

P2P Systems with Transactional Semantics *

Shyam Antony
shyam@cs.ucsb.edu

Divyakant Agrawal
agrawal@cs.ucsb.edu

Amr El Abbadi
amr@cs.ucsb.edu

Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA

ABSTRACT

Structured P2P systems have been developed for constructing applications at internet scale in cooperative environments and exhibit a number of desirable features such as scalability and self-maintenance. We argue that such systems when augmented with well defined consistency semantics provide an attractive building block for many large scale data processing applications in cluster environments. Towards this end, we study the problem of providing transactional semantics to P-Ring a P2P system which supports efficient range queries. We first extend a commonly used replication protocol in P2P systems to provide well defined guarantees in the presence of concurrent updates and under well defined failure assumptions. A multi-version concurrency control protocol called LSTP which leverages the guarantees of the replication protocol to provide transactional semantics is proposed. LSTP is designed to provide useful consistency semantics over P-Ring for read intensive workloads without sacrificing the scalability and other desirable properties inherent to the system. Under LSTP, read-only transactions are abort-free and non-blocking and the index stores no state for such transactions. We show that LSTP ensures no missed dependencies between transactions and guarantees basic consistency for read-only transactions when update transactions are serializable. The design of LSTP and its provable properties is a proof of concept that P2P systems can be augmented with transactional semantics. Results from a preliminary simulation study are also presented.

1. INTRODUCTION

Allowing programmers to concentrate on domain specific problems and relieving them from having to deal with general data management issues such as efficient access, storage, isolation and recovery is one of the fundamental reasons why many enterprise applications are layered on top of databases.

*This work was supported under the UC Discovery/NEC Award COM-05-10189.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

EDBT'08, March 25–30, 2008, Nantes, France.

Copyright 2008 ACM 978-1-59593-926-5/08/0003 ...\$5.00.

In other words, databases form a foundational building block for such applications. In recent years, cluster computing environments consisting of thousands of nodes distributed over a few data centers has become the defacto environment for service providers operating on very large scales of data, such as search engines (e.g., [3]), pub-sub systems (e.g., [13]) and storage systems (e.g., [2]). Distributed databases cannot cope with such an environment and such degrees of scale [15, 7] and hence are unsuitable to act as building blocks for such applications. However, building such applications from the "ground up" is extremely difficult, expensive and time consuming. Ideally, issues of scale should be made transparent to the application programmer and should be dealt with by generalized software components. In this paper, we position suitably augmented structured P2P systems, as a foundational software component for data processing applications that do not require a rich and full featured data model.

Structured P2P systems such as Chord [25] and P-Ring [9] may be viewed as *scalable, dynamic, self-maintaining, load-balanced, integrated data partitions and index structures*. These hybrid data partition and index structures provide a powerful abstraction for constructing distributed applications in cooperating environments at an internet scale. We argue that the use of these systems in cluster computing environments should also be explored due to the following advantages.

1. **Encapsulated Scalability:** As already mentioned, structured P2P systems are highly scalable. What makes them even more attractive is the fact that they completely encapsulate the details of scale and scaling from the application programmer. The software infrastructure can seamlessly adapt to a rapid change in the scale of the system without any change in the code.
2. **Simple yet Powerful Interface:** Structured P2P systems export a simple interface involving primitive operations such as lookup, insert and range query. Using such simple primitives, researchers have shown that it is possible to create a variety of applications such as pub-sub systems (e.g., [14]), storage systems (e.g., [16]), information monitoring systems (e.g., [12]) etc.
3. **Reduced Administrative Cost:** Administrative costs are significant in establishing and maintaining large scale cluster computing environments. For example, partitioning data as per load characteristics is one of the chief administrative task in systems where data is

partitioned among nodes. Since P2P systems are self-maintaining, the administrator is relieved from the responsibility of having to act on such occasions as the system dynamically adapts to changing circumstances.

4. **Efficient Infrastructure Sharing:** The ability of P2P systems to add or remove nodes seamlessly is particularly useful in cluster environments since the infrastructure can be effectively shared between different services with the proportion of resource allocation decided dynamically and automatically.

However structured P2P systems exhibit the following disadvantages as well which limit their usability in non-P2P settings.

1. **Best Effort Query Processing:** In order to handle the volatile nature of the P2P environment, structured P2P systems usually follow a best effort query processing paradigm wherein no guarantee is provided about the semantics or consistency of query results. This makes them unsuitable for cluster computing environments which usually serve customers rather than grass root driven P2P applications. For example, a customer of a pub-sub system will be unhappy if an important publication was not delivered and will be annoyed if the same publication is delivered multiple times. These kind of scenarios can arise if best effort query processing is followed. Even though some studies (reviewed below) have considered issues of query semantics for singleton queries, that is inadequate since applications often need consistency across multiple queries and updates. For example, in the pub-sub case, consistency between subscription queries and publication updates is needed. The goal of this paper is to address this inadequacy.
2. **Churn and Security concerns:** Structured P2P systems are arguably plagued by vulnerabilities to churn [22] and also malicious behavior [24] by participating nodes. Fortunately, these concerns are not important in a cluster environment which represents a much more controlled environment than the internet.

There is a growing body of work exploring P2P systems with stronger consistency guarantees. Issues of atomic data access were first raised in [19]. Linga et al. [17] raise issues of query correctness in P-Ring and our work builds on techniques discussed therein. Similarly, Bawa et al. [4] explored correctness conditions for aggregate queries which were subsequently used in [20]. But these efforts suffer from an important limitation, they are restricted to the semantics of singleton queries. Assuming that applications using very large scale data partitions would only interact with the data in terms of singleton queries or updates is too simplistic. In order to provide meaningful semantics to applications, one has to consider the semantics provided to groups of related queries and updates. *Transactions* are the traditional database abstraction for representing groups of related queries and updates. But even if a specific application in a large scale environment does not explicitly use the notion of transactions, lessons learned as part of developing transactional protocols can be extrapolated to such applications. For example, in the course of developing our protocol, we uncover the *slow committers problem*, which is the kind

of uncertainty an application must cope with in a large scale dynamic system. We develop techniques for P-Ring (a range partition) but they also carry over to Chord (a hash partition) provided Chord is modified to use consistent successor pointers (described in the system model).

Issues of replication and distributed transactions have long been studied in the database literature (e.g., [18, 27, 8]). However, none of these protocols were designed to scale to thousands of machines. These protocols often provide strong consistency which unfortunately translates to a scalability bottleneck. Furthermore, they often rely on primitives such as broadcast which are unsuitable for large scale systems. The protocol we develop avoids such pitfalls and minimizes the number of synchronization points in the system. The proposal by Türker et al. [26] is similar in spirit to our work in that it attempts to add notions of transactions to emerging paradigms.

