Efficient Implementation of Causal Consistent Transactions in the Cloud

(*extended abstract of the MSc dissertation*)

Taras Lykhenko
Departamento de Engenharia Informtica

Advisor: Professor Luís Eduardo Teixeira Rodrigues

***Abstract*—Distributed key-value storage systems, which offer weak coherence models, have emerged as a strategy for increasing the performance and scalability of systems in the cloud. However, weak consistency makes application development difficult, and there is a keen interest in offering other coherence models that are useful to developers without compromising scalability. The Transactional Causal Consistency (TCC) model is particularly relevant in this context. In this article, we introduce FastCCS, a new algorithm to support TCC in fewer communication rounds than previous work. Experimental results, in which we compare the performance of FastCCS with the performance of other systems proposed in the literature, show that FastCCS can support a rate of higher than the previous systems.**

I. Introduction

In this work, we consider a system in which applications are structured in read and write sequences of operations, which we refer to as transactions, which access data held in a key-value storage system. Concurrent execution of these transactions, without adequate concurrency control mechanisms, may yield different results than desired due to the chaining of multiple distinct transaction operations [1]. Strong coherence models, such as serializability [2], avoid this problem by ensuring that the result of concurrent transaction execution is equivalent to a serial execution of these transactions. Unfortunately, mechanisms that ensure serialization are either blocking or cause transactions to abort and rerun, severely limiting the performance of storage systems [3].

Due to the performance issues inherent in traditional transactional systems, the first key-value storage systems to support cloud application execution gave priority to performance and scalability [4], supporting only very weak coherence models, such as eventual consistency [5]. However, experience has shown that these models make application development difficult [6], so there is a keen interest in finding new coherence models and techniques to support these models that are useful to programmers without compromising scalability [7]. The Transactional Causal Consistency (TCC) model is particularly relevant in this context. This coherence model extends the Causal Coherence model, initially defined for single operations, allowing applications to read multiple objects from a causal snapshot and to perform atomic writing of multiple objects. The relevance of this model stems from the fact that causal consistency is the strongest consistency model that can be supported without compromising system availability in the presence of network failures or partitions [6].

It is worth to underline that, even in a centralized system, the effects of concurrent execution already make it difficult to offer guarantees of consistency, distribution further amplifies this complexity [8]. In particular, if different keys are stored on different nodes, the risk of a client reading incoherent versions is greater, as it is in practice impossible to ensure that the multiple effects of a single transaction are visible at the same time on all servers [8]. However, the ability to deploy, and the opportunity to partition data, and to store different partitions on different servers is crucial to ensuring the performance and scalability of cloud storage systems as it enables different requests to be processed in parallel by different servers, despite the complexity of the distribution [9].

TCC systems use non-blocking algorithms, which use control information (metadata), which is written, read, and stored together with the data, to verify whether the transaction has read from a causal cut. If the transaction has not made a mutually coherent set of readings, the transaction may be required to read new versions of the data (and this may occur more than once). The size of the metadata held by the algorithm has a significant impact on system performance. On the one hand, the larger the metadata volume, the less efficient the system is, as it can consume a not-insignificant fraction of system resources. On the other hand, a larger amount of metadata allows for greater accuracy in identifying the causal cut and can avoid redundant read rounds. Many systems choose to reduce the size of metadata by creating what we call false dependencies; that is, metadata indicates that two operations can be causally related when they are not.

In this article, we introduce Fast Causal Consistent Snapshot (FastCCS), a new algorithm to support TCC. FastCCS explores a new combination between the degree of concurrency the system offers, the size of the metadata, and the number of communication steps required to execute a transaction. In particular, while previous TCC algorithms require the storage

This work was supported by national funds through Fundao para a Ciłncia e a Tecnologia (FCT) as part of the projects with references UID/CEC/50021/2019 and COSMOS (financed by the OE with ref. PTDC/EEICOM/29271/2017 and by Programa Operacional Regional de Lisboa in its FEDER component with ref. Lisbon-01-0145-FEDER-029271).

system to be linearizable (which limits its parallelism) or to perform multiple communication rounds to read a consistent snapshot. In contrast, FastCCS offers TCC on partitioned storage systems using only two communication rounds in the worst case. Besides, when exposed to load profiles dominated by read transactions, FastCCS executes most transactions in just one round of communication. This is achieved by using metadata whose size is linear with the number of partitions in the system, more precisely by using vector clocks that have an entry for each partition of the key-value storage system.

II. Background

A. System Components

Clients connect to a nearby datacenter, and applications strive to handle requests entirely within that datacenter. Inside the datacenter, client requests are served by a front-end web server. Front-ends serve requests by reading and writing data to and from storage tier nodes.

In order to scale, the storage cluster in each datacenter is typically partitioned across 10s to 1000s of machines. As a primitive example, Server 1 might store and serve user profiles for people whose names start with 'A', Server 2 for 'B', and so on. As a storage system, FastCCS' clients are the front-end web servers that issue read and write operations on behalf of the human users. When we say, "a client writes a value", we mean that an application running on a web or application server writes into the storage system.

B. Causal Consistency

The causal dependencies of an operation are determined by happened-before relations ($\rightsquigarrow$) [10], which are defined by three rules:

- **Thread of Execution**. If a and b are two operations executed by the same thread of execution (for instance, by the same client), then $a \rightsquigarrow b$ if a happens before b.
- **Reads From**. If a is an update operation and b is a read operation that reads the value set by a, then $a \rightsquigarrow b$.
- **Transitivity**. If $a \rightsquigarrow b$ and $b \rightsquigarrow c$, then $a \rightsquigarrow c$.

Whether $w_a(k_a)$ and $w_b(k_b)$ are two write operations on the same or two separate keys, k_a and k_b. Let $r_a(k_a)$ and $r_b(k_b)$ be two read operations by the same client, where r_a is executed before r_b and where r_a returns the value written by w_a e r_b returns the value written by w_b. A storage system is said to be causally consistent if in the case that $w_a \rightsquigarrow w_b$ there is no write w'_b on the k_b key such that $w_b \rightsquigarrow w'_b \rightsquigarrow w_a$.

