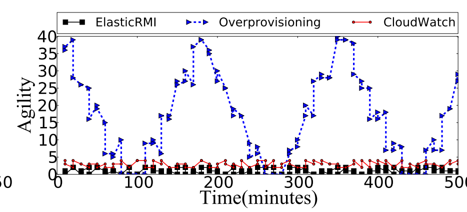
(c) Marketcetera – abrupt workload.



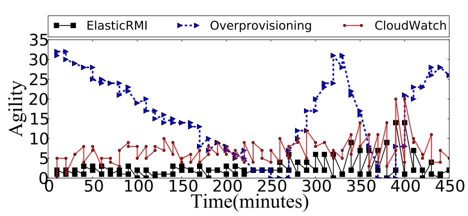
(d) Marketcetera – cyclical workload.



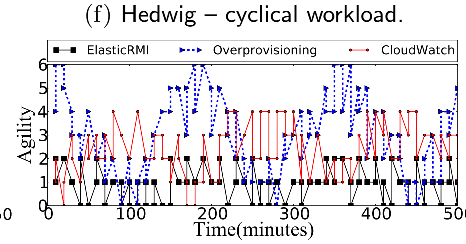
(e) Hedwig – abrupt workload.



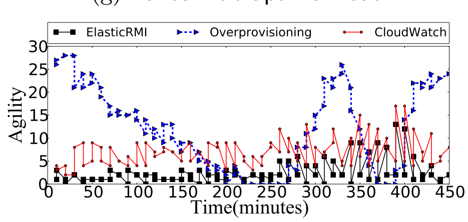
(f) Hedwig – cyclical workload.



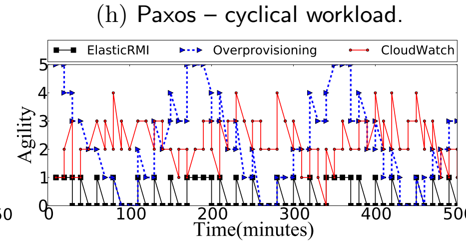
(g) Paxos – abrupt workload.



(h) Paxos – cyclical workload.



(i) DCS – abrupt workload.



(j) DCS – cyclical workload.

Fig. 8: Elasticity Benchmarking of Marketcetera order routing, Hedwig, Paxos and DCS. We compare the ElasticRMI implementation of these applications with two other systems described in Section 5.4.

tem [16], (3) Paxos, a consensus algorithm [18], and (4) DCS, a datacenter coordination service.

Marketcetera [13] Order Routing. Marketcetera is an NYSE-recommended [27] algorithmic trading platform. The order routing system is the component that accepts orders from traders/automated strategy engines and routes them to various markets (stock/commodity), brokers and other financial intermediaries using the Financial Information Exchange (FIX) protocol – the QuickFiX/J implementation. For fault-tolerance, the order is persisted (stored) on two nodes. The workload for this system is a set of trading orders generated by the simulator included in the community edition of Marketcetera [13].

Apache Hedwig [16]. Hedwig is a topic-based publish-subscribe system designed for reliable and guaranteed at-most once delivery of messages from publishers to subscribers. Clients are associated with (publish to and subscribe from) a Hedwig instance (also referred to as a region), which consists of a number of servers called hubs. The hubs partition the topic ownership among themselves, and all publishes and subscribes to a topic must be done to its owning hub. The workload for this system is a set of messages generated by the default Hedwig benchmark included in the implementation.

Paxos [18]. Paxos is a family of protocols for solving consensus in a distributed system of unreliable processes. Consensus protocols are the basis for the state machine approach to distributed computing, and for our experiments we implement Paxos using a widely-used specification by Kirsch and Amir [18]. The workload for this system is the default benchmark included in the `libPaxos` library [25].

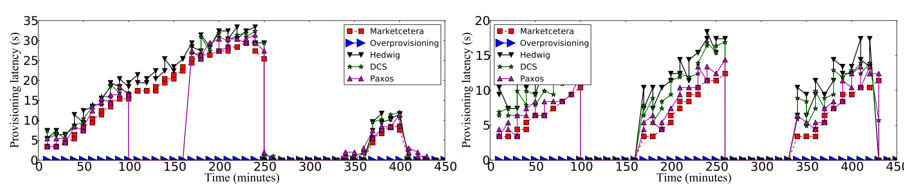
DCS. DCS is a distributed co-ordination service for datacenter applications, similar to Chubby [7] and Apache Zookeeper [31]. DCS has a hierarchical name space which can be used for distributed configuration and synchronization. Updates are totally ordered. The workload for this system is the default benchmark included in Apache Zookeeper [31].

