Information Sciences 264 (2014) 349–370

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Scalable service-oriented replication with flexible consistency guarantee in the cloud

Tao Chen *, Rami Bahsoon, Abdel-Rahman H. Tawil

School of Computer Science, University of Birmingham, Edgbaston, B15 2TT Birmingham, UK
School of Architecture, Computing & Engineering, University of East London, E16 2RD London, UK

A R T I C L E   I N F O

Article history:
Available online 28 November 2013

Keywords:
Replication
Consistency
Scalability
Cloud
Service oriented
Distributed application

A B S T R A C T

Replication techniques are widely applied in and for cloud to improve scalability and availability. In such context, the well-understood problem is how to guarantee consistency amongst different replicas and govern the trade-off between consistency and scalability requirements. Such requirements are often related to specific services and can vary considerably in the cloud. However, a major drawback of existing service-oriented replication approaches is that they only allow either restricted consistency or none at all. Consequently, service-oriented systems based on such replication techniques may violate consistency requirements or not scale well. In this paper, we present a Scalable Service Oriented Replication (SSOR) solution, a middleware that is capable of satisfying applications' consistency requirements when replicating cloud-based services. We introduce new formalism for describing services in service-oriented replication. We propose the notion of consistency regions and relevant service oriented requirements policies, by which trading between consistency and scalability requirements can be handled within regions. We solve the associated sub-problem of atomic broadcasting by introducing a Multi-fixed Sequencers Protocol (MSP), which is a requirements aware variation of the traditional fixed sequencer approach. We also present a Region-based Election Protocol (REP) that elastically balances the workload amongst sequencers. Finally, we experimentally evaluate our approach under different loads, to show that the proposed approach achieves better scalability with more flexible consistency constraints when compared with the state-of-the-art replication technique.
© 2013 Elsevier Inc. All rights reserved.

1. Introduction

Replication is a well-known technique applied by cloud service providers to enables scalable and highly available services. Traditionally, replication techniques aims to work with the underlying objects and data, recently however, service-oriented replication attracts plenty of interests for cloud computing providers. How to guarantee consistency among different replicas is a challenging issue for both replication paradigms. This is because consistency guarantee is not free of charge, at times the price paid for maintaining consistency is the lack of systems scalability, thus it is normally acceptable to relax certain consistency requirements for scalability as long as this does not compromise the integrity of the systems. Service Oriented Architecture (SOA) principles enable software engineers to efficiently develop applications in the cloud that allows for different services to be added either as standalone or in orchestration with other services. Therefore in most cases requirements for consistency and scalability varies depending on the services needed. Nevertheless, existing service replication approaches and middlewares [16,25,31,36,39,41,45,48] force engineers to choose between a rather restricted consistency

* Corresponding author at: School of Computer Science, University of Birmingham, Edgbaston, B15 2TT Birmingham, UK.
E-mail address: txc919@cs.bham.ac.uk (T. Chen).

0020-0255/$ - see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.ins.2013.11.024

or none at all. The inability for governing the trade-off between consistency and scalability may result for cloud systems to violate certain consistency requirements and not to scale well. Such inflexibility is mainly due to consistency enforcement protocols (e.g., atomic broadcast protocol) adopted by existing cloud replication approaches and middleware are strongly lay in communication level, which cannot satisfy application level consistency semantics. In addition, the on-demand property of cloud necessitates the need to efficiently add and remove physical nodes on the fly, while in traditional approaches this is researched regardless to the business requirements. As a result, the consensus processes during nodes change unnecessarily cause impacts to all application services.

In this paper, we present service-oriented, requirements aware approach and architecture that can be used to adapt with different consistency models at runtime based on predefined requirements. Our main contributions are:

- Firstly, we observe that there is a general lack of replication strategies specifically suited for service-oriented systems. Though traditional replication techniques tend to use data-oriented replication, we argue that such use is neither optimal nor straightforward in replicating cloud-based services. Therefore, we call for a novel approach for applying data-oriented replication techniques to the case of cloud-based and service-oriented replication. We introduce and formalize new notions for describing services. These are Atomic Services (AS), Composite Services (CS) and Redundant Services (RS) for service-oriented systems. The objective of such formalism is to address the fundamental differences between Service-Oriented Replication (SOR) and Data-Oriented Replication (DOR) in relation to the levels of granularity, which is the most significant distinctness between SOR and DOR.
- Secondly, we introduce and formalize the notion of "consistency regions". A consistency region is a logical domain, which uses consistency models, that autonomically (to great extent) manages the consistency requirements for a group of service(s) within the domain. We propose three distinct types of consistency regions. These are Conflict Regions – CR (i.e. services within this region are said to have conflicting requirements for consistency regardless of the session); Sessional Conflict Regions – SCR (i.e. a region including services of a particular session with conflicting consistency requirements); Non-conflict Region – NR (i.e. a region that does not impose any consistency constraints or requirements). Such separation has proved to be effective in isolating services, which are functionally independent with mutually exclusive consistency requirements. Our approach to regioning has proved to promote scalability and efficiency in managing consistency. Unlike traditional approaches, which depend on single and static consistency models (e.g., [31,45]), our approach proposes new regions based policies, which operates within regions for switching amongst various consistency models. We report on the design of two protocols: these are the Multi-fixed Sequencer Protocol (MSP) and the Region-based Election Protocol (REP). MSP aims at realizing the policies and guaranteeing the satisfaction of the consistency models in a region; it reduces the affect of single sequencer's bottleneck issues and eliminates the problem of a single point for failures. REP extends the election protocol to operate on regions. It elects new sequencers upon presence of failure and distributes the loads to multiple sequencers.
- Thirdly, we propose a layered middleware architecture to realize our contribution. The middleware flexibly manage replication for service-oriented and cloud-based systems; it includes a requirement-aware layer, which handles and expresses consistency requirements to lower level protocols benefiting from the consistency regions described above. We propose a solution for improving the reliability of the proposed middleware. The notion of regions enables for efficient tolerance and has limited the impacts caused by crashes under MSP and REP. We have experimentally evaluated our work using three replica nodes with varying loads, concurrent requests, and varying consistency requirements. We report on the response time when the consistency model changes from the most restricted mode to the most relaxed one. The results show that our proposal promotes flexibility of consistency and scalability because of the introduced notions of regions. From experiments, we also observed that as complexity increases composite and redundant services provide better response time because of partial replication (as when compared to no-regioning). Overall, our results prove that our approach allows for handling the trade-offs between consistency and scalability.

The resulting middleware, named Scalable Service Oriented Replication (SSOR) allows for engineers to easily specify service wised consistency and scalability requirements when dealing with cloud-based service systems and their replication. To limit the scope of this research, we assume that existing services' requirements rarely change at run-time. Such assumption is acceptable in numbers of cloud-based services. We also do not consider recovery properties in this paper.

The rest of this paper is organized as follows: we present the motivated scenario in Section 2; Section 3 identifies differences between service-oriented replication and data replication. In Section 4, we presents the notion of regions and relevant policies. The proposed SSOR with the two protocols MSP and REP are specified in Section 5. Section 6 discusses fault tolerance solutions in SSOR and Section 7 presents implementation details. The experiment results, evaluations of the effectiveness and a case study are demonstrated in Section 8. Finally related work, discussion and conclusion are presented in Sections 9, 10 and 11.

2. Motivation

This paper is motivated by the following scenario: assume a cloud provider offers book shops management services that allow online trading between books sellers and buyers. They initially provide service only for purchasing books and relevant

management in a cloud environment with the replication of services for scalability. Suppose the business does not require restricted consistency. For instance, strong consistency is only required for modifying books' details and adding new books service, whereas relaxed consistency constraints are preferable for adding book and deleting books services. Therefore in such case, the service provider may willingly accept to relax consistency and to trade it with enhanced scalability. Nevertheless, traditional approaches often assume a fixed consistency model, which forces service providers to accept and suffer from the latency caused by unnecessary consistency constraint.

Afterwards, as the business grows up, the bookshops management services may decide to extend the business by providing extra services, such as an electronic book downloading service. Also, they may be keen to provide specialized services with better performance for VIP buyers. However, traditional service replication approaches are not able to accommodate such requirements since the more services are integrated into the cloud, the higher latency are produced by the restricted consistency that originally provided. In this perspective, scalability quickly becomes an issue. To this end, the provider would have to choose either to violate consistency or apply heterogeneous replication approaches for certain services.

These aforementioned concerns lead to highly and timely demand for a solution that is capable to govern the trade-off between scalability and consistency of cloud-based services.

3. Service oriented replication versus data oriented replication

We start off by presenting the preliminary concepts. In this section, we differentiate data from service replication, and identify our notion of service-oriented replication. From the literature, there are two major approaches to achieve replication: either through replicating the service invocation or the underlying data items. The later one is also known as Data-Oriented Replication (DOR), while the former one is referred to as Service-Oriented Replication (SOR). In this work, we focus on the SOR as it derives from a service oriented perspective.

Service Oriented Architecture (SOA) and its applications in the cloud [40,49] have been widely researched over the last decade, but SOR is still at its infancy. Broadly, four major replication models are widely applied in the literature, these are: passive replication [8], active replication [38], semi-active replication [35] and Read-One-Write-All (ROWA) [7]. These models are all designed for DOR but yet there are not any specific models for SOR. However, Osrael et al. [30] conclude that traditional replication models in DOR are adoptable by SOR when considering the following differences: (1) Transaction model: DOR is compatible with traditional ACID transactions, while SOR can additionally be used for long running transactions. As argued by [30], transaction components are optional for replication, therefore distributed transaction protocols such as two phase and three phase commit protocols can be used for replication in a hot plunging-in fashion. This juncture does not have any impact on replication logic but mainly on the implementation details of transaction protocols. (2) Technology standard: SOR adapts web service whereas DOR may uses Common Object Request Brokers (CORBA), especially in the context of grid computing [9]. Differential standards do not impact the logic of replication but differs on the implementation of relevant middleware. For instance the well-known service replication middleware WS-Replication [36] is strongly bound to SOAP properties. (3) Granularity: Typically, SOR has a coarse-grained granularity whereas DOR allows for fine-grained granularity. Such property impacts the application of consistency models, for instance SOR may maintain unnecessary consistency between data items that should be processed concurrently as they are handled in the same service. On the other hand fine-grained granularity forces the system to maintain replicas for each data item and consumes much more resources as it is relatively easy to reach terabytes of data in the cloud. A coarse granularity also causes Redundant Nested Invocation (RNI) [32] issue in SOR. This issue occurs when the operations are non-idempotent or contains processes that return non-deterministic results (e.g., random number or local time), therefore a naive replication of invocation would result in distinct values on different replicas, which leads to inconsistency. In this paper, we mainly consider granularity differences and demonstrate techniques required to apply a DOR replication model to SOR.

To handle requirements from a service-oriented perspective and solve RNI issues, we distinguish services amongst three categories: *atomic services*, *composite services* and *redundant services*. Services that do not belong to any of these categories are not taken into account by the replication logic. Without lose of generality, the proposed notions are generic and not bound to any specific technology standard (e.g., web services).

***Atomic Service* (*AS*)** is a stand-alone service that either does not interact with other services or the interactions can be hidden. In principle, arbitrary services can be considered as ASs. The content of an atomic service does not need to be known by the invocation logic since full service redundancy is achieved. An AS can be either stateful or stateless.

***Composite Service* (*CS*)** achieves partial service redundancy for both stateful or stateless services whose content needs to be known. A CS consists of numbers of ASs, and instead of replicating the CS, we only make inclusive ASs replicable. Therefore the level of redundancy of a CS depends the number of ASs it is composed from. A CS can involve other CSs.

CSs need to fulfil a lazy replication, as the arguments of inclusive services are unknown until they are executed. This does not affect the replication logic for centralized replication models (e.g., passive replication). However, in active replication model, requests are replicated only after completion of the required service.

A service can be identified as ***Redundant Service* (*RS*)** if it is a stateless service and involves RNI issue. RS also applies lazy replication; hence the invocation of RSs is only allowed to occur on one replica after the returned values are replicated to other replicas. There is a many-to-many relationship between RS and another two services. More precisely, an AS or a CS may contain one or more RSs and an RS may associate with multiple ASs or CSs.