C. Transactional Causal Consistency

Causal consistency is defined for individual operations, regardless of how they are related, allowing sequences of read and write operations which results may not be as expected by programmers. Consider for example two write transactions $T^1 = w^1_a(k_a), w^1_b(k_b)$ and $T^2 = w^2_a(k_a), w^2_b(k_b)$ where $T^1 \rightsquigarrow T^2$ and two read transactions $T^3 = r^3_b(k_b), r^3_a(k_a)$ and $T^4 = r^4_a(k_a), r^4_b(kb)$. Causal consistency guarantees that in case that T^3 reads the value of k_b written by T^2 then, later it should read k_a written by T^2 (and not the previous value is written by T^1). This guarantee results from the fact that $w^1_a(k_a) \rightsquigarrow w^1_b(k_b) \rightsquigarrow w^2_a(k_a) \rightsquigarrow w^2_b(k_b)$ being independent from the way that operations are ordered in a transaction. However, causal consistency allows that T^3 reads a value written by T_1 in k_b and afterward reads a value written in k_a by T^2. Also, it allows that T^4 to read a value written by T^1 in K_a afterward reads a value in K_b written by T^2. Neither of these sequences violates causal consistency; However, introduces unexpected behavior.

Consider social network application, where the friendship relations are symmetric, and it needs to ensure the following invariant. If user u_1 is visible in the friends' list of u_2, then u_2 needs to be also visible in the fiends' list of u_1. Consider that k_i stores the friends' list of u_i and that the transaction T^1 establishes a new relation of friendship between u_a and u_b, and that T^2 erases that relation. In this case, both T^3 and T^4 would read a state that would violate the proposed invariant.

Now consider an application that manages a shared folder, in which k_a registers which users have access to the folder and k_b stores the content of the folder. Consider that T^2 that excludes a user from the list and afterward writes a new document to the folder witch that user should no longer have access. The sequences described above of the transaction T^4 would allow the transaction to read the old access-list and the new document, the application is unable to enforce the access restriction intended by the user.

Transactional Causal Consistency (CCT) avoids the anomalies described above by ensuring that all the effects of one write transaction are visible to other transactions or none at all. It should be noted that several systems extend causal consistency with support for read-only transactions [11]–[14]. These systems avoid the anomaly illustrated by T^4, although not the anomaly illustrated by T^3. Thus, TCC is stronger than causal coherence with support for read transactions, which in turn is stronger than causal coherence with no support for transactions. The interested reader may find a hierarchical comparison of the various consistency models that have been proposed in the literature in [6]. However, we emphasize that TCC remains weaker than snapshot isolation [15] that, in turn is weaker than serializability. These last two consistency models oblige to sort the write operations in a total order, which is not the case with the TCC.

III. Related Work

The implementation of different forms of transactions in systems with low consistency has been much studied in recent years, and it is possible to find different approaches to the problem in the literature. In this section, we will compare and discuss how the concepts and ideas presented in the related work have inspired us in the design of FastCCS. However, this comparison is not trivial due to different systems having different types of consistency and transaction isolation guarantees, so to make a reasonable comparison between the systems, this section will be subdivided into comparing the different techniques, the types of transactions and consistency guarantees. This comparison is summarized in Table I.

TABLE I

SYSTEMS THAT OFFER CAUSAL CONSISTENCY. R REPRESENTS THE NUMBER OF READ ROUNDS. NBR AND WTX RESPECTIVELY REPRESENT NON-BLOCKING READS AND WRITE TRANSACTIONS. N REPRESENTS THE NUMBER OF PARTITIONS, M THE NUMBER OF DATA CENTER AND TS THE PHYSICAL CLOCK VALUE.

System	R	NBR	WTX	Metadata Size	Strategy
ChainReaction [13]	≥ 1	✗	✗	O(M)	Sequencer
Orbe [12]	2	✗	✗	O(N x M)	Stabilization
GentleRain [14]	2	✗	✗	1 ts	Stabilization
COPS [11]	≤ 2	✓	✗	O(|deps|)	Explicit Check
Cure [6]	2	✗	✓	O(M)	Stabilization
Eiger [9]	≤ 3	✓	✓	O(|deps|)	Explicit Check
Wren [16]	2	✓	✓	2 ts	Stabilization
FastCCS	≤ 2	✓	✓	O(N)	Stabilization

Strategies can be classified into four broad categories according to the technique they use to ensure that the transaction is performed in a consistent cut, namely: stabilization, serialization, explicit dependency check, and explicit locks. Next, we will briefly discuss each of these techniques. The strategy in which the coherent cut is obtained by stabilization, do it by ordering the transactions in total order, based on a timestamp. A partition can satisfy a read operation made by a transaction that executes at the instant t when it knows that it has already become aware of the effect of all write transactions that were executed with timestamp lower than t. Systems such as Orbe [12], GentleRain [14], and Cure [6], use physical clocks as the timestamp to order the transactions. Using the physical clock has the advantage of them monotonically increasing without the presence of events. Unfortunately, since it is impossible to synchronize clocks with complete accuracy, partitions need to exchange information to know which timestamps are in the past of all other nodes. Moreover, it is possible due to clock skew that a client reads a value that the timestamp is the "future" for some partition, forcing the partition to stall the operation until its clock catches up. Wren [16] uses similar strategies though using hybrid clocks.

In systems where the consistent cut is obtained by serialization, a centralized component is used, which takes cognizance of all transactions, and orders them in full. This solution has the disadvantage of creating a bottleneck in the system, which limits the scalability of the system. One of the systems that implement this strategy is ChainReaction [13].

Systems that use explicit consistent cut verification require that all write transactions to be associated with metadata that captures the client's causal past. System like COPS [11] and Eiger [9] use this strategy. However, the client must piggyback causal dependency information and execute expensive dependency checks across partitions.

Some systems try to pursue general-purpose transactions. In their pursuit of general transactions, these systems all choose consistency models that cannot guarantee low-latency operations. Other systems such as Wren and Cure, try to give the give to the client a much weaker form of general-purpose transaction guaranteeing only TCC. In these types of transactions, the client is responsible for caching its writes and execute all read operations from a specific snapshot. However, the pre-set of the snapshot has a cost. In this case, one round of communication and was previously shown [8] this cost is not negligible.

Other system takes one step back and implements only read transactions without implementing any write transactions. Some of these systems are ChainReaction, Orbe, GentleRain, Cops. Dropping write transactions for lower latency introduces several artifacts that make it more difficult for programmers to reason about. Moreover, are incompatible with TCC.