5.3 Workload Pattern

To measure how well the system adapts to the changing workload, we use two patterns shown in Figures 8a and 8b. These two patterns capture all common scenarios in elastic scaling which we have observed by analyzing real world applications deployed in the HP Cloud using HP CloudSystem Matrix [17]. The abrupt pattern shown in Figure 8a has all possible scenarios regarding abrupt changes in workload – gradual non-cyclic increase, gradual decrease, rapid increases and rapid decrease in workload. A cyclic change in workload is shown by the second pattern in Figure 8b. So, together the patterns in Figures 8a and 8b exhaustively cover all elastic scaling scenarios we observed. Note however, that although the pattern remains the same for varying the workload while evaluating all the four systems, the magnitude differs depending on the benchmark used, i.e., the values of points A and B in Figures 8a and 8b are different for the four systems depending on the benchmark. Point A, for example, is 50,000 orders/s for Marketcetera, 75,000 updates/s for DCS, 24,000 consensus rounds/s for Paxos and 30,000 messages/s for Hedwig. We set Point B at 20% above Point A – note that the specific values of Points A and B are immaterial because we are only measuring adaptability and not peak performance.

5.4 Overprovisioning and CloudWatch

We compare the ElasticRMI implementation of the applications in Section 5.2 with the *existing* implementations of the same applications in two deployment scenarios – (1) Overprovisioning and (2) Amazon AutoScaling + CloudWatch [4][2]. The overprovisioning deployment scenario is similar to an "oracle" – the *peak* workload arrival rate i.e., point A for the abruptly changing workload and point B for the cyclic workload are known a priori to the oracle; and the number of nodes required to meet a desired QoS (throughput, latency) at A and B respectively is determined by the oracle through experimental evaluation. The oracle then provisions the application on a fixed set of nodes – the size of which is enough to maintain the desired QoS even at the peak workload arrival rate (A and B respectively). In a nutshell, the over provisioning scenario can be described as "knowing future workload patterns and provisioning enough resources to meet its demands". Overprovisioning is the alternative to elastic scaling – there are going to be excess provisioned resources when the workload is below the peak (A and B), but provisioning latency is zero because all necessary resources are always provisioned. In the CloudWatch scenario, we use a monitoring service – Amazon CloudWatch to collect utilization metrics (CPU/Memory) from the nodes in the cluster and use conditions on these metrics to decide whether to increase or decrease the number of nodes. The ElasticRMI implementation of the above applications, however, uses a combination of resource utilization and application-level properties specific to Marketcetera, DCS, Paxos and Hedwig respectively to decide on elastic scaling.



(a) Provisioning latency – Abrupt Workload (b) Provisioning latency – Cyclic Workload

Fig. 9: Provisioning latency in seconds for ElasticRMI and Overprovisioning (which is always 0). Provisioning latency for Amazon CloudWatch is not plotted because it is in several minutes and hence well above that of both ElasticRMI and Overprovisioning. You can see repeating patterns corresponding to the cyclic workload.

5.5 Agility Results

In this section, we compare the Agility of the ElasticRMI implementation of all four systems against Overprovisioning and CloudWatch.

