Towards Liquid Service Oriented Architectures

Daniele Bonetta
supervised by Cesare Pautasso

Faculty of Informatics
University of Lugano — USI
Lugano, Switzerland
{name.surname}@usi.ch

ABSTRACT

The advent of Cloud computing platforms, and the growing pervasiveness of Multicore processor architectures have revealed the inadequateness of traditional programming models based on sequential computations, opening up many challenges for research on parallel programming models for building distributed, service-oriented systems. More in detail, the dynamic nature of Cloud computing and its virtualized infrastructure pose new challenges in term of application design, deployment and dynamic reconfiguration. An application developed to be delivered as a service in the Cloud has to deal with poorly understood issues such as elasticity, infinite scalability and portability across heterogeneous virtualized environments. In this position paper we define the problem of providing a novel parallel programming model for building application services that can be transparently deployed on multicore and cloud execution environments. To this end, we introduce and motivate a research plan for the definition of a novel programming framework for Web service-based applications. Our vision called "Liquid Architecture" is based on a programming model inspired by core ideas tied to the REST architectural style coupled with a self-configuring runtime that allows transparent deployment of Web services on a broad range of heterogeneous platforms, from multicores to clouds.

Categories and Subject Descriptors

D.1.0 [**Programming Techniques**]: General

General Terms

Design, Performance, Languages

Keywords

REST, Web Services, Programming Models, Performance, Liquid Architectures

1. INTRODUCTION AND MOTIVATION

RESTful Web services have emerged as a radically simplified approach to design the plumbing of modern service oriented architectures. The simplicity of the HTTP protocol has been one of the winning features that have promoted this success, while the careful design of the protocol has enabled the Web to scale and exponentially increase its size over the past 20 years. However, it remains challenging to design, build and operate RESTful Web services that can exibit the same properties. Two technologies have recently appeared that could play a relevant role in the future of service oriented computing and require further research to determine their impact on the next generation of Web services; those technologies are clouds and multicores.

Cloud computing and multicore processor architectures are two emerging classes of execution environments that are rapidly becoming widely adopted for the deployment of Web services, and thanks to their intrinsic parallelism, services can be redesigned in order to exploit the available computational power. Nevertheless, both computing platforms require the software to correctly use a very large and potentially heterogeneous pool of available execution resources. For instance, in the near future multicore machines with more than 64 cores will become mainstream [18]. On such a computing platform, the correct usage of each core will became a relevant issue not only affecting the overall performance of the service, but also impacting its power consumption. The same consideration is also valid on Cloud computing platforms, where the correct usage of the available resources could result in reduced capital expenditure. Moreover, the dynamic nature of Infrastructure as a service (IaaS) cloud computing platforms, coupled with novel cloud pay-as-you-go economic models can encourage the development of applications capable of running on machines characterized by different computational capacity and different cost. For example, we envision applications that will be able to change their deployment configuration, or even to migrate from one cloud provider to another, according to specific economic models indicating in real time the cheapest cloud provider to support the actual load of a Web service.

To deal with such important scenarios, we think that the research and development of a novel programming paradigm allowing to build services that can be transparently deployed on both clouds and multicores could represent a challenging research topic. In the following section we define in detail the nature of this research, identifying the main challenges, and defining the concept of liquid architecture to deal with those challenges.

1.1 Research Challenges

Within the area of service oriented architectures, the development of services able to exploit different computational resources is a relatively new research area. The growing availability of parallel processing units present in both large

WWW 2011, March 28–April 1, 2011, Hyderabad, India.
ACM 978-1-4503-0637-9/11/03.

virtualized environments such as IaaS Clouds and modern multicore machines represents an opportunity for the definition of a new class of Web service architectures, which we call *liquid* architecture.

We define a liquid Web service as a software entity, capable of the following three properties:

1. *Transparent deployment*: the Web service has to be deployable on any kind of computing platform, with no need for any external reconfiguration effort. To give an example, a Web service capable of transparent deployment has to be deployable on any kind of multicore machine (from 1 core, up to the technological limit which may approach hundreds of cores in the foreseeable future), as well as on any kind of virtualized Cloud computing infrastructure, like for instance an Amazon E3 Virtual Cluster. The same applies on hybrid deployment scenarios, i.e., Clouds composed of virtualized multicore processors.

2. *Infinite scalability*: the transparent deployment has to guarantee the optimal level of scalability for the Web service, for the given computing platform. To give an example, a Web service deployed on a 8 core machine has to perform exactly 8 times better than it would perform on the single core environment. The same linear scalability has to be guaranteed on all possible heterogeneous scenarios.