Eiger has a much moderate approach. This system separates read from write transactions. Thus, it can optimize read transactions without introducing too much latency. More, over it is compatible with TCC.

Now we will discuss how FastCCS places in the related work. The goal of this work is to create a system that satisfies the following goals. Our system must support scalable and low latency transactions. It is making one strategy incompatible with our requirements. Serialization due to the limitation of the scalability. We opted not to use explicit dependencies as it would introduce too much metadata overhead at the client-side, due to the client needing to piggyback its operations. So, we chose stabilization as a strategy to be implemented. However, trying to avoid some downfalls of previous systems that implemented this strategy with physical clocks. We opted to use logical vector clocks that avoids blocking the client reads due to clock skew.

For the type of transaction, as we mentioned, one of our goals is low latency. General transactions have the disadvantage of requiring two rounds of communication. Moreover, as we FastCCS typical workload would be read-heavy, it would introduce unnecessary latency; however, as we want to provide TCC to avoid some anomalies related to write operations. Thus, we opted for read-only and write-only transactions, as they are compatible with TCC.

IV. FASTCCS SYSTEM DESIGN

In many cloud applications, read operations dominate the workload of the system [8]. For example, 99.8 % of Facebook distributed database operations [17] are reads and the latency of those operations is particularly important because a client request can lead to thousands of reads and some of these reads need to be done in sequence, and the critical path can reach dozens of reads [18].

Therefore, it is crucial to offer an algorithm that supports non-blocking read operations with as fewest as possible rounds of communication.

FastCCS focuses on providing read and write transactions for cloud application without increasing significantly the overall latency experience by the user compared to eventual consistency, however providing to the client higher consistency guarantees.

A. Client Library

This library is responsible for handling the splitting, routing, and re-assembly of the transactional requests. As the transaction can span multiple partitions, the client library is responsible for splitting the client's request, and so hiding the

internal structure of the replicas by forwarding requests to the correct partitions. For knowing the correct partition, it uses a hashing function that is provided by Cassandra [19]. Moreover, to hide the internal algorithm of the transaction and to give an illusion to the client that the transaction is executed as a single operation. The client library is responsible for the re-assembly of the client request, and in case of Eiger [9] and FastCCS to check if the first round of parallels reads was successful.

Also, the client library keeps track of the causal dependencies of the client to avoid introducing false dependencies between thread-of-execution done on behalf of different clients. Each client has a unique id that it uses to communicate with the client library so that the dependencies of different clients (e.g., operations done on behalf of c_1 are not entangled with operations of c_2). This unique id is also used to generate the transaction id, by simply appending the unique client id to a sequence number.

B. Designing the algorithm

In the recent works was proven that it is impossible to achieve one round read transaction while supporting TCC or as they were called in the literature *fast transaction* without compromising availability [20]. Thus, the read transaction protocol needs at least two rounds of communication always to return a consistent causal view over multiple keys. The optimal solution is to, in the majority of cases, only needing one round of communication and the second round for the cases that it fails. The success rate of the first round will influence the overall latency that the client experiences. So we can derive a two-step algorithm in which the first part is optimistic, that reads the most recent available values, then has a function that verifies if the first round was successful or not. The second part is for the times that the optimistic read fails. The more metadata that the system stores with the versions, the more precise is the assessment if the first round was successful. For example, if the system stores little metadata, the system could conclude that the first round was unsuccessful and need to execute a second round that would introduce more latency, even if, in reality, the result was causally consistent. This is called a *false dependency*. Moreover, the metadata needs to be precise enough to determine the second round read condition, or else, more rounds would be needed to return a consistent result. On the other end of the spectrum, the system could store more metadata that would reduce the false dependencies at the cost of a heavier first round and more storage overhead. Other solutions may include fetching more than one version of the keys in the first round, the more versions that are fetched, the lower the probability of the second round occurring, however, with the cost of more expensive first-round reads. It becomes clear that exits a balance between the number of rounds of communication and the size of the metadata.

C. Metadata

FastCCS implements more precise metadata with the intent to reduce to the minimum the number of second rounds without too large sized metadata that would detriment the overall performance of the system. We will now describe the metadata implemented in FastCCS.

FastCCS attaches to each read or write transaction a timestamp, that is materialized in the form of a vector clock of size N. Where N is the number of partitions in a datacenter, every time a write transaction changes the value of a key, it is created a new version of that key, which is stored with the commit timestamp of that transaction. The client also maintains a vector clock timestamp vc of size N of the last operation that it performed. This vector clock tracks the client's causal past. When a new version of a key is created, it is passed by three states of knowledge: pending, confirmed, visible. A timestamp associated with a pending transaction is temporary and will be updated when the transaction is confirmed. A key passes to the state of visible when all the keys in that write transaction are confirmed in all participating partitions. Every partition i keeps a sequence number that is incremented every time that a new version of a key is created in that partition. This sequence number is denominated as sn_i. Every partition i also maintains snapshot vector clock timestamp svc_i of size N, where every entry holds the maximum value of sn_j observed by the partition i, denominated as snapshot vector clock. Every key modified by a transaction with a commit timestamp lower or equal to the snapshot vector clock, are guaranteed to be confirmed in all the partitions. Every partition i updates the value of the entry $svc_i[i]$ every time a transaction is confirmed in that partition. Moreover, partitions periodically exchange the value entry of $svc_i[i]$. Every time that the snapshot vector clock is updated, the partition checks if there exists transactions in the state confirmed with a commit timestamp lower or equal than the svc_i and passes them to the visible state accordingly.

V. Write-Only Transactions

FastCCS's write-only transactions allow a client to atomically write many keys spread across many servers in the local datacenter. As we will see, the algorithm guarantees low latency because it takes at most 2 message RTTs in the local datacenter to complete and no operations acquire locks.

A. Write-Only Transaction Algorithm

The write transaction T is executed in two rounds of communication, in the following way. Let $\mathcal{P}(T)$ be the set of partitions that store keys modified by T. In the first round, the client chooses one of the partitions as the transaction coordinator as $cp \in \mathcal{P}(T)$. The client sends to each partition $\forall i \in \mathcal{P}(T)$ the new value of the modified keys and the coordinating identifier of the transaction, the vc_c and the number of participants in the transaction and waits for the responses. Each partition, upon receiving this message, increments its sequence number sn_i and creates a new pending version of the key to which it assigns a temporary commit timestamp ct_i where $ct_i[j] = -1, i \neq j$ and $ct_i[i] = sn_i$. If the partition is the transaction coordinator, the partition waits for the other partitions to respond to $\mathcal{P}(T)$ otherwise the value of sn_i is returned to the coordinator.