4. Trade-off between consistency and scalability

4.1. Consistency models

Consistency is a well-understood issue in any replication strategy. In this section, we review various consistency models and their realization techniques from the literature. Generally, consistency maintenance can be expensive in terms of scalability. A relaxed consistency usually achieves better scalability, but the price is that the state of each node may not be always the same; hence there is a non-trivial trade-off between scalability and consistency. Consistency models are referred to as the contracts between process and data for ensuring correctness of the system. Although these models are traditionally applied in terms of data consistency, recent research proposed to show that they could be well adopted in service-oriented environments [30]. In such context, a service typically controls a set of data and possibly consists of many other services, henceforth the granularity of data consistency model is subject to change form fine-grained to coarse-grained when applied in the context of SOR. In other words, the consistency restriction of a target changes from data to service. In this perspective, it is obvious that when the services satisfy certain consistency models, the consistency of effected data can be enforced as well. To achieve a formal definition of consistency models, we classify consistency model in terms of their levels of constraints.

Linerizability [24] is the most restricted model. A system is said to be linerizable if the following criteria are met [12]:

- **LC1**: *The interleaved sequence of operations meets the specification of a correct copy for an object or a service.*
- **LC2**: *The order of interleaving operations is consistent with the times at which the operations occurred in the actual execution.*

LC1 highly depends on the application domain. Due to the needs for real-time property, passive replication is the only replication model that satisfies LC2 [12]. We additionally identify a *Strong Sequential Consistency* model that is slightly weaker than linearizability. The properties for this model are reformed as follow:

- **SSC1**: *The same as in LC1.*
- **SSC2**: *The order of interleaving operations is consistent with the order in which each replica is executed.*

The model achieves similar consistency to linearizability in terms of ordering. However it does not require the real-time property, hence such model can be achieved using an active replication model. An even weaker form of consistency model is called *Casual Consistency* [19]. The criteria for casual consistency are shown as below:

- **CC1**: *The same as LC1.*
- **CC2**: *The order of interleaving operations is consistent with the causal order in which each replica is executed.*

This model works on data/services that are casually related. That is, within an interleaving series of operations, only the pair of processes that has happened-before relation needs to be consistent. This model can be realized by an arbitrary replication model since it only captures partial view of delivery order.

The weakest data centric model is *Sequential Consistency* model [23]. A system is said to be sequentially consistent if the following constraints are satisfied [12]:

- **SC1**: *The same as LC1.*
- **SC2**: *The order of interleaving operations is consistent with the program order in which each client is executed.*

Both active and passive replication can achieve this model. Note that if a system satisfies linearizability or sequential consistency, then it also satisfies sequential consistency but the converse is not always true.

All aforementioned consistency models are guaranteed by various ordering techniques. A First In First Out (FIFO) order ensures that messages from a sender are always delivered in the same order as they were sent; casual ordering defines a happened-before relationship for any pair of messages; while total order constraints the same order of events occurring on different replicas. Generally, there are two form of protocols by which consistency can be guaranteed: (1) two-phase protocol [16,39,41] where the first phase runs unordered but with reliable broadcast, out-of-order operations are rolled back when found. (2) Apply atomic broadcast (abcast) protocol over reliable communication [15,17,31,36,45], which prevents inconsistent ordering from happening. In this work, we are specially interested on the later category. Although various ordering techniques and protocols have been proposed to guarantee consistency models, it has been shown that they suffer from various limitations [10] since they mainly rely on the communication level. Such limitation is corollary according to the well-known end-to-end argument [46] which states that lower level facilities cannot ensure higher level semantics, it can at most optimize them. The details of various *abcast* protocols are discussed further in Section 5.2.

4.2. The notions of consistency region

We define consistency region as a logical unit to represent application state-level requirements for consistency and scalability. A service group may consist of many regions, by service group, we refer to a component or application that consists

of a number of services. We argue that certain consistency is unnecessary depending on different business purposes thus can be loosened for better scalability. The proposed region is used to define consistency boundaries that handles which group of services need to be ordered. Each region contains one or more services and every service that needs to maintain consistency is required to be associated with a region. Note that within SOR, ordered message delivery implies serializability property, in other words, the execution of an entire service is also ordered atomically. Specifically, the notion of regions is formalized as below:

Conflict Region (CR). The data centric conflict region is the most restricted region as it orders the delivery of all requests to any inclusive services. Services from different CRs do not need to maintain consistency and thus the unneeded consistency constraint can be dropped. This region is useful to isolate services requiring different consistency constraints, when strong consistency constraint is required and certain consistency amongst services is not need. However it may still be rather expensive due to the natural property of a coarser-grained granularity in SOR. Therefore the concept of **Concurrent Delivery Service (CDS)** is additionally introduced to represent causal relationships within a conflicting region, that is, the ability to concurrently delivery messages without ordering based on services' needs. For instance, assume a series of messages $m1$, $m2$ and $m3$ which should be originally delivered as $m1 \rightarrow m2 \rightarrow m3$. However if $m2$ and $m3$ can be delivered concurrently such as $m1 \rightarrow m2||m3$ then the waiting time of subsequent messages can be reduced, and thus a full concurrency ($m1||m2||m3$) becomes feasible. The difference between the notions of region and concurrent delivery is that if only $m1$ and $m3$ are allowed to be delivered concurrently, they are still delivered in their original order since $m1$ and $m3$ are transitively ordered by $m2$. Whereas transitive ordering would never occur for services from different conflict regions. However, transitive ordering causes the issue of non-deterministic concurrent events. We therefore assume that applications can accept such behaviors if concurrent delivery is allowed. Recall from the service categories we defined in Section 2, only AS may relate to CR since CS and RS are non-replicable.

Sessional Conflict Region (SCR). A sessional conflict region is client centric as each session is related to a particular client, and is a relaxed form of the CR. There is only one sessional conflict region in each service group, but requests to the inclusive services are delivered orderly in sessional basis and such services do not need consistency maintenance with those from other conflicting regions. Sessional conflict region can also apply concurrent delivery due to the same reason for conflict region. Similar to CR, only AS can be included in SCR.

Non-conflict Region (NR). Non-conflict region has no consistency constraint and are replicated directly without triggering the atomic broadcast process. NR is useful for services that allow for consistency violation.

In principle, excluding any read-only services from all of the three regions achieves the ROWA model. It is obvious that service requests are conflicting if the services are within the same CR or SCR when those requests are carried on the same session. We refer to such conflict region as **Atomic Region.**

To sum up, as illustrated in Fig. 1, each region belongs to at most one group, and each node may contain multiple service groups. There is a one-to-many relationship between a region and an AS, and regions from different service groups do not related to each others. A group can reside either in one cloud or cross multiple clouds. Systems administrators can set requirements for the classification of services (e.g., what category a service belongs to), in which regions these atomic services reside and which services can concurrently be delivered. This is feasible because service level requirements tend to be well known for systems administrators. The trading between consistency and scalability for a service could be realized by changing the region it belongs to. More concretely, consistency of a service can be governed by: moving it from a CR with more/less number of services to a CR with less/more services, changing its region category (e.g., from CR to SCR/NR) or allowing more/less services that it can be concurrently delivered with. By doing so, we are able to control the potential number of

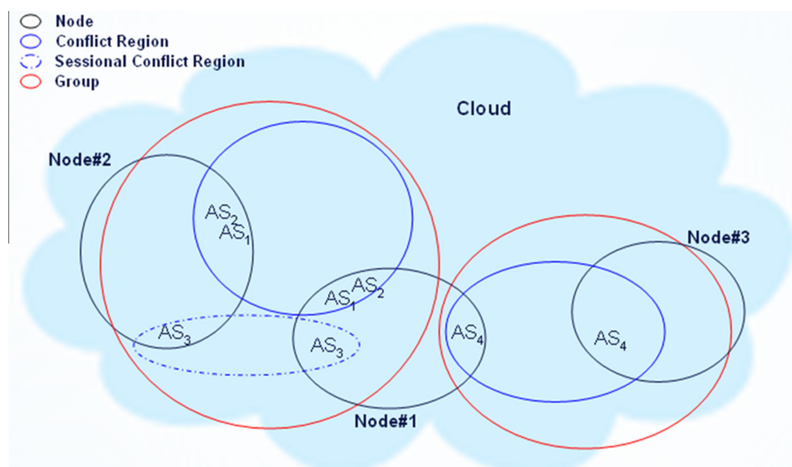


Fig. 1. High level diagram shows the relationships of regions, services, groups and cloud.

service requests that should be delivered in sequence. This is in turn allows for the control of trade-offs between consistency and scalability. In the next section, we explicitly specify the policies that enforce the above points.

4.3. Region based polices

Once the category of services and the regions they belong to are defined by the system administrator, our policies can then be used to coordinate corresponding reactions at the communication level protocols (e.g., abcast). To this end, we formalize region policies with respect to their ordering relationship and to the notion of services identified in the previous sections. Within the context of a service group, we denote R_i for the ith regions that could be conflict, sessional-conflict or non-conflict. AS_i represents the ith atomic services and SES_i are denoted for the ith sessions. CR represents all conflict regions whereas CR_i denotes the ith conflict regions, let SCR be the sessional conflict region and NR be the non-conflict region. In such context, CR_i, SCR and NR may contains a number of services. CDS is denoted as any pair of services that allows concurrent delivery, in particular, (AS_e,AS_f) represents messages for AS_f, which are concurrently deliverable with those for AS_e. To better explain the ordering of two arbitrary messages, let M_i represents a request message for AS_i with an associated region R_i, M_j represents the subsequent request message for AS_j within region R_j. M_i and M_j may be associated with SES_i and SES_j. ($i,j \in N$), (such sessions may not exist if the request is not sessional). Note that upon delivery, a request for CS is decomposed as multiple requests for the inclusive ASs of the said CS. Table 1 summarize the defined notations.

The representation of consistency requirements is said to be valid if the following policy hold:

$$\textbf{Policy 1}: \quad CR_1 \cap CR_2 \ldots CR_n \cap SCR \cap NR = \varnothing (n \in N) \tag{1}$$

The above policy implies that each AS is allowed to be associated with only one region. Once policy 1 is satisfied, for arbitrary request message M_i, the following subsequent policies determine the associated replication logic:

$$\textbf{Policy 2}: \quad AS_i \text{ is replicated with atomic broadcast protocol if } R_i \in CR. \tag{2}$$

$$\textbf{Policy 3}: \quad AS_i \text{ is replicated with atomic broadcast protocol if } R_i = SCR. \tag{3}$$

$$\textbf{Policy 4}: \quad AS_i \text{ is replicated without atomic broadcast protocol if } R_i \in NR. \tag{4}$$

Upon arrival of M_i and M_j on replicas, the following policies are used to determine whether they should be delivered in consecutive order:

$$\textbf{Policy 5}: \quad M_i \textit{ and } M_j \text{ are delivered in consecutive order if } R_i = R_j \ \& \ R_i, R_j \in CR \ \& \ (AS_i, AS_j) \notin CDS \tag{5}$$

$$\textbf{Policy 6}: \quad M_i \textit{ and } M_j \text{ are delivered in consecutive order if } R_i = R_j = SCR \ \& \ SES_i = SES_j \ \& \ (AS_i, AS_j) \notin CDS \tag{6}$$

Upon delivery, our abcast protocol (specify in Section 5.3) examines these policies and determines whether M_i and M_j are delivered in consecutive order or in parallel. To reason about the correctness of these policies, we systematically prove how regions and policies can be used to satisfy consistency models, ranging from weaker to stronger ones. In doing so, we are able to trade consistency for better scalability and vice versus. The previously defined notations are used in the following discussion.

Table 1
Summary of notations.

R_i	ith regions including conflict, sessional-conflict or non-conflict
AS_i	ith atomic services
SES_i	ith sessions
CR_i	ith conflict regions
CR	All conflict regions
SCR	Sessional conflict region
NR	Non-conflict region
(AS_e, AS_f)	Messages for AS_f are concurrently deliverable with those for AS_e
M_i	ith message
M_j	The subsequent message after the ith message

Lemma 1. *If a passive replication model is applied, then for* $\forall CR_i \in CR$*, the consistency of* $\forall AS_j \in CR_i$ *is linearizable* $(i,j \in N)$.

