An Architectural Style for Liquid Web Services

Daniele Bonetta and Cesare Pautasso
Faculty of Informatics, University of Lugano (USI)
{*name.surname*}@*usi.ch*

***Abstract*—Just as liquids adapt their shape to the one of their container, liquid architectures feature a high degree of adaptability so that they can provide scalability to applications as they are executed on a wide variety of heterogeneous deployment environments. In this paper we enumerate the properties to be guaranteed by so-called liquid service-oriented architectures and define a set of design constraints that make up a novel architectural style for liquid architectures. These constraints drive the careful construction of a pattern, the RESTful Actor (REactor), which enables to deliver the required scalability by means of replication of its constituent parts. REactors feature a RESTful Web service interface and a composable architecture which is capable of delivering scalability and high performance in a way that is independent from the chosen deployment infrastructure. We discuss how the REactor can be deployed to run on distributed (shared-nothing) execution environments typical of virtualized Cloud computing environments as well as on modern multicore processors with shared memory architectures.**

***Keywords*-Scalability, Performance, Web Services, REST, Architectural Styles and Patterns**

I. Introduction

Scalability to serve a very large number of clients and transparent reconfigurability across heterogeneous execution environments are two desirable characteristics of modern Web services. Whereas it is possible to design and build Web services which scale to handle millions of concurrent requests without visible performance degradation [1], the architecture of the services is typically designed taking closely into account the underlying hardware execution environment [2]. For example, a CPU-intensive software service designed to run on a homogeneous cluster of computers may not provide the same scalability when deployed on a different set of machines, such as a virtualized cluster of heterogeneous multicore processors [3].

In this paper we study how to design services able to scale across different execution infrastructures, from distributed memory environments to shared memory multiprocessor machines. The goal is to provide a set of design constraints (codified as an architectural style), which allow services to achieve scalability and transparent reconfigurability over heterogeneous deployment environments. Whereas it is possible to achieve such qualities by making correct use of existing service-oriented middleware technologies, our proposed pattern-based approach can potentially significantly reduce the design effort required as it can be supported by an autonomic run-time software infrastructure [4].

The paper makes the following contributions. We propose a novel architectural style for so-called *liquid* services and introduce the corresponding design constraints. These are sufficient to design service-oriented architectures which can scale by making efficient use of whatever hardware infrastructure is available. Following these constraints, we introduce an architectural pattern for the design of liquid services called "REactor" (from RESTful actor) and discuss how systems making use of it can exhibit "liquid" qualities.

The liquid architectural style constraints influence the architecture of liquid services at design-time. At run-time these constraints make it possible to build a runtime container, which can automatically identify the kind of reconfigurations needed in different operating conditions, and apply them at the right time to obtain the liquidity of the services that are supported by it [5]. While the description of such run-time is out of scope for this paper, initial results can be found in [6], [7].

The rest of this paper is organized as follows. We motivate the need for guidance in the design of scalable Web services in Section II. The background Section III defines scalability and introduces the CAP theorem. In Section IV we introduce the notion of liquid Web services, identifying the defining qualities for such services. In Section V we introduce the liquid architectural style as a set of constraints for the design of liquid Web services. Section VI introduces the REactor pattern and Section VII shows different scalable configurations for the REactor. In Section VIII we evaluate how services developed according to the REactor pattern comply with the constraints of the liquid architectural style. Section IX presents related work before we conclude in Section X.

II. Motivation

Systems designed to exploit a specific hardware infrastructure may significantly degrade their performance if deployed on different ones. This is partially due to the challenge of tuning the configuration of the software to optimally make use of the available hardware resources, but also due to more fundamental problems, such as the difficulty of providing communication primitives that hide the architectural differences of the execution platform (e.g., shared-memory, multicore servers vs. distributed-memory clusters [8]).

As a response to these challenges, dynamically reconfigurable (i.e., elastic) execution infrastructures are rapidly becoming one of the predominant choices for running service-oriented architectures. Technologies such as PaaS and IaaS [9] have introduced a novel paradigm for Web services deployment, where the hosting infrastructure for a service is partially or totally unknown at design time. With it, services are designed to be executed on a virtualized Cloud infrastructure which makes use of abstraction layers that make it difficult for the services to exploit and take advantage of the underlying physical hardware.

The flexibility offered by the virtualization layer has notable advantages in term of scalability, offering services the opportunity to pay for computing resources only when needed, with an economic benefit exploited by Cloud providers adopting the "pay-per-use" business model. Generally, Infrastructure as a Service (IaaS) Clouds provide a virtualized distributed memory infrastructure, which can be used by services running on a cluster of independent virtual machine instances, each with its own private memory space [10].

Another alternative for Web services deployment is represented by the shared memory model. This is found in most modern multicore machines, allowing a service to scale by making use of multiple processor cores [11]. Once all cores have been used, more shared memory machines can be added, obtaining a heterogeneous cluster of multicores.

It is very challenging to design services which can exploit such different runtime infrastructures, and scaling the same service architecture by exploiting the characteristics of a heterogeneous execution infrastructure requires the architecture to feature qualities such as flexibility, adaptability and dynamic reconfigurability. As the hosting environment changes, the way in which the architecture of the service is deployed should also change, possibly without affecting the original design of the service and the availability of the service.

Defining an architectural style for services with such qualities makes a key contribution to solve the problem of platform-independent design of services which can make optimal and efficient usage of their execution platform [12]. The guidance provided by the style and the pattern we propose can not only provide scalability guarantees over elastic, virtualized environments and non-virtualized ones.

