Adaptive and Dynamic Funnel Replication in Clouds

Guy Laden
IBM Research Haifa, Israel
laden@il.ibm.com

Roie Melamed
IBM Research Haifa, Israel
roiem@il.ibm.com

Ymir Vigfusson
School of CS, Reykjavik University
ymirv@ru.is

ABSTRACT

We consider the problem of strongly consistent replication in a multi data center cloud setting. This environment is characterized by high latency communication between data centers, significant fluctuations in the performance of seemingly identical virtual machines (VMs) and temporary disconnects of data centers from the rest of the cloud. In this paper we introduce the adaptive and dynamic Funnel Replication (FR) protocol that is designed to achieve high throughout and low latency for reads, to accommodate arbitrary latency/throughput tradeoffs for writes, to maximize performance in the face of VM performance variations and to provide high availability for read requests despite network partitions. FR is based on the idea of flexible write dissemination topologies which enables it to achieve, per message, the desired tradeoff between latency and throughput, depending on the message size, the observed network conditions, and the importance of latency as indicated by the client. We demonstrate the benefits of flexible dissemination topologies and show that in a cloud setting with N identical replicas FR can improve the write latency up to a factor of $\frac{N}{2}$ for $N \geq 2$ compared to the notable chain replication (CR) protocol at the expense of a slight decrease in the write throughput. In a setting with potentially high variability in the performance of replicas, e.g., as in Amazon EC2, FR can achieve throughput up to a factor of 16 higher than CR while also improving latency. FR does this by adopting a topology that consists of concurrent disjoint data replication paths so that load on high throughput paths is adaptively increased while load on congested replicas is reduced.

1. INTRODUCTION

Replication of objects in several locations improves the reliability, scalability and availability of web services, but at the price of increased storage and synchronization overhead [25, 16, 4]. In particular, clouds—typically composed of geographically disparate data centers [2]—use data replication to support i) geographic redundancy for fault-tolerance; ii) low access latency to customers; and iii) horizontal scalability. However, Brewer's famous CAP theorem [5] proclaims that *strong consistency* of data is at odds with availability in multi data center replication because network partitions in WANs are unavoidable. This is an unfortunate trade-off for cloud companies because availability is fundamental to their business (as witnessed, for instance, by a recent outage of Amazon's EC2). Although many cloud providers have chosen to weaken the consistency guarantees within their systems to guarantee availability [10], strongly consistent replication is fundamental to critical application domains such as e-commerce, including online banking and stock trading [17], and is also often adopted for certain parts of web applications and sites such as password/account management [23]. Developers also find it substantially easier to deal with strong consistency guarantees, as weaker semantics require applications to resolve or tolerate conflicts [10, 2] which complicates application code and prolongs the development time. Striking the ideal balance between strongly consistent replication and service availability has become a hot topic in the systems and database communities.

Our approach is to sacrifice only *write* availability during wide-area network partitions while supporting strong consistency. This design choice is motivated by the conventional wisdom that read-heavy workloads are dominant in cloud settings [8]. In general, *ROWAA (read one, write all available)* replication protocols, e.g., a recent variation of the chain replication (CR) protocol [26] called CRAQ [24], allow data to be read from a *single* local replica while still supporting strong consistency even under a network partition across data centers. This assumes there is no local network partition, which happens less frequently than a network partition in the WAN. Jimenéz-Peris and Patiño-Martínez [16] argue that ROWAA approaches provide superior scaling of availability to quorum techniques, e.g., Paxos [19], and claim that the availability of ROWAA protocols improves exponentially with more replicas [26].

In addition to higher read availability, the ability to read from a single local replica allows a CR protocol to achieve lower latency and message complexity for read requests compared to quorum-based protocols. A CR protocol also incurs lower message complexity for write requests compared to quorum-based replication protocols, including recent variations of Paxos, e.g., [20, 13, 18, 22]. The lower message complexity incurred by CR allows it to achieve higher throughput compared to non-ROWAA protocols in a setting with identical replicas, since more network bandwidth is available to transmit payload and fewer messages need to be processed per request [12]. It has been argued that ROWAA approaches exhibit better throughput than the best known quorum systems (except for nearly write-only workloads), and are consequently a better choice for replication in most real settings [16, 26]. CR also simplifies recovery from failures compared to quorum-based replication protocols, since in CR each replica is at least as knowledgeable about committed write requests as its predecessor replica in the chain. Hence, in CR upon a failure of the tail (leader)

LADIS 2011 Seattle, Washington

replica, the predecessor replica can be *immediately* take on the tail role without running any leader election or state transfer protocols. In contrast, the failure of a leader in a quorum-based replication protocol such as Paxos [19] requires a new leader to be elected and a complex state transfer protocol to be run (see Section 5). Finally, CR is simpler to understand, easier to develop and test, and its correctness can be easier to verify compared to Paxos. These properties are crucial for industrial deployments [8].