Proof. Because the fact that LC1 is mainly application dependent, thus we assume LC1 can be satisfied. The real-time property is hold once the passive replication is applied, hence the only criteria left is to prove the delivery orders seen by different replica are consistent. It is obvious that LC2 can be satisfied according to Policy 5, which implies the execution of services in CR_i is both atomic and ordered. □

Lemma 2. *For* $\forall CR_i \in CR$*, the consistency of* $\forall AS_j \in CR_i$ *is strong sequential* $(i, j \in N)$.

Proof. Suppose that arbitrary replication models are applied and SSC1 can be satisfied. For satisfying SCC2, each replica should see the same delivery order of message. Such criteria can be satisfied according to Policy 5, which guarantees that the execution of services in CR_i is atomic and ordered. □

Note that Lemmas 1 and 2 do not hold amongst those services that satisfy $(AS_k, AS_j) \in \mathrm{CDS}$ where AS_j, $AS_k \in CR_i$ and $(k,j \in N)$.

Lemma 3. *Consistency of* $\forall AS_j \in SCR$ *is sequential* $(j \in N)$.

Proof. Suppose that arbitrary replication models are applied and SC1 can be satisfied. For satisfying SC2, the services are only required to be consistent with their programs order in which they are executed by the client. From Policy 6, it follows that the delivery of services in SCR is atomic and ordered in terms of the same session, and a session is bounded to each particular client. As a result, SC2 is satisfied. However, the sequential consistency does not be hold amongst those services that satisfies $(AS_k, AS_j) \in CDS$ where AS_j, $AS_k \in SCR$ and $(k,j \in N)$. □

Lemma 4. *If* $\exists (AS_k, AS_j) \in CR_i$ *and they are CDS then the consistency of* AS_j, AS_k *and* $\forall AS_t \in CR_i$ *is casual* $(i, j, k, t \in N, j, k \neq t)$.

Proof. Suppose arbitrary replication models and CC1 can be satisfied. Policy 5 implies that delivery of AS_j and AS_k are not ordered or ordered transitively whereas delivery between AS_j or AS_k, while $\forall AS_t \in CR_i$ are atomic and ordered on all replicas. This fact also indicate that the other services maintain a constant happened-before relationship to $\mathrm{AS_j}$ and $\mathrm{AS_k}$ as required by the application, therefore $\forall AS_t \in CR_i$ and AS_j or AS_k are said to be casually related, thus CC2 can be satisfied. The achieved casual consistency is within the context of global linearizability or strong sequential consistency. □

Lemma 5. *If* $\exists (AS_k, AS_j) \in SCR$ *and they are CDS, then the consistency of* AS_j, AS_k *and* $\forall AS_t \in SCR$ *is casual in terms of the same session* $(j, k, t \in N, j, k \neq t)$.

Proof. Lemma 5 can be proved based on Policy 6, and the proof is very similar to that of Lemma 4. However, Lemma 5 lies in the same session, henceforth the achieved casual consistency is in the context of sequential consistency. □

At this stage, we have shown how our proposed notions of service, region and policies can be used to satisfy different consistency models. Note that if $\forall AS_j \in CR_i/SCR$ is not a read-only service, then the whole system actually achieves the ROWA variation of replication model.

5. Scalable service oriented replication

5.1. SSOR architecture in the cloud

In this section, we specify the SSOR architecture and how the proposed notion can benefit the underlying protocols. Cloud layers can be classified into 3 categories: Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS) and Infrastructure-as-a-Service (IaaS). SaaS provides fully working applications for users; PaaS enables management of developing and deploying applications on cloud environment; whereas IaaS applies visualization techniques to elastically manage a physical and virtual computing infrastructure. The proposed SSOR aims at the IaaS front as our framework assumes full control over the Virtual Machine (VM) and physical nodes. The architecture of SSOR is presented in Fig. 2, to improve flexibility, we have adopted a layered-based approach. Upon requests for a service, the middleware interrupts the workflow of service and examines whether replication/*abcast* is needed, after which the process would trigger the corresponding protocol manager that directly takes control of the protocol stack. Finally, the adaptor is invoked and pushes messages down to the virtual machines through the underlying Group Communication Service (GCS). Upon receiving a message, the processes are taken in the reversed manner.

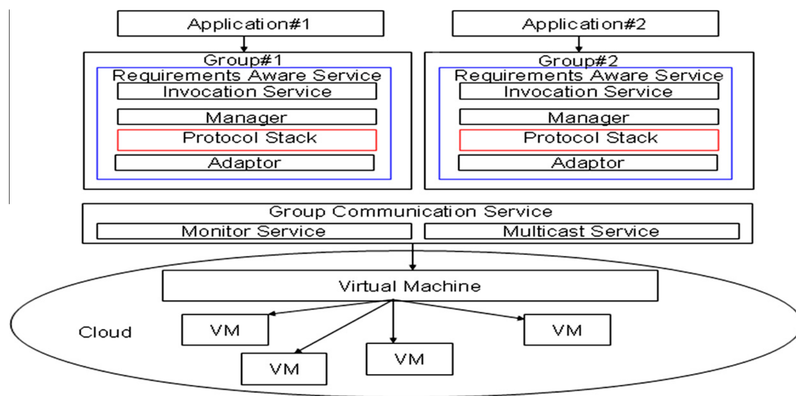


Fig. 2. High level diagram of SSOR architecture.

In our SSOR architecture, the *requirements aware service* component parses the predefined category of services and identifies to which region they belong to as defined by the administrator. This layer of service can be accessed throughout the entire replication process, in doing so, it is possible to serve low-level protocols with high-level application requirements. *Requirements aware services* are computed locally, thus they do not consume bandwidth but require identical setup on replicas. *Invocation service* can be used to obtain arguments, injecting additional data or invoking replicable services. Implementations of this service normally relies on proxies and Aspect Oriented Programming (AOP) techniques. Such service is particularly important as it is responsible for capturing the program order of RS and the services inside CS. Managers in the *manager* component are designed to coordinating actions for protocols, in particular, each protocol is attached with a dedicated manager. Based on the concept of micro-stack, the *protocol stack* maintains a batch of hierarchical protocols that participate during replication such as abcast and fault tolerance protocol. Transaction is optional in terms of replication hence it is not discussed here, however transactional protocols (e.g., two phase commit protocol) can be easily added to the stack and gain benefits from the requirements aware services. Both the monitoring and multicast services are provided by the underlying *Group Communication Middleware* (GCM). Lastly, *adaptor* components work as an intermediate component connecting SSOR to different GCMs. The application of GCM is useful for fault tolerance in the presence of faulty replicas, since it provides for managing groups membership and view delivery services [8]. In addition, it can be notified through the virtual machine manager or broadcasting features of the cloud infrastructure when a physical node is added or removed. Various GCM like JGroup [20] adapts stack of micro-protocol, which allows configuring reliability and view property on demand. The joining and leaving of replicas may happen simultaneously, however, the order of these simultaneous events and their consistency in relation to the messages sent are maintained by view synchrony [8]. We will discuss this in more details with fault tolerance issues in Section 6. Note that GCS is not required for replication but the reliability property provided greatly simplifies the implementation.

5.2. Atomic broadcast for guarantee consistency

As mentioned, consistency is guaranteed via ordering techniques, which is realized as *abcast* protocols. Traditional communication level *abcast* protocols are widely adapted in replication. They can be classified as sequencer based and sender/destination based [14]. Sequencer-based protocol includes fixed and moving sequencer protocols. The fixed sequencer approach applies a single sequencer, and has 3 variations [14]: unicast–broadcast (UB), broadcast–broadcast (BB) and unicast–broadcast–broadcast (UUB). Although the protocol achieves linear latency [17], the single sequencer can easily become a bottleneck and causes high impact in the presence of failure. The moving sequencer protocol [14] distributes the sequencer load to multiple ones based on token passing. However, it offers worse latency than fixed sequencer and does not provide better throughput [17]. Another broad of distributed agreement protocols is so-called sender/destination based approach, which includes: privilege based, communication history and destination agreement. The privilege based protocol allow only one node to send each time, thus it reduces throughput and performance. Communication history requires a logical clock and ensures order by learning history from each nodes, but it has bad performance since it relies on quadratic number of messages to be exchanged. Destination agreement protocols have a variety of variations, but it has relatively bad outputs in terms of latency and message complexity. Hybrid protocols are also exists [17], a complete survey of those protocols can be found in [14]. The aforementioned protocols are limited in flexible consistency supports, as they are all communication level and not aware of consistency and scalability requirements. To cope with this issue, we propose a novel protocol based on the traditional fixed sequencer protocol since it outperforms other protocols and obtains linear latency. The application of service category, region and policies allow for the proposed protocol to overcome drawbacks from using a single sequencer, and flexibly adapt predefined consistency models. Details are explained in the next section.

5.3. Multi-fixed sequencers protocol

Our Multi-fixed Sequencers Protocol (MSP) is a requirement aware variation of the fixed sequencer protocol. The novel aspect is that the traditional single fixed sequencer is physically split into different sequencers based on the fact that each atomic region requires a distinct sequencer. However MSP is different from the moving sequencer protocol, as it still relies on a fixed sequencer and thus prevents the extra delay caused by token passing. To reduce the load on the sequencer and achieve better performance, we extend MSP from the UUB fixed sequencer protocol.

As shown in Fig. 3, MSP requires three round trips for each service request to complete. The assignment of sequence and message delivery (execute service) are examined within each atomic region only (as define in policies 5–6), therefore synchronization is reduced. In particular, MSP obtains three major advantages over the traditional fixed sequencer approach: (1) The state of the application can be known at a lower level. (2) It reduces the impact of a bottleneck issues caused by single sequencer. (3) There is less impact to systems in the presence of sequencer failure since different sequencers can be tolerated simultaneously.

The concept of CDS within a conflict region is realized by a special mechanism: assume that within the context of the same atomic region, a message is in the form of (M, *seqno*, *concurrentno*), where M is the message content whereas *seqno* and *concurrentno* are referred to as one sequence. *Seqno* is used as the primary consecutive number of sequential messages whereas *concurrentno* represents consecutive number of concurrent messages. In particular, such mechanism consists of two phases, the first phase is termed **Assignment Conditions** on a sequencer. Its process can be demonstrated via the following scenario: initially a message obtains a sequence $(i,\emptyset)$ *or* (i,k) $(i,k \in N)$ for a service, and if the subsequent total number of j $(j \in N)$ messages that are concurrently deliverable with any previous requests, then they will be assigned sequences $(i,-1)_1$, $(i,-1)_2,\ldots,(i,-1)_j$. Eventually the sequencer would receive a sequential request, which cannot be concurrently delivered with one of the previous requested services, then the assigned sequence would be $(i+1,j)$. The process of assigning sequences is atomic and consecutive since they occur on the same sequencer of a CR.

The second phase named **Delivery Conditions** is executed on each replicas. More precisely, its process is shown as follow: Upon arrival of message (M, *seqno*, *concurrentno*) on a replica, the message would be delivered in consecutive order. *Concurrentno* = -1 indicates this message can be delivered concurrently. Whereas if *concurrentno* = $\emptyset$ or *concurrentno* = j $(j \in N)$ then the delivery should be sequential. In addition, a message with *concurrentno* = j implies that it cannot be delivered until the total number of j concurrent messages have been delivered. Therefore on each node, the delivery order is guaranteed to be $(i,\emptyset) \rightarrow (i,-1)_1 \| (i,-1)_2 \|,\ldots,(i,-1)_j \rightarrow (i+1,j)$. To understand detailed stages of the protocol, MSP can be summarized in four phases:

Request. To determine whether the subsequent phases are needed, MSP examines policies 2–4 in this stage. Requests for RS can broadcast immediately, whereas requests for AS and CS require to trigger the subsequent phase. Similar to RS, requests for AS within NR are replicated immediately.

Coordination. This phase is completed on the sequencers. Upon receiving request for a sequence, the sequencer identifies the region that associate with the requested service. If an incoming request is for AS, the sequencer replies according to the assignment conditions. However if the request is for a CS, then such request is decomposed and each inclusive ASs within the CS is assigned a sequence from their region, after which the sequences are replied back to the sender.