The second round begins when the coordinator receives a response from all partitions. The coordinator creates a commit timestamp for the ct_T transaction where $ct_T[i] = max(vc_c[i], sn_i), \forall i \in \mathcal{P}(T)$. The coordinator sends this value to all partitions in $\mathcal{P}(T)$. Receiving the value of ct_T, the partition passes all versions of keys modified by transaction T to the committed state, associating with these versions the value of the final timestamp. Finally, each partition waits until $svc_i[i] + 1 \geq ct_T[i]$, at which point it is sure that all transactions with a commit timestamp lower than ct_T are already committed in the partition i. When this condition is met, a response is sent to the client. The write transaction is considered terminated when the client receives a response from all the participating partitions, ensuring that all values that the transaction wrote are already committed on all partitions. The coordinator sends the transaction ct_T to the client. The client adopts ct_T as its new client timestamp ($vc_c = ct_T$).

VI. Read-Only Transactions

FastCCS's read-only transactions allow a client to read many keys spread across many servers from the local data-center consistently in at most two rounds of communication usually one in read heavy workloads. This is achieved by the first read returning the most recent committed values from the server, and the second read it's only executed if the first round fails to deliver a consistent view of the keys. The second read only happens if there is a concurrent commit over the same keys. We will go more in depth about the algorithm of the read-only transactions in this section.

A. Read-only Transaction Algorithm

The read transaction T runs in at most two rounds of communication as follows. Let $\mathcal{P}(T)$ be the set of partitions that store keys read by T.

In the first round, the client sends to each $i \in \mathcal{P}(T)$ partition the keys it wants to read, along with its vc_c and waits for a response from all partitions in $\mathcal{P}(T)$. Each partition, upon receiving this message, updates its snapshot vector clock by making $svc_i[k] = max(svc_i[k], vc_c[k]), \forall k \in N$. This is possible since vc_c represents the maximum timestamp the client has ever seen, and if the effects of a transaction are already visible it means that the transaction has already been committed on all partitions and thus making possible to advance svc to vc_c safely. This ensures that all transaction effects that the client has observed in the past will be visible, thus ensuring atomicity and causality.

Partitions return newer versions with $ct_T \leq svc$, svc, and the maximum commit timestamp read ctm, where $ctm[k] = max(ctm[k], ct[k]), \forall k \in N$.

When the client receives a response from all partitions. Client checks if $svc_i \geq ct_j, \forall i, j \in \mathcal{P}(T) \wedge i \neq j$ and updates its $vc_c = max(ctm_i[k], vc_c[k]), \forall i \in \mathcal{P}(T) \wedge \forall k \in N$. If the condition is met, the client ends the transaction and returns the values. If the condition is not met, the client has read of a snapshot that may or may not be consistent. Therefore, the client must start a second round of read operations on partitions that did not meet the previous condition by sending their new vc_c. Partitions receiving a request from the second reading round return the latest versions with $ct_T \leq vc_c$, and update their snapshot vector clock similar to the first round.

The second round ensures that a coherent causal snapshot will be returned, because the stabilization protocol ensures that all versions that the client read in the first round are already installed on all partitions, so it is not necessary to block reads until the new version is installed. Since the client reads versions that satisfy $ct_T \leq vc_c$, no extra rounds of communication will be required because the reads returned in the first round have a $ct_T \leq vc_c$.

VII. Garbage Collection of Obsolete Versions

To limit the number of versions that are maintained in the system, partitions need to implement some sort of garbage collection mechanism in order to delete versions that are not longer necessary. One way to garbage collect old versions is as follows. Partitions periodically exchange among them them the lower snapshot read by any active transaction; versions that are older than this snapshot can be safely deleted. However, this approach requires more communication among partitions, increasing the overall load on the network. Furthermore, in the presence of network partitions, garbage collection may stall, because partitions would not be able to compute the lower snapshot. FastCCS implements the garbage collection mechanism proposed in Eiger [9] that circumvents these problems by assuming that servers can have their physical clocks loosely synchronized. This garbage collection strategy limits old versions in two ways. First, transaction have a timeout that specifies their real-time duration. If the timeout fires the client library restart the transaction. Thus, servers only need to store values that were overwritten during this timeout period. Second, the partition only retains values that could be requested in the second round. Thus, a server only keeps versions that are newer than those returned in a first round within the timeout duration. This mechanism requires nodes to maintain additional metadata for each version of the data, namely the last time at which that version has been accessed.

VIII. Correctness

We have derived proofs of correctness for FastCCS' algorithms. Due to space constraints, these were not included in this paper, but can be seen in the full thesis.

IX. Faults

Like the correctness arguments, the reconfiguration strategies were not included in this paper. Furthermore, they are not evaluated, leaving them as future work.

X. Evaluation

A. Implementation

In order to evaluate the proposed system, it needs to be compared to other solutions that employ different techniques. The chosen ones were: Eiger [9], for offering read and write transaction with multiple rounds of reads (at most three)

and Wren [16] for comparing if two round of reads has a higher overhead than reading in at most two rounds although with a higher metadata overhead. In addition to the causally consistent systems, FastCCS is also compared with a system that only offers eventual consistency as this types of storage system are predominant in cloud applications. Is expected that eventual consistency should offer best latency, as it makes updates visible as soon as they are received and only needs one round to satisfy read transaction without any metadata that normally be used to enforce consistency. We implemented FastCCS as a modification of Eiger's fork of Cassandra. Wren was implemented also by modifying Eiger's fork of Cassandra and for Eiger we used the original code. For eventual consistency we used the original fork of Cassandra from which Eiger is based.

B. Experimental Setup

In our evaluation, we use two different experimental testbeds that complement each other, namely, we perform experiments using simulations and we perform experiments in a real deployment on AWS [21]. Simulations allow us to experiments with system sizes that we cannot afford to deploy in AWS. The AWS deployment allow us to assess the performance of FastCCS in a realistic setting, and simulations offer some form of validation of the results obtained.

