Efficient Inter-Cloud Replication for High-Availability Services

Abdallah Abouzamazem
School of Computing Science
Newcastle University
Newcastle upon Tyne, UK
Abdallah.Abouzamazem@ncl.ac.uk

Paul Ezhilchelvan
School of Computing Science
Newcastle University
Newcastle upon Tyne, UK
Paul.Ezhilchelvan@ncl.ac.uk

***Abstract*— Amazon's recent service disruption and investigations into the underlying causes of similar major outages indicate that cloud outages in future cannot be ruled out with certainty. This paper investigates the idea of tolerating outages by inter-cloud replication, i.e., through service replication on multiple, fail-independent clouds. A challenge in realizing this idea is to minimize performance degradation that inevitably arises when replicas on multiple clouds have to be kept in a mutually consistent state over the Internet. It is addressed by developing a new order protocol that makes the most use of the high bandwidth communication within a cloud and uses the Internet communication to minimum necessary. The protocol also deals with cloud outages and widely differing rates with which service requests can arrive at replicas in different clouds. Experiments performed confirm that the protocol reduces the ordering latencies considerably and also improves throughput.**

Keywords: State Machine Replication, Multi-Cloud replication, Input Ordering, Mencius, Order latency and throughput, node crashes, cloud outages.

I. INTRODUCTION

This paper addresses the availability concerns that arise when a service provider decides to avail cloud-based infrastructures, instead of his own dedicated infrastructure, to host his application. Clients can access a hosted application as a service using the Internet and processing a client's request typically involves accessing and even updating data at the back-end and returning a sequence of responses to the client and/or to client-specified destinations. Such third-party hosted services can range from enterprise services to Scientific computing. In the rest of the paper, we term these hosted services simply as *services*. Note that a (hosted) service would commonly make use of some of the services offered by a cloud provider; in fact, this usage reliance is a major attraction in opting for cloud infrastructures.

A service, even if built robustly, becomes unavailable when an underlying support service offered by the cloud provider becomes unavailable; we term the latter event as a *cloud outage*. Consequently, concerns about the availability of a hosted service arise because cloud outages have occurred in the recent past and subsequent investigations reveal that the underlying causes do not readily lend themselves to a permanent, fool-proof fix so that outages in future can be ruled out with certainty. These reasons form the rationale for our work and are expounded below.

Cloud users have recorded outages in major cloud providers over the last three years. Some of these outages had gone on even for hours: Amazon EC2 (2011) [1], Skype (2010) [19], Wikipedia (2010) [20], Paypal (2010) [15], Google (2010) [10], two incidents with Gmail (2009) [8, 9].

Detailed, post-outage investigations reveal that the 'root' events that eventually led to these outages in fact caused anticipated failures (see Table I) and the cloud system had automated procedures to recover from these failures. In many cases, the recovery procedures had been tested offline to industry standards. Despite all these, the recovery not only did not work but instead caused correct components to fail, leading to an outage. This happened because the system conditions that prevailed *at the time* did not satisfy some implicit design assumptions in the recovery procedures.

TABLE I. OUTAGES THAT LASTED FOR HOURS.

Service Outage	Root Event → '*supposedly tolerable*' *failure* → **incorrect recovery** → Outage Nature
EBS	Network misconfiguration→ *Nodes partitioning* → **re-mirroring** → many clusters collapsed [1]
Gmail	Upgrade → *some servers offline* → **bad request routing** → all routing servers went down [8]
Gmail	Maintenance → *a datacenter (DC) offline* → **Bad cross-DC re-mirroring** → many DCs down [9].
Paypal	Network failure → *Front-end systems offline* → **late failover** → global service interruption
Skype	System overload → 30% supernodes (SN) crashed → positive feedback loop → all SNs crashed [19]
Wikipedia	Overheated DC → *that DC offline* → **broken failover mechanism** → global outage [20]
App Eng	Power failure → *25% machines of a DC offline* → **bad failover** → all user apps in degraded state [10]

For example, the root-event in [10] was a power failure affecting 25% of machines; in Skype's case, it was a system overload that affected 30% of supernodes. In both cases, recovery procedures attempted at shifting the load from affected nodes to healthy nodes; the assumption was that some healthy nodes around a crashed one would be

sufficiently under-loaded to take on the extra load. On that occasion, this assumption was not met and the recovery attempt ended up compounding the overload problem. Table I, taken from [12], summarizes the root event for each major outage and the recovery procedure (shown in bold) that not only failed to accomplish recovery from a 'supposedly tolerable' failure (shown in italic) but also led ultimately to the outage. The picture that emerges from investigations on these outage incidents can be summarized as follows.

Some system conditions that were not encountered during offline tests emerged in actual deployment and caused bugs/oversights in the design of recovery procedures to manifest into failures which then lead to outages. Some researchers [12] propose regular online, fault-injection testing of recovery procedures (akin to 'fire drills' in institutions). The effectiveness of this approach is questionable given the large-scale and evolving nature of cloud systems; the probability of outage occurrences may well be reduced but cannot be assured to be zero.

Given the complexity involved in eliminating all possible causes of cloud outages [6], it would only be practical to assume that long-lasting outages are inevitable and that critical, hosted services be supported to *tolerate* outages. Accomplishing outage-tolerance requires service replication on N, N > 1, clouds, which can be termed as *inter-cloud replication*, and *managing* replication *efficiently* so that the service remains responsive even in the most demanding load conditions. The latter is the core objective of this paper.