III. Background

A. Scalability

The classic definitions of scalability focus on the ability of a system to handle an increasingly large amount of work [13]. Thus, scalability for a Web service can be defined as the ability for a specific service to provide a guaranteed response time for an increasingly large number of clients. A more formal definition considers the scalability of a system as a function of its efficiency expressed in the following terms:

$$S(n) = \frac{T_1}{T_n}$$

The scalability function $S(n)$ defines the scalability factor for a Web service S under the load of n concurrent clients. The constant T_1 identifies the average response time of the service when only one client request has to be processed at a time, while T_n represents the average response time with n concurrent clients. Excluding from this simple analysis superlinear behaviors, $0 \leq S(n) \leq 1$ holds. Services characterized by a constant value of S for any n are called *scalable* services. Scalable services with S constant and close to the maximum value ($\forall n, S(n) \sim 1$) are called services with an *ideal* scalability.

The scalability of a Web service can be obtained in two orthogonal ways, depending on the available hardware resources [14]. Services are said to *scale up* when they can handle an increasing number of requests because they have been allocated more hardware resources *within* the hosting computing infrastructure (e.g., more processor cores or more memory). Services are said to *scale out*, when they scale to handle a large number of requests as more computing resources have been added *to* the computing infrastructure.

The scale out solution can be applied to services with some degree of replication of their state and which have been implemented with a set of communicating processes deployed on a cluster of computers. Scalability is obtained simply by adding new nodes of commodity hardware to the cluster and replicating on them new service processes. This is the most commonly found approach to scalability in Cloud-based services as it has become very simple to scale a service by instantiating a new virtual machine and adding it to the pool of available resources.

The scale up solution is adopted with service architectures which require access to some kind of shared memory space. Services of this kind make use of thread-level parallelism and target massively multicore machines based on technologies such as cc-NUMA [15]. Modern multiprocessor server-class machines can handle a very high number of parallel threads, thus this approach to scalability is also highly relevant for scalable Web service design. For instance, commercial servers based on the POWER7 architecture can support up to 1024 concurrent threads [16].

Services designed to scale up have usually a hard scalability limit imposed by the hardware (i.e., they will be considered "scalable" up to a given number of clients). Still, they can be designed to exploit efficiently the computing resources on which they have been deployed. Conversely, systems designed to scale out can theoretically provide unlimited scalability [17], as the limit of maximum concurrent clients can be increased just by adding more machines to the service deployment infrastructure.

B. Distributed Systems and the CAP Theorem

As we have discussed, scalability could require to replicate a service by running it in a distributed environment. In the following we summarize a very important implication of applying distribution to the design of a service, which will be later used to evaluate the REactor pattern in Section VIII.

Given the following properties of distributed systems:

- *Consistency*: all nodes of the distributed system are able to see the same value of the same data at the same time.
- *Availability*: all possible node failures do not prevent the overall system from serving clients.
- *Partition tolerance*: no set of failures less than total network failure is allowed to cause the system to respond incorrectly.

The CAP theorem [18] states that it is impossible for a distributed system to guarantee the three properties at the same time. In the context of this paper, the main consequence of the theorem is that whenever a Web service needs to distribute its state across different partitions connected via a network, only one property out of availability and consistency can be guaranteed at the same time.

IV. Liquid Web Services

Relaxing all assumptions on the deployment infrastructure allows us to rethink how services are designed and propose a service architecture which can scale independently of its target deployment environment. To do so, it is important to focus on concurrency, state management and parallelism as primary design concerns.

In this section we outline the main characteristics and qualities for a specific class of service-oriented architecture, which we call *liquid* service-oriented architecture. Such services can be introduced using the following metaphor. Suppose to be given a liquid, and to have two different containers, both equivalent in terms of volume, but of different shape and material. Once poured in the first container, the liquid will adapt itself to fill the volume within the container's shape. The same liquid can be poured in the second container. Gravity will change the liquid's configuration and adapt it to shape of the new container.

A liquid Web service is a service that can be deployed across different runtime infrastructures with minimal manual reconfiguration efforts. Like physical liquids, the service perfectly adapts to the characteristics of its container while maintaining its intrinsic qualities unchanged.

Independently of the metaphor, we define a liquid Web service architecture as characterized by three key qualities:

Q I) Transparent Deployment. The same Web service design is deployable on multiple heterogeneous infrastructures. This means that a Web service should be designed with a sufficiently high level of abstraction so that it can be deployed on a wide class of runtime execution environments. More precisely, a liquid service has to be deployable on any kind of computing platform without manual reconfiguration. To give an example, a Web service capable of transparent deployment has to be deployable on any kind of shared-memory multicore machine (from dual-core, up to the technological limit), as well as on any kind of virtualized distributed Cloud computing infrastructure. For example, an Amazon E3 Virtual Cluster [19].

Q II) Scalability. A Web service has to guarantee scalability and predictable performance making efficient use of the resources available in any deployment scenario. To achieve this, the service design is kept orthogonal to the solution used to achieve the scalability of the service, which should ideally be able to both scale up and scale out. To give an example, a liquid service should be able to scale linearly on a multicore machine (at least up to the number of available cores) as well as on a Cloud IaaS virtualized infrastructure. This means that the service should offer mechanisms that allow it to dynamically adapt itself to exploit the best scalability mechanism for the available hardware. A liquid service which is given a heterogeneous pool of resources should also automatically exploit the pool of resources according to its actual utilization and workload levels. For example, the service should be capable of migrating itself from a local cluster to the Cloud when the number of requests becomes high (e.g., on workload peaks), and then migrate back to the cluster as the normal operating conditions resume.