However, CR also has two significant limitations that do not exist in quorum-based protocols such as Paxos: i) high write latency that grows linearly with the number of replicas; and ii) the write throughput and latency are determined by the slowest replica. We note that CR was designed for high-throughput in a LAN setting with identical replicas [26], and in such a setting the two aforementioned limitations are less significant. However, in a cloud setting, where services comprise multiple geographically disparate data centers, these two limitations are substantial, since the latency across data centers is high and in cloud computing platforms such as Amazon EC2 the CPU, disk, I/O, and network performance of seemingly identical virtual machine instances significantly differ from each other [11, 3]. We note that the chain topology of CR, in spite of its benefits described earlier, is ultimately responsible for these limitations in a cloud setting.

Our contributions. In this paper, we report on the adaptive and dynamic *Funnel Replication (FR)* protocol designed to eliminate the two aforementioned limitations of CR. In order to support strong consistency in FR a certain replica denoted as the *tail* orders and commits write requests that have been received by all replicas. The order of committing write requests at the tail is determined using an efficient *vector clock algorithm* that supports *deterministic total ordering* of all write requests, i.e., this total ordering can be *reconstructed* upon a tail failure. Therefore, as opposed to previous CR protocols and recent variations of Paxos [22], in FR *each* replica can handle a client write request (such a replica is denoted as a *head* replica) without the need to synchronize concurrent writes to a given object (which can delay the commit of a write request up to two communication steps in recent variations of Paxos [22]) and with minimally delaying the commit of concurrent writes as there are at most N concurrent writes to a given object with the same vector clock.

Each head replica can propagate a write request using *any* replication topology that includes all the replicas. Two examples for such a topology are a combination of chains (see Figure 1) or a star (see Figure 2). A replica that receives an uncommitted write request either forwards it to other replicas or sends a short acknowledgment message to the tail replica, depending on the replication topology. Once a request is committed by the tail, it propagates a notification of this to all replicas using a reverse chain topology, which allows easy recovery and enables all replicas to handle read requests.

As opposed to CR which is optimized for throughput, the flexibility in write request propagation topology allows FR to *dynamically (per message)* achieve the desired tradeoff between latency and throughput, depending on various criteria such as object size, the observed network conditions and the importance of latency as indicated by the client. Notably, in a multi data center cloud FR can employ D *parallel* chains (see Figure 1), one in each of the D data centers, in order to achieve write latency that includes only two WAN communication hops from each data center (since every replica can handle a write request). In contrast, in this setting, the write latency of CR protocols includes either $D-1$ or D WAN link delays if the client and the head replica are in the same and different data centers, respectively. In these scenarios, FR with a parallel chain topology also incurs the sending of additional D brief acknowledgement messages when compared to CR, and hence in a setting in which all the replicas are identical the lower write latency achieved by FR comes at the expense of a minor decrease in the write throughput, though *at any time* FR can revert to a chain topology in order to achieve identical latency and throughput as CR. In Section 4, we experimentally show that FR can reduce CR's write latency by up to a factor of 3.5 when D is 7. Additionally, in our experiments the latency of FR was substantially lower than CR (up to a factor of 2.5) in a single data center setting, while throughput was minimally impacted. In the latter setting, the size of the replicated object is assumed to be up to 5KB, which is larger than a typical object in Bigtable [9] or stock ticker packets in financial markets.