Marketcetera Order Routing. The relevant QoS metrics for the order routing subsystem are order routing throughput, which is the number of orders routed from traders to brokers/exchanges per second and order propagation latency, which is the time taken for an order to propagate from the sender to the receiver. We compare the elasticity of the three deployments of the order routing system described in Section 5.4. The results are illustrated in Figure 8. Figures 8c and 8d plot the agility over the same time period as in Figures 8a and 8b for all the three deployments. From Figures 8c and 8d, we observe that the agility of ElasticRMI is better than CloudWatch and overprovisioning. Ideally, agility must be zero, because agility is essentially a combination of resource wastage or resource under-provisioning. We observe that for abruptly changing workloads, agility of ElasticRMI is close to 1 most of the time, and increases to 5 during abrupt changes in workload. We also observe that the the agility of ElasticRMI oscillates between 0 and a positive value frequently. This proves that the elastic scaling mechanisms of ElasticRMI perform well in trying to achieve optimal resource utilization, i.e., react aggressively by trying to push agility to zero. In summary, the average agility of ElasticRMI is 2.26. As expected, the agility of overprovisioning is the worst, up to $40\times$ that of ElasticRMI. This is not surprising, because its agility does reach zero at peak workload, when the agility of ElasticRMI is 5, thus illustrating that overprovisioning optimizes for peak workloads. The average agility of overprovisioning is 17.2. CloudWatch performs much better than overprovisioning, but it is less agile than ElasticRMI. Its agility is approximately $2\times$ that of ElasticRMI on average, and it does not oscillate to zero frequently like ElasticRMI. Figure 8d shows that the agility of ElasticRMI is again better than that of CloudWatch and overprovisioning. We also observe that as in the case of abrupt workloads, the agility of ElasticRMI tends to decrease to zero more frequently than the other two deployments. Figure 8d also demonstrates the oscillating pattern in the agility of overprovisioning – the initial agility is high (and comes from *Excess*), and as the workload increases, *Excess* decreases, thereby decreasing Agility and bringing it to zero corresponding to Point B. Then *Excess* increases again as the workload decreases thereby increasing Agility. This repeats three times.

Hedwig. The relevant QoS metrics for Hedwig are also throughput and latency – the number of messages published per second and time taken for the message to propagate from the publisher to the subscriber. Figures 8e and 8f illustrate the agility corresponding to our experiments with Hedwig. From Figures 8e and 8f, we observe similar trends as in the case of Marketcetera order processing. ElasticRMI has lower agility values than the other two deployments, and the agility of ElasticRMI tends to oscillate between zero and a positive value. The agility values of CloudWatch are more than $7\times$ that of ElasticRMI, on average for abrupt workloads and $3\times$ that of ElasticRMI for cyclic workloads. As expected, the agility of over provisioning is the highest, and is worse than the values observed for Marketcetera in the case of cyclic workloads. We also observe a similar oscillating trend in the agility values of the overprovisioning deployment as in Marketcetera, but the agility values oscillate more frequently because $Req\_min(i)$ – the minimum capacity needed to maintain QoS under a certain workload changes more erratically than Marketcetera due to the replication and at-most once guarantees provided by Hedwig for delivered messages.

Paxos The relevant QoS metrics for Paxos are the number of consensus rounds executed successfully per second, and the time taken to execute a consensus round. Figures 8g and 8h illustrate the agility corresponding to our experiments with Hedwig. From Figures 8g and 8h, we observe similar trends to Hedwig and Marketcetera. The agility of CloudWatch in this case is 6.6× than of ElasticRMI, on average for abrupt workloads and 2.2× that of ElasticRMI for cyclic workloads. We also observe that the agility of ElasticRMI returns to zero (the ideal agility) most frequently among the three deployments.

DCS The relevant QoS metrics for DCS are the number of updates to the hierarchical name-space per second and the end-to-end latency to perform an update as measured from the client. Figures 8i and 8j illustrate the agility corresponding to our experiments with Hedwig. From Figures 8i and 8j, we observe that the agility of CloudWatch in this case is 7.2× than of ElasticRMI, on average for abrupt workloads and 3.2× that of ElasticRMI for cyclic workloads. As in the systems discussed above, we observe that the agility of ElasticRMI returns to zero (the ideal agility) most frequently among the three deployments.

5.6 Provisioning Latency

Figures 9a and 9b plot the provisioning latency of ElasticRMIfor both abrupt and cyclic workloads. We observe that the provisioning latency of ElasticRMI is less than 30 seconds in all cases, which compares very favorably to the time needed to provision new VM instances in Amazon CloudWatch (which is in the order of several minutes, and hence omitted from Figure 9). Provisioning latency is zero for the overprovisioning scenario, and that is the main purpose of overprovisioning – to have resources always ready and available. Also, we observe that as the workload increases, provisioning interval also increases, due to the overhead in determining the remote method calls that have to be redirected and also due to increasing demands on the resources of the sentinel object in ElasticRMI's object pools.

5.7 Overhead (See Appendix A)

6 Related Work

J-Orchestra [9,10] automatically partitions Java applications and makes them into distributed applications running on distinct JVMs, by using byte code transformations to change local method calls into distributed method calls as necessary. J-Orchestra's implementation seems to make it usable in the cloud, because it only seems to need the identities (IP addresses, port numbers) of compute nodes onto which the application has to be partitioned. The key distinctions between J-Orchestra and ElasticRMI are that (1) they attempt to solve slightly different problems – J-Orchestra tackles the complex problem of automatic distribution of Java programs while ElasticRMI aims to add elasticity to already distributed programs and (2) ElasticRMI partitions different invocations of a single remote method.

