Towards Explicitly Elastic Programming Frameworks

K. R. Jayaram
IBM Thomas J. Watson Research Center, NY, USA.

***Abstract*—It is a widely held view that software engineers should not be "burdened" with the responsibility of making their application components elastic; and that elasticity should be either be *implicit and automatic* in the programming framework; or that it is the responsibility of the cloud provider's operational staff (DevOps) to make distributed applications written for dedicated clusters elastic and execute them on cloud environments.**

In this paper, we argue the opposite – we present a case for explicit elasticity, where software engineers are given the *flexibility* to *explicitly* engineer elasticity into their distributed applications. We present several scenarios where elasticity retrofitted to applications by DevOps is ineffective, present preliminary empirical evidence that explicit elasticity improves efficiency, and argue for elastic programming languages and frameworks to reduce programmer effort in engineering elastic distributed applications. We also present a bird's eye view of ongoing work on two explicitly elastic programming frameworks – *ElasticThrift* (based on Apache Thrift [6]) and *ElasticJava*, an extension of Java with support for explicit elasticity.

I. Introduction

Elasticity is the ability of a distributed application to increase or decrease its use of computing resources to *maintain* a desired level of performance (sometimes referred to as *quality of service*) in response to changing workloads. Elasticity is a key aspect of scalability of distributed applications deployed in cloud computing environments, in both Infrastructure-as-a-Service (IaaS) and Platform-as-a-Service (PaaS) systems. Good *horizontal* scalability is a pre-requisite for elastic deployment of distributed applications – the application must increase its performance (e.g. higher throughput, lower latency) as it gets additional resources. During the software engineering lifecycle, once an application developer establishes horizontal scalability of his design/algorithm/software, the next key challenge is the design and implementation of effective elasticity management components, which (1) monitor the application's performance during execution, (2) determine when additional computing resources should be requested from the runtime/cloud management system, (3) provision relevant components on said resources and re-distribute the workload among available resources, and (4) continuously monitor itself at runtime and relinquish un-necessary resources when the workload decreases.

With the increasing availability, cost-/energy-effectiveness and usability of cloud/utility computing environments, elasticity has become an important aspect of modern software engineering. Designing and implementing robust elasticity management components is vital to *both* new distributed applications as well as *existing and legacy* distributed applications when they are deployed on/*migrated to* the cloud.

It is a widely held view that elasticity should be the responsibility of specialized programming frameworks and/or operational staff (DevOps); and that (original) application developers should not be burdened with designing elasticity management components. This paper presents a case for generic programming frameworks that support *explicit elasticity*, to offer software engineers the *flexibility* to engineer distributed applications that utilize resources effectively while maintaining their performance in response to varying user/customer workloads, i.e., developers explicitly specify when resources should be added or removed. This paper makes the following technical contributions:

- It outlines the current state-of-the-art in elasticity management, from domain specific frameworks for implicit elasticity to DevOps tools, detailing their benefits and shortcomings.
- It outlines the arguments against explicitly elastic programming languages and frameworks, while examining their various fallacies.
- It outlines the design of ElasticThrift (based on Apache Thrift [6]), a framework for polyglot elastic RPCs, aimed at meeting the needs of modern distributed applications whose components are programmed in different languages due to varying developer affinities and availability of robust libraries. ElasticThrift is applicable to many languages but specific to the RPC programming idiom.
- It describes, at a high level, the design of a new programming framework – *ElasticJava* (an extension to Java), for generic elastic distributed programming. ElasticJava targets a wide variety of application domains, and helps the developer focus on high-level elasticity decisions by masking low-level mechanics of performance monitoring, interaction with the cloud management system to provision virtual machines etc.
- It presents preliminary empirical evidence to support our hypothesis that explicit elasticity improves application performance and reduces resource wastage.

II. Elasticity Management Today

The two predominant practices for elasticity, namely DevOps tools like CloudWatch/AutoScaling and implicitly elastic frameworks like map-reduce [8], Spark [16], Dryad [9], etc. are popular for several reasons.

DevOps tools typically adopt a black-box approach to elasticity by using simple externally observable metrics, e.g.,

network traffic, CPU/memory usage, etc. to decide on elastic scaling at the level of virtual machines. IaaS clouds like Amazon EC2 provide tools to automate this, e.g., Amazon CloudWatch [1] and AutoScaling [2]. Mathematical modeling of externally observable metrics using e.g., Kalman filters [7] and wavelets [14] has also been shown to be effective for certain workload classes. All the above techniques operate at the level of virtual machines (VMs). The unit of scaling is the VM instance – VMs are added if additional resources are needed and terminated when they are under-utilized; additional bandwidth is provisioned according to the VMs' needs. DevOps tools are popular for several reasons: (i) software development and operation are separate concerns, enabling any distributed application to be deployed to the cloud as long as the components can execute inside VMs, (ii) developers do not need to learn new programming models – the removal of an under-utilized VM was simply treated as "node failure", and the addition of a VM treated as a "recovering node"; distributed applications have always been designed to handle failures (iii) hypervisors enable the collection of various metrics of VMs, like CPU, memory and network utilization, which are then sent to tools like CloudWatch. Operational staff only need to be trained to understand such metrics/tools; they can be used to manage several applications (possibly from different vendors) without understanding any application internals.