Inter-cloud replication is effective against outages as multiple clouds are very unlikely to suffer outages *at the same time* for two reasons. Different cloud providers tend to employ diverse recovery strategies and their systems are unlikely to encounter the adversarial system conditions at the same time. Inter-cloud replication has recently been investigated for secure storage [2] and our focus here will be on service availability through replication without compromising service responsiveness.

Replication requires that the replicas perform state updates in a consistent manner despite concurrent user access, server crashes and, additionally here, outage occurrences. Replica consistency, in turn, requires that the user requests be identically ordered at all replicas and the overheads incurred for ordering the requests predominantly influences responsiveness of a replicated service. In this paper, we first propose a 2-tier structure for organizing inter-cloud replication: replication *within* a cloud forming a crash-tolerant but outage-prone entity that is then replicated *across* multiple clouds. Secondly, we propose an outage-tolerant order protocol that mirrors this 2-tier replication structure. It is designed to make the most of the high bandwidth communication within a cloud and to minimally use the slow, inter-cloud communication over the Internet. Finally, we run experiments to show that the ordering latencies are reduced considerably compared to the inter-cloud replication pursued by the conventional 1-tier approach. These three aspects are the main contributions of the paper.

The paper has the following structure. Next section describes the rationale behind the design choices that led to our 2-tier inter-cloud replication structure. Section III presents the two-stage order protocol for the proposed 2-tier replication structure. Section IV is on performance comparison that shows that our approach leads to reduced ordering latencies and improved throughput. Section V concludes the paper.

II. Challenges and Design Choices

We use the term *server* to mean the virtual or physical machine within a cloud that implements a hosted service. Figure 1 depicts the *default* inter-cloud service replication in which the service is hosted by one server in each of N, N = 3, clouds which we call *sites*. Servers s_j and s_k, $0 \leq j,k \leq 2$, communicate with each other over the Internet and through the firewalls of their respective sites. Internet communication is assumed reliable with retries being automatically carried out when a packet loss or corruption is detected. We assume that servers communicate using TCP/IP.

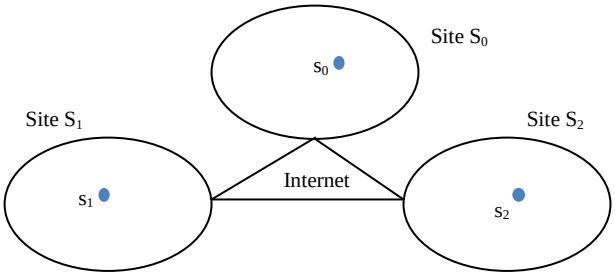

Figure 1. Inter-Cloud Replication: Default Approach.

A client of the service can submit its request to any one server. Suppose that requests r_0 and r_1 are submitted respectively to s_0 and s_1 and lead to, upon being processed, updates u_0 and u_1 on the service state. (If, say, r_0 is read-only, then u_0 is null.) Though only two servers directly received one request each, all three servers must carry out both the updates and must do so in the same order: either u_0 followed by u_1 or *vice versa*; otherwise, the replication is incorrect. This leads to the *requirement* that the replication management maintain the *abstraction* of a *single server* with a single input queue: all requests, irrespective of the site where they are submitted at, enter the single, abstract queue in *some* order and update, one at a time, the service state held by the single, abstract server.

Assumption A1: A server can fail only by crashing and a crashed server recovers or is replaced after an interval of some arbitrary duration. Further, message communication delays between two operative servers are finite but a fixed upper bound on these delays does not exist.

The absence of a fixed bound on delays encapsulates (i) the possibilities of message propagation, especially over the Internet, being unduly delayed, e.g., due to congestion or a transient breakdown, at arbitrary moments, and (ii) hence the impossibility of estimating a bound that is guaranteed to hold on the delays that are to unfold in near future. In particular, it does *not* mean that the communication can be unreliable nor the delay bound should not be estimated, but merely admits that the delays *can* exceed any estimate of the bound.

A. *False Suspicions in Dealing with Crashes*

In maintaining the 1-server abstraction mentioned earlier, a server crash must be dealt with at some level, irrespective of the replication strategy used. Typically, an operative server observes the crash of another server when the latter's expected response is absent for 'too long'; for example, if an 'are-you-alive' probe is not responded to within a timeout of sufficiently long duration, a crash *suspicion* is raised. A suspicion correctly indicates a crash only if the timeout duration used is at least as long as the round-trip delay prevailing at that moment; otherwise, it is a false alarm. Thus, a suspicion can be correct or false when it is raised; moreover, a correct one is never contradicted and a false one, on the other hand, is nullified with the passage of time when the expected response arrives later than expected.

To deal with crashes and the uncertainty over suspicions, the following assumptions are made.

Assumption A2. N > 2 and at most $\lfloor (N-1)/2 \rfloor$ servers can fail *at the same time*.

A2 requires that the causes of server crashes not be correlated. Throughout the paper, we assume that N=3 and the sites involved are denoted as S_0, S_1 and S_2 (see Fig. 1, 2).

Assumption A3. False crash suspicions are subsequently realized to be false and there is an unknown instance of time after which a correct server is not suspected.

Assumption A4. Correct suspicions are eventually made.

Assumptions A3 and A4 are collectively called *eventual perfect detection* (denoted as ◊P) in the literature [4].