Apart from high latency between data centers, public clouds, e.g., Amazon EC2, and virtualized environments in general are characterized by potentially large fluctuations in resource availability and performance of VMs [11, 3]. Choosing the appropriate write dissemination topology for this environment can have a large impact on replication performance. In Section 4 we show that by employing multiple head replicas with a star topology from each FR can achieve up to a factor of 16 higher throughput than CR when the outgoing bandwidth is congested at some replica, without foregoing the aforementioned latency benefits. This is since FR is able to adaptively shift load on outgoing bandwidth links from the most congested replicas to ones with more available bandwidth whereas CR's throughput is constrained by the slowest replica in the single data path topology (chain). The resiliency of topologies with multiple disjoint data paths (as the one mentioned above) to slow nodes has been observed in multicast systems [7].

2. SYSTEM MODEL

We consider a system composed initially of an ordered view $V_0 = \{r_0, r_1, ..., r_{N-1}\}$ of $N \geq 1$ replicas, where the replica with the highest index (r_{N-1}) is denoted as the *tail* replica. For a replica r_i, we denote replicas r_{i-1} and r_{i+1} as its *predecessor* and *successor* replicas, respectively.

As in previous CR protocols (e.g., [24, 26]), we assume a highly available *master* service that detects failures of replicas and informs all replicas of the new view V_{i+1} upon a failure of a replica $\in V_i$. Such a master service can be implemented using a consensus-based locking service [19, 24, 26] like ZooKeeper [15] or Chubby [6]. The master service is required only upon bootstrap, informing the replicas about V_0 and upon a replica failure/addition/removal, and it is not in the critical path of read and write requests. Thus, a single master service can be used by many cloud services e.g., many FR replication groups [6]. We note that the replicas can themselves implement the master service in parallel to running the FR protocol.

Like previous CR protocols [24, 26] and also for simplicity we assume the fail-stop model and reliable and FIFO links. In practice, we can leverage a tightly bound clock synchronization protocol, e.g., [21], and a lease protocol similar to the "master lease" described in [8] in order to allow FR to assume the less restrictive crash failure model. Finally, FR supports the strong consistency semantics known as *linearizability* [14] that provides the guarantee that i) all read and write operations to an object are executed in some sequential order; and ii) a read operation to an object always returns the *latest* written value. These semantics are easy to understand and simplify the design of applications.

3. THE FR PROTOCOL

We now briefly describe FR's *write* and *read* protocols and how failures are handled. Due to space limitations we do not describe

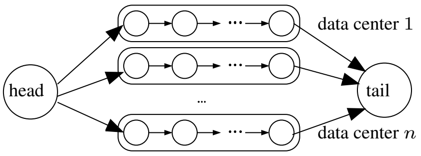


Figure 1: Multi data center topology.

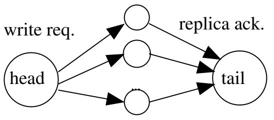


Figure 2: Star topology achieves optimal write latency.

replica addition or deletion operations.

The write protocol. FR retains the notion of a single tail replica from CR, however FR is not limited to a chain dissemination topology and can accommodate *any* topology for propagating a write request to all replicas. In addition to propagating the request itself it is necessary in FR to propagate vector clock meta-data for it (described below) from all replicas towards the tail replica. This meta-data is used by the tail to determine the ordering of each request. Depending on the dissemination topology, this meta-data may be sent to the tail using special *acknowledgement (ack) messages* containing meta-data only (no user payload), or it can be piggybacked onto the write requests as they are sent between the replicas. Either way, write requests and ack messages are *funneled* through the replicas until reaching the tail.

Write requests in FR are received from clients at so-called *head* replicas which forward the requests to other replicas according to the dissemination topology. As opposed to CR in which only a single replica can serve as the head replica, in FR every replica may function as a head in addition to assisting other heads in forwarding write requests.

An example dissemination topology from a certain head replica is depicted in Figure 2. The head replica forwards the write request to all other replicas in the first message round and all the replicas send acknowledgements to the tail replica in the second message round. An additional example is illustrated in Figure 1. In this "parallel chains" topology, the head replica for a request forwards it to the first replica in each of the D chains, and each such replica forwards the request to the next replica in the local chain until the last replica in each chain sends an ack message to the tail aggregating meta-data from all the replicas in the local chain. In this example, there are D acks from the last replica in each of the D chains. This topology strikes a particular balance between latency and throughput by employing chains within a data center but not between data centers due to the latency considerations discussed above.