This paper makes the following technical contributions:

1. We extend the replication model in P-Ring to allow concurrent updates and extend a simple replication protocol to support such updates. The basic idea of the protocol, to replicate on successor peers is frequently used and hence not novel. However, our extension to this protocol, is the first protocol to leverage consistent successor pointers to provide guarantees about semantics in the presence of concurrent updates.
2. We develop a transactional protocol that is well suited for large scale, read heavy environments. The protocol allows read-only transactions to be non-blocking and abort-free and the index is stateless for read-only transactions. To the best of our knowledge, there exists no known transactional semantics in such large scale systems.
3. We expect our protocol to be a significant proof of concept that transactional semantics can be added to P2P systems in cluster settings.

Road Map: We describe the system model and some design objectives in Section 2. The replication scheme which provides durability and concurrent update semantics is the subject of Section 3. The primitives of the LSTP protocol is described in Section 4 and the protocol itself is Section 5. Section 6 provides some preliminary empirical results and Section 7 concludes.

2. PRELIMINARIES

2.1 System Model

We assume a cluster computing model consisting of thousands of nodes distributed over a few data centers and connected by a low latency, high bandwidth, reliable network. We assume a communication model in which there are known upper bounds on per-hop latency and processing delays. We also assume that nodes are able to establish reliable FIFO sessions with each other. A high level view of the system model is shown in Figure 1. The system has three distinct components, viz., P-Ring, the user pool and the free peer pool.

P-Ring: P-Ring [9] acts as a distributed data store with fast access to the data. Each item, i , stored in the index has two components, $\langle k_i, v_i \rangle$ where k_i is the data key of the

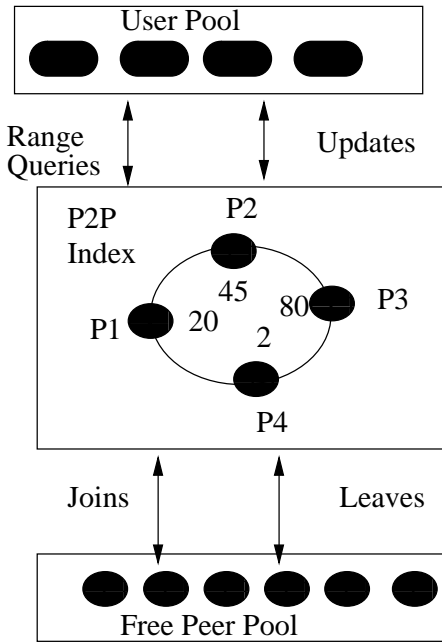


Figure 1: High level system model

item and v_i is the value of the item. Similar to Chord [25], the peers are organized in a ring topology. The data key space is contiguously distributed over the ring as shown in Figure 1. Unlike Chord, here data keys are not hashed but stored in the ring according to their actual value. A peer is the *owner* of the sub-range of the data space associated with it. For example in the figure P2 owns [20, 45) and the entire data key space is [0, 100). A range query [lb, ub] is evaluated by contacting the owner of lb and by following the successor pointers along the ring. For example to evaluate [8, 25] first P1 is contacted which returns the data in [8, 20) and forwards the query to P2. Failures of peers during the range query evaluation process would result in the relevant sub-range queries being retried by the querying process. The figure shows a flat data space but multiple namespaces [15] can be used to store different types of data in the same index. To achieve fast logarithmic routing across the ring, each peer maintains $O(\log_a(N))$ long distance pointers, where N is the number of nodes in the network and $d \geq 2$, is a configurable parameter. The peers are assumed to follow a *fail-stop* failure model.

User Pool: The user pool is the set of processes that query and update P-Ring. A transaction is the unit of interaction of a user process with P-Ring and is a sequence of range queries (reads) and writes. There is no designated transaction manager. Instead the user processes and P-Ring collaborate to emulate the functionality of a transaction manager. Each user process is responsible for maintaining the state of its transaction. For ease of exposition, we make the following assumptions. We assume that user processes that update the index have access to stable storage to log their updates. In the event of a crash, upon recovery, the user process should use the log to resume or abort its updates. We can avoid making this assumption by “pushing” the functionality of stable storage into the index; i.e, the index in addition to storing data, also stores the log of user

processes. We also assume that user processes have access to a local cache. This local cache should be used for repeating reads, i.e, when a transaction reads the same part of the index more than once. The local cache assumption allows us to sidestep the so called “phantom problem”. However later in the paper we discuss how repeating reads can be supported correctly without the need for a local cache. The distinction between user processes and peers is virtual and the same physical node may support both processes provided the respective failure models are obeyed.

Free Peer Pool: Since the data keys are not distributed across the ring using hashing, skewed data distributions would result in the overloading of some regions of the ring. Therefore there is a need for dynamic load balancing which is achieved using the free peer pool. When a peer in the ring is overloaded it tries to balance the load with its successor. If the successor is also overloaded, it invites a free peer to join the ring as its successor and split its load. In either case the overloaded peer transfers a suffix of its region of ownership to its successor resulting in a *data range transfer*. Similarly when a peer is underloaded it tries to balance the load with its predecessor by requesting it to yield a suffix of its range of ownership. If the predecessor is also not sufficiently loaded it leaves the ring and its region of ownership is taken over by its successor. Note that in all these cases ownership transfer is unidirectional, i.e, from a predecessor to a successor and never in the opposite direction. This is a restriction we add to the load balancing protocol developed in [9]. Intuitively, this prevents the problem of a query moving forward and thereby missing data which is concurrently moving backward and also ensures rollover if data is replicated in successors.

2.2 Consistent Ring

As already mentioned, the index is organized in a ring topology. After the initial routing, queries are evaluated by following the successor pointers in the ring. Hence the structural integrity of the ring is important to ensure that no relevant peers are missed during the processing of a range query. Structural integrity is enforced by adapting the protocol developed in [25]. Briefly, each peer maintains a list of k successor pointers and a single predecessor pointer. These pointers are continuously maintained by each peer through periodic synchronization of the list with its successor. However this scheme allows temporary inconsistencies in the ring. As a simple example, consider three successive peers p_1, p_2 and p_3 in the ring. Suppose p_2 introduces a new peer p' into the ring and before information about p' has been propagated to p_1 through the synchronization process, p_2 crashes. Now p_1 discovers the crash and hence assumes that its new successor is p_3 but the actual successor is p' . This temporary inconsistency in the ring structure will be eventually resolved by the protocol but range queries processed during the temporarily inconsistent phase will “jump over” p' and hence miss some of the result set. To avoid this problem, [17] introduced the notion of consistent successor pointers which ensures that, provided there is no network partition, a range query which follows the consistent successor pointers will not “jump over” any peer. While this scheme was developed specifically to ensure that range queries will not miss parts of the result set, a side effect is that any message which is propagated by forward hopping through the consistent successor pointer will not “jump over”

any peer. Similarly, a simple handshake protocol, ensures that messages which hop through the system by following predecessor pointers will not miss any peer “on the way” as well. We exploit the ability of messages to hop over consistent pointers in our replication scheme in Section 3. Our work inherits the absence of network partitions assumption from the consistent ring [17] protocol.