B. *Primary-Backup vs. State Machine Replication*

In primary-backup (PB) replication, one of the servers is designated as the primary and the rest as backups. Backup servers forward the requests that they directly received, to the primary which processes the requests and returns the updates to all backups. When the backups agree that the primary has crashed, one of them becomes the new primary.

In state machine replication (SMR, for short), servers exchange with each other the requests that they received directly, order all requests identically and process them in that order [17, 18]. Each server responds only to those requests that it directly received.

Note that servers do not exchange state updates in SMR; whereas, in PB replication, the primary server has to disseminate all state updates to all backups and hence can become a performance bottleneck. So,

Choice 1: SMR is chosen with an objective to reduce the ordering overhead as small as possible. In fact, as we shall see later, our approach requires that a server disseminate a directly received request once to every other remote site.

Choice 1 is orthogonal to N > 2 in A2 and PB replication is not possible when false suspicions can occur *and* when N = 2: replication fails when both servers act as primary after (falsely) suspecting each other of having crashed and this situation is commonly referred to as *split-brain syndrome*.

C. *Default approach to Outage Tolerance*

A *default approach* to providing outage tolerance is to use the default replication approach by considering that a server crash is *equivalent* to a site outage from the perspective of remote servers, even though server crashes are far more common compared to site outages. However, the default approach is readily implementable using the state of art protocols. One of the most efficient protocols available for implementation is Mencius [13] which

- is specifically designed for servers communicating over the Internet using TCP/IP,
- assumes ◊P detection,
- offers several optimization features, and
- enables a wrongly isolated server to receive the updates it missed while it was isolated.

D. *Proposed Approach to Outage Tolerance*

In this paper, we investigate an alternative approach to outage-tolerance which distinguishes site outages from server crashes; a server crash in one site would not even be visible to servers at remote sites. In the proposed approach, server s_k, $0 \le k \le 2$, in Fig 1 is replaced by a group of *n*, *n*>2, replicated servers: s_{0k}, s_{1k}, …, s_{nk}, which are connected to each other by the high-bandwidth network within the cloud. Each s_{jk} knows the address of, and communicates directly with, every other $s_{j'k'}$, $0 \le k, k' \le 2$, $0 \le j, j' \le n$ using TCP.

Assumption A5: Among the *n* servers of an operative site, A3 and A4 hold and at most $\lfloor (n-1)/2 \rfloor$ servers can remain crashed at any time.

Assumption A6. At most $\lfloor (N-1)/2 \rfloor$ sites can suffer an outage *at the same time*.

Throughout the paper, we assume N = *n* = 3, for brevity, as in Fig 2, even though N ≥ 2 and n > 3 are possible. In the proposed approach, replication therefore involves 9 servers of which up to 3 servers (one in each site) can remain crashed in the absence of any site outage. One option is to let these 9 servers execute an order protocol (e.g., Mencius) for identically ordering the requests to be processed, with the bound *f* on the number of servers that can crash, set to 3.

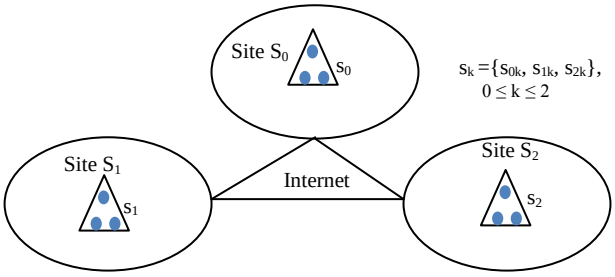

Figure 2. Inter-Cloud Replication: Proposed Approach.

This option, in addition to having the potential to increase the worst-case overheads due to *f* being large [7], cannot handle a site outage at all: a site outage makes three servers unavailable; if each of the operative sites already has one crashed server within, then *f* is exceeded by 2. So,

Choice 2: request ordering is done in two stages:

- Intra-site ordering using any known order protocol (Mencius, here) for $f = \lfloor (n-1)/2 \rfloor = 1$, followed by
- Inter-site ordering using a new protocol proposed for $\lfloor (N-1)/2 \rfloor = 1$.

The new, inter-site ordering protocol treats the server triplet in a given site as a single, reliable unit that can crash *only* when the site suffers an outage. This means that a server in one site does not have to suspect crashes of a remote server; it only has to suspect outages of remote sites.

As outages are major events, we assume that servers detect site outages *reliably*, e.g. by using an external oracle or using large timeouts, thus ruling out false outage suspicions. So, we make these additional assumptions:

Assumption A7. False outage suspicions do not occur.

Assumption A8. An operative server in an operative site correctly and eventually detects a site outage.

Assumptions A7 and A8 are collectively called *perfect detection* (denoted as P) in the literature [4]. The advantages of the proposed, 2-stage approach are demonstrated by comparing ordering latencies and throughput against those observed in the default approach realized though Mencius. Recall that Mencius is developed for request ordering over the Internet, and hence is the best choice for the default approach; however, the same cannot be said for using it for intra-site ordering as the replicas are connected by high-speed networks. Nevertheless, performance comparisons indicate that two-stage ordering offers considerable performance benefits which lead to a conclusion that benefits of 2-stage ordering would be more, had a protocol (e.g., [14, 3]) designed specifically for high-speed networks been chosen for intra-site ordering.