Simulations have been performed using PeerSim [22], which was augmented with some modules that help in increasing the fidelity of the results. Namely, PeerSim has been configured with extensions that simulate FIFO point-to-point channels with configurable network latency and finite bandwidth. Each node has a bandwidth limit of 1 Gb/s, and each message is randomly delayed with an average value of 0.5 ms (latency observed between servers in an AWS data center [23]). Since the performance of the various algorithms depends fundamentally on the number of rounds and the need to wait for a causal cut, to speed up simulations we do not aim at capturing CPU utilization or overhead of the disk access time. This allow us to quickly obtain results that approximate well enough the real setting. However, as we will note later, in some systems, the CPU can also be a bottleneck, in particular in systems that have to manage large amounts of metadata. These effects are only captured in the real deployment.

For the real deployment we have built a prototype of our system using Cassandra, a well known key-value store which is very used in the industry. For running the experiments, the prototype was deployed in AWS. In these deployments, each partition run within m4.2xlarge instance with 8 vCPUs and 32 GB of memory. Each client machine runs a client library that issues read and write transactions eagerly. In order to benchmark the proposed architecture we used the a modified version of the stress test of Cassandra. The modifications we have introduced were the minimum necessary to make it work with the new client library.

TABLE II
PARAMETERS OF THE DYNAMIC WORKLOAD GENERATOR.

Parameter	Default	Range
Keys/Read	5	2-64
Keys/Write	5	2-64
Partitions	8	2 - 64
Value Size (B)	128	2 - 1024
Write fraction	0.05	0.01-0.5

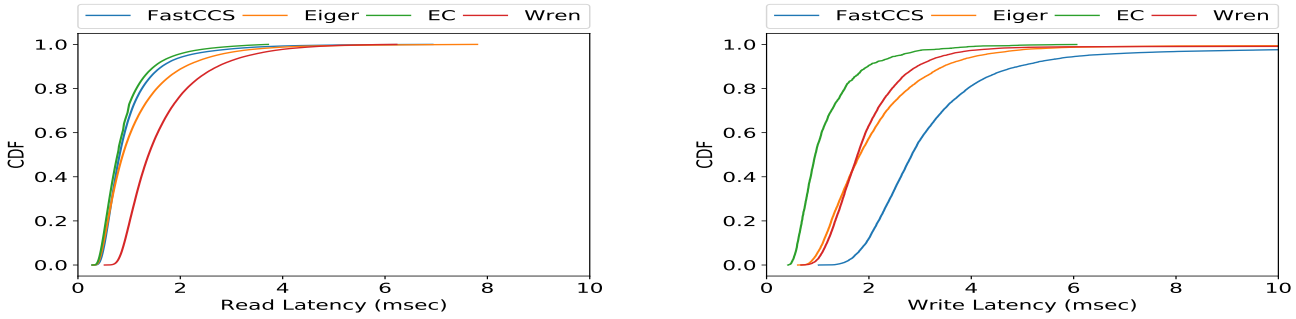
XI. LATENCY

The first experiment compares the latency observed by the clients when they perform read and write operations using different systems. It is expected that read operations present a lower overall latency when compared to write operations. In fact, our target systems have been designed to optimize the latency of read operations, which are assumed to be the most frequent operations.

Figure 1 depicts the cumulative distribution function (CDF) of the latency observed by the clients for both read and write operations. These numbers have been collected using the deployment on AWS.

We start by discussing the performance of reads. Not surprisingly, the lowest latency if offered by a system that offers only eventual consistency. This can be explained by the fact that this system is the one that requires less metadata and less coordination (operations only need one round of parallel requests to return a result). Interestingly, FastCCS closely follows the performance of an eventual consistent system for read operations. This happens because FastCCS implements a good tradeoff between metadata and accuracy: it does not require clients to maintain complex dependency trees but avoids most false positives, and therefore allows most reads to execute in a single communication round. Eiger has less precise metadata compared to FastCCS, which leads to more false dependencies which, in turn, often generate a second round of reads. Moreover, in Eiger, a write transaction becomes visible as soon as the partition receives the commit, so the second round of reads may target a pending commit value; the partition needs to issue a request to the coordinator of that transaction to confirm if it is safe to return the new version. Eiger also needs to maintain a dependency tree in the client, which adds additional overhead in the client, that introduces even more latency. Finally, Wren presents a read latency that is higher than all the other systems as it needs two rounds to return a causal consistent view of the keys.

Looking at the latency of write operations, the tradeoff implemented by FastCCS becomes clearer. In order to favor read transactions, FastCCS sacrifices write latency, as the client waits for the update of the write transaction to be installed in all partitions. Thus, FastCCS is particularly well suited for read-heavy workloads, where slower write operations are not able to have a significant impact on the overall throughput significantly, as we will see next.



(a) (b)

Fig. 1. Cumulative distribution function for each systems read and write latencies.

A. Throughput

This section analyzes the throughput achieved by the different systems. We measure throughput as the number of transactions executed by the client per second. All results are normalized with regard to the throughput achieved by the eventual consistency consistency; this makes easier to assess the overhead incurred when one increase the consistency guarantees to TCC. Furthermore, in this section we present results obtained both using simulations and the real deployment on AWS. This allows us to validate our simulation environment.

1) Effect of Tx Lenght on Read Transactions: Figure 2 presents the throughput of the different systems, when executing transactions with a write/read ratio of 0.05, as a function of the objects accessed in each *read* transaction. As in can be observed, FastCCS is able to closely follow the performance of system that only offers eventual consistency, with a penalty in the order of 10%. Both Eiger and Wren are much worse. For transactions that touch a small number of items, the number of communication rounds of each protocol dominates the performance and, therefore, Wren exhibits a poor throughput. However, for transactions that touch a lot of objects, the performance of Wren starts approximating the performance of other systems, even surpassing Eiger for long transactions. This is due to a combination of three factors.

First, a large number of keys is penalizing for Eiger. In fact, in Eiger, when more keys are accessed in the first round it becomes more likely that the first round will return an inconsistent view, which subsequently increases the likelihood of reading from a key that has a pending version, which subsequently increases the number of commit checks. Moreover, if the number of keys per request increases, the second round is more expensive as more keys will need to be returned in the second round of reads. Second, as the number of keys increase, the costs associated with CPU utilization and with disk access time start to become dominant, and the overhead of the additional round introduced by Wren becomes less relevant. Third, we have also noticed that as the read transaction size influences the number of dependencies in the client, and this dependency tree is only cleared when the client issues a write transaction. Moreover, as the client has a low probability of issuing a write transaction, the overhead of maintaining the dependency tree increases, which negatively influences the client's performance.