2.3 Design Considerations

Our target environment is a P-Ring consisting of a few thousand machines supporting millions of concurrent transactions which are predominantly read-only in nature. We also assume that transactions are short-lived in general but also that there are a few long-running read-only transactions that execute range queries spanning large portions of the ring (These correspond to statistics gathering transactions which are common in large scale systems). We have two main design objectives; high scalability and low overhead. A major constraint on the concurrency protocol design is the performance behavior of the replication protocol that we design for durability (Section 3). To exploit the read-heavy workload, the replication system is biased against updates which are relatively more expensive compared to range queries. Given these objectives and constraints, what direction should we take in designing a concurrency control protocol (CCP)? CCPs can be broadly categorized into three distinct flavors; locking protocols, optimistic protocols and version based protocols.

A number of strong reasons can be cited against locking based protocols in our target environment. Firstly, it is fairly complicated to implement a correct locking based protocol using the guarantees provided by the replication scheme since techniques such as lock coupling are necessary for range queries. Secondly, given the overwhelmingly read-only nature of the target workload, update transactions will suffer from starvation unless they are provided with some priority in locking. However once update transactions are given priority there will be an enormous amount of blocked transactions in the system. Note that updates take longer to execute since our replication scheme is an update everywhere with retries when necessary scheme and hence will block read-only transactions for a longer duration. Finally, we will have to store state in P-Ring even for read-only transactions to keep track of the lock holders and since any state which is not replicated could be lost, read-only transactions effectively become update transactions in the sense that they have to execute updates to make sure that state corresponding to their locks are not lost.

Given the read-only nature of the workload, optimistic protocols are seemingly attractive. Furthermore, it has been shown [26] that it is possible to design distributed optimistic concurrency control protocols that do not need a global coordinator. However the main disadvantage of this approach is that aborts cascade and even read-only transactions may have to abort. This disadvantage is particularly severe on read-only transactions that execute range queries spanning a large portion of the data space since they will often be forced to repeatedly abort. Note that these transactions do the most work in terms of data transfer and hence aborting them is expensive. Also, in this approach, read-only transactions may be forced to block on commit.

Given the disadvantages and constraints discussed above, we base our design on the multi version approach. We be-

lieve that LSTP (Large Scale Transaction Protocol), the protocol we develop, effectively addresses these issues. For example, in LSTP, read-only transactions are non-blocking and abort-free. Furthermore, no state is stored in P-Ring for read-only transactions which is particularly important given the nature of our replication scheme. This ensures good response time for short-lived read-only transactions and the abort-free property relieves long running read-only transactions from having to waste any work.

3. DURABILITY: REPLICATION SCHEME

In many P2P systems such as P-Ring [17], CAN [21] etc., it is implicitly assumed that there is a unique user process (peer) corresponding to each data item. The user process periodically refreshes the item by contacting the peer in the system storing the item. The item is discarded if there is no refresh within a specified timeout. Only the designated user may update the item. In other words in this model which we term the *user refresh model* the responsibility for sustaining and updating the item in the system is bound to some user. In our system model, the system assumes the responsibility of sustaining the item and mediating between concurrent updates by different users. One of the most commonly ([10, 17]) used availability schemes in P2P systems with a ring topology is to replicate an item on the k successors of the owner of the item. The failure assumption is that there cannot be more than k near simultaneous crashes among k successive peers in the ring.

The simple scheme of replicating an item in $k + 1$ successive peers with the owner of the item at the head of the list gives rise to a variety of questions, when we need clear semantics for updates under dynamic system operation. In order for this scheme to work, there should be no “gap” in the list of replicas of an item; otherwise a situation could arise where a sequence of crashes results in a peer without a copy of an item taking over as the owner of the item. How do we ensure that no gaps occur in the replica list of an item? Crashes results in a reduction in the number of replicas for some items. How do we restore the number of replicas per item as k in the presence of concurrent system events such as crashes? These issues are explained under the heading “replica maintenance” below. How do we propagate an update on an item among the different replicas of an item while avoiding the problem of “lost updates”? This question is answered under the heading “update propagation” below. For this replication scheme to work, for every item there should have been a short time interval, such that $k + 1$ distinct copies of the item existed in the system within that time interval. But for a newly inserted item, there are 0 copies in the system to begin with? How do we go from 0 copies to $k + 1$ copies within a short time interval? This question is answered under the heading “replication bootstrapping” below. How the replication scheme coexists with concurrent data redistribution for load balancing purposes is answered under the heading “concurrent data distribution” below. Finally, when we consider all these different components together, what guarantees can we provide to the application? That is answered towards the end of this section.

Replica Maintenance: As explained above, for each item i stored in the system, the replicas of i form a list of contiguous peers in the ring with the owner of the subrange, which includes i , at the head of the list. The system asynchronously tries to maintain the length of this list as

$k + 1$. This is achieved by *periodically* forwarding *replica maintenance messages* through the list starting from the head (owner of the item) of the list. The message carries a count which is an estimate of the distance between the peer currently processing the message and the owner of i in number of hops. The last peer in the list uses this distance estimate to increase or decrease the length of the list by 1. The length of the list increases when the last peer in the list creates a replica for i on its successor or when a peer in the list introduces a new peer into the system as its successor. The length of the list decreases when the last peer in the list discards its copy of i or if some peer in the list crashes. If the successor neighborhood of i remains stable then the number of copies converges to k . The peer join algorithm from [17] is modified (see concurrent data redistribution below) such that a joining peer receives copies of the items from the peer introducing it as its successor before participating in ring activities. Since replicas are only discarded from the end of the list and also since copies are created in new peers joining the list in the middle, no “gap” can never arise in a replica list. Note that these arguments hold only because we assume the fail-stop model for peer crashes. The next lemma whose correctness follows from the correctness of the consistent successor pointer protocol [17], formalizes this “no gaps” concept. Replica messages can be succinctly batched for efficiency.