3. *Very long life*: since a service is supposed to be kept in operation for a very long period, an architecture capable of the prior two properties should also be capable of a self-organizing deployment mechanism, permitting to its configuration to change during the service life cycle. To give an example, such liquid Web services should be able to adapt at runtime to the available computing power (if deployed on a cloud environment). If deployed on a multicore processor, a liquid Web service should be able to save energy by using only the necessary amount of cores, leaving powered off all the unused resources for future peaks of requests.

The design and development of Web services capable of the above three properties represents to us a meaningful research challenge, and the maturity of modern computing platforms is a stimulus to our work.

2. RELATED WORK

The idea of treating Clouds and Multicores as a single computing environment has been introduced by David Wentzlaff et al. within the contest of the Fos Operating System [22]. They propose the development of a modern Operating System to target at the same time and in parallel the two different computing platforms, using a well defined service based architecture. Within Fos every OS function is implemented as a Web service exposing OS kernel functionalities. For instance, a process loaded in the Fos Operating System will interact with a scheduler service, a malloc service, a file I/O service, and so on. The adoption of such a service oriented structure is of great interest, as it permits the transparent deployment of the Operating System on both Clouds and Multicores. Other examples of distributed Operating Systems that have somehow tried to address similar scenarios with peculiar solutions are Amoeba [15], Sprite [17] and Clouds [11]. Unlike the case of a distributed OS, in our scenario we are more interested in Web services instead of general applications. Due to this reason, the Operating System approach seems to be too generic for our scenario, and we think that the research challenges introduced in the prior section could be targeted with an approach more focused to the programming model level.

In the area of programming models and tools, plenty of different solutions have been proposed across the last decades, to build parallel and concurrent applications. Notable in this area are some relevant programming paradigms from the high performance computing field, like the Skeletal model [9] and the Flow based model [14]. All of these models embrace a simple but very effective idea of parallelism: if you have the need for scalable applications, at some points you will have the need to isolate and replicate some portions of your computation. We consider this aspect as relevant for our approach, but despite of the power of such parallel programming models, none of them has been designed explicitly targeting the Web service scenario. Many of them have been designed for local (in-machine) concurrency, or to exploit process/thread level parallelism, and this represents a limit for distributing services over the Web.

Important efforts in the development of scalable systems have also been done in the definition of concurrency models more focused on the semantic of cooperating processes than in the communication primitives. Belonging to this area are well known models like the Actor Model [1], the Agent-Oriented programming paradigm [5], the Active Object [12], or the Partitioned global address space model [8]. All of these concurrency paradigms offer interesting solutions to the scalability problem, and some of them have shown how general concepts can be efficiently implemented in real-world applications. Good examples for this class are the Microsoft CCR/DSS framework [10], Scala [16], and the X10 [7] programming languages.

Out of the realm of parallel programming, relevant efforts have been done through the development of frameworks and instruments explicitly designed for Cloud computing platforms [21]. Microsoft Azure is a relevant example in this area. Thanks to its service oriented layer, the Azure platform allows to deploy in the cloud complex data-intensive applications [13], as well as any kind of Web service, including third party components [6].

3. PROPOSED APPROACH

As introduced in Section 1, there is a lack of programming abstractions and techniques which prevents Web services to fully exploit both Clouds and Multicores. Also, none of the solutions discussed in Section 2 seems to provide all of the technological and conceptual tools needed for the development of Web services capable of satisfying the requirements enumerated in Section 1.1.

To address these limitations, we propose an approach based on the definition of a novel programming paradigm explicitly targeting the development of liquid Web services.

We structure our approach around three key elements: (1) a service oriented *programming model* for the constrained design of the liquid service, (2) a service oriented *standard library support* for the implementation of Web services' key features (sharing of state, composition, and reflection), and (3) a smart *runtime support* for the runtime self-adaptation needed to implement the liquid deployment properties of

the Web service. Taken all together, these three components will contribute to the definition of a novel programming paradigm for the development of liquid Web services.

3.1 Service Oriented Programming Model

To exploit the high parallelism level of both Clouds and Multicores, Web services have to be programmed according to a parallel programming model. Any other approach not taking into account this need for an intrinsic parallel architecture of a Web services would result in not providing the necessary flexibility to scale the service. Of the several parallel programming paradigms proposed until now, we see in the Actor model [1] approach a good candidate solution for our scenario.

The Actor model has shown its validity in programming languages like Erlang [2] and Scala [16], demonstrating its flexibility and scalability on large concurrent applications. According to the Actor model, we propose to develop Web services as composed of several entities cooperating through messages. However, unlike the standard Actor model, we propose a novel variation refining the communication mechanism employed.