Q III) Composability. Different liquid Web services should be composable to enable the definition of more complex services, and the composition should have at least the quality Q I. Also, service composition could also be used to adapt the service to exploit certain characteristics of the deployment infrastructure.

V. The Liquid Architectural Style

To guide the design of liquid services with the qualities identified in Section IV we have collected the following design principles and architectural constraints, which together make up the *liquid* architectural style. These are: fine-grained decomposition, underspecified communication, explicit interaction semantics and explicit composition rules.

C I) Fine-grained decomposition. The service should be composed of multiple, autonomous, fine-grained components. The grain of each component should be as fine as possible to give maximum flexibility in the way components can be deployed, dynamically replicated and migrated. Components encapsulate both the basic functionality provided by the service as well as its state. At a specific time instant, the overall state of the service is the union of the states of its components. Components may be stateless.

C II) Underspecified communication. Components should interact with other components by means of some communication mechanism. In order to provide the highest possible flexibility at deployment time we minimize the constraints we make on the actual connector through which component cooperate to deliver the service's functionality. Our assumption conforms with the choice of an asynchronous message-based *coordination* mechanism used in most service-oriented architectures. Still, we do not constrain nor make any assumptions about the *communication* mechanism that will be used at runtime to carry messages between components. In other words, the *connector* used to link the components within a service should be left underspecified so that once the architecture is deployed the most appropriate communication mechanism available at runtime can be used.

C III) Explicit and well defined interaction semantics. Another important constraint is that messages exchanged between components should explicitly expose the effect of the interaction on the state of the components. In other words, each message exchanged between components should indicate whether or not accepting it would change the state of the component processing it. This implies that every interaction among components of the same service should happen using a fixed set of interfaces, and for every interface the effect on the state should be defined. This leads to the definition of a fixed set of uniform interfaces that every component should eventually implement. Not only we can distinguish messages carrying read-only requests, but – since each component carries its own private, share-nothing state – at every moment in time it is possible to identify service state changes and isolate which components they affect. For instance, components that read data from other components should indicate in their request message that the action would not have an impact on the receiver's state. Instead, components requiring others to change state should point out that their message will cause a state transition. This constraint of explicitly labeling the safety properties of an interaction is important to ensure the scalability of stateful Web services by means of replication [20].

C IV) Explicit composition rules. To guarantee C III also for composite services (Q III), another constraint has to be introduced: it should also be specified how components can be composed recursively out of other component interfaces, and the result should not violate the C III constraint. In other words, for every incoming message with a given interaction semantics, it has to be known which kind of outgoing messages can be exchanged with other components. This way a composite component can accept a message without violating its associated interaction semantics. For instance, a message to perform a read-only operation on a composed service, should not result in a change of state of one of the services called by its receiver.

VI. REactors: RESTful Actors

In this Section we will follow the design principles and constraints identified in the prior section to introduce a novel architectural pattern, called REactor (RESTful Actor).

Method	Property	Recursive Composition
Create	$\emptyset$	Create, Read, Update, Delete
Update	Idempotent	Update, Read
Delete	Idempotent	Delete, Read
Read	Idempotent, Safe	Read

Table I: Recursive Composition for Each CRUD Method

Whereas there are many possible designs that deliver the qualities of liquid services, this pattern codifies a simple and minimal approach to achieve the same qualities while following the previously enumerated constraints.

A. REactor Pattern

1) A RESTful Web service is designed as the composition of one or more REactors. A REactor is an autonomic software component characterized by a private state, which is made accessible to other REactors through a uniform interface. REactors are globally addressable through unique identifiers which collectively form the address space of a service. To comply with the fine-grained decomposition constraint (C I) it is important to provide a fine-grained addressing mechanism to identify REactors and their state (which could be empty). REactors found within RESTful Web services may use URIs as identifiers and the resources published by a RESTful Web service may be associated with one or more REactor.

2) The uniform interface of a REactor allows it to react to incoming messages targeting specific resources managed by the REactor. When applying the pattern it is necessary to define the mapping between resources to REactors. The actual communication mechanism chosen to deliver and route the messages between REactors will be defined at runtime in compliance with the underspecified communication constraint (C II). This way, the operations performed by REactors can be designed by referring to resources and the runtime takes care of hiding the difference between resources local to the same REactor and remote ones.

3) Interactions between REactors can happen only by mean of state transitions. Each state transition is triggered by a message containing the *operation* required, and referring to the portion of state (i.e., the resource) that will be affected after the message has been processed. Like resources of the REST architectural style, REactors can be seen as passive consumers of incoming messages. As a refinement, however, it is also possible for REactors to generate further outgoing messages while processing an incoming message, permitting in this way REactors'composability (Q III).
We further refine the REactor pattern by including a set of possible methods associated with their uniform interface. These methods are explicitly mentioned in the messages exchanged by the REactors, and can be one of: Create, Read, Update and Delete. The choice of a CRUD-like uniform interface makes it easy to comply with the constraint requiring to explicitly label the interaction semantic (C III).

In fact, labeling all the possible operations with the CRUD uniform interface provides every state transition with specific algebraic properties. Read, Update and Delete methods are idempotent, which means that the same operation can be executed multiple times on the same resource producing the same result. Moreover, the Read operation is also a safe operation, which means that the operation is side-effects free. These three methods can thus be repeated an arbitrary number of times in the event of communication failures.