LEMMA 1. *Let a replica maintenance message forward hop along $p_0, p_1, p_2, \dots, p_x$ where p_0 is the owner of item i . Then when the message arrives at p_x every peer which owns some part of $[k_i, p_x.lb]$ has a copy of i .*

Update Propagation: When a user process issues an update on an item i , it is first routed to the owner of i . Then it forward hops across the replica chain of i until it reaches the last replica of i or a replica on which the update is a duplicate of an earlier update. Then, the update starts backward hopping back towards the owner during which it is executed at every hop destination. The update succeeds when it is executed on the owner which sends an acknowledgment to the user. If any of the peers crash during this process, the user will timeout while waiting for the acknowledgment from the owner and will retry the update. It is safe to retry updates because we ensure that duplicates are detected using rules imposed by the transactional protocols described later. Note that updates are executed on the replica list in reverse order. Also recall that range queries are evaluated at the head of the replica list. Together, these two observations imply that an update on an item, which is visible to a range query has been executed on all the existing replicas of the item at the time of evaluation of the range query. Thus the problem of lost updates arises if and only if all replicas of an item crashes. We already assumed that there cannot be more than k near simultaneous crashes in the successor neighborhood of any peer. Therefore, as long as we ensure that no update is executed on an item before $k + 1$ distinct copies of the item exist in the system, the problem of lost updates does not occur.

Replication Bootstrapping: Before the first update is executed on an item, the system has to ensure that copies of the item are present in at least $k + 1$ peers within a short time interval. When an update is received at the owner of an item, with data key k_i , which is not present in the owner’s data store, the owner ignores the update and starts the

replication bootstrapping process. It first sends a forward hopping message and every peer adds (k_i, UNBORN) where UNBORN is a special value to indicate that the item’s replication bootstrapping is not complete yet, to its data store if it is not already present there. The forward hopping stops either when it is received at a peer with (k_i, BORN) in its data store or the forward hopping reaches the end of the (k_i, UNBORN) list with a hop-count $\geq k$. Then the UNBORN state is converted to BORN in a backward hopping process similar to update propagation. From the above description it follows that the first time an item’s state is converted from UNBORN to BORN, the item with key k_i is present in at least k peers within a short time (the time needed for k hops) and hence by the failure assumption and replica maintenance protocol, the replica list of the item will henceforth never reduce to 0. Note that items in the UNBORN or BORN state only act as placeholders for future updates and do not appear in the result set of range queries.

Concurrent Data Redistribution: Updates can be executed concurrently with replica creation and data redistribution (for load balancing) as follows. While creating a new replica, the items are transferred in key order. If an update is received for an item that has already been sent then it is forwarded to the new replica using the same FIFO channel which after executing the update forwards it back simulating the usual backward hopping step. Similarly, during a sub-range split for data redistribution items are discarded in reverse key order at the previous owner.

The copy present in the owner of an item, i , is designated as the primary copy of the item, when $\{k_i, \text{BORN}\}$ makes it to its data store. Non-primary copies are secondary copies and exist to take over as the primary when the primary crashes. Reads are always using the primary copies of items. The *history of a copy of i* is the totally ordered set of all updates which were successfully executed on that copy. When a new copy is created it inherits the history of its source. We say an update on an item i *succeeds* the moment it is present in the history of every copy of i . The transactional primitives rely on the replication scheme to provide the following guarantees.

1. If an update u on an item i is present in the history of the primary copy of i at time τ^u then u should be present in the history of all existing copies of i at time τ^u .
2. \exists a **replication timeout** τ^R such that an update will not be executed on any copy of i τ^R time units after u was last issued into the system.

The first guarantee is provided by the replication scheme developed so far and its correctness follows from the reverse hopping execution of updates over consistent pointers. The second guarantee is easy to achieve but the need for such a guarantee and its implications are interesting. When two messages which originate around the same time are routed towards the same item, there is no guarantee which message arrives first at the owner of the item. This is true even though we make the pairwise FIFO channel assumption since different messages to the same item may be routed through different paths. The transactional protocol is designed to cope with such order of arrivals. The transactional protocol can also handle duplicate updates. But in some special cases there are restrictions such as a write message must

not arrive after an abort message. Since messages are acknowledged by owners, it is usually sufficient to wait for an acknowledgment. However in the presence of crashes, there is a need for a safe timeout value. We determine such a value by imposing an upper bound on the number of hops an update message is allowed to make. Messages which exceed that number of hops are discarded. Note that we assume that the maximum per-hop latency is known. A conservative value for such a timeout will not have an adverse impact on efficiency since the timeout is repeatedly used only in times of churn which we expect to be rare in cluster settings.

4. LSTP: PRIMITIVES

In general MVCCPs (Multi Version Concurrency Control Protocol) [6] works as follows. Whenever an item is updated, instead of overwriting it, a new version of the item is created. Reads avoid blocking by reading older versions of items when necessary. The protocol has to ensure that while the data read may be stale, it is never inconsistent. We first describe six primitive operations and then express our protocol in terms of the primitive operations. The six primitives are *write*, *conditionalWrite*, *commit*, *abort*, *lookup*, and *range query*. Note that these are primitive operations used by LSTP. A write by the application process does not translate to a simple write primitive but rather a more complicated protocol involving write and lookup primitives as defined in the next section. Similar arguments are also true for range query primitives.

A $write_{tid}(k_i, v)$ (abbreviated as $write(i)$ when tid is clear from the context) when executed on a copy of i results in the creation of a new version of i which includes the key k_i , new value v , a state s and the transaction id tid of the transaction attempting the write. A newly created version is in an uncommitted state and is migrated to the committed state if a $commit_{tid}(i)$ update is received and discarded if an $abort_{tid}(i)$ update is received. We allow at most one uncommitted version of an item. Sometimes it is necessary to atomically execute a write only if a condition holds and hence we define a primitive called the $conditionalWrite_{tid}(B, i)$ which is identical to the usual $write_{tid}(i)$ except that the write only succeeds if some condition from a small set of pre-defined conditions is satisfied. These conditionalWrites are analogous to the read-modify-write primitives available in centralized systems. Let $uncommitted(i)$ denote the uncommitted version of item i . Similarly, let $lastCommitted(i)$ denote the version last (in chronological order) committed on i .