The communication mechanism usually implemented with Actors is based on message passing. While this simple abstraction is powerful and very expressive, we think that it is too general, and we propose to change it with something more constrained. To this end, we propose a novel implementation of the Actor model based on three main design aspects. We call this variant of the notion of Actors *Reactors*: RESTful actors.

In detail, in our programming model each Reactor is a complete RESTful Web service. We provide all the programming abstractions necessary to specify each Reactor's business logic, but we restrict the way Reactors communicate to the REST uniform interface of HTTP verbs. According to the REST architectural style each Reactor can be seen as a resource which can be manipulated through state transfer operations. We can call this programming model a service oriented programming model where every Reactor is at the same time a service provider and a service consumer.

Second, each Web service application has to define at least one specific Reactor, called the *Entry Reactor*. This particular element will act as the Web service's front-end, accepting requests from external clients and providing responses to their requests.

Finally, we can distinguish two general classes of Reactors: managed and unmanaged. Managed Reactors are Reactors whose business logic has been expressed within the application, by the Web service programmer. Unmanaged Reactor are proxies to external, third-party RESTful Web services used as external components by managed Reactors to provide the final response.

The previous three fundamental design choices permit to express any kind of Web service as a composition over a set of concurrent entities (Reactors) that are accessible and can be manipulated through their uniform interface.

To guarantee scalability, Reactors providing only pure functional behaviors should be freely replicable by the runtime. The scalability of more complex, stateful should be guaranteed using a set of distributed shared memory functionalities provided by the liquid services standard library in form of services.

From an implementation perspective, we plan to experiment with the use of JavaScript as the main programming language for our Reactor model. JavaScript is a rapidly maturing language which is starting to appear also in server-side deployments. We plan to leverage its extensibility features by adding a number of built-in objects that will be used at design time to specify the RESTful interfaces. In this way, we will provide a programming environment based on JavaScript where a Web service will be programmed as a set of predefined JavaScript prototype objects (JavaScript Reactors), that will be compiled and executed as pure RESTful Web services.

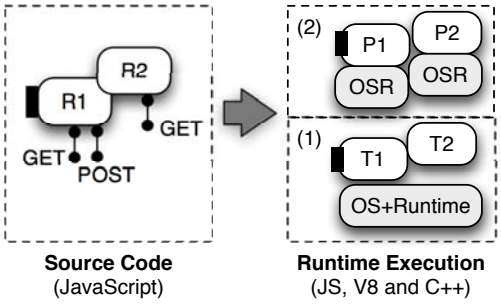

Figure 1: Compilation and adaptive deployment: Reactors are defined in the JavaScript source code with their RESTful interfaces. At runtime, the deployment support instantiates one thread per Reactor if the deployment has to be performed on a Multicore machine (1), or instantiates two different processes on two different machines if on a Cloud computing environment (2).

3.2 Liquid Runtime Support

We expect that the design of Web services as a set of concurrent Reactors will be of great help to achieve the envisioned liquid properties at runtime. The runtime execution support should implement three features: Reactors compilation, Reactors communication, and Reactor deployment.

First, based on the JavaScript source code, the runtime should be able to compile each Reactor as an independent entity, with a global view on all the other Reactors, but executed on a separate JavaScript frame, and of course on a separate thread or process. Also, in a multicore execution environment, each standalone hardware thread or process should be scheduled to run just within a fixed set of cores or CPUs, to reduce contention, take advantage of locality and optimize resources utilization. In a Cloud environment, similar considerations should drive the placement of the Reactors within the appropriate virtual machine instances.

Second, the runtime should implement a late-binding mechanism to chose the optimal communication primitive to be used to connect the Reactors, according to the specific deployment the runtime is targeting (see Figure 1). For a shared memory deployment (e.g., on a multicore machine), we expect that it would be beneficial to implement Reactors communications with very efficient zero-copy shared-memory buffers rather than using the full HTTP/TCP/IP stack. Changing the deployment context to the Cloud, the communication mechanisms should be implemented through the intra-Cloud communication middleware offered by the cloud provider, or if none is available, through standard HTTP. Considering a more advanced, hybrid scenario (e.g. a cluster of distributed multicore machines), the runtime

should be able to take similar decisions in order to let the distributed Reactors communicate in the most efficient way. The runtime should take special care of the Entry Reactor and publish it on a standard HTTP server. Likewise Unmanaged Reactors should be efficiently bound to their external RESTful API providers.