4) Liquid Web services are expressed as a composition of multiple REactors. This means that every REactor can communicate with others through the exchange of messages. Thus, we have to deal with the composability of such methods in order to respect the basic algebraic properties provided by each of them. The goal is to avoid that, as a consequence of an idempotent operation performed on a REactor, a non idempotent operation is performed on another REactor. Concerning the global state of the service, the first operation would become non-idempotent, violating the explicit interaction semantic constraint (C III).
We can solve this issue defining the recursive composition rules (C IV) for each of the CRUD operations shown in Table I. Given the set of CRUD methods, we identify a subset of valid methods to be called on other REactors during the execution of composite operations. This way, we can guarantee that every REactor will be compliant with the explicit interaction semantic (C III), no matter if the operation received is coming directly from the client, or from any other REactor. To give an example, consider the Update operation. A REactor receiving a message for the execution of such operation on its state will be enabled to update also other REactors, as well as to read their state. Still, it will not be allowed to create or delete any other external resources. This guarantees the composite Update operation to be idempotent.

B. REactor State Management

The REactors' state is organized as a set of *resources*. Any REactor can publish one or many resources. Any resource can be accessed and modified using its uniform interface (i.e., the CRUD methods). As a consequence, any possible message exchanged between REactors could result in a state-related operation, or in a pure functional (i.e., stateless) one. This data-oriented vision of REactor interactions simplifies the definition of interaction patterns among REactors, because there can be only four outcomes: 1) a resource present in a REactor can be accessed with a Read request on its URI. 2) A resource present in a Reactor can be updated just by sending a new value through an idempotent Update message to its URI. 3) A new resource can be added to a Reactor with a Create operation. 4) A resource can be removed with an idempotent Delete operation directed to the desired URI.

To understand the effect each operation has on the state of a REactor, coupled with the functional behavior required to implement the operation, we give the following classification. On the one hand, we distinguish whether operations only depend on the input provided by the message invoking them (stateless) or whether they make use of the state of the REactor (stateful). On the other hand, we distinguish whether operations are implemented with arbitrarily complex behavior ($f()$) which may – for example – involve conditional state changes or the invocation of other REactors, or simply give direct access to the underlying state through the CRUD uniform interface (I).

Resource	Method	Operation Type
/time	Read	logical
/calendar/2011/feb/7/	Read	physical
/calendar?date=2011/feb/14	Create	hybrid

Table II: Sample Calendar RESTful Web service

State:		$f()$	I
	Stateful	hybrid	physical
	Stateless	logical	$\emptyset$

- logical operations expose a pure functional (i.e., stateless) behavior.
- physical operations give direct access to the state of a resource through the CRUD uniform interface.
- hybrid operations publish complex functionality which could make use of the state of the resource.

Since physical operations give a direct access to the state of resources, they can be seen as a special case of hybrid operations, where the arbitrary functionality ($f()$) is replaced by an *identity* function (I).

To give an example, consider the simple Web service described in Table II. The service implements a basic calendar API, exposing two resources accessible through three methods, for a total of three operations. The behavior of the service is the following. By reading the /time resource, the service responds with the current time. This operation is performed on a logical resource, since /time does not require to save data to complete the operation. By reading the /calendar resource with a /2011/feb/07/ argument, the service responds with a piece of data (for instance a JSON object) listing all the appointments scheduled for the given date. This operation is performed on a physical resource, since the data is directly available in the REactor's state and no additional functional behavior needs to be applied in order to retrieve it. Finally, by creating new data in the /calendar resource, a new appointment is added, if the date exists (e.g., it is a valid date format). This is an hybrid operation, since it can be performed on the resource only if the date provided is correct. This means that additional behavior is needed to decide if the method will be allowed to modify the state of the resource.

C. REactor Architecture

The internal architecture of a REactor is depicted in Figure 1, where two REactors with their main components are shown.

1) The *Acceptor* component is responsible for receiving request messages and routing them according to the type of operation. Requests concerning physical operations are directly routed to the State Manager component. Requests dealing with logical operations are directly processed by the Functional Processor component. Finally, requests dealing with hybrid operations are first routed to the Functional Processor component, and then eventually directed to the State Manager component. Thanks to its routing role and the explicit interaction semantic of REactor's messages, this component can also cache the results of certain requests, for instance logical operations that are executed through a Read operation. Likewise, the caching mechanism could be implemented to speed up operations which read a physical resource, until it gets modified by an update operation, causing the invalidation of the cache.

2) The *Functional Processor* component is responsible for handling logical and hybrid operations, and executing the semantic of their methods applied to the corresponding resources. Operations could include arbitrary functional behavior, including the exchange of messages with other REactors. To this end, the functional processor component is also able to communicate with external REactors as it can use the resource addresses to locate the corresponding REactors. The component is stateless as it does not keep any private state. Instead it relies on the State Manager for managing the state of the resources affected by hybrid operations.

3) The *State Manager* component is responsible for handling physical and successful hybrid requests. The component features a persistent state manager, and is the only stateful component of the REactor. To keep the generality of the REactor pattern, we do not make any assumption about the persistence technology used to implement this component. In the simplest case, the state is managed with a simple key-value data store, which should also feature some kind of transactional guarantees. For example, we suggest to use multi-versioning concurrency control (MVCC [21]). This solution, based on "append-only" data structures (for instance, a B-tree) has the advantage that data representing a resource is never overwritten, but as changes are applied, newer versions of the same resource are just appended at the tail of the data structure. The append process is guaranteed to be atomic for concurrent modifications to the same resource's data, and the implicit versioning control made available by adding timestamps to the append mechanism enables complex data manipulations with transactional properties.