We say a version of an item becomes *live* when the write which creates the version succeeds. A version remains live unless the corresponding $abort_{tid}(i)$ succeeds. This definition of liveness can be viewed as the generalization to versions of the liveness definition in [17]. Note that by this definition a committed version in the primary copy of an item remains live until it is purged. Therefore the live versions of an item include all the committed versions in the primary and possibly an uncommitted version. The consistent range query protocol developed in [17] guarantees that if a range query $[a, b]$ starts at time τ_1 and successfully ends at time τ_2 , all items in the range $[a, b]$ which were live during the entire time interval $[\tau_1, \tau_2]$ will be retrieved and also that no item which was not live anytime in $[\tau_1, \tau_2]$ will be retrieved and the other items may or may not be retrieved. Using the consistent range query protocol in our case would

result in retrieving for each item in $[a, b]$ at least all the versions which were live throughout $[\tau_1, \tau_2]$. To avoid this, we augment the range query primitive. In the augmented range query, the owners of $[a, b]$ are again contacted using the same consistent range query protocol but the peers instead of simply returning the item, perform a function called $evalVersions$ on the existing versions of i and return the result. Similarly, a lookup returns specific information about the versions of a specified item rather than the entire version history. The details of the functions evaluated by the peers are described later. But it is important to remember that the read operations involve active cooperation of peers rather than simple data transfer.

The replication layer provides no guarantee about the ordering of concurrent updates on the different replicas of an item. So we impose a set rules on the update primitives to obtain an ordering property which lays the foundation for the transaction protocols.

1. A $write_t(i)$ on an item i is accepted only if no uncommitted version of the item exists. If there exists a version already with tid t or if $t == uncommitted(i).tid$ t is treated as a duplicate. Otherwise $write_t(i)$ is rejected.
2. A user can issue a $commit_t(i)$ only for those items for which the corresponding write is known to have succeeded. But if a user wishes to abort a write which was rejected or has not succeeded, the user has to wait for at least the duration of replication timeout before first issuing $abort_t(i)$ into the system.
3. A $commit_t(i)$ or an $abort_t(i)$ is accepted only if t equals $uncommitted(i)$. It is treated as a duplicate otherwise.

From the first and third rules, it follows that a $write_t(i)$ can only be followed by either a $commit_t(i)$ or an $abort_t(i)$ in the history of any copy of i . Any other attempted write will be rejected or is a duplicate. In particular for a $write_{tid}(i)$ that succeeds this property holds in the primary of i and hence by the first replication guarantee, it must also hold in every subsequent primary of i .

The second rule together with the two replication guarantees ensures that if a $commit_t(i)/abort_t(i)$ and $write_t(i)$ are present in the history of a copy of i , the write always precedes the abort/commit. This is because a commit is sent only after a write is known to have succeeded (and hence already present in the history of all copies) and an abort is sent after waiting for the replication timeout (and hence the write will not be added to the history of any copy after that). Note that commit or abort updates are never rejected. They are either accepted or treated as a duplicate.

LEMMA 2. *Let a write w on an item i succeed at time τ . Then, at time τ , the last operation in the history of every copy of i is w .*

This lemma follows directly from rules 1 and 2.

The committed versions of a copy of an item can be ordered by the order in which they were committed; call such an ordering the *commit ordering* of the copy.

THEOREM 1. *At any time τ , the commit ordering of a primary copy of i is a prefix of the commit ordering of every secondary copy of i .*

This theorem can be proved easily using the above lemma and rule 3. In fact we can prove a stronger result; the commit ordering of the primary lags behind that of any secondary by at most one version. It follows that if reads are always evaluated at the primary copies then if one read observes a committed version then no subsequent read will miss that version.

5. LSTP: CONSISTENCY AND ISOLATION

5.1 Protocol Outline

Many MVCCPs maintain consistency using timestamps. One such MVCCP is the popular snapshot isolation(SI) which we use as the starting point for our protocol. In centralized SI [5], a transaction always reads from a snapshot of the (committed) data as of the time the transaction started, called its start or read number. When an update transaction is ready to commit it gets a commit number which is larger than any existing read or commit number. An update transaction aborts if it is detected that any of its update conflicts with a concurrent transaction (that has committed). When an update transaction commits, its changes are visible to all transactions which start with read numbers greater than the committing transaction’s commit number. Now let us consider how the protocol should be adapted to ensure the same semantics in the P-Ring case. Assume that we have a failure free coordinator (we’ll later show how this can be integrated into the ring with fault tolerance.). A transaction obtains the commit or read number from the coordinator. Now, if the coordinator maintains a mapping between the transaction id and the commit number of every committed transaction we can ensure SI semantics. This mapping is necessary because the commit numbers cannot be propagated instantaneously to all the objects updated by a committing transaction. When a reading transaction does not have enough information in the ring to conclude if a version belongs to a committed transaction with commit number less than the reading transaction’s read number, it probes the coordinator to find out. While this solution works, the coordinator has become a bottleneck of the system since it is repeatedly contacted by every transaction. One straightforward solution is to distribute the transaction-id commit number mapping and also the read number generation in the ring so that the coordinator needs to be contacted only for obtaining commit numbers. This makes the solution much more scalable but unfortunately gives rise to a subtle race condition. When the transaction id - commit number mapping was stored in the coordinator, the granting of a commit number and updating the transaction id - commit number mapping can be done “atomically“. That atomicity is no longer possible because the commit number is granted by the coordinator but the mapping is not stored in the coordinator. Instead it is written into the ring by the committing transaction and any transaction which tries to lookup this mapping during this interval enters into a race condition with the committing transaction.

We attack the above mentioned race condition in two distinct ways. First, we show that if a transaction is missed due to this race condition then it will not cause any inconsistencies via indirect dependencies. Then we develop a solution that avoids the case where inconsistencies due to direct dependencies can occur. We show that this solution while providing weaker isolation than snapshot isolation, still has a

number of interesting consistency properties. This solution ensures scalability because the only global synchronization point in the system is the commit number generator which is contacted only by update transactions at the time of committing. The read-only transactions which form the bulk of the workload never contact the commit number generator. Our second objective for the protocol was to ensure low overhead when compared with no transactional semantics. We show that the read phase reads at most three versions per item resulting in an overhead of a factor of 3. But in practice, as shown in our simulation section, the overhead is much lower for read-heavy workloads. For updates we introduce an extra lookup per update resulting in an overhead of one lookup per update. We believe, these overhead factors, are well within acceptable limits in return for the consistency properties we provide.