III. TWO-STAGE ORDERING OF REQUESTS

Figure 2 depicts the replication context for 2-stage ordering. A client submits its request to any site and the site can forward the request to any of the 3 servers. If a client suspects a site outage, he can re-submit his request at another site; similarly, if a server is suspected to have crashed, requests sent to it may be re-forwarded to another server. Duplicates are ordered like the original requests but are filtered out prior to being processed.

A. Background: Mencius

We highlight those aspects of Mencius which are only relevant to 2-stage ordering and refer to [13] for full details. For presenting these relevant aspects, we will assume the context of the default approach depicted in Fig 1 where servers s_0, s_1 and s_2, respectively in sites S_0, S_1 and S_2, decide on an identical processing order for the client requests.

An execution of any order protocol involves running an infinite sequence of *ordering instances*; each instance is uniquely numbered using a natural number $i \in \{0,1,2,\ldots\}$ assigned in the increasing order with no gaps. Several instances may be run concurrently at any given moment. The i^{th} instance assigns some request r the order number $o=i$. The assignment is irreversible and unanimous across all servers. The requests are processed (by all servers) as per the o assigned to them.

Mencius divides the (infinite) set of all natural numbers into three disjoint (infinite) subsets, when there are three servers (as assumed here). It uniquely assigns a subset to each server and a server initiates instances numbered from the subset assigned to it.

For brevity, let server s_k, $0 \le k \le 2$, be assigned the subset of $\{k, k+3, k+6, \ldots\}$. A counter C_k (initialized to k) holds the instance number to be initiated next by server s_k. When s_k receives a request r from a client, it initiates an instance for r with number $i = C_k$, increments C_k by 3 and coordinates the i^{th} instance so that o of $r = i$.

If all servers are busy receiving client requests, at least three instances would be concurrently running. If, on the other hand, one server hardly receives requests or receives requests at a smaller rate, then the instance numbers of that server will not get assigned or will get assigned at a reduced rate. The requests of busy servers cannot be processed if the numbers assigned to them are larger than the number yet to be assigned by the slow server. To handle this problem, servers are permitted to *skip* instances by assigning the skipped numbers to *null* requests. Mencius ordering guarantees the following three properties:

1. If a server learns that a non-null r is assigned the order number o, then r has been submitted by a client; (*validity*)
2. If a server seeks to order a non-null r and remains operative for long enough, then r would be learnt to have been assigned an order number; (*liveness*) and,
3. Whenever a server learns that a null or non-null r is assigned an order number o, then every other operative server will eventually learn, or has already learnt, that r is assigned the order number o (*unanimity*).

B. Two Stage Ordering

We first present the principle involved in 2-stage ordering, which consists of three steps. The three servers of each site k, s_{0k}, s_{1k} and s_{2k}, execute Mencius (only) amongst themselves and order all requests submitted at site k. Thus, three executions of Mencius, one in each site, take place in parallel. Note that these executions do not require inter-site communication over the Internet, but only intra-site message exchanges over the high-speed networks of each cloud. Each of them leads to servers of a given site generating an identical stream of requests. This stream is referred to as the *local stream* (local to a given site) in Fig 3 where the *site index*, abbreviated to SI, denotes the value of $k \in \{0,1,2\}$.

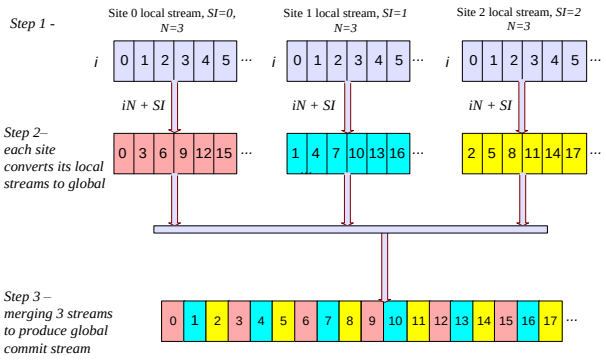

Figure 3. Basic idea in 2-Stage Ordering.

The next two steps involve merging these three local streams into a global stream, while preserving the order within each local stream. An ordered request in the local stream of site k with order number *o* is assigned a *global order number go* as: $go = 3 \times o + k$. The global stream is then composed as the stream of requests in all three streams, arranged in the increasing order of their *go*.

Next, we present some definitions and then describe a protocol for implementing the last two steps in a distributed manner. The protocol is then extended to deal with outage detections and sites receiving requests at unequal rates.

Definition 1: $r' <_l r$. Let r' and r be two requests that have been ordered using Mencius in a given site and assigned the order numbers o' and o respectively. $r' <_l r$ if and only if $o' < o$. Note that $r' <_l r \Rightarrow go' < go$.

Definition 2: $r' < r$. Let r' and r be *any* two requests that have been assigned the global order numbers go' and go respectively. $r' < r$ if and only if $go' < go$.

We will denote a server s_j, $0 \le j \le 2$, in site k, $0 \le k \le 2$, as s_{jk}, and the assignment of a global order number *go* to a request *r* as (*r*, *go*). Recall that the *unanimity* property of Mencius guarantees that when a server s_{jk} assigns *o* to *r* (through Mencius ordering), every other operative server $s_{j'k}$, $j' \neq j$, will assign or would have already assigned *o* to *r*. So, any assignment (*r*, *go*) originating at site k is already, or would soon be, known to at least two servers in site k.

Definition 3: *Stability* and *global stability* for (*r*, *go*). An assignment (*r*, *go*) is said to be *stable* in site k if it is, or would be, known to at least two servers in site k. (*r*, *go*) is said to be *globally stable* if it is stable in at least two sites.