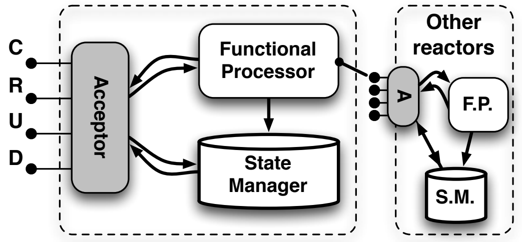


Figure 1: Reactor Pattern General Structure

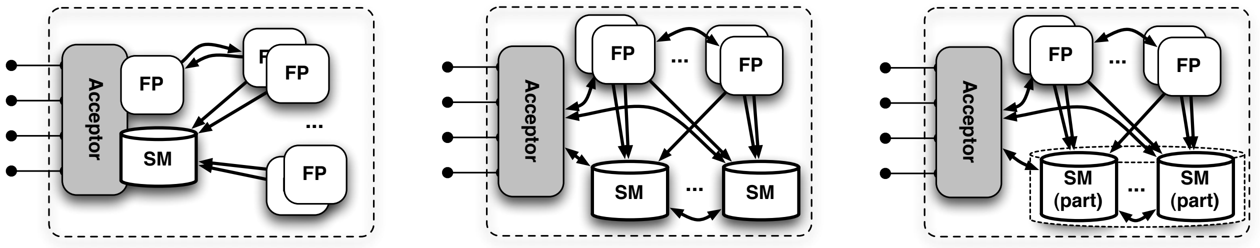
VII. Scaling REactors

Services designed according to the REactor pattern can scale following three different mechanisms. Each is based on the partial or full replication of some REactor's components, coupled with different degrees of guarantees on the consistency of the state owned by the REactor's State Manager component.

1) The *Scale f* configuration (Figure 2 a) enables the REactor to scale by replicating the Functional Processor component. In this configuration the Acceptor component becomes a load balancer which distributes messages to the Functional Processor replicas according to a given scheduling and load balancing strategy. As the State manager is not replicated, the state is guaranteed to be consistent.

2) The *Scale AP* configuration (Figure 2 b) enables the REactor to scale by replicating both the Functional Processor component and the State Manager. Each replica holds a full copy of the State manager component. However, copies are not guaranteed to be consistent. This has the effect that operations on the same resource performed by different replicas of the REactor could give different answers. The consistency across replicas is not guaranteed, however every replica attempts to synchronize its state by propagating changes to the others. This is made possible thanks to the MVCC approach to design the State manager component, which enables the automatic reconciliation of conflicts between replicas. Like for the Scale f case, in this configuration the Acceptor serves as a load balancer dispatching the incoming requests. Also, the component can track which REactor replica owns the most recent version of the state. This enables the Acceptor component to act as a coordinator for the state synchronization across replicas.

3) The *Scale CP* configuration (Figure 2 c) enables the REactor to scale by replicating both the Functional Processor component and the State manager. However, unlike the Scale AP mechanism, the state is not *fully* replicated. Instead, the state is partitioned across different replicas through *sharding* [20]. This guarantees that each replica will handle an unique version of a subset of the REactor's state, providing de facto the guarantee of consistency for that specific piece of data. However, this configuration also means that in case of a failure for a specific replica data availability cannot be guaranteed. In case of a failure, all hybrid and physical



(a) Scale f (b) Scale AP (c) Scale CP

Figure 2: Reactor Scalability Mechanisms

operations associated with a failed replica will fail until the replica becomes available again. In this configuration the Acceptor component becomes a router for client requests, as it is responsible for managing the sharding mechanism by sending all requests to the correct partition of the REactor state.

VIII. Discussion

REactors guarantee the qualities of liquid Web services by implementing the constraints identified for the liquid architectural style. The correlation between the qualities and constraints of the liquid architectural style and the design of the REactor is discussed in the following.

A. *Transparent Deployment*

Transparent deployment Q I is guaranteed by the REactor pattern (in any of the possible configurations) thanks to its compliance to constraints C I and C II.

First, fine-grained decomposition (C I) enables the runtime to deploy different parts of the service (i.e., different REactors) on different infrastructures, in a way which is unconstrained by the service design.

The second constraint, underspecified communication (C II), enables the runtime to perform the late binding of the best communication mechanisms available in the target deployment environment. Since every interaction between REactors is expressed in term of operations (that is, resources and methods of the uniform interface) invoked through a message-passing coordination primitive, nothing is specified on the type of communication to be used at runtime. Thus, the design of Web services remains unaware of the actual communication primitives that will be used. This enables the deployment of services transparently adapting their structure to the available communication infrastructure. The deployment of the REactors making up a Web service on different execution environments becomes just a problem of late-binding abstract message exchanges to concrete and compatible software connectors. In other words, the REactor pattern permits to postpone the decision about the communication mechanism to be used until run-time. This means that at any moment in time it is possible to automatically reconfigure which connectors have been used to link the REactors which make up a given service.

B. *Composability*

Composability (Q III) is guaranteed by the REactor pattern (in any of the possible configurations) thanks to its compliance to the constraint C III and also C IV.