Acquire sequence. This phase is triggered when the sender acquires sequence form the sequencer. The sender broadcasts messages with the sequence to other nodes immediately if the target AS does not contain RS. Otherwise it executes the service according to delivery conditions and records the returning results of inclusive RSs in their program order and finally, broadcasts to the others upon execution is completed. If the replied sequence is for CS, the sender executes the CS as a whole and delivers each inclusive service based on delivery conditions after all the sequences of inclusive ASs are received. The arguments of inclusive ASs and any RSs contained should be recorded in their program order. Eventually, the sender informs other replicas to execute the inclusive ASs with those arguments.

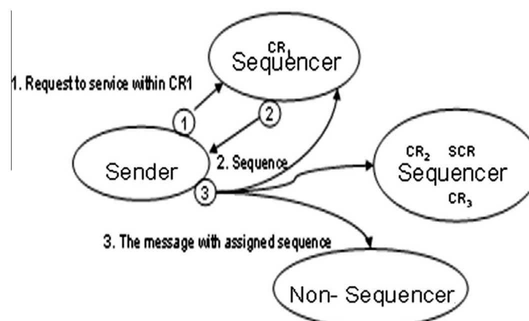


Fig. 3. Multi-fixed Sequencer Protocol (MSP).

Agreement. When replica nodes receive message from a sender, it delivers the message based on delivery conditions. If RS is involved, then there is need to extract the corresponding returned values contained in the message. Note that composite service is not executed as a whole, but only the inclusive ASs are replicated in their program order. Messages for services belonging to NR can be delivered immediately upon arrival.

MSP can be adapted with passive replication which rather resembles to active replication with fix sequencer protocol.

5.4. Region based election protocol

In this section, we present a simple but powerful Region based Election Protocol (REP) for distributing the regions of sequencers. Region distribution is required when new nodes join or leave as these cases could result in the change of sequencers. This is because there is often the case that some machines are faster and more reliable than others [48], therefore the administrator may configure powerful nodes for larger regions. For the regions of which a node wants to become the sequencer, we refer to as the interests of the said node. A region is distributed according to two conditions: (1) A CR or SCR will be assigned to the oldest node that is interested in it. (2) Equally distribute those CRs that no one is interested into all the other nodes. Therefore for each region, a node that is interested on it and currently responsible for fewer regions would have higher chance to become a sequencer for the said region. As shown in Fig. 4, where REP consists of three phases:

- **Join**. The new joining node broadcasts its interests after it is initialized. Upon receiving interests from the new node, an existing node records these interests after which it replies with its own interests and the regions that it holds. If the existing node is a sequencer, it also replies with a decision indicating whether it would move its region to the new node, based on the aforementioned conditions.
- **Record**. Upon collecting replies from all nodes, the new joining node adjusts new region distribution accordingly and broadcasts its regions if it becomes a new sequencer of any regions.
- **Agreement**. In case of region distribution change, each node (including the new joining node itself) assigns the regions sequencer to the new node and removes the region from its original sequencer.

To prevent inconsistent global view of regions' distribution, all incoming requests for MSP are suspended until the election process is completed on the new joining node. Upon completion of the *Agreement* phase, for each atomic region, the new joining node also needs to determine the initial *expected_seqno* and *expected_concurrentno* (*expected_seqno* and *expected_concurrentno* indicates if a given sequence is the next one that should be delivered) based upon the smallest sequences of all suspended messages. Thanks for the help of view synchrony, which guarantees that all nodes receive messages in the same views. As a result, reliability of this process can be ensured. In case of no messages are suspended, the *expected_seqno* and *expected_concurrentno* are obtained directly through state transfer [8] as such case implies that no requests sent from the end clients before its REP protocol complete. The new sequencer can obtain the latest correct state (*seqno*, *concurrentno*) through fault tolerance protocol, which will be discussed in Section 6. REP only requires local election when a sequencer crashes as all nodes maintain common view of regions distribution; hence such mechanism efficiently improves elastically in the cloud.

6. Fault tolerance in SSOR

The end-to-end fault tolerance between end clients and cloud services has been provided through replication. Thus we are more concerned about the reliability of SSOR on the occurrence of node failure. In particular, we mainly aim at discussing crash failure in this work. Consistency is maintained by ordering, which relies on consecutive sequences, therefore faulty nodes may cause termination issue such that the MSP waiting for sequences that would never arrive, we call those sequences as *never-arriving sequences*. For instance, a crashed non-sequencer may send a request for a sequence and be successfully assigned a sequence. Afterwards, the sender could crashes before it triggers the last broadcast and consequently, other nodes

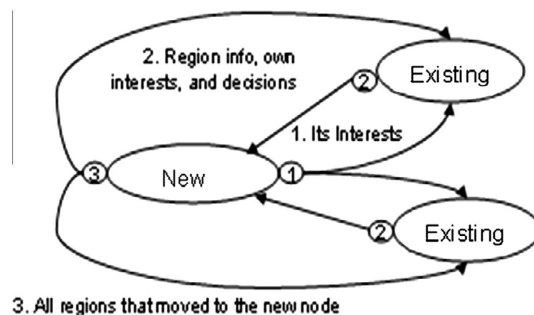


Fig. 4. Multi-fixed Sequencer Protocol (MSP).

are in the undetermined state in which they are waiting for a sequence that will never be received. We adopt the notion of view synchrony provided by GCS and demonstrate how it can be used to solve fault tolerance issue under different circumstances.

6.1. View synchrony

Thanks to the all-or-nothing property provided by the underlying reliable communication and View Synchrony (VS) [37], the tolerance processes can be simplified. More formally, the following guarantees of VS are provided by the underlying GCS.

- **VS1**: If a correct message multicasts message m then it will eventually deliver m.
- **VS2**: A correct node p delivers a message m at most once.
- **VS3**: If a process p delivers message m in view V, then $p \in V$.
- **VS4**: If a process delivers view V_1 then V_2, then no other processes delivery V_2 before V_1.
- **VS5**: If a process sends message m in view V_1, and another process delivery m in view V_2, then $V_1 = V_2$.

VS1 and VS2 are essential properties provided by reliable communication and the rests are ensured by view synchrony. However, traditional view synchrony does not work on unicast and it does not ensure serializability after delivery. Such properties are non-trivial for fault tolerance as the order of tolerance tasks and view delivery cannot be ensured when multi-threading is involved. Therefore we identify two extra properties, which ensure serializability and work for unicast:

- **VS6**: If a process delivery message m, then the sender of m belongs to current view
- **VS7**: The processes that triggered by message delivery are serializable with view delivery.

VS6 ensures that no received messages from crashed node are delivered even for unicasted messages. VS7 ensures serializability of the processes from VS4 and VS5, in other words, it guarantees that all nodes sees the operations followed by message and view delivery in the same order, even in a multi-threading environment.

6.2. Tolerate non-sequencers crash

Tolerating crash on non-sequencers is simple since in such case the sequencer would be notified by GCS with information regarding the crashed nodes, such information can be used to determine the never-arriving sequences. Non-sequencer crash can be handled as: on each sequencer, any requests that successfully obtain a sequence are memorized till the beginning of the *Agreement* phase in MSP. Such memory is useful because when failure occurs, the existence of memorized sequences requested by the crash node implies that those sequences would never be triggered by *Agreement* phase in MSP, therefore the sequencer is able to tell other nodes exactly which are never-arriving sequences. This solution is guaranteed to perform correctly as long as the underlying view synchrony properties and reliability can be ensured. We further discuss the non-sequencers crash tolerance by examining every possible circumstance in MSP:

- *Circumstance 1*

The sender may crash after it sends sequence requests and before the sequencer reply by unicast. According to VS6, it follows that no requests for sequence from the crashed node can be delivered after the new view has been installed. From VS7, it is ensured that view delivery is suspended until the sequence is assigned. If a request for a sequence from the crashed node arrives right before the view installation, the assigned sequence can be memorized by the sequencer who then notifies other node about it. VS3 ensures such notification is only delivered on a correct node. According to VS5, if the message arrives after the view installation, then such message would not been seen by the sequencer.

- *Circumstance 2*

A sender may crash after it sends its final broadcast in *Agreement* phase, which is triggered by multicasting; VS5 ensures that no agreement message is delivered after view is delivered which implies that the corresponding assigned sequence are memorized by the sequencer. If the agreement message arrives on the sequencer right before the view installation, VS7 guarantees that view delivery is suspended until the memory of corresponding assigned sequence is removed from cache and the agreement phase is completed. Such enforcement ensures that no sequences that have been received for agreement are mistakenly seen as missing ones. VS3 ensures that it is delivered in a correct node. According to VS5, if the message arrives after view installation, then such message would not be seen by the sequencer.

Eventually, each correct node knows the sequences that assigned to the crashed nodes, which would never be received. To avoid deadlock, whenever a message delivery is failed on the delivery conditions within an atomic region, the node then examines if the excepted sequence is missing. If this is the case, *expected_seqno* or *expected_concurrentno* would be increased immediately.

6.3. Tolerate sequencers crash

Crash tolerance on sequencers is relatively complex and expensive. Similar to non-sequencers crash, sequencers crash could cause nodes to await for a never-arriving sequence. The approach proposed by [18] tolerates crash on fixed sequencer through making redundant copies of these sequencers, but this requires extra consistency constraint among those copies. [5] propose a decentralized technique for solving the termination issue caused by consecutive sequences following a crash. SSOR applies a variation of centralized approach based on the notion of region, which reduces the impact of tolerance. More precisely, each node memorizes the requests from clients till the required service is completed. Upon the occurrence of a sequencer crash, firstly REP helps to elect a newer sequencer locally; each node then propose its sequences which are assigned by the crashed sequencer as well as their *expected_seqno* and *expected_concurrentno* to the new sequencer. Finally, the new sequencer decides *seqno* and *concurenntno* according to the largest among *expected_seqno* and *expected_concurrentno* of assigned sequences. The new sequencer can then inform other nodes of never-arriving sequences during the consecutive serial of sequences. At the last stage, retransmission is triggered for those memorized requests that have not yet been assigned a sequence. View synchrony ensures the consistency between view and message delivery amongst nodes. Note that associated processes after message delivery are serializable with each other, including the delivery of views. Because services are divided into regions, hence the processes for electing a new sequencer can be performed without impacting all the services in the cloud. Such improvement can benefits to the cloud electricity since it is often the case that nodes are shutdown for better resource utilization.

One integral problem when tolerating sequencers crash is how to identify never-arriving sequences. Thanks to view synchrony, missing sequences between the largest *expected_seqno/expected_concurrentno* pair and the largest assigned sequence would be those assigned sequences that never being seen by the requester. Therefore to prevent waiting forever. any sequences that are smaller than the largest *expected_seqno/expected_concurrentno* pair would be discarded. Note that during the step of releasing cached sequences, a series of concurrent deliverable sequences are modelled as a whole. That is, none of the concurrent sequences are released until the subsequent non-concurrent sequences arrive. The reason for making this assumption is presented in Lemma 8. The following are correctness proofs of the idea to identify never-arriving sequences in the case of sequencers crash.

Lemma 6. *Under the assignment condition, for a series of sequences that starts with* $(i,\emptyset)$ *or* (i,k) *and, ends with* $(j,\emptyset)$ or $(j,-1)$, *the never-arriving sequences can be estimated as* $\{(n,\emptyset)|\ i+1 \leqslant n \leqslant j-i\}$ $(i,j,n,k \in N)$.

Proof. According to the assignment conditions in MSP, it is easy to know that *seqno* equals to the subsequent *seqno* -1, therefore the lemma is always true if the crashed sequencers and other nodes perform correctly. Note that the numbers of concurrent delivery (sequences with *concurrentno* as -1) may occur during the start and end of sequence, however they are not seen by any of the existing nodes hence they can be considered as never existed. □

Lemma 7. *Under the assignment condition, for a series of sequences that starts with* $(i,\emptyset)$ *or* (i,k), *end with* (j,t), *then the never-arriving sequences can be estimated as* $\{(n,\emptyset)|i+1 \leqslant n \leqslant j-i-1\}$ *followed by* $(j-i-1,-1)_1, (j-i-1,-1)_2,\ldots,(j-i-1,-1)_t$ $(i,j,k,n,t \in N)$.