Another closely related work is Self-Replicating Objects (SROs) [22], a new elastic concurrent programming abstraction. An SRO is similar to an ordinary .NET object exposing an arbitrary API but exploits multicore CPUs by automatically partitioning its state into a set of replicas that can handle method calls (local and remote) in parallel, and merging replicas before processing calls that cannot execute in replicated state. SRO also does not require developers to explicitly protect access to shared data; the runtime makes all the decisions on synchronization, scheduling and splitting/merging state. Live Distributed Objects (LDO) [24] is a new programming paradigm and a platform, in which instances of distributed protocols are modeled as live distributed objects. Live objects can be used to represent distributed multi-party protocols and application components. Shared-state and synchronization between the objects is maintained using Quicksilver [23], a group communication system. Automatic scaling is not supported and must be explicitly implemented by the programmer using the abstractions provided by LDO.

RMI extensions with support for *multicast* interaction between a client and a group of *replicated* server objects for fault tolerance include multi-RMI [30], mRMI [20] and the Manta high performance Java system [19]. Here, each remote method invocation is multicast to a *replicated* set of server objects and the method returns multiple times giving the client multiple return values. The systems discussed above are clearly different from ElasticRMI where a remote method invocation is directed to a single object.

Quality Objects (QuO) [6] is a seminal framework for providing quality of service (QoS) in network-centric distributed applications. When the requirements are not being met, QuO provides the ability to adapt at many levels in the system, including the middleware layer responsible for message transmission. In contrast to QuO, ElasticRMI attempts to increase quality of service by changing the size of the remote object pool, and does not change the protocols used to transmit remote method invocations.

7 Conclusions

We have described the design and implementation of ElasticRMI and have demonstrated through empirical evaluation using real-world applications that it is effective in engineering elastic distributed applications. Our empirical evaluation also demonstrates that relying solely on externally observable metrics like CPU/RAM/network utilization decreases elasticity, as demonstrated by the high agility values of CloudWatch. We have shown that our implementation of ElasticRMI reduces resource wastage, and is sufficiently agile to meet the demands of applications with dynamically varying workloads. Through an implementation using Apache Mesos, we ensure portability of ElasticRMI applications across Mesos installations, whether it is a private datacenter or a public cloud or a hybrid deployment between private data centers and public clouds. We have demonstrated that ElasticRMI applications can use fine-grained application specific metrics without revealing those metrics to the cloud infrastructure provider, unlike CloudWatch.

References

1. A. Birrell and B. Nelson. Implementing Remote Procedure Calls. *ACM TOCS*, Feb 1984.
2. Amazon Web Services (AWS) Inc. Amazon CloudWatch. http://aws.amazon.com/cloudwatch/, 2012.
3. Apache Pig. http://pig.apache.org.