1) Protocol: Normal Part

This part is presented by assuming that all sites receive client requests nearly at the same (non-zero) rate and outage suspicions (false or correct) do not arise. It carries out three activities: A server s_{jk}

- ensures that a locally generated assignment (*r*, *go*) is globally stabilized; (*Activity 1*)
- enters a globally stable (*r*, *go*) (which may have been generated locally or remotely) into the local copy of the global stream GS_{jk} as the go^{th} entry, $GS_{jk}[go]$, only after all r', $r' <_l r$, have been entered at $GS_{jk}[go']$; (*Activity 2*)
  - more formally, activity 2 maintains the invariant: if $GS_{jk}[go] = r$ then (*r*, *go*) is globally-stable *and* $\forall r' <_l r$: $GS_{jk}[go'] = r'$;
- processes $GS_{jk}[go]$ only after all $GS_{jk}[go'] < GS_{jk}[go]$ are processed. (*Activity 3*)
  - formally, $GS_{jk}[go']$ *not* processed $\Rightarrow \forall\ go > go'$: $GS_{jk}[go]$ *not* processed.

Each server s_{jk} maintains GS_{jk} as above and executes the following five steps to carry out activity 1; the first two steps are executed at the site where (*r*, *go*) is generated and the last three at a remote site.

Step 1.1: When s_{jk} generates (*r*, *go*), it transmits *propose*((*r*, *go*), k) in parallel (using distinct TCP/IP sockets) to all servers in each remote site.

Step 1.2 (a) s_{jk} waits until it is known that the TCP communication in step 1.1 terminates for at least two servers in each remote site; (b) s_{jk} then transmits *propose*((*r*, *go*), k) to each server in the local site k to indicate that (*r*, *go*) is now globally stable.

Since a TCP connection is reliable, any operative destination will receive *propose*((*r*, *go*), k) so long as s_{jk} is operative until its TCP transmission completes. When s_{jk} executes step 1.2(b), an operative remote site is guaranteed to have one correct server having *already* received *propose*((*r*, *go*), k) even if another server in that site crashes soon after receiving *propose*((*r*, *go*), k) from s_{jk}. Steps 1.3-1.5 below ensure that (*r*, *go*) gets stable at any operative remote site. Note also that if s_{jk} crashes or if site k suffers an outage before s_{jk} can complete step 1.2(a), *propose*((*r*, *go*), k) from s_{jk} will not be sent within site k.

Optimization: A server s_{jk} executes Steps 1.1 and 1.2 only when a timeout of random duration expires, and only for those requests *r* that s_{jk} locally-ordered (using Mencius) during the timeout duration and for which a *propose*((*r*, *go*), k) has not been received from another server in the local site. This arrangement reduces the likelihood of more than one server attempting to globally stabilize the same (*r*, *go*). Timeout durations may be exponentially distributed with mean = 3 times the upper bound estimates of intra-site transmission delays observed, as Mencius takes at most 3 sequential intra-site communication to order a local request. (Mencius maintains an estimate of the upper bound on delays due to its ◊P assumption).

A server s_{jk} executes the following steps to globally stabilize (*r*, *go*) proposed from a remote site k':

Step 1.3: When s_{jk} receives *propose*((*r*, *go*), k') from a server of (remote) site k', it forwards the proposal to all other servers in the local site.

Step 1.4: When $s_{j'k}$ receives *propose*((*r*, *go*), k') from a server of local site, it forwards the proposal to all other servers in the local site, if it has not already done so.

Step 1.5: A server of site k concludes that (*r*, *go*) of site k' is stabilized in the local site (and also globally), if it (i) has sent *propose*((*r*, *go*), k') within the local site and (ii) has received a *propose*((*r*, *go*), k') from another local server.

Observations: When a request *r* is locally ordered in site k, the assignment of *go* to *r* and the global-stabilization of (*r*, *go*) are not affected by how fast or how slow the requests are being submitted in other sites. More specifically, a server s_{jk} does not have to await any acknowledgement (above the TCP level) from the remote servers for the *propose*((*r*, *go*), k) it transmitted in step 1.1.

Similarly, entering a globally stabilized (*r*, *go*) into GS depends only on whether earlier *local* requests have been globally stabilized (see *Activity* 2). However, processing a local request *does depend* on how fast/slow remote requests being globally stabilized (see *Activity* 3).

2) Dealing with idle or non-busy sites.

When s_{jk} receives *propose*((*r*', *go*'), k') from a server of (remote) site k' for a non-null *r*', the *desirable* situation would be that s_{jk} or some other local server has already assigned (*r*, *go*) or is likely to do so shortly such that $go \ge go' - k' + k$, i.e., $go - k \ge go' - k'$. Let o_{mx} denote the largest Mencius instance initiated within site k at the time when s_{jk} receives *propose*((*r*', *go*'), k'). For the desirable situation to

prevail we should have $o_{mx} \geq (go' - k')/3$, given that o is related to go in site k as $go = (3 \times o + k)$.

Recall that server s_{jk} maintains counter C_{jk} to hold the instance number to be used next in Mencius ordering (see subsection IIIA); also that C_{jk} is incremented by 3. To ensure that $o_{mx} \geq (go' - k')/3$, s_{jk} should have $(C_{jk} - 3) \geq (go' - k')/3$; if it does not, it initiates new instances using null requests until $C_{jk} \geq (go' - k')/3) + 3$.