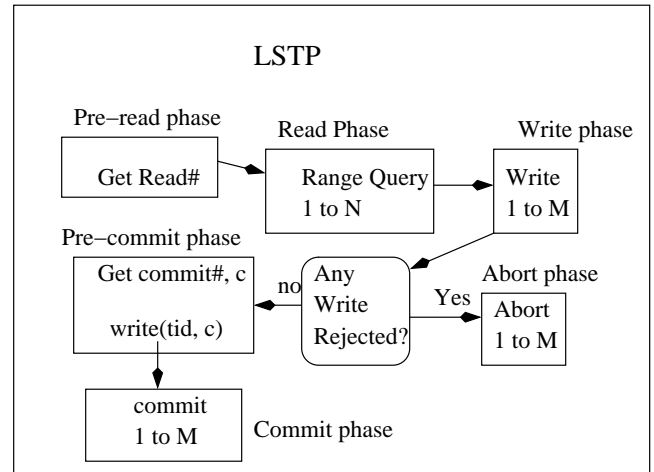


Figure 2: Phases of a Transaction

As shown in Figure 2, LSTP is composed of a number of non-overlapping phases. A transaction starts by obtaining its read number by probing random peers in the ring during a pre-read phase. Then, it enters the read phase. During this phase for every item it wishes to read, it tries to read the latest version with commit number not greater than its read number. As discussed earlier, due to the distributed nature of the transaction id - commit number mapping in our system, there may be some uncertainty about the commit number of some concurrent transactions. The reading transaction does not block in the event of such uncertainty and instead stores the id of such transactions locally in a set called the U-set. The U-set is built concurrently with the read phase and members of the U-set are always missed by the reading transaction. Read-only transactions have only these two phases. Update transactions enter into the write phase after their read phase. A write is allowed to succeed only if the read number of the writing transaction is greater than or equal to the commit number of the transaction that created the latest version of the item. Otherwise it enters the abort phase and aborts all attempted writes. Transactions which successfully complete their write phase, obtain a commit number from the commit number generators in the ring and write the commit number at a designated position in the ring, in a pre-commit phase. Finally, an update transaction enters its commit phase during which it propa-

gates its commit number to every version it created during its write phase.

5.2 Protocol Details

Commit number generation in the ring: The commit number generator has to satisfy the property that every time it responds to a request it should return a monotonically increasing number. We introduce a special key γ whose associated value is a counter. This item behaves just like any other item except that the only update allowed on the item is an increment request. The request is never rejected and if it makes it to the primary of the γ key, the primary increments the counter and returns the value to the user along with the acknowledgment. From the first replication update consistency property it follows that the counter in any secondary of the γ key, is incremented at least as many times as the counter in the primary. Also, before returning a commit number, the counter is incremented once in the primary, thereby guaranteeing that commit numbers granted are strictly monotonically increasing. Note that under our no-k-successive-peer failure assumption, commit number generators are fault tolerant. An update transaction has to obtain only one commit number which is obtained after all its writes are complete. Also, for the commit number generators neither lost requests nor duplicate requests impact the correctness since the granted commit numbers must only increase monotonically and need not be contiguous. There is no need to serialize concurrent commit number requests provided such requests are issued only after a transaction completes its write phase and the granted commit number is greater than its read number.

Pre-Commit Phase: After obtaining its commit number by contacting the commit number generators mentioned above, an update transaction with id tid writes within a separate namespace of P-Ring, the item $\{tid, commit\# \}$. Recall that the ring can be divided into different namespaces to store data of different types in the same ring. In our case, we divide the ring into two namespaces; one to store the actual data and the other to store the commit numbers of transactions. This write need not be committed since transaction ids are unique and hence there is no chance of contention on the item. Transactions which need to know the commit number of a transaction with id tid , can do so using a $lookup(tid)$. Note that such a lookup is not frequently needed since commit numbers are also stored with committed versions.

Commit/Abort Phase: During its commit phase, an update transaction sends a commit update to all the items on which an uncommitted version created by the transaction exists. The versions upon receiving the update are converted to committed versions and the commit number of the transaction are stored along with the versions. Abort phase is also similar except that the transaction has to wait for the replication timeout before starting it and uncommitted versions are discarded upon receiving an abort update. Read-only transactions have no commit or abort phase and hence are abort free.

Generating read numbers: Unlike the commit number which is needed only by update transactions before their commit, read numbers are needed by all transactions before they perform any other operations. Therefore it is necessary to decentralize the process by which transactions obtain read numbers. In LSTP, a new transaction probes a

few random peers in the ring and uses the maximum commit number known to those peers as its read number. Each peer locally learns some commit numbers from the commit numbers stored in the data for which the peer is the owner. These commit numbers are either those stored with committed versions or items in the transactional namespace. This locally learned information is continuously updated by gossiping with other peers and using the maximum known commit number. Therefore, the only guarantee about the read number of a transaction is that it corresponds to the commit number of some transaction T_r which had completed its pre-commit phase before this transaction starts performing operations. In snapshot isolation, the read number of a new transaction is greater than the commit number of all transactions which committed earlier. The weaker guarantee provided to read numbers in LSTP is the price of decentralizing the read number generation and the consistency properties we develop has to build upon this guarantee. The closer the read number of a transaction is to the maximum granted commit number, the more recent is the transactions “view“ of the index.

PROPERTY 5.1. *For any committed (committing) update transaction T , $commit\#(T) > read\#(T)$.*

The above property holds because the read number is the commit number of some transaction which has obtained its commit number before T started. Also, T obtains its own commit number after its read and write phases are over and finally granted commit numbers are strictly monotonically increasing.

A transaction T_1 is said to *directly depend* on another transaction T_2 if in the dependency serialization graph there is a WW edge or WR edge directed from T_2 to T_1 . Similarly, T_1 *directly anti-depend*s on T_2 if there is a RW edge directed from T_2 to T_1 . Finally, a transaction T_1 is said to *depend* on a transaction T_2 , if in the dependency serialization graph, there is a chain of direct dependencies directed from T_2 to T_1 .

Protocol 1 LSTP:Write Phase

```

1:  $W :=$  write set of a transaction  $T$ .
2: for each item  $i$  in  $W$  do
3:    $T$  obtains  $tid$  and  $c$  by performing  $lookup(i)$ .
4:    $tid :=$  txn id of the last committed version of  $i$ .
5:    $c :=$  commit number of txn corresponding to  $tid$ .
6:   if  $tid$  is in  $T$ 's U-set or  $c > T$ 's  $read\#$  then
7:      $T$  enters abort phase.
8:   else
9:      $T$  sends a conditionalWrite  $CW$  to owner of  $i$ 
10:     $CW$  requires that the last committed version of  $i$  remains
         $tid$ .
11:    if  $CW$  is rejected then
12:       $T$  enters abort phase.
13:    end if
14:  end if
15: end for

```

Write Phase: The write phase is listed in Protocol 1. In order to write on an item i , a transaction T with read number r , first looks up (line 3) the last transaction T_{tid} to commit on i . If T_{tid} has commit number greater than r (line 6), T aborts. T also aborts if T_{tid} belongs to the U-set (explained later). Otherwise T attempts a conditionalWrite. The write is conditional to ensure that some other transaction did not commit in the time interval between the lookup by T and the write attempt by T . If the condition does not

hold, the write will be rejected. Finally, recall from the basic primitives that a write will be rejected if an uncommitted version of an item exists. If the conditionalWrite is rejected due to either of the two above mentioned cases, T aborts. To summarize, T succeeds in creating a new version of i , only if all the existing versions of i were created by transactions having commit number $\leq r$. This observation when combined with Property 5.1 gives rise to the following property.