Finally, the runtime should implement features for the real time monitoring of available computing platforms, as well as all the policies for the reconfiguration features required to achieve dynamic adaptation driven by external (e.g. Cloud variable pricing) or internal (e.g. Multicore power consumption) factors.

Since the target language is JavaScript, the runtime will be implemented with core communication libraries and Reactors objects written in high performance event-loop based C++ code, and part of the runtime logic support written using standard JavaScript. We are currently evaluating Google V8 [1] as a possible hosting virtual machine that will be utilized by the runtime to execute the JavaScript code, and Node.js [2] as the evented I/O networking framework. The JavaScript virtual machine will be embedded in the native core of the runtime, who will take care of Reactor allocations on different frames and their migration across the Cloud.

3.3 Liquid Services Standard Library

Every programming environment provides a set of reusable functionalities in the form of a standard library. For example, the standard library of the C language usually contains all the functions that are present in every implementation of the language. In our programming model we foresee the need for two features: a shared memory and a service composition engine.

The shared memory is needed for scalability reasons. We think that Reactors requiring to share great amounts of data (for instance, a database connection), should have access to a common high performance shared memory in which they can store data that has to be delivered to many other Reactors. This does not conflict with the constraint of ensuring that all Reactors comply with the same uniform interface. In fact, also the Reactor providing access to the shared memory can use the basic GET/PUT/DELETE primitives shared by the interfaces of all Reactors. In the same way the runtime provides the optimal interconnection between the Reactors, we expect the runtime to manage the implementation of the shared memory with the most efficient technology within the given deployment environment.

The second functionality is the service composition engine, and is required to provide Reactors with a very efficient reflection mechanism. Every Reactor will be allowed to ask the service composition engine for the execution of an orchestration over a subset of other Reactors. In this way, every Reactor can "take control" of other Reactors, and coordinate them for specific goals. Also, this component is very important for unmanaged Reactors: the presence of a service composition engine as a primitive entity in the framework allows Reactors to consume not only simple third party Web APIs, but also to compose them, and execute the composition.

From an implementation perspective, we plan to reuse the following technologies to implement our standard library: memcached[3] as the shared memory component, and JOpera[4] as the service composition engine [19].

4. METHODOLOGY

The methodology that will be adopted for the development of the programming framework is based on three main milestones to be reached in parallel for all of the three key components of the framework.

For each of component the first milestone is to define a complete set of specifications and requirements. For this first milestone, the runtime support will be defined in term of its core mechanisms and functions. A complete set of C++ native objects will be defined and all the native interfaces to be exposed to the JavaScript code will be identified. For what concerns the programming model, we plan to identify the complete set of methods and functionalities to be exposed to the JavaScript level for the definition at design time of the RESTful actors. For the standard library, nothing has to be specified for this milestone as the two core components of the library will be accessible to other actors using standard HTTP verbs (this has allowed us to start with the development of one of the components that will finally compose the service oriented standard library).

The second milestone will require to have a running prototype of the entire system, able to be deployed on different shared memory machines, to implement, stress and test the runtime support on multicores. This milestone will require to have a core set of native objects implementing high performance communication and mechanisms for the local shared-memory interaction of Reactors.

The third and final milestone will require to have a running prototype of the framework with a running version of the liquid runtime deployment support deployable on a Multicore and on an Amazon virtualized cluster as well.

For each milestone we plan to validate our approach developing test-case and small benchmark applications and comparing them with real world existing Web services. For the second milestone we plan to develop a business process execution engine entirely written in our framework, while for the third milestone we plan to develop a complete Web framework (like Lift[5]) to evaluate the behavior of the runtime under very intensive data loads.

4.1 Results and Current Status of the Research

According to the three milestones identified above, we have begun working on the definition the complete set of specifications for the programming model and the runtime support. In the meanwhile, we are also working on the development of one of the two key components of the shared memory standard library: the service composition engine. In our preliminary work [3, 4, 20] we have explored and optimized the performance of a business process execution engine on modern multicore machines. In particular, our work has been focused on scalability optimizations of the engine for different multicore machines. To this end, we have proposed a self-configuration startup mechanism based on the notion of hardware-awareness: the service composition engine is able to identify certain characteristics of the hosting hardware platform (e.g. number of cores, number of CPUs,

[1] http://code.google.com/p/v8/
[2] http://nodejs.org
[3] http://memcached.org
[4] http://www.jopera.org
[5] http://www.liftweb.net

number of caches, etc.) and according to those low level information it is able to reconfigure its deployment policies, for instance by changing how threads are scheduled by the OS.