4. AWS Inc. Amazon Auto Scaling. http://aws.amazon.com/autoscaling/, 2012.
5. B. Hindman and A. Konwinski and M. Zaharia and A. Ghodsi and A. Joseph and R. Katz and S. Shenker and I. Stoica. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *NSDI'11*. http://incubator.apache.org/mesos/.
6. BBN Technologies. Quality Objects (QuO). http://quo.bbn.com/, 2006.
7. M. Burrows. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *OSDI '06*.
8. C. Cachin and R. Guerraoui and L. Rodrigues. *Reliable and Secure Distributed Programming*. Springer-Verlag, 2011.
9. E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic Java Application Partitioning. In *ECOOP '02*.
10. E. Tilevich and Y. Smaragdakis. Portable and Efcient Distributed Threads for Java. In *MIDDLEWARE '04*.
11. Elastic Map Reduce. http://aws.amazon.com/elasticmapreduce/.
12. Facebook Inc. Apache Thrift. http://thrift.apache.org/, 2012.
13. G. Miller and T. Kuznets and R. Agostino. Marketcetera Automated Trading Platform. http://www.marketcetera.com/site/, 2012.
14. Giraph. http://incubator.apache.org/giraph/.
15. Hadoop. http://hadoop.apache.org.
16. Hedwig. https://cwiki.apache.org/ZOOKEEPER/hedwig.html.
17. HP Inc. CloudSystem Matrix – Complete, Integrated, Open Solution for Building and Managing IaaS. http://h18004.www1.hp.com/products/blades/components/matrix/index.html, 2012.
18. J. Kirsch and Y. Amir. Paxos for Systems Builders. http://www.cnds.jhu.edu/pub/papers/cnds-2008-2.pdf.
19. J. Maassen and T. Kielmann and H. Bal. Generalizing Java RMI to Support Efficient Group Communication. In *In ACM Java Grande Conference 2000*.
20. J. Rajajczak. MRMI – Multicast Remote Objects. http://staff.ii.pw.edu.pl/~ratajcza/mrmi/mrmi.doc/jr.mrmi.server.MulticastRemoteObject.html, 2003.
21. J.Ousterhout et. al. The Case for RAMCloud. *CACM*, 2011.
22. K. Ostrowski and C. Sakoda and K. Birman. Self-replicating Objects for Multicore Platforms. In *ECOOP'10*.
23. K. Ostrowski and K. Birman and D. Dolev. Quicksilver Scalable Multicast (QSM). In *NCA '08*.
24. K. Ostrowski and K. Birman and D. Dolev and J. Ahnn. Programming with Live Distributed Objects. In *ECOOP '08*.
25. LibPaxos. http://libpaxos.sourceforge.net/.
26. Memcached. http://www.memcached.org.
27. NYSE/Marketcetera Press Release. NYSE Technologies and Marketcetera Launch New Era Software-as-a-Service Trading Platform on SFTI Network. *http://www.marketcetera.com/site/node/153*, 2009.
28. R. Escriva and B. Wong and E. Sirer. HyperDex: a Distributed, Searchable Key-Value Store. In *SIGCOMM'12*.
29. SPEC Open Systems Group (OSG). Report on Cloud Computing to the OSG Steering Committee. http://www.spec.org/osgcloud/docs/osgcloudwgreport20120410.pdf, 2012.
30. W. Cazzola and M. Ancona and F. Canepa and M. Mancini and V. Siccardi. Enhancing Java to Support Object Groups. In *ROOTS '02*.
31. Zookeeper. http://zookeeper.apache.org/.

A Appendix – Overhead

In this section, we examine whether using ElasticRMI creates any overhead in the absolute performance of distributed applications. Through extensive experiments, we have observed that overheads are negligible (typically between 2% and 3%). To determine the overhead, we compare the elastic scalability of an ElasticRMI implementation, where utilizing the nodes available is the responsibility of ElasticRMI, to a manual provisioning scenario where the order routing system is provisioned on all the nodes available, irrespective of the frequency of incoming orders. We keep the number of senders (traders/automated engines) and receivers (brokers/exchanges) constant and increase the number of nodes available to the routing system. The results are shown in Figure 10.

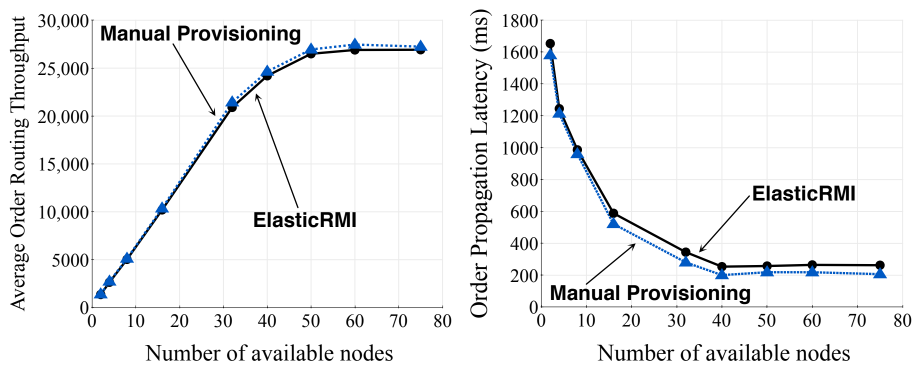


Fig. 10: Marketcetera order routing throughput and latency

Figure 10 demonstrates the elastic speedup [29] of the ElasticRMI-based implementation. The default implementation of the order routing subsystem is elastic, and the ElasticRMI-based implementation has a similar elastic speedup. Using ElasticRMI has a constant overhead when compared to the default implementation, which uses message passing and ActiveMQ, and this is due to the constant overheads imposed by RMI marshalling/unmarshalling as well as the time taken by the ElasticRMI runtime to ramp up to handle the workload. The throughput and latency of of the ElasticRMI-based implementation have a negligible difference when compared to manual provisioning – the difference is statistically insignificant as shown in Figure 10. We observe similar behavior for the other four systems evaluated previously, namely Paxos, DCS and Hedwig.