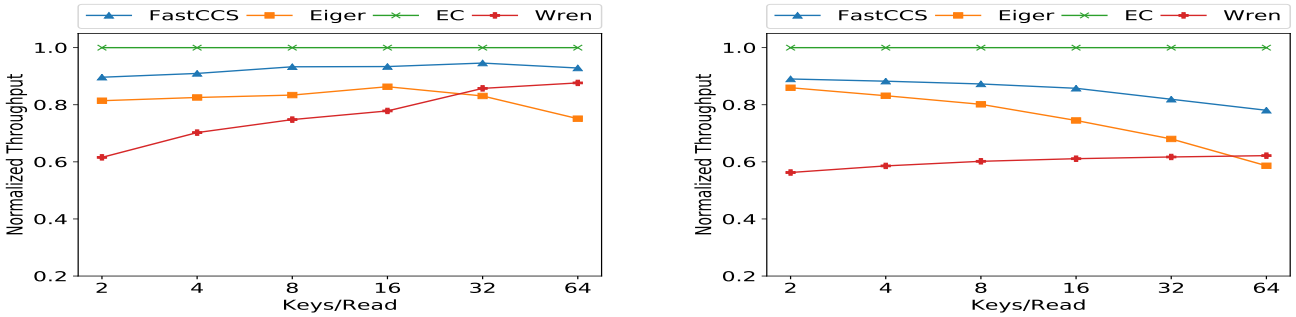
In the case of FastCCS, when the read transaction size increases, the second round also becomes more likely and more expensive as it needs to return more values. However, as the probability of the second round due to false positives is low, and there is no need for a third round, FastCCS is still able to offer overall better performance than Eiger.

The figures obtained with the simulated environment show that the simulator can capture with reasonable accuracy the performance of the different systems for small transactions. For large transactions, the simulator no longer provides an accurate estimate of the performance; this is due to the fact that the simulator is not able to simulate CPU or disk usage.

2) Effect of the Write/Read Ratio: Figure 3 presents the throughput of the different systems, when executing transactions with different write/read ratios (all transactions access 5 objects of 128 bytes).

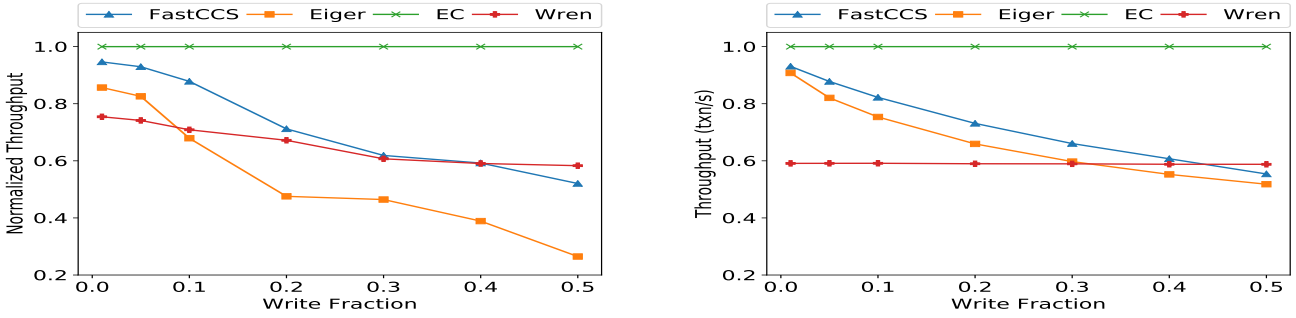
The figure unveils an interesting limitation of Eiger. For residual write ratios (for instance, 0.01%), Eiger and FastCCS have almost the same number of second-round reads. Therefore, one could expect that both systems would exhibit the same performance, as predicted by our simulations. In reality, the throughput of Eiger is 10% lower that that of FastCCS in the AWS deployment. The cause for this difference lies in the way Eiger keeps dependencies in the client. Eiger maintains a tree of dependencies that can only be purged when a write transaction is executed. For small ratio of write transactions the dependency tree keeps increasing and the CPU utilization at the client becomes a bottleneck.

The figure also shows throughput of FastCCS decreases, as the percentage of write transaction increase. This is due to the higher write transaction cost and a higher probability of a second round. In fact, it is possible to observe that for large write/read ratios, Wren eventually outperforms the FastCCS. This happens because write operations terminate earlier in Wren. Interestingly, the simulations can also capture this fact, despite the limitations in accuracy previously discussed. Nevertheless, for most realistic write/read ratios, FastCCS outperforms both Eiger and Wren.



(a) AWS (b) Simulations

Fig. 2. Read throughput as a function of Tx length: AWS vs simulations.



(a) AWS (b) Simulation

Fig. 3. Throughput as a function of the write/read ratio: AWS vs simulations.

B. Scalability

The ability to distribute the data across different servers is essential for scalability. Thus, all cloud storage systems split the data into logical partitions and then let a different set of servers handle each partition. If a transaction accesses data that are in different partitions, coordination among different servers is required. This experiment focuses on understanding how the system can scale horizontally. We do so by studying the effect of the number of partitions on the system throughput.