3) Dealing with an outage detection.

It consists of 5 steps: (i) flushing all proposals received directly from the detected site, say, k', for stabilization at the local site; (ii) estimating the largest *go* of all *propose*((*r*, *go*), k') which might have been considered as globally stable within k' just before outage; (iii) agreeing on the estimates of (ii) and isolating k'; (iv) continued isolation of k', if necessary, and (v) re-admitting k' after recovery.

Step 2.1: When server s_{jk} detects an outage of site k', it acts on any *propose*((*r*, *go*), k') that it has received directly from site k', but has not yet handled as per Step 1.3. After having handled all such *propose*((*r*, *go*), k'), s_{jk} sends a 'flush' message to other local servers. Step 2.1 ends when s_{jk} (i) has sent its flush message and received one from another local server, and (ii) knows that all *propose*((*r*, *go*), k') that it received directly from k' are stable in site k. (Step 1.5 stipulates how s_{jk} can deduce whether a *propose*((*r*, *go*), k') is stable in site k; also, recall that whenever another server in site k, say, $s_{j'k}$ receives a *propose*((*r*, *go*), k') from s_{jk}, it deals with the received proposal as per Step 1.4.)

Remark 1: Suppose that the assignment (*r*, *go*) was generated at site k' and was regarded to be globally stable by servers of k' prior to site outage. Due to Step 1.2, at least two servers in site k or at least one correct server if site k has a crashed server would have received *propose*((*r*, *go*), k') prior to outage detection as the latter is assumed to be perfect. Step 2.1 is not completed until s_{jk} receives a flush message from another server in site k; therefore, if s_{jk} has not received *propose*((*r*, *go*), k') directly from site k', it must receive that proposal from a local server while awaiting to receive a flush message. When s_{jk} receives the proposal for the first time from a local server, it executes Step 1.4 and also deduces that *propose*((*r*, *go*), k') is stable in site k.

Step 2.2: Server s_{jk} generates a special request called *site_revoke*, denoted as, *SR*. The request has two fields *SR*.site and *SR.G* which are respectively set to k' and the largest (*r*', *go*) from site k' which s_{jk} knows to be stable in site k. s_{jk} then initiates a Mencius instance to order *SR*.

Remark 2: If a server of site k' has transmitted *propose*((*r*, *go*), k'), say, to just one server in one remote site, say, k, prior to the outage of site k', then no server in site k' would have regarded (*r*, *go*) as globally stable and this fact cannot be deduced accurately by remote servers. When site k has three operative servers, they may deduce the (local) stabilization of *propose*((*r*, *go*), k') either before or after forming their respective *SR*. Hence, *SR.G* of two servers may not be the same and the difference refers to the proposals *not* considered as globally stable in the failed site. Each of these proposals can either be treated as null or be made globally stable and processed prior to isolating site k'; but the decision needs to be the same for all servers which is accomplished by intra-site ordering of *SR* requests.

Step 2.3: Server s_{jk} waits until its GS_{jk} has at least two *SR* entries from each of the two operative sites. Let SR^1_1 and SR^1_2 be the first two entries in GS_{jk} originating from one given site and entered in that order; similarly, let SR^2_1 and SR^2_2 be the first two entries from the other site.

G_{jk} = maximum of $\{SR^1_1, SR^1_2, SR^2_1, SR^2_2\}$; all requests from site k' in GS_{jk} are regarded to be null for all $go \in [G_{jk}+3, G_{jk}+w]$, where w is a predefined window length.

Step 2.4: When (*r*, $go > G_{jk}+w$) is globally stabilized, steps 2.2 and 2.3 are repeated with *SR*.site=k' and $SR.G=G_{jk}+w$, if site k' has not yet recovered and the service state of its servers not been updated until then.

Step 2.5: When server s_{jk} learns that site k' has recovered and the service state of its servers has been updated, it generates a special request called *site_admit*, denoted as *SA*, with fields *SA*.site and *SA.G* set to k' and the current value of G_{jk}, respectively; steps 2.2 and 2.3 are executed with *SA* (instead of *SR*) and G_{jk} computed at the end of step 2.3 is the first *go* to be used by the servers of the re-instated site k'.

IV. PERFORMANCE COMPARISON

We compare the performance of 2-Stage ordering and of Mencius in the default approach when site outages and crashes do not occur. The comparison metrics are 3-fold:

Throughput determines the average number of requests ordered by a server per second. (Note: ordering refers to global ordering in the 2-Stage approach.) Recall that all servers order all requests irrespective of the submission site. Hence, the ideal throughput value should be the sum of the average rates at which requests are being submitted in each site. A shortfall indicates server saturation.

Minimum latency denotes the time elapsed between the instance when a request is submitted and the instance when the *first server* orders that request; similarly, **maximum latency** refers to the duration required for the *last operative server* in the system to order that request. Note that the first and last servers could be different for different requests.

To conduct experiments, both protocols are implemented in Java and each server is a 1.86 GHz Intel Core (TM) 2 PC with 2.0 GB memory running Fedora 12. The default approach used 3 machines as in Fig 1; and in 2-stage ordering, a group of 3 machines forms a site as in Fig 2. All the machines were part of a single cluster (connected by a LAN) and the inter-site communication was both simulated and emulated, leading to three classes of experiments.