We note that the dissemination topology may be *dynamically* determined by each replica along the dissemination path. FR's ability to support any dissemination topology and head combination allows it to achieve perfect load-balancing, as it can *precisely* and *dynamically* determine the load incurred on each replica. As we show in Section 4, FR's dissemination flexibility also allows it to work around slow replicas by incurring more load on fast replicas.

After the tail commits a write request, it sends an acknowledgement to the client and also propagates notification of the commit along the *chain* of predecessors towards r_0. This notification enables handling read requests from every replica as discussed in the read protocol subsection.

Write request ordering. In order to provide strong consistency, write requests to the same object should be ordered consistently across failures. In CR protocols the (single) head replica determines the request ordering and all other replicas receive the requests in that order due to the chain topology. Since FR supports multiple heads and flexible propagation an alternative solution is needed. We note that the ordering can not be arbitrarily chosen by the tail replica because if it fails the new tail replica must be able to re-create an identical ordering.

Our solution is to introduce a vector clock for write requests and define an associated total order. This order is used by the tail to determine when a request may be committed. On failure of the tail, the new one uses the same information and commit policy so that the same ordering is chosen.

We associate a vector clock with every write request as follows. Each replica maintains an integer counter that is incremented by one when a new request arrives from a client. We define the vector clock of a request req to be the vector $\langle vc_0, ..., vc_{N-1}\rangle$ where vc_i is the value of the counter at replica r_i at the time of receiving req. The counters are propagated along with data and ack messages so that the tail eventually has the full vector clock for every request. Every replica also maintains a local state where it saves the counter it associated with every write request as long as that request remains uncommitted. After failure, this state is sent to the new tail so that it too has all the relevant information. We note that each replica except the tail only deals with tracking and sending its own counters and does not need to be aware of the counters of other replicas.

Given this definition of a vector clock we can view a write request as a tuple (key, value, vector clock, originating head replica). Note that a request can be uniquely identified by the head at which it was received (originating head) and the value of the counter given to that request at that head.

We define a total order on requests: request a *commits-before* request b if and only if: for each index i in the vector clocks of a and b, $a.vc_i \leq b.vc_i$ *and* there exists at least a single index j such that $a.vc_j < b.vc_j$ *or* $a.originating\_head < b.originating\_head$.

Using this total order the following policy is used at the tail to determine what requests to commit and when. The tail maintains a set of uncommitted write requests for every object and adds entries or updates vector clocks in exiting entries as data/ack messages are received. An uncommitted write request req is committed by the tail as soon as the following conditions hold: i) The request data has been received by the tail; and ii) The request vector clock has been fully determined, i.e., the counter assigned to the request by every one of the replicas is known to the tail; and iii) Of all outstanding requests for this object, the request is earliest in the *commits-before* relation.

Regarding condition iii), it is important to take into consideration requests for which not all the vector clock counters have been received. When comparing a request a in which all counters in the vector clock are known to request b that has some unknown counters it is possible to conclude that a commits-before b even before all counters in b are known if for all indexes i in the vector clocks which are known in both a and b, $a.vc_i \leq b.vc_i$ *and* either there exists at least a single index j such that $a.vc_j < b.vc_j$ *or* $a.originating\_head < b.originating\_head$. This is due to the assumed FIFO property of the network links which ensures that if the tail has received a counter from a replica, then it is not possible to later receive a smaller valued counter.

Since the tail delays committing of some requests until the above conditions hold for them it is important to note that once the vector clock for a write request is fully determined by the tail, at most $N-1$ write requests may be committed before the delayed write is committed. Finally, we claim but do not prove that after committing a write request req, the tail will never later receive a request that should have been committed before req.

The read protocol. FR's *read* protocol is similar to the "clean (committed)/dirty (uncommitted)" protocol described in [24]. Each replica maintains two tables: *Clean* maps an object's key to its last clean value and *Dirty* maps an object's key to a set containing uncommitted write requests to the object. In a given replica, object obj is clean if and only if the local set of uncommitted write requests associated with obj is empty. Otherwise, the object is dirty. When a write request first arrives at a replica an item is added to the relevant set in the Dirty map. Once notification is received from the tail that the request has been committed the item is removed from the set in the Dirty map and the value in the Clean map is updated to the new one.