Proof. this is very similar to Lemma 6, except it ends with (j,t) which expects a number of t concurrent deliverable sequences. This lemma is true according to the assignment conditions between concurrent sequences and sequential sequences, which imply that there are a total numbers of t never-arriving concurrent sequences apart from the never-arriving sequential sequences. □

Lemma 8. *Under the assignment condition, for a series of sequence that starts with a number of k concurrent sequences* $(i,-1)$ *and, end with* $(j,\emptyset)$ *or* $(j,-1)$, *the never-arriving sequences can be estimated as* $(i+1,k)$ *followed by* $\{(n,\emptyset)|i+1 \leqslant n \leqslant j-i\}$ $(i,j,n,k \in N)$.

Proof. According to the assignment conditions, it is easy to know that the subsequent sequential sequence is $(i+1,k)$. To ensure this lemma is true, the memorized concurrent sequences should only be removed if a subsequent sequential sequence is received. Consequently, k is the total number of concurrent sequences with a *seqno* of i. Once $(i+1,k)$ is identified the rest is essentially the same as Lemma 6. □

Lemma 9. *Under the assignment condition, for a series of sequence that starts with a number of k concurrent sequences* $(i,-1)$ *and, end with* (j,t), *the never-arriving sequences can be estimated as* $(i+1,k)$ *followed by* $\{(n,\emptyset)|i+1 \leqslant n \leqslant j-i\}$ *and* $(j-i-1,-1)_1, (j-i-1,-1)_2,\ldots,(j-i-1,-1)_t$ $(i,j,k,n,t \in N)$.

Proof. This is essentially a concatenation of both Lemmas 7 and 8 and this is true if they are hold. □

In the following we discuss further interesting circumstances when sequencers crash:

- *Circumstance 1*

The sequencers may crash after certain sequences have been replied. In such case, VS6 ensures that no assigned sequences from the original sequencer are delivered; VS7 guarantees that all assigned sequences that are received before the view installation is known by the sender node. Therefore all nodes are consistent and no sequences are falsely identified as never-arriving sequences. In addition, the retransmission for requesting sequences would not be redundant since all messages that are not marked as 'received' implies that they have not been received before retransmission.

- *Circumstance 2*

The sequencer and other non-sequencers may crash at the same time. For the crashed non-sequencers, VS5 ensures that no *Agreement* phase of MSP is delivered after a view is delivered; VS7 guarantees that if a request for *Agreement* phase of MSP from the crashed non-sequencers arrives right before the view installation, then view delivery is suspended until the corresponding assigned sequence is removed from cache and the execution of service is completed. As a result, all nodes are consistent and the never-arriving sequences requested by the crashed non-sequencers can be eventually discovered.

The above solution can be used with REP for acquiring the most recent state when there is a need to change the sequencer.

6.4. Discussion of reliability in SSOR

Realistic scenarios are far more complex, different events may occur interleaving with each others. For instance, nodes may join in when different sequencers crashes; or a new sequencer may crash even before the election protocol is completed. We then further discuss how those relevant issues can be resolved.

The REP and tolerance protocol for sequencer crash can perform correctly only if all the nodes share common knowledge regarding how the regions are distributed. However in case of nodes joining and leaving simultaneously, the distribution of regions may become non-deterministic since both of the two protocols require changing the sequencer. Therefore, we identify a **Region Distribution Synchrony** (This is not needed for tolerance of non-sequencer crash, since in such case no information regarding to the distribution of regions is required), based on the properties provided by view synchrony. Note that although each delivery of view implies a numbers of joining/leaving nodes, each run of REP and sequencer crash tolerance protocol can only work for a particular node. This can be achieved by running the protocol for each joining/leaving node according to their order in the delivered view, and GCS guarantees that every node sees a consistent order. Region Distribution Synchrony specifies rules for maintaining synchronization when reallocating region sequencers between protocols:

- **Rule 1**: *All nodes handle protocol for joining nodes in the same order.*

This is an essential rule to maintain a synchronized order guarantees that no competition occurs, since two or more simultaneous joining nodes may compete for the same region. This can be easily realized based on VS4.

- **Rule 2**: *All nodes handle protocol for sequencer crash in arbitrary order.*

Handling different sequencers crash simultaneously is feasible because of the fact that there is only one sequencer for each consistency region.

- **Rule 3**: *All nodes handle protocol for joining nodes and protocol for sequencer crash in the same order.*

This is needed because a joining node may be interested on a region that is currently handled by a tolerance protocol due to crash of the original sequencer. The REP protocol for joining nodes is only allowed to be run after the completion of the crash tolerance protocol. However the tolerance protocol only needs to wait until the *Agreement* phase of REP complete as a new sequencer has already been elected for certain regions by that stage. This rule can be easily realized based on VS4.

It is also possible that a crash occurs before a protocol completes. For example, the new joining nodes may crash before REP completes and a new elected sequencer may crash during the tolerance protocol. In the following we discuss how Region Distribution Synchrony can be used to resolve these situations.

A joining node crashes during REP can be known by the nodes who assigned regions to the joining node and they can simply switch the regions back without any broadcast, therefore such process does not need to follow region distribution synchrony. VS5 guarantees that the last *Agreement* phase of REP is consistent amongst nodes, that is, nodes either see the phase as not complete at all hence the case is identified as crash during election, or see it before the delivery of view regarding to the crash, in which case it is identified as sequencer crash and the normal sequencer tolerance approach would be used. A new elected sequencer crashes during tolerance of a crashed sequencer can be solved in a similar way: regions of

new sequencers are marked as 'pre-elected' before REP is completed. When a new sequencer crashes then other nodes elect the next proper node as the new sequencer.

REP and tolerance protocol itself also suffers termination issues. However a solution is relatively easy: during the protocol for a joining/leaving node, each node holds a record of the members for current view. Upon nodes crashing, a node that is under waiting state examines if the crashed node is the node that it is waiting for. If this is the case then such node is removed from the waiting list. A node can move onto the next action if the criteria for terminating a waiting stage have been satisfied. Correctness is ensured by VS3 and VS4. This process does not need to respect the rules of region distribution synchrony.

7. Implementation details

As mentioned in Section 5.1, the proposed SSOR consists of various modules. Each of those modules has been implemented from the ground up using Java. In particular, we have implemented four major protocols in the *protocol stack*: MSP, REP, fault tolerance and replication. The essential logic of each protocol is to handle data passed from the upper/lower protocol and then send the processed result to the lower/upper one. Additionally, those protocols are specified in an XML configuration file; henceforth, the SSOR can easily allow the adding or removing of protocols. Each protocol has a corresponding manager which is responsible for coordinating actions; for instance, our region-based policy is implemented in the *manager* component for the replication protocol. *Requirements-aware service* component of the architecture obtains and maintains information regarding consistency requirements of services, specified in an XML configuration file. Administrators and system designers can specify region and service definitions in the configuration file. Once the category of services and the regions, which they belongs to are known, the policies presented in Section 4.3 can be used to adaptively decide on the predefined consistency model and its corresponding actions during runtime. *Invocation service* component is implemented based on the AOP concept; more precisely, the invocations of AS or CS are interrupted and suspended by SSOR and they are suspended until the required processes are completed.

The *Group Communication Service* component plays an integral role for ensuring reliability of SSOR. We have chosen JGroup [20], as the underlying GCS provider. JGroup manages the message transmission of SSOR via UDP-based reliable multicast, which guarantees VS1 and VS2 as described in Section 6.1. Such reliable multicast relies on two-entropy gossip protocols, where the first round is an unreliable UDP multicast; the second round is a gossip message that is triggered periodically to ensure each node does receive the messages. Instead of using the JDK supported serialization approach for sending objects, we have implemented a lightweight streaming approach: communication among nodes relies on transmitting simple primitive and string values. Additionally, we predefine an index for each class where possible and send those indices instead of class names. Following such an approach, we prevent sending objects/functions signatures and thus largely reduce the size of each message. The membership service provided by JGroup ensures VS3–VS5; in particular, the underlying GCS notifies our fault tolerance protocol when a faulty node or a new node is discovered. For better reliability, we realized our Region Distribution Synchrony at the top of GCS's membership service. In addition, we have modified JGroup's view synchrony protocol to include unicast (for ensuring VS6) and have implemented an additional thread synchronization mechanism to guarantee serializability between view delivery and message delivery (for ensuring VS7) on a node.

8. Evaluation

In this section, we first demonstrate how SSOR can be adopted by cloud providers through a case study. We also statistically evaluate our approach based on results obtained from experiments. More precisely, our evaluation is carried out through comparing scalability of SSOR with the traditional approach that applies fixed sequencer protocol that only allows for fixed consistency constraint. In particular, our experiments are carried out under different loads and consistency requirements. We have chosen to compare this protocol because the literature [14,19] indicates that it outmatches others in terms of performance. However the results are comparable to other traditional solutions that apply passive replication since they are rather similar. The active replication model is selected in all experiments since it is relatively easy to implement.

As the objective of our approach is to handle the trade-off between scalability and consistency, therefore it is sufficient to examine flexibility of our approach under fixed number of physical machines. The overheads as well as impact of lazy replication are also evaluated. To show how our approach benefits for the elasticity of cloud, we also demonstrate the performance of our approach when replicas are added/removed dynamically.

8.1. Use case

Let us consider the case of a provider named Cloudplus, in which consumer's demands for consistency can vary from weak to strong. By using our approach, consumers can promote better scalability via specifying the level of consistency required. SSOR is designed to assists each application replica, thus there is a distinct SSOR instance for each replica of a service-based application. More precisely, upon deployment of an application, the SSOR engine instantiates a new SSOR instance, which includes all the components mentioned in Section 5.1 on the same VM of the application. The consumers only need to provide their consistency requirements of region and the service definitions in an XML document. This information is then handled by SSOR. Note that SSOR would not trigger any replication if the number of replica equals to one. To achieve interruption of services, SSOR needs to be instantiated within the same application server (e.g., Tomcat) as the application.

Consequently, it would be able to act as a proxy for those services via AOP techniques. To avoid unnecessary communication, the provider may assign a unique group ID for each set of replicas of an application, thus only the nodes in the group with the same ID are able to send messages to each others. SSOR does not require any extra configurations on the underlying resource management module since the GCS detects new/crashed replica by listening to the activity of nodes within the same group. In particular, when a new replica is needed, the SSOR instance and the XML document are replicated along with the application. To conclude, it is clear that SSOR can be used as a 'black-box' middleware, which is independent from the above service-based application and the underlying resource management modules.

As businesses grows up, a consumer may wish to change his/her consistency requirements or s/he is willing to improve scalability by switching to a more relaxed consistency model, or s/he has developed new services to consolidate the application. In such cases, the only amendment required is to change the regions and service definitions in the XML file, SSOR would synchronize such file across all the replicas and then trigger redeployment.

8.2. Experiments setup

Our experiments were simulated on a mini cloud, which consists of three physical machines with a capacity of 1.85 GHz CPU and 1 GB memory in a LAN, numbers of virtual machines are running on each of the physical machine. The testbed is a service-based application that is being replicated across all three nodes, such application consists of three atomic services for performing account activity, named *createUser*, *login* and *removeUser*. We denote those services as service 1, 2 and 3 respectively. Each of the atomic services has the computational complexity of 245 ms and a simulated single response time of around 365 ms per request per service. One extra service named *updateUserActivity* is designed as composite service.

8.3. Evaluating performance, scalability and consistency

We first evaluate the performance of our SSOR with regards to consistency. The experiments are carried out on 3 fixed replicas, with equivalent numbers of concurrent requests (3–39) sent to each of the three services, and the average time taken for each service are recorded. Table 2 illustrates the setup parameters for experiments a to e. We compare the achieved responsiveness and the performance when no replicas are used, we then calculate the overhead caused by consistency constraints. We first assume that the three services require strong sequential consistency, for example the application has restricted rules that the activity of a user account should be consistent, hence all the three services can be put into one conflict region. Fig. 5a shows the performance of traditional UB-based fixed sequencer protocol. While Fig. 5b shows the performance of SSOR with one conflict region, therefore essentially it realizes UUB variation of fixed sequencer protocol. In comparison, the traditional approach has a maximum of 80.8% average overhead for the three services, while SSOR suffer a maximum of 75% overhead. They are very close since they actually realize similar approaches.