Class I – In this class of experiments, a LAN that physically connects all servers and hence LAN *is* the Internet. Class I experiments attribute the smallest possible values (approximately 3 milliseconds) to communication delays, thus exposing how much a protocol can gain in performance due to short delays over the 'Internet'.

Class II – Internet delays are simulated using DummyNet [16]. They were fixed at 25 milliseconds (ms), 50 ms and 100 ms.

Class III – Internet delays are emulated using the traces taken in a real experiment [5, 11]; magnitudes and variations of delays between different sites are different.

TABLE II. THROUGHPUT IN CLASS III EXPERIMENTS.

Throughput		
1/λ ms	2-Stage	Default
20	138	120
38	76	70
75	39	37
150	20	19
1000	3	3
10000	0.3	0.3

Request Arrivals: In all our experiments, all three sites are equally loaded; the inter-arrival intervals between successive requests at a *given site* are uniformly distributed with a mean that is referred to as the *average inter-arrival interval* per site and is denoted as (1/λ); the overall request submission rate is therefore 3λ, considering all 3 sites.

1/λ is varied from 20ms (heavier load), 38ms, 75ms, 150, 1000 ms and 10000ms (lighter load). For these values, the system (of 3 sites) receives requests at the rate of 150, 80, 40, 20, 3 and 0.3 requests per second, respectively.

An experiment consists of 30000 requests being submitted at each site, with an exception of only 10000 requests when 1/λ = 10000 ms. In the 2-stage ordering, requests at a given site are equally distributed to the three servers. An experiment was repeated 3 times when 1/λ = 10000ms, and 5 times for all other 1/λ; the results we report next are the averages over these experiments.

Finally, it would be useful to view the following results together with the rationale behind conducting the three classes of experiments: Class I represents an extreme case, perhaps a *futuristic* scenario, where the Internet traffic becomes as fast as today's LANs. Class III represents the most *practical* scenario in which the Internet delays observed are being used. Class II results aim to show *the trend* in throughput and latencies as the internet delays are varied.

A. Results

1) Throughput

Table II shows throughputs for each protocol in Class III experiments. For 1/λ = 20 ms, the throughput observed is below the ideal (150 per second), and the drop is larger for the default approach. This indicates that 1/λ=20 is the smallest we could have used. As 1/λ increases, the throughput approaches to the ideal value. This trend is present in all experiments we carried out and hence we show the relative performance of the protocols in Table III.

We define Gain in throughput, Gain_T, as: $(T_{2S} - T_D)/T_D$, where T_{2S} and T_D are the throughput values observed in 2-stage ordering and in the default approach, respectively. Table III presents Gain_T (in %) all of which are non-negative, indicating that $T_{2S} \geq T_D$. This is not surprising as there are 3 servers per site in 2-Stage ordering. On the other hand, it also shows the advantages that these extra servers bring. Further, referring to Table II, the default approach starts saturating for 1/λ = 150 ms which amounts to the overall submission rate of just 20 requests per second, where as the 2-stage ordering shows a similar trend at 40 requests per second. We also observe that gain in Class I and Class III experiments are similar, indicating that the delays over the Internet do not affect the throughput much.

TABLE III. GAIN_T *VERSUS* 1/λ.

Gain_T in % 1/λms	**Class I**	**Class II**			**Class III**
		25ms	**50ms**	**100ms**	
20	13	12	12	11	15
38	7	7	7	6	9
75	3	3	3	3	5
150	5	5	5	5	5
1000	0	0	0	0	0
10000	0	0	0	0	0

2) Latency

TABLE IV. MAXIMUM AND MINIMUM LATENCIES (CLASS III)

1/λ ms	2-Stage Latency in ms max (min)	Default Latency in ms max (min)
20	1614 (1012)	2697 (2191)
38	1412 (874)	2429 (2050)
75	1210 (781)	2060 (1660)
150	1036 (641)	1870 (1550)
1000	820 (376)	1569 (1065)
10000	925 (374)	1400 (1038)

Latencies observed in Class III experiments (see Table IV) lead to two interesting observations. The maximum latencies in 2-Stage ordering are smaller than the minimum latencies in the default approach. This shows that the traditional order protocols that involve multiple, sequential inter-site communications tend to yield large latencies for inter-cloud replication. Secondly, the differences between maximum and minimum latencies in 2-Stage ordering are larger and are more pronounced for larger 1/λ values (i.e., at lower submission rates). This is attributed to the phenomenon whereby when a local request *r* is locally ordered by Mencius, every earlier $r' < r$ is either globally stabilized already or well in the process of becoming stabilized. This reduces the ordering delay for such *r*.

Similar to Gain_T, we define Gain_L for latencies to compare the two approaches. Gain_L, as: $(L_{2S} - L_D)/L_D$, where L_{2S} and L_D are the latency values observed in 2-Stage ordering and in the default approach, respectively. Gain_L_max and Gain_L_min denote Gain_L when maximum and minimum latencies are considered. They are presented in Tables V and VI. Note that a negative Gain indicates that the 2-Stage ordering produces smaller latencies

and hence performs faster; also that the 2_stage ordering gets faster as the delays over the Internet increase.

TABLE V. GAIN_L_MAX *VERSUS* 1/λ.

%GainL_max ---------- 1/λ	Class I	Class II			Class III
		25ms	50ms	100ms	
20	32	-19	-38	-53	-40
38	23	-19	-41	-55	-42
75	22	-27	-43	-56	-41
150	26	-24	-47	-54	-45
1000	24	-14	-38	-48	-48
10000	17	-11	-41	-47	-34

TABLE VI. GAIN_L_MIN *VERSUS* 1/λ.