At any replica, a read request to a clean object is *locally* handled by sending the client the local clean value. A read request to a dirty object issued to a non-tail replica requires *querying* the tail regarding to the object's last committed value as it appears at the tail replica. The tail replica typically can handle both a read request to a dirty object and a query request, with the exception that after a failure of a tail replica the new tail replica must first resolve all the uncommitted write requests to a given object prior to handling either a read or a query request to this object.

Handling failures. Upon receiving a notification on a failure of a replica r from the master, a replica r_i removes from its local data structures all the acknowledgements received from r. Additionally, if r_i has an uncommitted write request req whose head replica is r, then it waits ic (where c is a constant typically set to the RTT) time units before sending req to all replicas, assuming r_i did not hear the retransmission of req by another replica in the last ic time units.

On notification of failure of a tail, all replicas send to the new tail their meta-data about uncommitted writes, i.e., the counter they associated with each of these write requests. The new tail must wait to receive this information from all replicas before answering any requests. Once this information is received the new tail is able to reach commit decisions that will be consistent with those of the old tail and may start handling requests.

The latency incurred by the above approach for dealing with failures may be improved upon using the *update protocol* outlined below which proactively synchronizes uncommitted writes across the chain so that at a tail failure there is less work to do. Periodically the tail sends its predecessor an update message which contains information about all the uncommitted writes it is aware of. Each replica that receives an update message merges the received information with the local information on uncommitted write requests, updates the local data structures, and forwards the merged information to its predecessor replica. Additionally, if a replica r_i is informed by the update protocol that it is missing an uncommitted write request req that another replica r_j has, then r_i requests req from r_j. If a replica cannot reach its predecessor within a given timeout it sends an update message to its predecessor's predecessor. A replica which did not receive an update message in a long time (but less than the failure detection timeout) initiates one itself. We note that this optional update protocol allows FR to achieve either lower or equal recovery latency compared to CR by bypassing a failed replica, and hence upon a detection of a replica failure by the master in FR uncommitted write requests are expected to have been replicated to more replicas compared to CR, which accelerates the latency of these requests.

4. EVALUATION

In this section, we present an experimental evaluation of the write protocols of FR and CR/CRAQ [24]. We do not evaluate the read protocols since those of FR and CRAQ (a variation of CR that like FR allows all replicas to handle read requests) are comparable, and hence the performance of these two protocols is also expected to be similar, though substantially better than the read performance of CR that allows only a single replica to handle a read request.

We implemented the write protocols of CR and FR in C using the BSD socket API, TCP connections, and non-blocking I/O. The testing environment consists of two hosts running client processes and up to six hosts running replicas. All the machines run Linux 2.6.32, use the default TCP settings, and are connected to the same 100Mbit switch. The hosts have 2.4GHZ to 3.0GHZ Intel Xeon CPUs. The two machines running client processes have two cores and 2GB/4GB of RAM while the rest of the machines have four cores and 16GB/32GB of RAM. In all the experiments CPU and RAM utilization is relatively low and we don't expect the variations between machines to have affected the evaluation.

The FR variant we tested (denoted as FR/STAR) made use of a star topology for write propagation (see Figure 2): the head for a given write request sent it to all replicas, each of which then sent an ack message to the tail. All replicas were head replicas (including the tail) and could accept requests from clients.

Latency experiments. For small messages it is expected that the write latency of CR is inferior to that of FR/STAR, since each message will traverse the chain topology serially compared to the parallel dissemination of FR/STAR. However, for writes of larger messages, which consist of multiple Ethernet packets, a CR implementation could forward data to the next replica in the chain before having processed the entire message, and hence in CR a large message can be concurrently processed by different replicas along the chain. FR/STAR harnesses a different type of concurrency: in FR/STAR if a message transmission time is substantially smaller than the link latency, then a write request forwarded by a given head to all replicas concurrently travels over the network links. However, increasing the message size also increases the transmission time so eventually FR/STAR loses the benefit of this concurrency. For a given network, FR can run in the background latency measurements in order to find the message size starting from which a chain propagation can achieve lower latency than a star propagation. Therefore, FR can always achieve write latency that is better or equal to the write latency achieved by CR. Below, we report on our study investigating the interplay between message size and link delay.