Suppose that the application allows for non-deterministic behaviors between the services *removeUser* and *login*, then another experiment can be carried out with the same setup, but *removeUser* and *login* are allowed to be delivered concurrently to improve scalability. From Fig. 5c, it is clear that the performance is relatively better and achieves a maximum overhead of 46.2%. This is obviously a better performance since the requests for *removeUser* and *login* are not restrictedly ordered. However, this experiment is non-deterministic since whether the system scales is strongly dependent on the number of consecutive request for *removeUser* and *login* services. Assuming that there is no need to maintain consistency amongst the three services and thus they can be assigned into 3 distinct conflict regions. As the results showing in Fig. 5d, dropping unnecessary consistency by defining consistency boundaries provide a significantly better performance and only 24.2% maximum overhead. In addition, the performance of each service is relatively better since they are not related to each other.

Table 2
Summary of setup for experiments a–e.

Experiment	Regions	Involved services	Conflict occurrence conditions	Achieved consistency model
a	N/a	CreateUser, login and removeUser	Any requests to the three services	Linerizable
b	Conflict region 1	CreateUser, login and removeUser	Any requests to the three services	Strong sequential
c	Conflict region 1, with 2 concurrent deliverable services of login and removeUser	CreateUser, login and removeUser	Requests between createUser and createUser, createUser and login, createUser and removeUser	Causal
d	Conflict region 1	CreateUser	Requests between createUser and createUser	Strong sequential
	Conflict region 2	Login	Requests between login and login	Strong sequential
	Conflict region 3	RemoveUser	Requests between removeUser and removeUser,	Strong Sequential
e	Sessional conflict region	CreateUser, login and removeUser	Any requests to the three services from the same user	Sequential

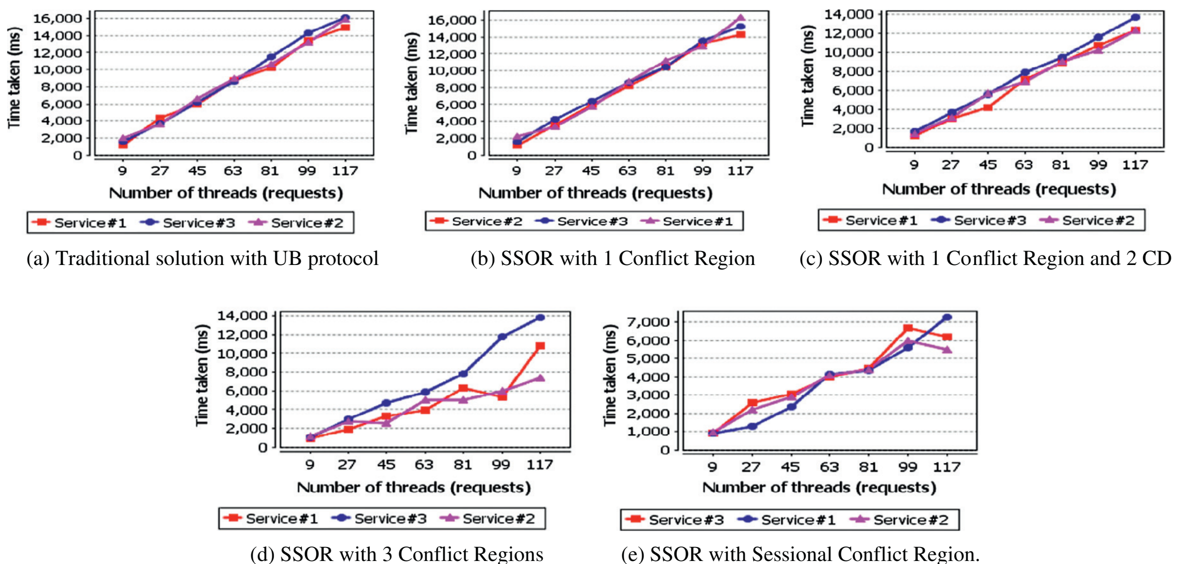
Now, suppose sequential consistency is preferable in the majority of cases since only the three services' outputs need to be consistent for a particular user. As shown in Fig. 5e, configuring the services with sessional conflict region outperforms by 28.8% when compared with when no replication involved, since it simulates the best case where every request comes from a distinct client. We can conclude that SSOR outperforms traditional solutions with fixed sequencer protocol in terms of flexibly making trade-off between scalability and consistency. In addition all experiments obtain linear latency since MSP is a variation of the fixed sequencer protocol. The achieved scalability tends to be more significant when the complexity of services increases.

Fig. 6 illustrates the scalability of all three atomic services for all experiments in Fig. 5. For each experiment, scalability is statistically measured by the average performance degradation of each request when the numbers of requests increase. More formally, we define:

$$\text{Scalability} = \frac{P_{max} - P_{min}}{W_{max} - W_{min}}$$

where P_{max} and P_{min} denote performance of the maximum and minimum concurrent requests respectively. W_{max} and W_{min} represents the maximum and minimum simulated workload which in our case are 117 and 9 respectively. As shown in Fig. 6, a smaller value of the y-axis statistically indicates a better scalability. We can also observe that experiments a and b again obtain similar results due to the fact that they are essentially based on similar approaches.

One interesting observation is that the scalability of removeUser service in experiment d only performs slightly better than in c. This is because we randomly send requests to the nodes, and the sequencer of the region of *removeUser* service may potentially receive more requests, which causes additional overhead and thus downgrade the performance. However, other experiments from b to e prove that by loosening certain consistency and defining consistency boundaries via the proposed region, scalability can be increased gradually. As a result, we can conclude that SSOR significantly outperforms traditional fixed consistency approaches.



(a) Traditional solution with UB protocol (b) SSOR with 1 Conflict Region (c) SSOR with 1 Conflict Region and 2 CD

(d) SSOR with 3 Conflict Regions (e) SSOR with Sessional Conflict Region.

Fig. 5. Experiments results of SSOR and traditional approach with fixed consistency (service 1 = createUser, service 2 = login, service 3 = removeUser).

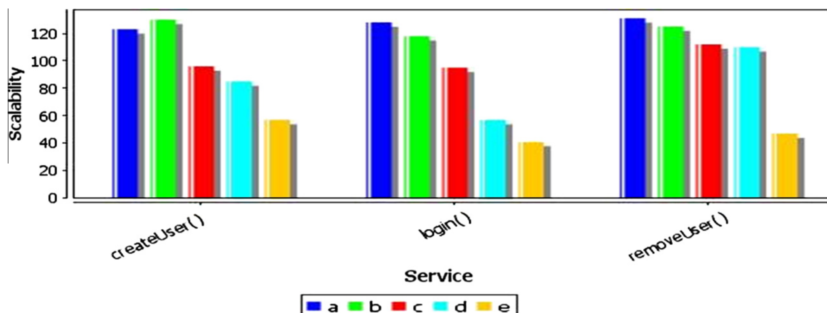


Fig. 6. Scalability of experiments a to e in Fig. 5.

8.4. Overhead

Comparing average overhead with respect to the numbers of replica is achieved by sending a total number of 30 requests for each of the three services concurrently. The average overhead under certain number of replicas and consistency requirements is recorded. The simulation is based on one machine only as we aim at a comparative study in this experiment. As illustrated in Fig. 7, when only sequential consistency is desirable, the overhead ranges from 50% to 150% (all requests come from different clients) with 2–7 replicas. On the other hand, the overhead produced when 3 distinct conflict regions are used is slightly higher, around 125–440%. As mention before, performance achieved by one conflict region with 2 concurrent deliverable services is non-deterministic; therefore the corresponding line of 1 conflict region with 2 CD in the figure is relatively unstable (150–860%). In comparison, traditional approach produces larger overhead (210–940%) than any of the other three, and this would hugely downgrade the scalability when such strong consistency constraint is not required amongst all of the three services. To this end, we have similar results as in previous experiments.

8.5. Impact of lazy replication

Two more experiments are conducted for evaluating the impact of lazy replication in CS and RS. We setup a service named *updateUserActivity* with computational complexity of 245 ms. To simulate RS, we modified the atomic service *login* to include the function of return current local timestamp. The complexity of *login* service is near 0 and the system requires the returned timestamp is absolutely consistent in all nodes, therefore the *login* service shall be considered as a redundant service. The performance of these two services is compared with traditional approaches that do not allow for adjusting redundancy under different redundancy levels. We apply three distinct nodes and 30 requests are concurrently sent to those services. As shown in the above Fig. 8a, the lazy replication in CS introduces extra overheads as compared to the traditional approach when the complexity is below 203 ms. However, as the complexity increases (from 203 ms to 890 ms), CS outperforms the traditional one as it only replicate partial computation. We obtain such observation because although lazy action delays the overall time taken for each round, the benefits gained is that redundant computation is reduced, which decreases the waiting time for subsequent processes. Such benefit is significant only if the reduced computation is complex enough. As shown in Fig. 8, lazy replication performs better when such complexity is more than 203 ms, and the more complexity it has, the more improvements can be obtained. Lazy replication may violate uniform property provided by the underlying components as the node may crash after execution on the contacted node but before multicasting. Similar to the experiment of CS in Fig. 8a, the application of RS for solving non-deterministic and RNI issues is evaluated by comparing performance with the traditional approach. As shown in Fig. 8b, the lazy replication on RS has a slightly worse responsiveness due to the fact that the RS has nearly zero computation complexity, therefore the penalty caused by delaying the multicasting process cannot be

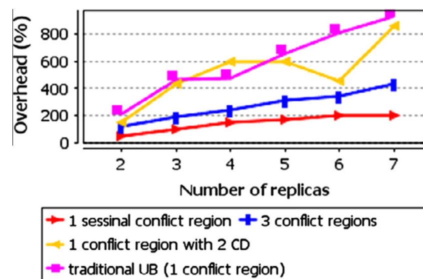
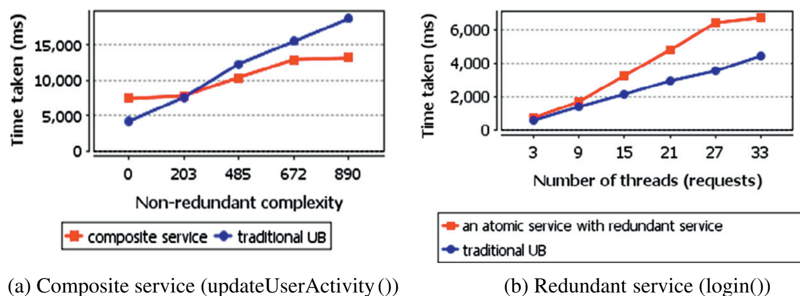


Fig. 7. Overhead under different number of replicas.



(a) Composite service (updateUserActivity ()) (b) Redundant service (login())

Fig. 8. Impact of lazy replication.

traded through running less redundant computations. However if RS has a reasonably higher complexity, then it would obtain similar results for the experiments of CS, as shown in Fig. 8a.

8.6. Elasticity

To evaluate elasticity, two clients simultaneously send requests to *createUser* and *login* services, each associated with two different conflict regions, hence no conflict occur between the two services. Three nodes are used and two of which are elected as the sequencers for the two conflict regions. The requests from each client are sent synchronously, that is, subsequent requests are only sent if the previous ones have been completed. Elasticity is evaluated by monitoring the performance of each service when the sequencer of a region needs to be changed as a result of either new joining nodes or a crash failure.

Fig. 9 shows the performance of these two services initially with one node, and then when there are two more nodes join, one of which is required to become the new sequencer. We can clearly see that the latency of the 42nd request for *createUser* reaches 950 ms, this is due to the overhead produced by changing the corresponding sequencer. On the other hand the *login* service does not suffer such delays since it belongs to another region. Fig. 10 shows the performance when two of the nodes crashes, one of which is the original sequencer. At the 25th request, the *login* service suffers high latency due to the need for reaching a consensus when the sequencer of its regions crashes, while the *createUser* service only suffer slightly delay because of view synchrony.