To assist the application developer and reduce the complexity of engineering robust distributed applications, several researchers and organizations have proposed domain-specific programming frameworks for *implicit elasticity*, where elasticity is the responsibility of the framework and not the developer (examples are various "big-data" frameworks like map-reduce [8], Dryad [9]; event stream processing systems like IBM System S, etc.). Such frameworks are typically designed with the goal to keep developers' task simple and high-level, i.e., writing SQL-like queries (as in Shark,Pig) or map-reduce functions while masking lower level details like distribution, load balancing and elasticity. Such frameworks take data and operations (queries, map-reduce functions, etc.) as input and apply optimized domain-specific algorithms to infer the correct number of worker tasks, mapping of tasks to VMs/machines (scheduling) and other runtime parameters necessary to complete the analytics job within the specified deadline. Elasticity is implicit, based on either the volume/velocity of data and/or desired deadlines. However, such frameworks are limited to data analytics, and many distributed applications do not fit their programming model.

III. Why Explicit Elasticity

Programmable Elasticity in distributed applications has remained under-explored. This is especially vital for applications that do not fit the programming model of (implicit) elastic "analytics" frameworks like Hadoop, Pig, etc., but require high performance, scalability and elasticity.

Consider multi-tier web applications which have conditional flows. The best example, as illustrated in Figure 1, is an electronic commerce application like an online store, that is typically split into a web frontend, inventory management system, pricing database, customer tracking and ad serving components. A simple visit to the store (from a browser) will impact the web frontend, customer tracking and ad serving components. However, a purchase will impact the pricing database and inventory management system as well. Hence, depending on the context of the visit, two different (code) paths are exercised. There is a causal relationship between $Simple$, and B_1, C_1, C_2, B_2, i.e. $Simple \rightarrow B_1 \rightarrow C_1 \rightarrow C_2 \rightarrow B_2$. Also, $Purchase \rightarrow D_1 \rightarrow E_1 \rightarrow E_2 \rightarrow D_2$. Simply observing the network traffic to the web server is not sufficient to make a good elastic scaling decision, e.g., during thanksgiving sales or after the release of popular phones, the $Purchase$ path may be exercised heavily. And components serving that path should be scaled to increase revenues without worrying much about customer tracking or ad serving. In general, for multi-tier applications with distinct causal paths, engineering elasticity management components requires an understanding of causal paths, path profiling and path frequency estimation along with using internal application logic to make effective decisions.

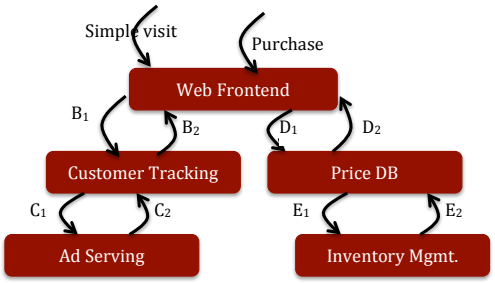

Fig. 1. Example multi-tier application with multiple causal paths

The second class of applications are *datacenter infrastructure applications* like key-value stores (e.g., memcached [12], Hyperdex [5]), consensus protocols (e.g., Paxos [10]), distributed lock managers (e.g., Chubby [4]) and message queues. 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 add additional compute nodes will only degrade its performance further. In the case of a distributed cache like memcached [12], different elements of the cache may have varying access frequencies, and need different levels of elasticity. Again, this is an argument for explicit or programmable elasticity where application-specific logic has to be used for elastic scaling.

IV. Explicitly Elastic Programming Frameworks

We propose two explicitly elastic distributed programming frameworks: (1) ElasticThrift, which is specific to the remote-procedure call (RPC) programming style, but compatible with a wide variety of programming language runtimes, and (2) ElasticJava, an extension of Java which supports a wide variety of programming idioms but is specific to a single language runtime, i.e., Java. Both these frameworks are under development, and we present an overview in this paper along with preliminary results.

A. ElasticThrift

Modern distributed applications are often *polyglot*, composed of components programmed in different languages. This is primarily because of *developer affinity* to languages that "fit the problem at hand", and the robustness of certain libraries. For example, developers may have an affinity to JavaScript and PHP for front-end work on web applications and to C++ for scenarios like programming wireless sensors.