In the WAN latency experiments, we measure the time between receiving a message at the head and committing it at the tail. To avoid clock synchronization issues we ran the head and tail processes on the same physical server in these experiments. Each point of data in the latency experiments represents the average measured latency of a thousand runs, with the top and bottom 5% of the results discarded. The BSD socket option TCP_NODELAY was used in these tests to direct the operating system to immediately send messages instead of buffering small messages. Since there is only one head in CR, a write request from a client not co-located in the same data center incurs an additional WAN link latency to reach the head. This WAN hop is avoided in FR where every replica may be a head. To reflect this advantage of FR we make the assumption that clients are uniformly distributed over data centers and for the measured results of CR we added the average latency entailed by the extra hop, computed as $LL\frac{N-1}{N}$ where LL is the WAN link latency and N is the number of replicas (data centers).

Figure 4(a) and Figure 4(b) show the ratio between the write request latency of CR to FR/STAR in a simulated WAN environment with 5 and 7 replicas, respectively, each residing in a different data center. All WAN links between the data centers incur the same latency which was simulated by inserting an artificial delay (sleep) on receipt of any message (acknowledgement or data) by a replica. The request latency ratio is shown as a function of the WAN link latency. We vary the link latency from 10ms (which we note is also similar to a LAN environment with disk access) to 160ms, and show curves for message sizes ranging from 1KB to 1MB.

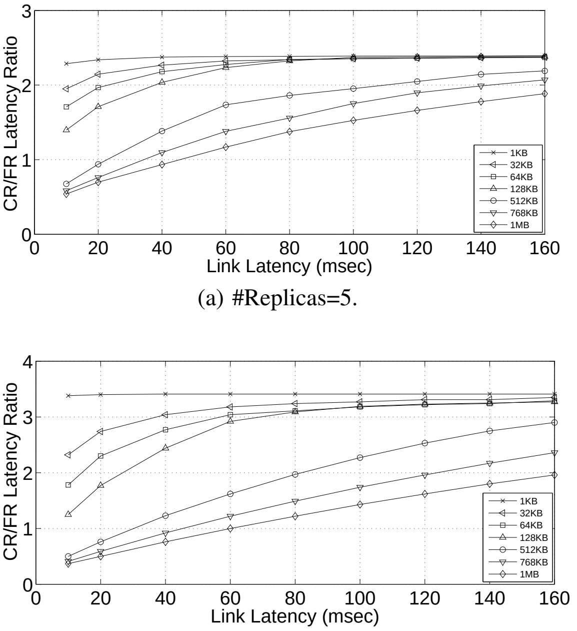
With 7 replicas, for a message with a size up to 128KB, FR improves latency for all the WAN links we simulated. Above 128KB message size and for WAN link delays of up to 60ms CR typically has the advantage over FR/STAR. For WAN link delays above 60ms, FR/STAR dominates CR for all the message sizes we tested. For message sizes up to 1KB which are sent in a single Ethernet packet CR's pipeline provides no concurrency and thus the latency factor is given by the number of links traversed in a chain $(7-1=6)$ divided by the number of message rounds in FR/STAR (2). The resulting latency factor is more than 3 due to the extra hop incurred on CR clients that are not co-located with the single head. With 5 replicas the results are qualitatively similar to the results with 7 replicas, though the performance gap between CR and FR/STAR is smaller since in CR the chain is two WAN hops shorter while FR's latency is unaffected.

The improvements in performance are dramatic for some delay / message-size combinations which indicates that in a WAN environment selecting the appropriate write propagation topology is important. As mentioned above, per message, depending on the message size, the observed network conditions, and the importance of latency as indicated by the client FR can dynamically select the propagation topology in order to always achieve better or identical latency to the latency achieved by CR.

Finally, in a LAN setting our results (omitted for brevity) show that FR can reduce the latency up to 61% over CR for message sizes up to 5KB. In a LAN setting with replicas synchronously performing a disk I/O to persist the data FR/STAR may have an advantage far beyond 5KB since in FR the disk I/O's will be parallelized and not serialized as in CR. The WAN experiments with 10ms link delay (typical disk I/O times) back up this claim.