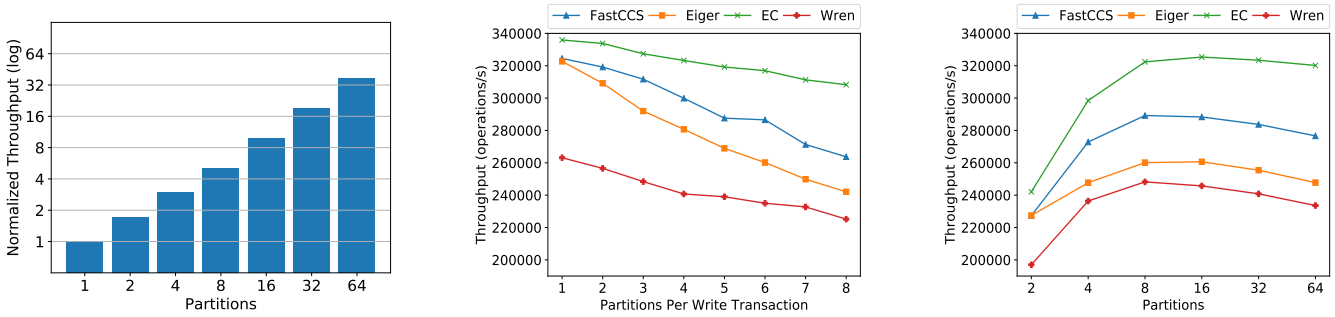
Given that the number of communication rounds used by each protocol has a direct impact on the achievable throughput, we also show the average number of communication round used by the different systems. For these experiments, we have been able to deploy the system on AWS using up to 64 partitions. Unfortunately, we could not afford to run experiments on a real deployment using a larger number of machines, as this was outside our budget. Therefore, we resorted to simulations to estimate the performance of the system in scenarios that go up to 1024 machines.

1) Horizontal Scaling: We start by showing that FastCCS is able to horizontally scale and sustain additional clients as more servers are added to the system. For this experiment, we augmented the number of partitions and augmented proportionally the number of clients, such that the operations submitted by N client machines are fully loading N partitions. Moreover, we proportionally increase the number of keys to avoid any artifacts due to an increase in concurrency between clients. Transactions always access 5 objects selected at random.

Figure 4(a) shows the throughput for FastCCS as we scale the number of partitions from 1 to 64 (note that both axes are in log scale). The bars show the throughput normalized against the throughput of 1 partition. FastCCS scales out as the number of partitions increases. However, this increase is not linear from 1 to 8 partitions. The configuration with 1 partition has the benefits of batching: all operations that involve multiple keys are executed on a single machine. As the number of partitions increases, the transactions span across multiple partitions, and thus the system is no longer able to exploit batching effectively. This effect was also present in the original evaluation of Eiger [9].

As we increase the number of partitions, the differences due to lack of batching no longer become relevant. In fact, in a system with many partitions, most transactions tend always to access 5 different partitions. Nevertheless, the ability of the system to scale perfectly is limited due to a number of overheads that are associated with the maintenance of multiple partitions, such as background stabilization procedures or increased size of metadata. Next, we describe a number of experiments that provide some insights for the causes of the observed impairments to horizontal scaling.

2) Partition Overhead: In order to better understand the sources of overhead that become visible when many partitions are used, we have run a series of experiments where we increase the number of partitions while keeping the workload



(a) Normalized throughput of FastCCS changing the total number of Partitions (AWS) and proportionally changing the number of clients and keys. Bars are normalized against 1 partition.

(b) Partitions per Write

(c) Total Number of Partitions

Fig. 4. Changing the Number of Partitions (AWS).

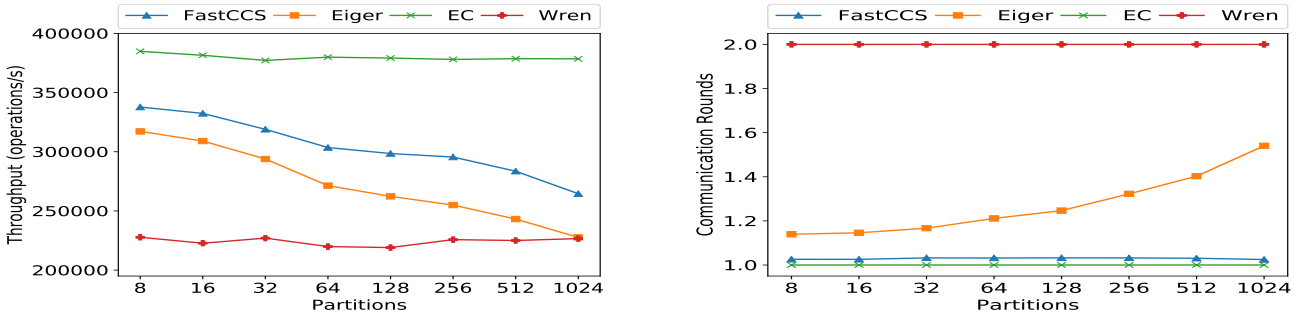
constant. Figure 4 shows the results obtained with the deployment at AWS. We have performed two experiments. In the first experiment (4(b)), we fixed the total number of partitions to 8, and we varied the number of partitions accessed by each transaction. In the second experiment (4(c)), transaction access 5 objects at random, and we have changed the total number of partitions (the higher the total number of partitions is, the more likely is that a transaction touches 5 different partitions).

We start by discussing the effect on the number of partitions accessed by each transaction. Naturally, even the eventual consistent system degrades its performance as the number of partitions accessed by a given transaction increases, as more servers need to be contacted. This happens because Cassandra maintains a number of background bookkeeping tasks that become heavier with the number of nodes increase. Still, not surprisingly, the eventually consistent system is the one that is less affected by an increase in the number of partitions, and it does not require nodes to coordinate, and the amount of metadata maintained is minimal. The performance of FastCCS, Eiger, and Wren follow a similar trend when the number of partitions accessed by each transaction increases. Because these protocols require more coordination among partitions, the effect of using a larger number of partitions is more noticeable than in an eventually consistent system. For instance, with Eiger, because the metadata used to capture causal dependencies has low accuracy, the number of times the protocol requires a second or even a third round of communication. FastCCS has the same coordination overhead as Eiger but, due to more precise metadata, it is less likely to require a spurious second round of reads.

We now discuss the effect of changing the total number of partitions. In this experiment, we have kept the workload constant across all experiments. When there are few partitions, served do not have enough capacity to serve all the clients' requests. Therefore, the throughput of the system is limited by the lack of capacity of the servers. As we increase the number of partitions, we are able to distribute the load of the clients among different servers, and the throughput increases. This growth stops when the number of partitions reaches 16. In this scenario, the capacity of the servers is no longer a bottleneck, and adding more servers does not provide any help (we recall that the workload is fixed). From this point, we can start observing the effect of some amount of overhead that is induced by having a large number of partitions. This overhead as two main sources: one source are background activities, such as the stabilization protocol used by FastCCS and Wren, and another source is the increased size of the metadata (for instance, larger vector clocks). The combined effect of these two factors slows down the system when the workload remains constant (naturally, the larger system could sustain a higher peak workload, but this is not depicted in these plots). Note, however, that even the eventually consistent system experiences this effect, given that the underlying key value store also has some bookkeeping tasks whose overhead increases with the number of servers.