Despite the use of multiple languages, many distributed applications follow a service-oriented architecture, where components make remote procedure calls (RPCs) to other components. Despite inducing tight coupling between components, RPCs remain a popular paradigm for distributed programming, including in infrastructure applications because of their simplicity in helping developers conform to a service-oriented architecture without necessarily using RESTful HTTP interfaces (that have huge overheads and impact performance).

ElasticThrift is a cross-language elastic RPC framework, and an extension to Apache Thrift [6]. Apache Thrift is an open-source cross language serialization and RPC framework. ElasticThrift uses an Interface Definition Language (IDL) similar to Apache Thrift. The ElasticThrift IDL is designed to make describing application types and service interfaces straightforward and language independent. The IDL Compiler reads IDL files and outputs serialization code and RPC stubs in a variety of languages. Consider a project where a Python web script needs to make a call to a Java program responsible for processing credit cards. In Apache Thrift, this is designed by creating an IDL file describing the service interface over which the Python code will communicate with the Java server. ElasticThrift's IDL is similar to Thrift's IDL, except that it allows additional decision making for elasticity, at a high level. This IDL can then be compiled by the ElasticThrift IDL Compiler, which will generate Python and Java code along with required network sockets and serialization to allow the two programming languages to interact through ElasticThrift's runtime system. ElasticThrift aims to enable rapid development of complete polyglot services in a few lines of code.

```
elastic service AirlineBooking {
  string fareQuote(string flight, string date);
  string book(string flight, PassengerDetails d);
  int scaleUp() {
    fql = EThrift.getAvgLatency(fareQuote);
    bl = EThrift.getAvgLatency(book);
    if(fql >= 0.5 * bl)
      return 1;
  }
  ...
}
```

Fig. 2. An outline of an elastic component implemented in a service-oriented manner to serve RPCs using ElasticThrift IDL. The actual implementations of the method calls can be language specific, but the developer can use application specific metrics (like average latency of execution of the RPC above) to make elastic scaling decisions

Figure 2 illustrates a simple elastic server implemented with ElasticThrift, where the developer uses application-specific logic to decide when to add new servers – the developer adds servers when the average latency to retrieve an airfare is more than half of the latency involved in booking a ticket. This is for illustrative purposes only, and the developer can declare variables in the IDL and use complex conditions on them to decide on elastic scaling. The ElasticThrift runtime periodically calls `scaleUp` to determine the number of servers that have to be added to the server pool – in this case, the programmer has decided to add one server at a time. All IaaS-level interactions to get additional VMs to instantiate servers is performed by the ElasticThrift runtime.

B. ElasticJava

While ElasticThrift focussed on elasticity in the RPC paradigm, we wanted to investigate the benefits of providing several other elastic programming idioms, including elastic classes, event handlers, methods and stream graphs. We have chosen these idioms because most datacenter applications we examined consisted of components composed of remote methods, event handlers and graphs of interconnected data streams. ElasticJava is an extension of Java, and consequently the unit of elasticity is an object. The programmer is only concerned with deciding when to add or remove objects to the application. Consider an electronic commerce application requiring elasticity, as illustrated in Figure 1. The ad serving system may be represented by a collection of objects, which is elastic, i.e., objects are added to the collection during workload spikes and removed when they are no longer required to meet desired QoS. The basic programming abstraction in ElasticJava is the elastic class, which may consist of one or more elastic methods, event handlers and stream graphs. An elastic method executes like any remote method, except that the developer can instruct ElasticJava runtime to add more objects to handle said method calls when workload (e.g., frequency of invocations) increases. ElasticJava thus also provides a transparent way to replicate objects without the explicit use of ordered broadcasts or consensus protocols.

```
elastic class AdServer {
  Budget b;
  elastic List<Ad> getTopRelevantAds(...) {
    if(b.getValue > 0) { .. }
  }
  elastic handler(AdvertisingBid bid){ ... }
  int scaleUp() {
    float lat = EJava.getAverageLatency(this.
     getTopRelevantAds);
    if(lat > 200 && b.getUpdateRate() < 60)
      return 2;
  }
  int scaleDn() {...}
}
```

Fig. 3. An example of an elastic class in ElasticJava which uses internal application logic to decide when to add or remove objects at runtime.

Figure 3 illustrates a simple elastic class in ElasticJava, with one elastic method and event handler. The programmer has to implement these methods following usual "single-machine shared memory" programming practices. Although, at runtime, the elastic class is instantiated into a set of objects which may reside on different JVMs on different machines, all aspects of distribution and bootstrapping, i.e., interacting with IaaS

clouds to obtain additional VMs, start JVMs and creating objects is handled by the ElasticJava compiler and runtime. All shared variables are stored in a distributed key-value store shared among all objects in ElasticJava's runtime. All accesses to shared state, both reads and writes are mediated by the shared key-value store. We are also investigating using group communication protocols like atomic broadcast in lieu of key value stores. We have observed that group communication has performance advantages when accesses to shared state are predominantly reads, while key-value stores are effective when accesses are atleast 40% writes. Similar to ElasticThrift, the programmer can implement `scaleUp` and `scaleDn` to signal the number of objects that have to be added or removed from the elastic object pool. These well-known methods are called periodically by the runtime to make elastic scaling decisions.