The well defined and fixed CRUD recursive decomposition rules defined for the REactor pattern (Table I) guarantee that any operation requiring the interaction with composite resources (i.e., resources belonging to different REactors) will be handled with the right semantic, thus guaranteeing that at any time any composite REactor would behave as a non-composite one, respecting the explicit fixed interaction semantic defined by the constraint C III.

C. *Scalability*

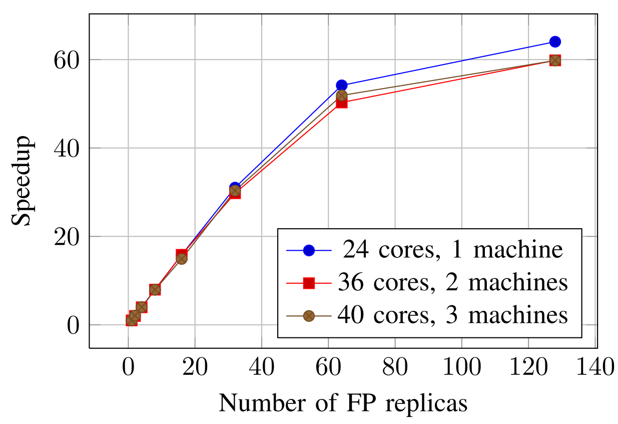
The previous two qualities (Q I and Q III) are guaranteed by the REactor pattern independently to the configuration of any of its internal components. Scalability (Q II) cannot be guaranteed by the REactor pattern in its basic centralized configuration, but can be achieved thanks to the scalable reconfiguration mechanisms described in Section III.

Scalability is guaranteed by constraints C I and C III. The first constraint, fine-grained decomposition, enables a REactor to be distributed across multiple independent components (the Acceptor, the State Manager and the Functional Processor). Having an explicit interaction semantic (C III) permits to replicate such components in the way described in Section III, taking advantage of caching for safe operations.

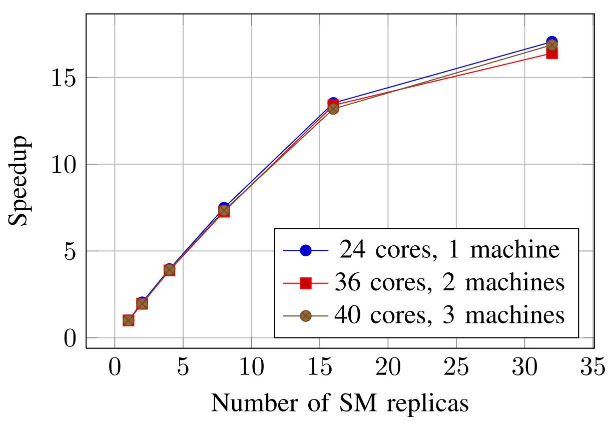
The combination of the two constraints permits to scale a REactor by replicating its internal components. The number

	T	S	Co	C	A	P
Centralized	✓	x	✓	✓	✓	x
Scale f	✓	f	✓	✓	✓	x
Scale AP	✓	✓	✓	x	✓	✓
Scale CP	✓	✓	✓	✓	x	✓

Table III: Properties Guaranteed per Scalability Mechanism. (T)ransparent deployment, (S)calability, (Co)mposability, (C)onsistency, (A)vailability and (P)artition tolerance.



(a) Scale f



(b) Scale AP/CP

Figure 3: Scalability Evaluation for the ScaleF and ScaleAP/CP Mechanisms

of replicas on any of the scalable configurations can grow as much as required since components do not share any information with one another. The possibility to replicate components coupled with their share-nothing design does not constrain the run-time infrastructure supporting the reactors as long as it can provide the necessary routing of messages from the acceptor to the replicas.

As a consequence of the replication-based scalability offered by the reactor pattern, a REactor based service has to deal with the consequences of the CAP theorem. The REactor pattern can be reconfigured in order to decide which property to respect.

This is summarized in Table III, where depending on the specific configurations of the REactor (as presented in Section VII) any combination of the CAP properties can be guaranteed.

The first Centralized configuration cannot scale beyond a certain limit. The Scale f configuration provides scalability only for stateless requests (indicated with f), while both Scale AP and Scale CP can be scaled thanks to the replication/partitioning of their state. Availability is provided by Scale f and Scale CP configurations, but cannot be guaranteed for the Scale AP case because of the partitioned state. Consistency is guaranteed by Scale f (where the state remains centralized) and Scale CP, but only eventual consistency can be guaranteed for Scale AP, thanks to MVCC-based replication. Finally, partition tolerance is guaranteed by both Scale AP and Scale CP, but cannot be guaranteed for Scale f, since the state is centralized.

D. *Scalability Evaluation*

The REactor pattern guarantees both transparent deployment and composability by design. To demonstrate how scalability is not affected by different deployment scenarios we have performed a preliminary evaluation of two REactor-based Web services. The results of the experiments are presented in Figure 3. In the evaluation, we have developed two distinct web services, and we have deployed them on three different shared-memory and distributed-memory heterogeneous configurations. The services have been scaled according to the different scaling mechanisms presented in Section VII, measuring for any number of replicas of REactor's internal components the average throughput.