B Appendix – Fault Tolerance

ElasticRMI has also mostly preserved the fault- and fault tolerance model of Java RMI, where objects typically reside in main memory, and can crash in the middle of a remote method invocation. Thus, "failure" is defined as the *crash* of a process executing on a JVM either due to a hardware fault or a programmer error. We assume the *crash-stop* fault-model as Java RMI, where crashed processes are assumed not to recover – a recovering process is simply treated as a new process. The ElasticRMI runtime is a non-trivial distributed system and is subject to many failure-scenarios:

Failure of client objects: Since the client object and its stub reside on the same JVM, they do not fail independently. If a client fails during a remote method execution, the invocation may complete but the return values may not be delivered to the client. The failure of a client is not notified to a server object just like plain Java RMI.

Failure of sentinel: If the leader of the skeletons in a pool, i.e., the sentinel fails, the other skeletons in the pool elect a new sentinel using leader election algorithms [8]. The sentinel and its object fail together because they reside on the same JVM. Method executions pending at the sentinel also fail (due to the failure of the associated object), and a corresponding exception is returned by the client-side stub to the client application (which times out waiting for a return value from the sentinel). Even if a method invocation has not started executing at the sentinel, the sentinel's failure causes the invocation to fail, because the client stub has no way of knowing whether execution started when the sentinel fails.

Failure of an object in the elastic object pool: The failure of an object (and its associated skeleton on the same JVM) is handled in a similar way to the failure of the sentinel, except that leader election isn't necessary. The client-stub times out waiting for the return value from the client and throws a corresponding exception to the client application. Even if a method invocation has not started executing at the object, its failure causes the invocation to fail, because the client stub has no way of knowing whether execution started when the object fails.

Failure of the Key-Value Store: The failure of the key-value store (HyperDex) can be detected by all objects in the elastic object pool, because `get(...)` and `put(...)` method calls between any object and the key-value throw exceptions. The object then terminates the execution of the method and sends a corresponding message to the client-stub, which throws a corresponding exception to the client application.

Mesos-related failures: Mesos-related failures are detected by the ElasticRMI runtime, when Mesos throws exceptions indicating either that additional slices are unavailable or that a Mesos process has crashed. If slices are unavailable, the addition of objects to the elastic object pool is suspended until they are available. The runtime continues to periodically call the object's `changePoolSize` method to examine whether the addition or removal of objects is necessary; but actual addition is suspended until slices are available. If Mesos (i.e. its master node) crashes, the ElasticRMI runtime suspends all further additions, removals and polling until Mesos recovers. But existing objects in the elastic object pool continue to execute, and handle remote method calls.

Failure of ElasticRMI *server-side runtime processes:* If any runtime daemon/process of ElasticRMI fails, all objects shutdown after sending a SHUTDOWN message to the client-stubs that have pending method invocations.

```java
//Custom decider class to make application-level scaling decisions
class MyDecider extends Decider {
    int poolSizeKVS, poolSizeMBS, poolSizeDMQ;
    public MyKVStore kvs; //This decider monitors a KV Store
    public MyBackupStore mbs; // and a backup store
    public DistributedMQ dmq;
    int getDesiredPoolSize(ElasticObject o) { //Called by ElasticRMI runtime
        String cName = Decider.getClassName(o);
        //Make a decision on the size of the dmq pool
        if(cName.equals("DistributedMQ") {
           //Latency for inserting into queue is more than 100ms
           //Increase the size of all 3 elastic object pools in concert
           if(dmq.getAvgLatency() > 100) {
               poolSizeKVS++;
               poolSizeMBS++;
               poolSizeDMQ++;
               return poolSizeDMQ;
           }
        }
        ...
    }
    ...
}
class MyKVStore extends ElasticObject {
    ...
    MyKVStore(Decider d) {
        ...
        super(d); //associate d with this object pool
    }
}

class MyBackupStore extends ElasticObject {
    ...
    MyBackupStore(Decider d) {
        ...
        super(d); //associate d with this object pool
    }
}

class DistributedMQ extends ElasticObject {
    MyKVStore kvs;
    MyBackupStore mbs;
    DistributedMQ() {
        MyDecider d = new MyDecider();
        kvs = new MyKVStore(d);
        mbs = new BackupStore(d);
        d.kvs = kvs; //associate kvs with d
        d.mbs = mbs; //associate mbs with d
        d.dmq = this; //associate dmq with d
        super(d);//associate d with this object pool
    }
    long getAvgLatency() {...}
    ...
}
```