In both scenarios, performance of both services are relatively unstable when there are three correct nodes (after the 42nd request in Fig. 9 and before the 25th request in Fig. 10), this is due to the fact that when more than one node exists, requests are distributed to all nodes, hence requests routed to a sequencer node would have better performance than those reaching other nodes. To conclude, the notion of region has successfully limited the overhead produced by node crash and sequencer change, as only those services within certain regions are affected.

9. Related work

Table 3 provides a summary of the related work which we elaborate as below.

9.1. Replication and consistency in database systems

Managing consistency has been previously in the design of replicated database. The treatment for scalability in these solutions was implicit; furthermore, the solution has not explicitly looked at the tradeoffs between scalability and consistency in the design. We argue that the fundamentals for managing consistency in data-oriented database systems and service-oriented cloud are different due to differences between data and service-oriented replication, as highlighted in Section 3. Data-oriented replication considers consistency for each single data item. However, consistency in service-oriented replication tends to be more complicated as it includes the consistency among different business processes and

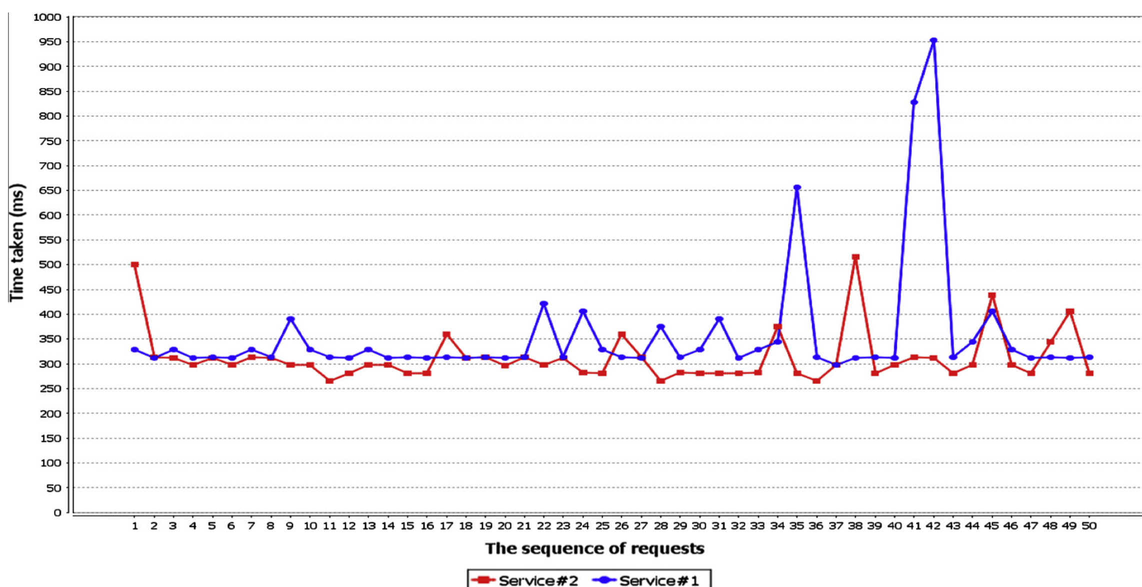


Fig. 9. Performance when two nodes join. (service 1 = createUser(), service 2 = login()).

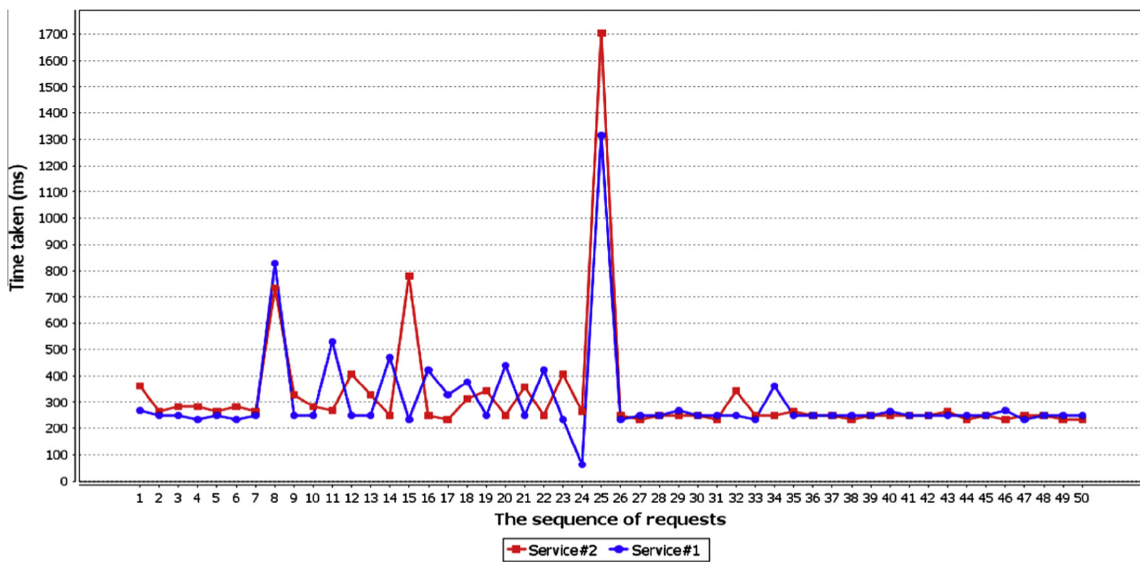


Fig. 10. Performance when two nodes crash. (service 1 = createUser(), service 2 = login()).

Table 3
A compendium of related work on replication and consistency.

Literature	Replication granularity	Technology standard	Consistency model	Strength	Weakness
[4,31,36,45,47,49]	Service-oriented	Web service	Linearizability	Scalable architecture that reduces communication amongst replicas	Fixed consistency model
[16,39,41]	Data-oriented	Generic	Strong sequential	Improves scalability based on the assumption that data conflict is rarely occurring.	Fixed consistency model, expensive roll back process.
[1.33]	Data-oriented	Generic	Causal	Aim for better scalability via optimizing the abcast protocol with relaxed causal consistency	Fixed consistency model, semantic of data is difficult to identify without knowing the context.
[2,3,11,22,28,29,46,48]	Data-oriented	Generic	Dynamic	Improve scalability by relaxing consistency as required	Consistency requirement of single data item is difficult to assume properly during design time.
[6,13,25,26,27,34,42]	Data-oriented	Generic	Eventual	Provide good scalability by having relatively weak eventual consistency model	Although they allow to influent consistency by selecting quorum value, the overall consistency model is still restricted to eventual consistency.

violated data. Cassandra [25], for example, is a distributed storage systems for managing data-oriented replication; it replicates data with quorum-based protocol to guarantee consistency. Although it offers three quorum values for consistency configuration, the resulted consistency belongs to the eventual consistency model [43], which is known to be the weakest client-centric consistency model. Similarly, Dymano [13] supports consistency via quorum-based protocol and additionally, it allows for better availability by leveraging on certain consistency upon the presence of failure. Despite the fact that the aforementioned systems and our SSOR are aimed for replication, their approaches are fundamentally different from our work. Firstly, they aim at data-oriented replication whereas we address replication for service-oriented systems, accounting for business processes and violated data. Secondly, the eventual consistency is weak and insufficient for dealing with service-oriented replication with stronger consistency requirements. Our proposed SSOR overcomes this limitation by providing various consistency models, which ranges from the strongest to the relatively weaker ones in an attempt to explicitly promote scalability of replication. A variety of approaches have been proposed for solving scalability issue in replication through optimizing the *abcast* protocols at communication level; a complete survey of these approaches is detailed in [14]. [1,33] propose solutions named generic broadcast that learn application semantic of partial ordering amongst data items, hence delivered in order only when necessary; this is similar to the concurrent delivery principle presented in this paper, their solutions however learn casual relationships by monitoring the content of transmitted data.

9.2. Replication and consistency in service-oriented systems

Several approaches have been proposed for service oriented replication. [4] propose a three tiers replication architecture that provides better scalability when replicating over WAN, with passive replication in the middle tier. WS-Replication [36] provides availability guarantee for replicated services based on active replication and JGroup, which uses the UB fixed sequencer protocol. Ye and Shen [45] propose a middleware using active replication as well, but it relies on communication history based one the *abcast* protocol. [31] present a middleware based on a variation of passive replication, which is rather close to active replication. All of the above approaches are either not intend to consider scalability and consistency in replication or are strongly bound to a specific standard (web service), as a result they cannot satisfy consistency requirements at the application level and scalability tends to be limited. Experiment results show that SSOR under various loads when certain consistency is desirable achieves better scalability than traditional approaches. Various optimistic solutions [16,39,41] are proposed based on the assumption that inconsistency is likely to occur in low probability; hence *abcast* is not need for all the messages. However it may also suffer scalability issue since it requires rollback whenever an out-of-order invocation is detected and no trade-off can be made when rollback is needed. In addition, rollback may be impossible or rather difficult to achieve if the service involves hidden components.

9.3. Replication and consistency in the grid

The relationship between scalability of replication and consistency has been widely studied in the context of grid computing. [3] propose a Quality of Service (QoS) aware replication strategy in the grid. QoS refers to dependability requirements including scalability and consistency. Their approach however requires users to specify the relevant 'importance' of different data, which is used to make trade-offs decision at runtime when conflicts occur. [29] argue that replication in grid should work in conjunction with scheduling, therefore they propose a combined algorithm that determines when to schedule task and when to replicate task. An extensive survey regarding replication and consistency in grid has been done in [2]. However, these work differentiate from ours in the sense that they focus on data whereas we concentrate on service.

9.4. Replication and consistency in the cloud

[11,22,28,46] are the closest work to our work. In particular, [22] propose an approach for trading consistency and scalability of transactional processes in the cloud, based on the needs of an application. The notion of category they proposed is similar to the region policies, but they mainly aim for storage levels. [46] proposes a consistency model to support the trade-off between consistency and scalability by detecting inconsistency with a set of metrics for consistency spectrum on data items. However, inconsistency detection implies the needs for rollback, which may be impossible or relatively difficult to achieve. Our solution achieves this trade-off by satisfying different consistency models on services that demands different requirements; hence SSOR essentially avoids unexpected inconsistency from happening. IDEA [28] is a self-managed middleware that adaptively satisfy system's requirements, and it is based on the continuous consistency model provided in [46]. Because of the passive property of inconsistency detection, IDEA can cope with requirements changes on the fly. In contrast, their approach is client centric whereas our proposal is per-service basis and transparent to end-users. In addition, unlike the detailed specification of consistency levels in [46], our approach depends on a choice of zero or unlimited conflicts via different consistency models, which tends to be adequate for a number of real-world applications. [11] propose a framework named Harmony, which can adaptive the consistency level (as numeric value) at runtime depending on the ratio of read and write process. Their approach is data-oriented, however.

The approach described in [47] is a cloud-based queuing service that enables high availability. However their approach forces the user to select either strong consistency or none at all. The paper in [44] describes a cloud-based transaction replication middleware named CloudTPS, this solution is implemented with the assumption that strong transactional consistency is required. Cloud-based storage systems are popular in terms of investigating the correlation between consistency and scalability. The research in [6] conducts a new consistency model, which considers a new dimension of different cloud providers based on Dymano's quorum strategy. Unlike our work, the approach is data-oriented and they can guarantee single consistency model only. Based on the same motivation of this paper, [48] also report on attempt to promote scalability by leveraging different consistency level. However, unlike our proposal, their approach is driven from the eventual consistency model, where the variation is that the solution is able to change the extent to which consistency can be violated. Instead of providing different consistency models to various scenarios, there also exist a consistency model that is static, but can cope with most scenarios of consistency and scalability requirement [34]. Such model is extended from the eventual consistency model, it is designed to be sufficiently strong to ensure most scenarios that demand good consistency, but also weak enough to allow good scalability. However, the applicability of this model depends highly on empirical studies, since the application scenarios tends to be unpredictable in the reality. [26] have specifically look at consistency and scalability problem in the cloud and they propose a new I/O model to reach a good tread-off between these two attributes. [27,42] propose architectural approach that specifically copes with scalability issue within the context of cloud federation whereas we look at single cloud provider scenario. However, since we did not make any assumptions with regards to the number of cloud providers, the proposed SSOR can be easily extended to the case of cloud federation.