PROPERTY 5.2. *Let T_1 and T_2 be any two committed (committing) transactions. If there is a WW edge directed from T_1 to T_2 then $\text{commit}\#(T_2) > \text{commit}\#(T_1)$.*

Also note that T_1 overwrites T_2 only if T_2 has created a committed version of i . Since a transaction creates committed versions only after finishing its precommit phase successfully, the next property follows.

PROPERTY 5.3. *Let T_1 and T_2 be any two committed (committing) transactions. If there is a WW edge directed from T_1 to T_2 then the pre-commit phase of T_1 ended before the write phase of T_2 ended.*

Protocol 2 EvalVersions

```

1: Definition of evalVersions:
2: Evaluated by the owner of a given item i.
3: To answer range query request by a txn with read# r.
4: answer = {lcv, plcv, u} triplet.
5: Initially, lcv = plcv = u = null
6: if commit# of last committed version of i < r then
7:   lcv = latest committed version with commit# < r.
8:   plcv = version immediately preceding lcv, if any
9: else
10:  lcv = last committed version of i.
11:  plcv = version immediately preceding lcv if any
12:  if uncommitted version of i exists and has read# < r then
13:    u = uncommitted version of i
14:  end if
15: end if
16: return {lcv, plcv, u}

```

Read Phase: In its read phase, a transaction T executes a series of range queries using the augmented range query primitive. The read number, r , of the transaction is attached along with the query. The peers evaluate the function EvalVersions (see protocol 2) using r as the parameter on the versions of every item which falls within the query's range. The result set of this function is at most three versions per item which are then returned to the querying transaction. The transaction then uses the commit and read numbers associated with these versions and its local U-set (explained later) to pick one version per item which is presented as the actual result of the range query. First we describe, how the three versions are selected, given a read number. Then we discuss why the three versions are sometimes necessary and always sufficient to produce a consistent result set.

The peers processing a range query use the following algorithm (see Protocol 2, EvalVersions) to select the at most three versions per item. If the last committed version of item i has commit number $\geq r$, then the latest committed version ($\text{lcv}(i)$) of i with commit number $\leq r$ and the version penultimate to the $\text{lcv}(i)$ are returned. Thus in this case at most two versions are returned (It is at most because the penultimate version may not exist). If the last committed version of item i has commit number $< r$, then again the

$\text{lcv}(i)$ (which in this case is synonymous to the last committed version of i) and the version penultimate to it are returned. In addition, if an uncommitted version of i exists and has read number (stored along with the version) less than r , that is returned as well for a total of at most three versions.

Next, let us discuss the algorithm (see Protocol 3) used by a user process with read number r in deciding which of the three versions per item is appropriate. Let $\langle u, \text{lcv}, \text{plcv} \rangle$ denote the 3 versions where a version which does not exist is set to null. Let T_u, T_{lcv} and T_{plcv} be the respective transactions that created these three versions. We already know that the commit numbers of T_{lcv} and T_{plcv} are $\leq r$. From lines 8, 10 and 12 of Protocol 3, it is clear that the uncommitted version is used only if it is confirmed (using a lookup into the transaction space) to have a commit number $\leq r$. That is, irrespective of which of 3 versions is chosen, the chosen version has commit number $\leq r$. This fact, when combined with Property 5.1, gives rise to the following property.

PROPERTY 5.4. *Let T_1 and T_2 be any two update transactions. If there is a WR edge from T_1 to T_2 , then $\text{commit}\#(T_2) > \text{commit}\#(T_1)$.*

Combining Property 5.2 and 5.4 we get

PROPERTY 5.5. *Let T_1 and T_2 be any two transactions. If T_2 directly depends on T_1 , then $\text{commit}\#(T_2) > \text{commit}\#(T_1)$.*

A simple argument based on induction extends the above property as follows.

PROPERTY 5.6. *Let T_1 and T_2 be any two update transactions. If T_2 depends on T_1 , then $\text{commit}\#(T_2) > \text{commit}\#(T_1)$.*

Also note that an uncommitted version u of an item is used as a result of a read, only if a lookup of T_u succeeded, i.e., T_u has finished its pre-commit phase. Also note that a write phase follows a read phase. These two observations when combined with property 5.3, gives rise to the following property.

PROPERTY 5.7. *Let T_1 and T_2 be any two update transactions. If T_2 depends on T_1 , then T_1 finished its precommit phase successfully before T_2 's write phase ended.*

From among the $\langle u, \text{lcv}, \text{plcv} \rangle$ triplet, the version chosen by a reading transaction T corresponds to the version with largest commit number less than r and not present in T 's U-set. The first part of the two conditions is as expected but the second part is unusual and unique to the conditions of our system. What is the U-set and why is it necessary? As already mentioned, u is the chosen version only if it is confirmed by the $\text{lookup}(u.\text{tid})$ that u has commit number $\leq r$. Otherwise u is ignored, i.e., T misses T_u . If u happened to have commit number greater than r then missing T_u is the expected behavior. But what if u has commit number $\leq r$? This is possible because T_u might have been granted a commit number less than r but was slow in writing this commit number into the index. We call this the *slow committer* problem because had T_u been quick enough in writing its commit number in the transaction namespace, this problem would never have occurred. Since T has missed T_u , the following two conditions are necessary to avoid inconsistency.

- I1. T depends on no other transaction that depends on T_u .
- I2. If T reads another item on which T_u has created a version, T misses T_u on that item as well.

The next property shows that the inconsistency represented by the absence of condition I1 can never occur.