Our previous work on ElasticRMI [11] considered a library for RPC-style programming in Java as opposed to a full fledged language like ElasticJava with support for elastic thread pools and graphs of interacting event handlers for stream processing. The use of specific keywords (**`elastic`**, **`handler`**) enables program analyses to infer all paths a request to one component takes – all possible causal paths can be determined and profiled following standard techniques [3].

V. Preliminary Evidence of Effectiveness

Elasticity is measured by a metric called *agility*, which averages the amount of (1) under-provisioned resources (that degrades application performance under increasing workload) and (2) over-provisioned (and hence wasted) resources. The mathematical definition of agility is available at [15] (standardized by SPEC), and lower values are better. This is because the ideal value of agility is zero, corresponding to neither under- or over-provisioning. The system is evaluated with rapidly varying workloads (following sinusoidal, abrupt and step-like patterns) per SPEC guidelines [15]. Our previous work on ElasticRMI [11] examined empirically the use of elastic methods (only) for explicit elasticity. The results were promising (at least 2× better agility when compared to DevOps tools under all workload patterns) and inspired our work on ElasticThrift and ElasticJava. We have evaluated the elasticity of the popular Apache Cassandra key-value store using ElasticJava and the ACME Airline Reservation system [13] using ElasticThrift. Under sinusoidal workload patterns, the agility of ElasticThrift and ElasticJava implementations with explicit elasticity is 78.9% and 87.6% better than that of the native versions scaled using CloudWatch and AutoScaling. All experiments were conducted on Amazon EC2. Under more extreme variations in workload (three spikes of more than 45%), the performance of the explicitly elastic versions is even better – 159.8% and 212.3% better agility than CloudWatch and AutoScaling.

VI. Conclusions

To summarize, we have outlined why predominantly used techniques for elasticity are ineffective for many datacenter applications, either due to multiple causal paths or complex application logic (e.g., distributed mutual exclusion). We propose to develop explicitly elastic programming frameworks that enable application developers to use fine-grained application-specific information in elastic scaling. We outline and are developing two frameworks simultaneously – ElasticThrift and ElasticJava – to experiment with explicit elasticity and gain insights into whether elasticity is best incorporated directly into programming languages like Java or into IDLs like Thrift, both of which are popular in distributed programming today. Early results both from our previous work on ElasticRMI and from experiments with ElasticJava and ElasticThrift are positive, indicating that explicit elasticity improves the adaptability of a distributed application to changing workloads. We are also working on combining static and dynamic slicing techniques to infer causal paths in large-scale distributed applications, and on the feasibility of program transformation techniques to inject elasticity management into them.

References

[1] Amazon Web Services (AWS) Inc. Amazon CloudWatch. *http://aws.amazon.com/cloudwatch/*, 2012.

[2] AWS Inc. Amazon Auto Scaling. *http://aws.amazon.com/autoscaling/*, 2012.

[3] Thomas Ball and James R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[4] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[5] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Hyperdex: A distributed, searchable key-value store. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 25–36, New York, NY, USA, 2012. ACM.

[6] Facebook Inc. Apache Thrift. *http://thrift.apache.org/*, 2012.

[7] Anshul Gandhi, Parijat Dube, Alexei Karve, Andrzej Kochut, and Li Zhang. Adaptive, model-driven autoscaling for cloud applications. In *11th International Conference on Autonomic Computing (ICAC 14)*, pages 57–64, Philadelphia, PA, June 2014. USENIX Association.

[8] Hadoop. http://hadoop.apache.org. 2015.

[9] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[10] J. Kirsch and Y. Amir. Paxos for Systems Builders. *http://www.cnds.jhu.edu/pub/papers/cnds-2008-2.pdf*, 2008.

[11] K. R. Jayaram. Elastic remote methods. In *ACM/IFIP/USENIX International Middleware Conference (MIDDLEWARE'13)*, pages 143–162. Springer Berlin Heidelberg, 2013.

[12] Memcached. http://www.memcached.org. 2015.

[13] Netflix Inc. The ACME Air Travel Reservation System. *https://github.com/aspyker/acmeair-netflix*, 2015.

[14] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 69–82, San Jose, CA, 2013. USENIX.

[15] SPEC Open Systems Group (OSG). Report on Cloud Computing to the OSG Steering Committee. *http://www.spec.org/osgcloud/docs/osgcloudwgreport20120410.pdf*, 2012.

[16] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, 2012. USENIX.