Throughput experiments. Our throughput experiments investigate how FR/STAR and CR react to reduced outbound network bandwidth at one of the replicas. The motivation for these experiments is a use-case where replicas run within virtual machines (VMs), e.g., in public clouds. In these virtualized environments, it is typical to have a dynamically varying amount of outbound link traffic available for replication as VMs on the same physical machine are brought up/down and react to changing request loads. For example, this has been observed to occur in the Amazon EC2 cloud [11, 3], and indeed cloud computing platforms provide very little guarantees regarding the (network) performance of seemingly identical VM instances [1]. For example, in Amazon EC2 if each VM on a physical host tries to use as much as possible of either CPU, memory, storage, or network resources, then each VM will receive an equal share of that resource, though if a shared resource is under-utilized then the VM will often be able to consume a higher share of that resource while it is available [1]. The study in [11] found that performance spikes in Amazon EC2 occasionally occur with an average duration of one to three minutes.

While any (ROWAA) replication algorithm must transfer all user-data to all the replicas via their inbound links, the use of outbound link bandwidth is more varied across protocols. We note that in the commonly deployed full-duplex switched Ethernet technology, the inbound and outbound links are isolated so that a server's outbound link may be congested independent of the load on its inbound link. In CR, all user-data is sent over all outbound links (except the tail's) and thus we expect overall throughput to approximate the throughput of the most congested outbound link. In FR there are two types of traffic sent over outbound links: acknowledgement messages and user-data. Overall throughput in FR is greatly impacted by acknowledgement message throughput from all replicas, though these messages are typically much smaller than user-data and the amount of bandwidth they require is usually very modest. User-data in FR on the other hand is only sent by a replica when it functions as a head/helps a head replica to forward data (or to answer read requests). In FR/STAR, a head with a congested outbound link will have its throughput as head impacted, though the impact this has on overall system throughput is limited since all other replicas continue at full speed as explained in Section 1. Specifically, if the other heads are able to saturate the network (which occurs in our experiments), then the impact of losing a head will be minor.



(a) #Replicas=5.

(b) #Replicas=7.

Figure 3: CR/FR latency ratio as a function of the WAN link latency.

In all the throughput experiments we employed eighteen clients processes, nine on each of two servers that were not part of the set of servers running the replicas. Each data point in the throughput experiments represents the average number of commits per second performed by the tail replica over a sixty second run. In the FR experiments all replicas function as a head replica for some requests. Each client process is directed at a single head replica and there were at least three client processes directing requests to each head. In our FR implementation replicas prioritized sending ack messages over data requests which is particularly important for the congested replica.

Figure 5 shows the ratio between the write request throughput of CR to FR/STAR for 3 and 6 replicas as a function of the percentage of congestion on the outbound network link of a single replica (the congested replica was always the second replica in the CR/FR chain). The message size for the two curves in Figure 5 is 1KB. As the level of congestion on the outbound link is increased, the relative performance of FR compared to CR improves. FR's throughput remains relatively high and stable for all the tested congestion percentages while CR's throughput is reduced in direct relation to the throughput of the congested link as we explain in Section 1. With no congestion, FR throughput is up to 15% worse than CR's. How-

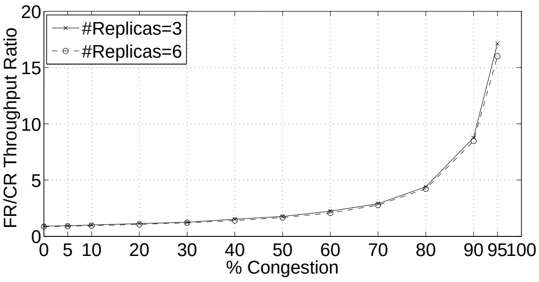


Figure 4: FR/CR throughput ratio as a function of the percentage of congestion on the outbound network link.

ever for congestion percentages of 20% and over FR's outperforms CR, and for high levels of congestion the factor of improvement is more than sixteen. Experiments with larger message sizes yielded almost identical results but for very small messages, e.g., 32 bytes, FR's throughput was significantly impacted by the overhead of the acknowledgement messages.

The above throughput experiments show that FR seamlessly adapts to work around outbound network congestion for a typical performance spike that occasionally occurs in Amazon EC2 [11]. Additionally, the above results highlight the potential of FR to adapt to VM heterogeneity, a direction we plan to investigate in future work. We note that while we focussed on congestion of outbound traffic on head replicas, FR (as opposed to CR) can similarly adapt to other resource bottlenecks on head replicas. An example is CPU usage if a significant computation must be performed by heads prior to replication.