10. Discussion

One important decision of our approach is to apply GCS as the underlying infrastructure. This is because GCS offers reliability and fault tolerance properties required by any replication system. However, GCS itself is not designed for scalability and thus the scalability of SSOR could be influenced by the performance of GCS. Clearly there is a trade-off between reliability and performance in such case. Our implementation also shows that GCS is used as a black-box layer as part of our architecture, therefore the proposed notion of region and protocols can be realized without GCS. However, excluding GCS implies that the complexity of fundamental design and implementation would be significantly increased, as it is impossible to assume any reliability of communications.

Based on the current stage of our research and observation from experiments, SSOR still suffers from two main limitations: (1) Runtime changes of regions and categories of service are not yet been handled. This is because our approach aims for achieving various fixed consistency models rather than satisfying continuous consistency, which is usually based on numeric value. (2) From the experiments, bottlenecks still exist within a single region. Although we introduced the concurrent deliverable service, the result of such specification is non-deterministic. This is mainly due to the fact that whether a single region scales well within a defined concurrent deliverable service strongly depends on how requests are interleaving with each other. Less consecutive requests being sent to concurrent delivery services would result in a degrading performance and vice versus.

Another interesting issue, but that is beyond the scope of this paper is that the way services are partitioned into different regions determines how the application scales, which implies that services within the same CR may still suffer from scalability issues if those services cannot be separated properly. Therefore such design decision that made by system administrators plays an integral role.

Our framework targets the foundational aspects in managing the trade-offs between consistency and scalability. The experiments use virtual machine running on a mini-cloud to demonstrate that our approach is fundamentally capable to deal with these trade-offs and to leverage better scalability with more relaxed consistency (and vice versa). The results yield valuable insight into the problem and demonstrate definitive improvement on the foundational aspects in contrast to the conventional service-based replication approach with static consistency. Though we are running our experiments in a controlled environment (i.e., a mini-cloud), the fundamental reasoning in applying the approach to a real setting is the same but with larger number of VMs. In this context, the only factor that could influence the results of our experiments when replicated in such environment is the VM interferences caused by scaling up the number of VMs. The VM interference is another research direction e.g., [21], which is out of the scope for this paper.

11. Conclusion

In this work, we presented a Scalable Service-Oriented Replication (SSOR) middleware solution that is capable of satisfying consistency requirements in service-oriented and cloud-based applications. To address the fundamental differences between DOR and SOR, we proposed new formalisms to describe services in SOR. We showed, by ways of systematic proofs how the proposed notion of consistency regions can be used to satisfy various consistency models and manage consistency requirements for groups of services within a region. To flexibly guarantee different consistency models at runtime, we defined new policies that operate within regions. Also, to realize our notions, formalisms and policies in practice we developed two novel protocols, Multi-fixed Sequencer Protocol (MSP) and Region-based Election Protocol (REP). Specifically, MSP aims at guaranteeing the satisfaction of the consistency models in a region, REP is responsible for flexibly balancing the workload amongst sequencers through electing new sequencers upon the presence of failure and distributes the loads to multiple sequencers.

To ensure reliability of the SSOR middleware, we illustrated our solutions in details for tolerating sequencers and non-sequencers crashes as well as the concept of region distribution synchrony for handling simultaneous node crashing. Experiment results showed that in contrast to traditional service replication approaches, SSOR promotes flexible consistency and thus, improves scalability of the managed cloud-based applications. We observed that as complexity increases, composite and redundant services provide better response time as a result of partial replication. In addition, the results of our experiments show that the distribution of services into different regions in SSOR significantly reduces the impact caused by crash failure. This implies that our SSOR could also be beneficial for improving elastically in the cloud.

In future research, we will investigate the applicability of self-managed replication for services that can cope at run-time with requirements changes. We also plan to research the trade-off problem for other quality attributes such as the availability and costs of various resources.

References

[1] M.K. Aguilera, C. Delporte-gallet, H. Fauconnier, S. Toueg, Thrifty generic broadcast, in: Proceeding of 14th International Symposium on Distributed Computing (DISC'00), 2000, pp. 268–282.

[2] T. Amjad, M. Sher, A. Daud, A survey of dynamic replication strategies for improving data availability in data grids, Future Gener. Comput. Syst. 28 (2) (2012) 337–349.

[3] V. Andronikou et al, Dynamic QoS-aware data replication in grid environments based on data "importance", Future Gener. Comput. Syst. 28 (3) (2012) 544–553.

[4] R. Baldoni, C. Marchetti, A. Termini, Active software replication through a three-tier approach, in: Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS'02), 2002, pp. 109.
[5] R. Baldoni, C. Marchetti, S. Tucci-Piergiovanni, A fault-tolerant sequencer for timed asynchronous systems, in Proceedings of the 8th International Euro-Par Conference, 2002, pp. 578–588.
[6] D. Bermbach et al., MetaStorage: a federated cloud storage system to manage consistency-latency tradeoffs, in: Proceedings of the 4th IEEE Conference on Cloud, Computing, 2011, pp. 452–459.
[7] P. Bernstein, V. Hadzilacos, N. Goodman, Concurrency Control and Recovery in Database Systems, Addison-Wesley, 1987.
[8] K.P. Birman, Building Secure and Reliable Network Applications, Prentice Hall, 1996.
[9] R.S. Chang, C.Y. Lin, C.F. Lin, An adaptive scoring job scheduling algorithm for grid computing, Inform. Sci. 207 (2012) 79–89.
[10] D. Cheriton, D. Skeen, Understanding the limitations of causally and totally ordered communication, in: Proceeding of 14th ACM Symposium on Operating System Principles, 1993, pp. 44–57.
[11] H.E. Chihoub et al., Harmony: towards automated self-adaptive consistency in cloud storage, in: Proceeding of IEEE International Conference on Cluster Computing, 2012, pp. 293–301.
[12] G. Coulouris, J. Dollimore, T. Kindberg, Distributed Systems: Concepts and Design, third ed., Addison-Wesley, 2001.
[13] G. DeCandia et al, Dynamo: amazon's highly available key-value store, in: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles, ACM, 2007, pp. 205–220.
[14] X. Défago, A. Schiper, P. Urbán, Total order broadcast and multicast algorithms: taxonomy and survey, ACM Comput. Surv. (CSUR) (2004) 372–421.
[15] R. Ekwall, A. Schiper, A fault-tolerant token-based atomic broadcast algorithm, IEEE Trans. Depend. Secure Comput. 8 (5) (2011) 625–639.
[16] P. Felber, A. Schiper, Optimistic active replication, in: Proceeding of 21st International Conference on Distributed Systems, 2001, pp. 333–341.
[17] R. Guerraoui, R.R. Levy, B. Pochon, V. Quema, High throughput total order broadcast for cluster environments, in: Proceedings of the International Conference on Dependable Systems and Networks, 2006, pp. 549–557.
[18] C. Hsiao, G. Chiu, A fault-tolerant protocol for generating sequence numbers for total ordering group communication in distributed systems, J. Inform. Sci. Eng. (2006) 1265–1277.
[19] P. Hutto, M. Ahamad, Slow memory: weakening consistency to enhance concurrency in distributed shared memories, in: Proceeding of 10th International Conference on Distributed, Computing Systems, 1990, pp. 302–311.
[20] JGroup: A Toolkit for Reliable Multicast Communication <http://www.jgroups.org>.
[21] Y. Koh et al., An analysis of performance interference effects in virtual environments, in: IEEE Symposium on Performance Analysis of Systems and Software, 2007.
[22] T. Kraska, M. Hentschel, G. Alonso, D. Kossmann, Consistency rationing in the cloud: pay only when it matters, in: Proceedings of the Very Large Data Base Endowment, 2009, pp. 253–264.
[23] L. Lamport, How to make a multiprocessor computer that correctly executes multiprocess programs, IEEE Trans. Comput. (1979) 690.
[24] L. Lamport, On interprocess communication, Distrib. Comput. (1986) 77–101.
[25] A. Lakshman, P. Malik, Cassandra – a decentralized structured storage system, in: Proceedings of the Workshop on Large-Scale Distributed Systems and Middleware (LADIS '09), Big Sky MT, 2009.
[26] D. Li et al, A new disk I/O model of virtualized cloud environment, IEEE Trans. Parallel Distrib. Syst. 24 (6) (2013) 1129–1138.
[27] J. Lloret et al, Architecture and protocol for intercloud communication, Inform. Sci. (2013).
[28] Y. Lu, Y. Lu, H. Jiang, Adaptive consistency guarantees for large-scale replicated services, in: Proceeding of National Academy of Sciences, 2008, pp. 89–96.
[29] N. Mansouri, G.H. Dastghaibyfard, E. Mansouri, Combination of data replication and scheduling algorithm for improving data availability in data grids, J. Netw. Comput. Appl. 36 (2) (2013) 711–722.
[30] J. Osrael, L. Froihofer, K.M. Goeschka, What service oriented middleware can learn from object replication middleware, in: Proceedings of the 1st Workshop on Middleware for Service Oriented Systems, 2006.
[31] J. Osrael, L. Froihofer, M. Weghofer, K.M. Goeschka, Axis2-based replication middleware for web services, in: IEEE International Conference on Web Services, 2007, pp. 591–598.
[32] J. Osrael, L. Froihofer, K.M. Goeschka, Replication in service oriented systems, J. Softw. Eng. Fault Toler. Syst. (2007) 91–117.
[33] F. Pedone, A. Schiper, Generic broadcast, in: Proceeding of 13th intl. symposium on distributed computing (DISC), 1999, pp. 94–108.
[34] S. Peluse et al., When scalability meets consistency: genuine multiversion update-serializable partial data replication, in: Proceeding of international conference on distributed, computing systems, 2012, pp. 455–465.
[35] D. Powell, M. Chéréque, D. Drackley, Fault-tolerance in Delta-4∗, ACM Operat. Syst. Rev. SIGOPS 25 (2) (1991) 122–125.
[36] J. Salas, F. Perez-Sorrosal, M. Patiño-Martínez, R. Jiménez-Peris, WS-replication: a framework for highly available web services, in: Proceedings of the 15th International Conference on World Wide Web, 2006.
[37] J.H. Saltzer, D.P. Reed, D.D. Clark, End-to-end arguments in system design, ACM Trans. Comput. Syst. (1984) 277–288.
[38] F.B. Schneider, Implementing fault-tolerance services using state machines approach: a tutorial, ACM Comput. Surv. (1990) 299–319.
[39] Y. Shen, Providing reliable web service though active replication, in: 6th IEEE/ACIS International Conference on Computer and Information, Science, 2007.
[40] Q.Z. Sheng et al, Behavior modeling and automated verification of web services, Inform. Sci. (2012).
[41] A. Sousa, J. Pereira, F. Moura, R. Oliveira, Optimistic total order in wide area networks, in: Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems, 2002, pp. 190–201.
[42] D. Villegas et al, Cloud federation in a layered service model, J. Comput. Syst. Sci. 78 (5) (2012) 1330–1334.
[43] W. Vogels, Eventually consistent, Commun. ACM 52 (1) (2009) 40–44.
[44] Z. Wei, G. Pierre, C.H. Chi, CloudTPS: scalable transactions for web applications in the cloud, IEEE Trans. Service Comput. 5 (5) (2012) 525–539.
[45] X. Ye, Y. Shen, A middlware for replicated web service, in: Proceeding of IEEE International Conference on Web Service, 2005, pp. 631–638.
[46] H. Yu, A. Vahdat, Design and evaluation of a continuous consistency model for replicated services, in: Proceeding of Operating Systems Design and Implementation, 2000.
[47] Z. Zhang et al, A cloud queuing service with strong consistency and high availability, IBM J. Res. Dev. 55 (6) (2007) 1–12.
[48] W. Zhu, M. Woodside, Optimistic scheduling with geographically replicated services in the cloud environment (COLOR), in: Proceeding of 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid, Computing, 2012, pp. 735–740.
[49] Y. Zhu et al, A human-centric framework for context-aware flowable services in cloud computing environments, Inform. Sci. (2012).