5. CONCLUSIONS AND FUTURE WORK

In this paper we have introduced the concept of *liquid* Web service. Liquid Web services are software services capable of automatic deployment and self-configuration in order to transparently target different computing platforms, from multicores to clouds, from shared-memory architectures to distributed-memory architectures.

We have proposed a novel programming paradigm based on the notion of RESTful actors (called Reactors), and we have motivated our decision to start with the development of a programming framework that will implement that concept. Reactors are autonomous software entities that cooperate concurrently within the application, but unlike traditional actors they communicate through a RESTful uniform interface as opposed to message passing. Each Reactor is a loosely coupled entity that can be independently migrated and replicated depending on the performance needs of the overall application, which is designed as a composition of late-bound Reactors.

To support the Reactors model, we have depicted the main characteristics that the runtime support will provide (compilation, deployment, monitoring, adaptation), and we have introduced some functionalities that have to be exposed to every Reactor in the form of a standard library (shared memory, reflective composition).

During the course of the next few years we will work on the development our framework as planned in Section 4, with particular attention to the evaluation and validation of each milestone for each component of our framework for liquid service oriented architectures.

Acknowledgements

This work is partially funded by the Swiss National Science Foundation with the SOSOA project (SINERGIA grant nr. CRSI22_127386).

6. REFERENCES

[1] G. Agha. *Actors: a Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[2] J. Armstrong, R. Virding, C. Wikstrom, and M. Williams. *Concurrent Programming in Erlang*. Pentice Hall, 1996.

[3] D. Bonetta, A. Peternier, C. Pautasso, and W. Binder. A Multicore-aware Runtime Architecture for Scalable Service Composition. In *2010 IEEE Asia-Pacific Services Computing Conference*, pages 83–90, 2010.

[4] D. Bonetta, A. Peternier, C. Pautasso, and W. Binder. Towards Scalable Service Composition on Multicores. In *On the Move to Meaningful Internet Systems: OTM 2010 Workshops*, pages 655–664. Springer, 2010.

[5] J. M. Bradshaw, editor. *Software Agents*. MIT Press, 1997.

[6] J. Cała and P. Watson. Automatic Software Deployment in the Azure Cloud. In F. Eliassen and R. Kapitza, editors, *Distributed Applications and Interoperable Systems*, volume 6115 of *LNCS*, pages 155–168. Springer, 2010.

[7] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar. X10: an Object-Oriented Approach to Non-Uniform Cluster Computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, 2005.

[8] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarría-Miranda. An Evaluation of Global Address Space Languages: Co-Array Fortran and Unified Parallel C. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 36–47. ACM, 2005.

[9] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1991.

[10] F. Correa. Actors in a New Highly Parallel World. In *International Conference on Software Engineering 2009*, pages 0–3, 2009.

[11] P. Dasgupta, R. LeBlanc Jr, M. Ahamad, and U. Ramachandran. The Clouds Distributed Operating System. *Computer*, 24(11):34–44, 2002.

[12] R. G. Lavender and D. C. Schmidt. Active Object – An Object Behavioral Pattern for Concurrent Programming, 1996.

[13] E. Meijer. Democratizing the Cloud. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 858–859, 2007.

[14] J. Morrison and J. Morrison. *Flow-Based Programming: A New Approach to Application Development*. Van Nostrand Reinhold, 1994.

[15] S. Mullender, G. Van Rossum, A. Tananbaum, R. Van Renesse, and H. Van Staveren. Amoeba: A Distributed Operating System for the 1990s. *Computer*, 23(5):44–53, 2002.

[16] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An Overview of the Scala Programming Language. *LAMP-EPFL*, 2004.

[17] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The Sprite Network Operating System. *Computer*, 21(2):23–36, 2005.

[18] D. Patterson. The Trouble with Multi-core. *Spectrum, IEEE*, 47(7):28–32, 2010.

[19] C. Pautasso. Composing RESTful Services with JOpera. In *Software Composition*, pages 142–159, 2009.

[20] A. Peternier, D. Bonetta, C. Pautasso, and W. Binder. Exploiting Multicores to Optimize Business Process Execution. In *IEEE International Conference on Service-Oriented Computing and Applications*, pages 131–138, 2010.

[21] B. Rimal, E. Choi, and I. Lumb. A Taxonomy and Survey of Cloud Computing Systems. In *INC, IMS and IDC, 2009. NCM'09. Fifth International Joint Conference on*, pages 44–51. IEEE, 2009.

[22] D. Wentzlaff and A. Agarwal. Factored Operating Systems (FOS): the Case for a Scalable Operating System for Multicores. *ACM SIGOPS Operating Systems Review*, 43(2):76–85, 2009.