In order to estimate the performed of the system for a higher number of partitions, we needed to resort to simulations. The results are depicted in Figure 5. Note that, as before, we assume that the workload is fixed. Thus, the maximum throughput is bounded by the number of clients. Therefore, adding more partitions only increases the overhead generated by background tasks, such as the stabilization protocols, and larger metadata (i.e., larger clocks). The simulations show a trend that is aligned with the results obtained with the real experiment. However, as we have seen in previous experiments, the results from the simulator are an upper bound on the real performance, because the simulator is not considering CPU utilization, which also increases as the metadata increases.

We have also measured the average number of rounds required by read operations in the different systems, as the number of partitions grows (Figure 5(b)). This is interesting because it highlights that the overhead induced by a large number of participants manifests in different ways for different protocols. For Eiger, the loss of performance can be mainly attributed to the fact that the average number of rounds increases with the number of partitions, which affects the throughput of the system. Thus, because Eiger uses less metadata than FastCCS, it needs to perform a second round more often.



(a) Throughput

(b) Average Number of Rounds

Fig. 5. Changing the Number of Partitions (Simulations).

FastCCS, instead, does not suffer from this problem. Most reads can be performed in one round, no matter how large is the system. Unfortunately, this positive feature is obtained at the cost of using more metadata, which grows linearly with the system size. This amount of metadata also affects negatively the performance of the system (even if the results are still better than Eiger).

XII. Conclusions

Stronger consistency models ease the life of programmers, making it more easy to reason about. However, as stated by Amazon and Google, the increase in user perceived latency leads to a concrete revenue loss. For example, Amazon estimates that a 100ms latency increase leads to 1% revenue loss [24]. So it is important to find consistency models that can offer low latency. This thesis has described the design, implementation, and evaluation of FastCCS. By using more precise metadata, we showed that FastCCS reduces the number of communication rounds needed to implement TCC. In fact, to our knowledge, FastCCS is the first system that implements TCC with at most two rounds of communication, reducing the overall latency that the client experiences.

References

[1] P. A. Bernstein, P. A. Bernstein, and N. Goodman, "Concurrency control in distributed database systems," *ACM Computing Surveys (CSUR)*, vol. 13, no. 2, 1981.

[2] C. H. Papadimitriou, "Serializability of concurrent database updates," *Journal of the ACM*, vol. 26, no. 4, 1979.

[3] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Highly available transactions: Virtues and limitations," in *Proc. of VLDB*, Trento, Italy, Aug. 2013.

[4] M. K. Aguilera, J. B. Leners, and M. Walfish, "Yesquel: Scalable sql storage for web applications," in *Proc. of SOSP*, Monterey, CA, Oct. 2015.

[5] W. Vogels, "Eventually consistent," *Communications of the ACM*, vol. 52, no. 1, 2009.

[6] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro, "Cure: Strong semantics meets high availability and low latency," in *Proc. of ICDCS*, Nara, Japan, Jun. 2016.

[7] P. Ajoux, N. Bronson, S. Kumar, W. Lloyd, and K. Veeraraghavan, "Challenges to adopting stronger consistency at scale," in *Proc. of 15th HotOS*, Kartause Ittingen, Switzerland, May 2015.

[8] H. Lu, C. Hodsdon, K. Ngo, S. Mu, and W. Lloyd, "The SNOW theorem and latency-optimal read-only transactions," in *Proc. of OSDI*, Savannah, GA, Nov. 2016.

[9] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Stronger semantics for low-latency geo-replicated storage," in *Proc. of NSDI*, Lombard, IL, Apr. 2013.

[10] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto, "Causal memory: definitions, implementation, and programming," *Distributed Computing*, vol. 9, no. 1, 1995.

[11] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: Scalable causal consistency for wide-area storage with cops," in *Proc. of SOSP*, Cascais, Portugal, Oct. 2011.

[12] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel, "Orbe: Scalable causal consistency using dependency matrices and physical clocks," in *Proc. of SoCC*, San Jose, CA, Oct. 2013.

[13] S. Almeida, J. a. Leitão, and L. Rodrigues, "Chainreaction: A causal+ consistent datastore based on chain replication," in *Proc. of EuroSys*, Prague, Czech Republic, Apr. 2013.

[14] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel, "Gentlerain: Cheap and scalable causal consistency with physical clocks," in *Proc. of SoCC*, Seattle, WA, Nov. 2014.

[15] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Feral concurrency control: An empirical investigation of modern application integrity," in *Proc. of SIGMOD*, Victoria, Australia, May 2015.

[16] K. Spirovska, D. Didona, and W. Zwaenepoel, "Wren: Nonblocking reads in a partitioned transactional causally consistent data store," in *Proc. of DSN*, Luxembourg City, Luxembourg, Jun. 2018.

[17] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li *et al.*, "TAO: Facebook's distributed data store for the social graph," in *Proc. of ATC*, San Jose, CA, Jun. 2013.

[18] P. Ajoux, N. Bronson, S. Kumar, W. Lloyd, and K. Veeraraghavan, "Challenges to adopting stronger consistency at scale," in *Proc. of HotOS*, Kartause Ittingen, Switzerland, May 2015.

[19] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," in *Proc. of LADIS*, Big Sky, MT, Oct. 2009.

[20] D. Didona, P. Fatourou, R. Guerraoui, J. Wang, and W. Zwaenepoel, "Distributed transactional systems cannot be fast," in *Proc. of SPAA*, Phoenix, AZ, Jun. 2019.

[21] Amazon, "Amazon Elastic Compute Cloud (EC2)," https://aws.amazon.com/ec2/.

[22] A. Montresor and M. Jelasity, "PeerSim: A scalable P2P simulator," in *Proc. of P2P*, Seattle, WA, Sep. 2009.

[23] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Highly available transactions: Virtues and limitations," in *Proc. of VLDB*, Trento, Italy, Aug. 2013.

[24] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie, "How far can client-only solutions go for mobile browser speed?" in *Proc. of WWW*, Lyon, France, Apr. 2012.