5. RELATED WORK

Quorum-based systems, e.g., Paxos [19], can support strong consistency semantics, can achieve constant latency for both read and write operations, and are resilient to failures, since only a majority of the replicas need to be accessed upon each read or write operation. However, the latter advantage is also a major disadvantage in a mostly read workload, incurring higher read latency and overhead compared chain and primary-backup approaches. In order to overcome this limitation, quorum-based systems often use a "master lease", allowing only the master to handle a read operation locally without accessing the rest of the replicas [2, 8]. The "master lease" can be extended to all replicas while incurring quadratic message complexity and possibly increasing the write latency up to the lease timeout. Recently, several variants of Paxos were proposed e.g., Fast Paxos [20], CoReFP [13], Generalized Paxos [18], and Mencius [22] in order to improve either its latency or/and throughput. These protocols either suffer from collisions or increased message complexity compared to Paxos [22]. Compared to all the aforementioned protocols, by relying on a master service FR can achieve lower read and write latencies as well as lower message complexity. FR also allows easier recovery than quorum-based protocols.

ROWAA protocols, e.g., CR and primary-backup, supports handling of a read operation by single replica and exhibit better throughput than the best known quorum systems (except for nearly write-only workloads), and hence compares favorably to a quorum-based approach in most real settings [26]. In the primary-backup approach the primary replica is a bottleneck since all the replication overhead is induced on it.

Finally, as opposed to previous CR protocols, FR supports any dissemination topology that includes all the replicas, which allows FR to dynamically achieve the desired tradeoff between latency and throughput as well as to seamlessly adapt to work around a slow replica, and hence achieving higher throughput and similar latency in a setting with heterogenous replicas.

6. REFERENCES

[1] Amazon EC2 Instance Types. http://aws.amazon.com/ec2/instance-types/.

[2] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J. M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR 11*.

[3] S. K. Barker and P. Shenoy. Empirical evaluation of latency-sensitive application performance in the cloud. In *MMSys '10*.

[4] K. Birman, D. Malkhi, and R. van Renesse. Virtually synchronous methodology for dynamic service replication. TR MSR-2010-151, Microsoft Research, November 18 2010.

[5] E. A. Brewer. Towards robust distributed systems. (invited talk). In *PODC*, 2000.

[6] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI 06*.

[7] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth multicast in a cooperative environment. In *SOSP*, 2003.

[8] T. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *PODC*, 2007.

[9] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI 06*.

[10] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, 2007.

[11] J. Dejun, G. Pierre, and C.-H. Chi. Resource provisioning of web applications in heterogeneous clouds. In *WebApps '11*, 2011.

[12] M. Demirbas. Metadata: Mencius: building efficient replicated state machines for wide-area-networks (wans), Tuesday, October 19, 2010.

[13] D. Dobre, M. Majuntke, and N. Suri. Contention-resistant fast paxos for wans. TR-TUDDEEDS-11-01-2006, Dept. of CS, Technische Universität Darmstadt.

[14] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12:463–492, July 1990.

[15] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *USENIXATC 10*.

[16] R. Jiménez-Peris, M. Patiño Martínez, G. Alonso, and K. Bettina. Are quorums an alternative for data replication? *ACM Trans. Database Syst.*, 28:257–294, September 2003.

[17] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency rationing in the cloud: pay only when it matters. *Proc. VLDB Endow.*, 2:253–264, 2009.

[18] L. Lamport. Generalized consensus and paxos. Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

[19] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[20] L. Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[21] C. Lenzen, T. Locher, and R. Wattenhofer. Tight bounds for clock synchronization. *Journal of The ACM*, 57:1–42, 2010.

[22] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: building efficient replicated state machines for wans. In *OSDI'08*.

[23] S. E. Perl and M. Seltzer. Data management for internet-scale single-sign-on. In *Proceedings of the 3rd conference on USENIX Workshop on Real, Large Distributed Systems - Volume 3*, 2006.

[24] J. Terrace and M. J. Freedman. Object storage on craq: High-throughput chain replication for read-mostly workloads. In *USENIX*, 2009.

[25] R. van Renesse and R. Guerraoui. Replication techniques for availability. In *Replication: Theory and Practice*, volume Lecture Notes in Comupter Science 5959, pages 19–40, 2010.

[26] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, 2004.