The first set of experiments (Fig. 3 a) has been performed on a stateless service, scaled according to the Scale f policy. Upon any request, the service performs a mathematical computation with fixed average response time (200ms). The service has been run with an increasing number of requests (i.e., an increasing number of concurrent clients), up to the scalability limit. The service has been deployed on three different infrastructures. First, the service has been deployed on a single shared-memory multicore machine (4 Nehalem processors CPUs, 6 cores each, with disabled SMT). Second, the same experiment has been performed deploying the REactor on two multicore machines (the prior one plus a 2 Opteron processors machine, 6 cores each, for a total 12 cores). Finally, the experiment has been repeated adding a third machine to the pool (a single Core2Quad processor, for a total of 4 cores). The chart shows clearly that the scalability of the REactor is not influenced by the three configurations. In fact, all three different scenarios have a similar scalability profile, and the scalability limit is nearly identical for the three cases.

To test stateful services (that will stress more the network than the CPUs), the second set of experiments has been performed on the same set of machines, scaling the service by replicating the State Manager component. Independently of the data management policy (Scale AP or Scale CP), the graph shows that also in this case linear scalability is guaranteed for the three deployment scenarios up to reaching the saturation point.

IX. Related Work

The term "Reactor" has been previously used in [22], where it was used to present a programming model inspired by Datalog [23] and the Actor model. The model described in the paper shares with our work the general intent, but our pattern is different from their programming model. The term "liquid architecture" has been previously used in [24], where it was adopted to describe a class of purely decentralized p2p architectures. In this paper we give a different definition focusing on service-oriented architectures. We first presented the vision of Liquid Web services in [25]. This paper significantly expands upon the initial position statement as it gives a detailed description of the Liquid architectural style constraints and introduces the REactor pattern. The design principles behind the REactor pattern have been mainly inspired by the Actor concurrency model [26] and the REST architectural style [27].

The Actor model is a concurrency model based on autonomous share-nothing entities, called Actors, cooperating concurrently to the achievement of a common goal through message passing. The main advantage of the Actor concurrency model is that it hides complex concurrency management mechanisms (such as locks) from the programmer, resulting in a clean and effective instrument to develop parallel and concurrent applications. The Actor model has been widely adopted and is natively supported in many programming languages like Scala [28] and Erlang [29]. Also, a notable number of frameworks and libraries are promoting its adoption in other languages [30]–[32]. The Actor model has been designed to deal with very general concurrency problems, and does not have a specific notion of how each actor's state might change. Introducing the notion of state, and the explicit interaction semantic constraint, we enable REactors to be used as a basic building block for liquid Web services. There have been many other extensions proposed to modify and tailor the actor model to specific domains. Examples are the Self Replicating Objects model [33] and the Active Object pattern [34].

Representational State Transfer (REST) is the architectural style for large-scale distributed hypermedia systems, such as the Web. The architectural style is based on four main constraints [27]: *Resource identification through URI*, *Uniform Interfaces*, *Self descriptive messages* and *Stateless interactions*. The constraints identifying the liquid architectural style defined in Section V have been deeply influenced by REST. However, the main distinction between REactors and RESTful Web services is that a REactor can also directly interact with other REactors through their uniform interface. The explicit composability and the presence of an explicit computational element within REactors is another difference which further distinguishes REactors from ordinary RESTful Web services. Thus, REactors could be seen as a more refined class of RESTful Web services. REactors also share the notion of computation as a first-class element with the CREST architectural style [35].

The idea of considering Clouds and multicores as a single computing environment has been introduced by David Wentzlaff et al. within the contest of the Fos Operating System [36]. 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 service-oriented architecture. Other examples of distributed Operating Systems that address similar scenarios are Amoeba [37], Sprite [38] and Clouds [39].

X. Conclusion

In this paper we have identified the main qualities for a novel class of Web services, called liquid Web services, capable of transparent deployability, composability, and scalability. Liquid Web services are of great importance for modern service oriented architectures as they permit to decouple the design of a Web service from the runtime infrastructure on which it will be deployed. To develop Liquid Web services we have introduced a set of architectural constraints (fine-grained decomposition, underspecified communication, explicit interaction semantics and explicit interface composition rules) used in the definition of the liquid architectural style. As guidance to apply the constraints we have also provided a novel architectural pattern, called REactor (RESTful Actor) and we have shown how services developed according to the REactor pattern can guarantee such liquid qualities over heterogeneous execution environments (i.e., Clouds and multicores). We have begun to apply the REactor model to the development of a new service-oriented programming model. This will be complemented by the development of a smart runtime supporting the execution of liquid Web services on IaaS Clouds and multicore machines.

Acknowledgment

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

References

[1] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J. Léon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing Scalable, Highly Available Storage for Interactive Services," in *Proc. CIDR*, 2011.

[2] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, *Web Services: Concepts, Architectures, Applications*. Springer, 2004.

[3] A. Fox, S. Gribble, Y. Chawathe, E. Brewer, and P. Gauthier, "Cluster-based scalable network services," *ACM SIGOPS Operating Systems Review*, vol. 31, no. 5, pp. 78–91, 1997.

[4] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, January 2003.

[5] S. Dustdar, C. Dorn, F. Li, L. Baresi, G. Cabri, C. Pautasso, and F. Zambonelli, "A roadmap towards sustainable self-aware service systems," in *Proc. of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, 2010, pp. 10–19.

[6] C. Pautasso, T. Heinis, and G. Alonso, “Autonomic resource provisioning for software business processes,” *Information and Software Technology*, vol. 49, no. 1, pp. 65–80, 2007.

[7] D. Bonetta, A. Peternier, C. Pautasso, and W. Binder, “A multicore-aware runtime architecture for scalable service composition,” in *APSCC*, 2010, pp. 83–90.

[8] G. F. Pfister, *In search of clusters*. Prentice-Hall, 1998.