%GainL_min ---------- 1/λ	Class I	Class II			Class III
		25ms	50ms	100ms	
20	30	-26	-42	-48	-54
38	21	-32	-43	-60	-57
75	16	-33	-45	-57	-53
150	39	-33	-48	-56	-59
1000	60	-13	-31	-48	-65
10000	69	-20	-37	-51	-64

V. CONCLUSIONS

We have proposed an approach for replicating a critical service on N, N=3, outage-independent clouds to tolerate a single cloud outage. It involves replicating the service not only across clouds but also within each cloud. For intra-cloud replication, we have chosen to use state machine replication and explained the rationale behind this choice. In this form of replication, service responsiveness is predominantly dependent on the overhead imposed by the order protocol. We have addressed this concern by proposing a two-stage ordering protocol and compared its overhead with that of Mencius [13] using which, as we have described earlier, an outage-tolerant replication can be readily implemented as well. Experiments carried out confirm that the proposed approach makes the best use of shorter intra-cloud communication delays, leading to substantial throughput gains and reduced latencies. Finally, we note that our approach can also work for N=2 (with a minimum of 6 servers), while the default approach requires N≥3.

VI. REFERENCES

[1] Amazon.com, "Summary of the Amazon EC2 and Amazon RDS service interruption in the US East Region", http://aws.amazon.com/message/65648/, April 2011.

[2] C. Basescu, C. Cachin, I. Eyal, R. Haas, A. Sorniotti, M. Vukolic, and I. Zachevsky, "Robust data sharing with key-value stores", Proc. 42nd IEEE/IFIP International Conf. Dependable Systems and Networks (DSN2012), IEEE Press, June 2012.

[3] M.Biely, Z. Milosevic, N.Santos and A. Schiper, "S-Paxos: Offloading the leader for high throughput state machine replication", in IEEE Symp. on Reliable Distributed Systems (SRDS), Oct 2012.

[4] T.D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems", in JACM, 43(2), pp.225-267, March 1996.

[5] Y. Chen, A.Romanovsky, A. Gorbenko, V. Kharchenko, S. Mamutov, O. Tarasyuk, "Benchmarking Dependability of a System Biology Application", Proc. 14th IEEE Conference on Engineering of Complex Computer Systems, pp.146-153, 2009.

[6] G. Ferro, "Complex Systems Have Complex Failures. That's Cloud Computing", in Ethereal Mind, http://etherealmind.com/complex-systems-complex-failures-cloud-computing/, April 2011.

[7] M.J. Fischer and N.A.Lynch, "A lower bound for the time to assure interactive consistency", Information Processing Letters, 14(4): 183-186, June 1982.

[8] Gmailblog, "More on Today's Today's Gmail Issue", http://gmailblog.blogspot.com/2009/09/more-on-todays-gmail-issue.html , Sept 2009.

[9] Gmailblog, "Update on today's Gmail Outage", http://gmailblog.blogspot.com/2009/02/update-on-todays-gmail-outage.html , Feb 2009.

[10] Google. "Post-mortem for February 24th, 2010 Outage", https://groups.google.com/forum/?fromgroups#!topic/google-appengine/p2QKJ0OSLc8, Feb 2010.

[11] A. Gorbenko, V. Kharchenko, S. Mamutov, Y. Chen, O. Tarasyuk and A.Romanovsky, "Exploring Uncertainty of Delays as a Factor in End-to-End Cloud Response Time", Proc. 9th European Dependable Computing Conference, pp. 185-190, May 2012.

[12] H.S. Gunawi, T. Do, J.M. Hellerstein, I. Stoica, D. Borthakur and J. Robbins, "Failure as a Service (FaaS): A Cloud Service for Large-Scale, Online Failure Drills", Technical Report, Univ. of Califorina at Berkeley, http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-87.html, July 2011.

[13] Y. Mao, F.P. Janqueira and K Marzullo, "Mencius: Building Efficient Replicated State Machines for WANs", in OSDI, 2008, pp. 369-384. http://www.usenix.org/events/osdi08/tech/full_papers/mao/mao.pdf

[14] P. Marandi, M. Primi and F. Pedone, "High Performance State Machine Replication", Proc. IEEE International Conference on Dependable Systems and Networks (DSN), 2011, pp. 454-465.

[15] PayPal. "Details on PayPal Site Outage Today", https://www.thepaypalblog.com/2010/10/details-on-paypals-site-outage-today/, October 2010.

[16] L. Rizzo. "Dummynet: a simple approach to the evaluation of network protocols". In *SIGCOMM Computer Communications Review*, 27(1):31– 41, 1997.

[17] F.B. Schneider. "Implementing fault-tolerant services using the state machine approach: A tutorial", in *ACM Computing Surveys*, 22(4) pp. 299–319, Dec. 1990.

[18] F.B. Schneider. "Replication management using the state-machine approach". In *Distributed Systems* (ed. S. Mullender), ACM Press, pp. 169–197, 1993.

[19] Skype.com. "CIO Update: Post-mortem On the Skype Outage", http://blogs.skype.com/en/2010/12/cio_update.html, 2010.

[20] Wikimedia.com. "Wikimedia Technical blog: Global outage", http://blog.wikimedia.org/2010/03/24/global-outage-cooling-failure-and-dns/, March 2010.