Fig. 11: Making application-level scaling decisions

C Appendix – Making Application-level Decisions

ElasticRMI also supports decision making at the level of the application using the `Decider` class. Applications can define performance monitoring components through (elastic) classes extending `Decider`. Such components (i.e. elastic objects that monitor other elastic objects) are also instantiated by ElasticRMI, on separate virtual nodes, and can be associated with any of the application's elastic classes by using the appropriate constructor (see Figure 3). It is the developer's responsibility to ensure that elastic objects being monitored communicate with the monitoring components, either by using remote method invocations or through message passing. The ElasticRMI runtime assumes responsibility for calling `getDesiredPoolSize` method of the monitoring class to get the desired size of each elastic object pool, and determines whether objects have to be added or removed.

Figure 11 illustrates the use of the `Decider` class. In this case, a distributed message queue application (`DistributedMQ`) has two internal components, namely a key-value store (`MyKVStore`) (to store messages) and a backing store (to store logs, history, etc.). Note that this is only an illustration – our goal in this section is not to design a message queue. The constructor for `DistributedMQ` instantiates the decider class (`MyDecider`) and associates the decider with the `DistributedMQ` elastic object, `MyKVStore` elastic object and `MyBackupStore` elastic object. An application-level scaling decision here is to scale all tiers together so that an increased load at the message queue due to the addition of objects to its object pool does not overwhelm either the `MyKVStore` object or the `MyBackingStore` object. Hence `MyDecider` adds increases the pool size of all these pools together. The ElasticRMI runtime is responsible for calling `getDesiredPoolSize` to get the desired pool size of each of these elastic objects, because the programmer has notified the runtime to use a decider by calling the constructor of `ElasticObject` through `super(d)`.

D Appendix – Discussion

In this appendix, we answer some frequently asked questions that are not essential to understanding the core concepts behind ElasticRMI.

D.1 Cluster-level Elasticity Management

A relevant question to ask in the context of programmable elasticity is whether applications (even while using `Decider` class) should make scaling decisions. Should all elasticity decisions be left to a "cluster manager" which has a global view of the *entire datacenter*? Will that lead to an elasticity solution that is optimal for the whole datacenter? We disagree with this line of thought. To the best of our knowledge, there is no such cluster manager which can automatically handle elasticity requiring the programmer to specify the priority or deadline assigned to each application executing in the data center. There are cluster schedulers (e.g. Apache Mesos, Omega from Google), but they do require the application to specify the number of nodes required, or the programming framework (e.g. Hadoop) to infer the number of nodes from the program and communicate this number to the scheduler. But, even if one were to design such a cluster manager, it would need the application to send requests for additional nodes – application priority and deadline alone aren't sufficient – what if several applications have the same priority? Also, many datacenter applications like paxos, distributed hash tables, etc. have no deadlines. Hence, designing an cluster/datacenter-level elasticity manager that gets no input from the applications in the form of resource requests is extremely hard if not impossible. And ElasticRMI enables the application to *make such requests*. It is up to the cluster manager or the scheduler to grant them – in fact, in our current implementation, Mesos may not grant resource requests and the ElasticRMI runtime propagates this decision back to the decider class and sentinel.

D.2 RMI Registry and References

Applications can use various mechanisms to obtain reference stubs to elastic objects. For example, an application can register its elastic objects with RMI's simple naming facility – the RMI registry. The RMI registry can store references to elastic objects, because the ElasticRMI pre-processor ensures their type compatibility. All stubs contain information about the sentinel, which is guaranteed to be present in failure-free runs of the elastic object pool. The `ElasticObject` class provides methods that can be used by an application-defined class to create references to elastic objects, which can then be registered with the RMI registry. Alternatively, an application can pass and return elastic object references as part of other remote or elastic invocations. For simplicity of presentation, we omit these details from Figure 3.

D.3 Singleton Class

An elastic class is not a singleton class. One can create multiple elastic object pools from an elastic class even under the purview of the same ElasticRMI runtime system. Each object pool can have its own unique reference registered in the `rmiregistry` – of course, each pool should use a different identifier/service name.