[9] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, “A view of cloud computing,” *Commun. ACM*, vol. 53, pp. 50–58, April 2010.

[10] R. Moreno-Vozmediano, R. Montero, and I. Llorente, “Elastic management of cluster-based services in the cloud,” in *Proc. of the 1st workshop on Automated control for datacenters and clouds*, 2009, pp. 19–24.

[11] D. Patterson, “The trouble with multi-core,” *Spectrum, IEEE*, vol. 47, no. 7, pp. 28–32, 2010.

[12] R. Taylor, N. Medvidovic, and P. Oreizy, “Architectural styles for runtime software adaptation,” in *Proc. WICSA/ECSA 2009*, 2009, pp. 171–180.

[13] E. Luke, “Defining and measuring scalability,” in *Scalable Parallel Libraries Conference, 1993., Proc. of the*. IEEE, 2002, pp. 183–186.

[14] M. Michael, J. E. Moreira, D. Shiloach, and R. W. Wisniewski, “Scale-up x scale-out: A case study using nutch/lucene,” *International Parallel and Distributed Processing Symposium*, vol. 0, p. 441, 2007.

[15] R. Iyer, “Performance implications of chipset caches in web servers,” in *Performance Analysis of Systems and Software, 2003. ISPASS. 2003 IEEE International Symposium on*. IEEE, 2003, pp. 176–185.

[16] R. Kalla, B. Sinharoy, W. Starke, and M. Floyd, “POWER7TM: IBM’s Next Generation Server Processor,” *IEEE Micro*, 2010.

[17] A. Datta, K. Dutta, H. Thomas, and D. VanderMeer, “World Wide Wait: a study of Internet scalability and cache-based approaches to alleviate it,” *Management Science*, vol. 49, no. 10, pp. 1425–1444, 2003.

[18] S. Gilbert and N. Lynch, “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services,” *ACM SIGACT News*, vol. 33, no. 2, p. 59, 2002.

[19] W. Vogels, “Web services at amazon.com,” in *IEEE SCC*, 2006.

[20] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory and Practice*. Addison-Wesley, 2007.

[21] P. A. Bernstein and N. Goodman, “Concurrency control in distributed database systems,” *ACM Comput. Surv.*, vol. 13, pp. 185–221, June 1981.

[22] J. Field, M.-C. Marinescu, and C. Stefansen, “Reactors: A data-oriented synchronous/asynchronous programming model for distributed applications,” *Theor. Comput. Sci.*, vol. 410, pp. 168–201, February 2009.

[23] J. D. Ullman, *Principles of database and knowledge-base systems, Vol. I*. New York, NY, USA: Computer Science Press, Inc., 1988.

[24] C. Bayrak and C. Davis, “The liquid architecture: a non-linear peer-to-peer distributed architecture with polymorphic message passing,” *SIGSOFT Softw. Eng. Notes*, vol. 28, p. 2, May 2003.

[25] D. Bonetta and C. Pautasso, “Towards liquid service oriented architectures,” *WWW2011 Doctoral Symposium*, 2011.

[26] G. Agha, “Actors: a model of concurrent computation in distributed systems,” MIT, Tech. Rep. AITR-844, 1985.

[27] R. Fielding, “Architectural styles and the design of network-based software architectures,” Ph.D. dissertation, University of California, Irvine, 2000.

[28] 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.

[29] R. Virding, C. Wikström, and M. Williams, *Concurrent programming in ERLANG (2nd ed.)*, J. Armstrong, Ed. Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1996.

[30] R. K. Karmani, A. Shali, and G. Agha, “Actor frameworks for the jvm platform: a comparative analysis,” in *Proc. of the 7th International Conference on Principles and Practice of Programming in Java*, ser. PPPJ ’09, 2009, pp. 11–20.

[31] Haskell-actors: Actors for haskell. [Online]. Available: http://code.google.com/p/haskellactor/

[32] Akka: A java and scala framework with actors, stm and transactors. [Online]. Available: http://www.akka.io

[33] K. Ostrowski, C. Sakoda, and K. Birman, “Self-replicating objects for multicore platforms,” *ECOOP 2010–Object-Oriented Programming*, pp. 452–477, 2010.

[34] R. Lavender and D. Schmidt, “Active object: an object behavioral pattern for concurrent programming,” in *Pattern languages of program design 2*. Addison-Wesley Longman Publishing Co., Inc., 1996, pp. 483–499.

[35] J. Erenkrantz, M. Gorlick, G. Suryanarayana, and R. Taylor, “From representations to computations: the evolution of Web architectures,” in *Proc. ESEC/FSE 2007*, 2007, pp. 255–264.

[36] D. Wentzlaff and A. Agarwal, “Factored operating systems (fos): the case for a scalable operating system for multicores,” *ACM SIGOPS Operating Systems Review*, vol. 43, no. 2, pp. 76–85, 2009.

[37] S. Mullender, G. Van Rossum, A. Tananbaum, R. Van Renesse, and H. Van Staveren, “Amoeba: A distributed operating system for the 1990s,” *Computer*, vol. 23, no. 5, pp. 44–53, 2002.

[38] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch, “The sprite network operating system,” *Computer*, vol. 21, no. 2, pp. 23–36, 2005.

[39] P. Dasgupta, R. LeBlanc Jr, M. Ahamad, and U. Ramachandran, “The clouds distributed operating system,” *Computer*, vol. 24, no. 11, pp. 34–44, 2002.