PROPERTY 5.8. *Let T_u be a slow committer wrt transaction T having read number r . Then there exists no other update transaction with commit number less than or equal to r that depends on T_u .*

PROOF. For contradiction, assume that such an update transaction, T_x , exists. Since r is the read number of T , it has to be the commit number of some transaction T_r which finished its pre-commit phase before T started. This implies, every transaction with commit number less than r , finished their write phase before T started. Hence T_x should have finished its write phase before the pre-commit phase of T_u ended (since T_u is a slow committer wrt T). This contradicts Property 5.7. \square

Protocol 3 LSTP:Read Phase

```

1: Actions by a transaction T to evaluate a range query [a, b].
2: U = U-set of T, empty before the first range query.
3: R is the result set for range query [a, b], initially empty.
4: T contacts peers owning [a, b] using range query primitive.
5: for each item i in [a, b] do
6:   Owner of i evaluates function evalVersions on versions of i.
7:   Owner responds with lcv, plcv, u triplet as defined in evalVersions (protocol 2).
8:   if u ≠ null and txn id, u.tid, is not in U then
9:     T obtains uc# using a lookup of u.tid in transactional namespace.
10:    uc# := commit# of txn with id u.tid.
11:    if u.tid is not present in transactional namespace then
12:      U = U ∪ u.tid
13:    else if uc# ≤ read# of T then
14:      R = R ∪ u
15:      continue to next item
16:    end if
17:  end if
18:  if (txn id of lcv is not in U) then
19:    R = R ∪ lcv;
20:    continue to next item
21:  else if plcv ≠ null then
22:    R = R ∪ plcv
23:  end if
24: end for

```

Since by Property 5.8 no transaction with commit# $< r$ depends on T_u and also since T only depends on transactions with commit numbers $\leq r$, there is no chance of T observing some other transaction which depends on T_u . The inconsistency specified in condition I2 can still occur because even though T_u had not finished its pre-commit phase the first time a version created by it was encountered by T , T_u might have finished its pre-commit phase by the time T again encounters a version created by T_u on another item. This is avoided by T by remembering the id of T_u in a set of transactions called the U-set. When T encounters a committed version of some item j , created by a member of the U-set, by condition I1, it has to be lcv(j). Therefore T uses the version penultimate plcv(j) to it. Note that plcv is guaranteed to be not created by a slow committer since there is a WW edge between T_{plcv} and a transaction with commit number $\leq r$ (See Property 5.8). Thus T ensures both I1 and I2.

A transaction id tid is assumed to belong to a potential slow committer and added to the U-set of a transaction T if all of the following conditions hold.

- C1. T encounters an uncommitted version u_{tid} on an item i for the first time during its read phase.
- C2. The read number stored along with u_{tid} is less than T 's read number.
- C3. The last committed version (if any) of i has commit number less than T 's read number.
- C4. A lookup(tid) by T returns false.

Conditions C1 and C4 ensure the definition of a slow committer. Conditions C2 and C3 are used to reduce the number of false positives in the U-set and thereby reduce the need for the lookup in C4. False positives in the U-set arise because it is not possible to distinguish between an active transaction (i.e, a transaction that is still in its read or write phase) and a slow committer (neither of which store any state in the transactional key-space). From Property 5.1 and the definition of a slow committer it follows that the read number of a slow committer wrt a transaction T must be less than T 's read number. This is used by condition C2 to reduce false positives. Condition C3 reduces false positives by using the fact that the write protocol ensures that within a commit ordering the commit numbers increase monotonically. It can be shown that if the U-set of a transaction T is built according to rules C1 to C4 then if T reads the same part of the index more than once, the result set will be identical. Therefore LSTP can support repeating reads without access to a local cache.

PROPERTY 5.9 (PROPERTY OF CYCLES-1). *Let G be the dependency graph of a system using LSTP, then G cannot have a cycle composed entirely of WW and WR edges.*

The proof is a straight forward application of Property 5.6.

PROPERTY 5.10 (PROPERTY OF CYCLES-2). *Let G be the dependency serialization graph of LSTP. Then there exists no cycle in G with only one RW edge.*

PROOF OUTLINE. Let $T_1T_2...T_NT_1$ be a cycle in G . Let T_NT_1 be the single RW edge in the cycle. The path $T_1T_2...T_N$ is made entirely of WW and WR edges and hence $T_N.read\# \geq T_1.commit\#$. Let i be an item which caused the RW edge between T_N and T_1 . Then T_1 must be in the U-set of T_N (otherwise T_N would not have missed the write by T_1 and hence there would be no RW edge). But then the edge T_1 to T_2 cannot exist (by Property 5.8) unless the cycle is of the form $T_1T_NT_1$. But this implies there is a WW or WR edge between a transaction in T_N 's U-set and T_N which is not possible by the specification of the protocol. \square

The above property is also shared by snapshot isolation (SI) which hints at a close relation between LSTP and snapshot isolation. But in SI, any cycle in the dependency serialization graph must not only have at least two RW edges but two adjacent RW edges [11]. The adjacency condition is very significant only in some special cases. For example, if in an application adjacent RW edges can never occur, then SI guarantees serializability but LSTP doesn't. Therefore LSTP provides transactions with weaker but similar

notions of isolation as does snapshot isolation. The widespread use of SI in read heavy environments (for example, snapshot isolation is used in ORACLE and recent versions of SQL server) leads us to hypothesize that LSTP will also be useful in many large scale read heavy environments.

Next we describe the formal consistency properties of LSTP. A transaction T is said to be provided with *basic* (aka *update* or *external*) consistency if the values read by T are the result of a serial execution of some subset of committed update transactions and each update transaction in the serial transaction execution executes the same steps as it did in the concurrent execution [27]. A concrete example for a practical use of basic consistency is the broadcast environment described in [23].

THEOREM 2. : *There are no dependency misses in LSTP*

THEOREM 3. : *LSTP provides transactions with basic consistency when update transactions are serializable.*

The two properties of cycles are sufficient conditions for the above two theorems to hold [1].

Informally Theorem 3 implies if the updates are well behaved, the read-only transactions are guaranteed to be consistent. This is precisely the characteristic of read heavy environments since concurrent, contending updates are rare. Indexes which are updated in bulk is an example workload which exhibits this kind of characteristic. Thus LSTP is ideal for read-heavy environments.

Purging of Old Versions: So far in the discussion, we have ignored the purging of old versions. The commonly used strategies for purging old versions using either the age of the version or bounding the maximum number of versions retained per item are also applicable in our setting. But a couple of points are worth noting. Irrespective of the strategy followed, it is necessary to retain at least the last two committed versions of an item due to the slow committer problem. If the workload consists of long running read-only transactions and the abort free nature of read only transactions is desired, then retaining old versions for a long time is inevitable. Version management in P2P systems is an interesting problem in its own right and has applications in other domains such as collaborative document editing.

6. PRELIMINARY EVALUATION

We implemented LSTP in a simple round based C++ simulator and obtained some empirical results. These results are very preliminary in nature due to the simplicity of the simulation environment and limited scope of the experiments. We are currently working on a comprehensive and realistic implementation. Our goal in this preliminary evaluation is to obtain some insight about the overhead imposed by LSTP when compared to simply executing the queries and updates as unrelated singletons. The size of the index is 100000 data items distributed over 10000 peers. A replication factor of 3 is used. We experimented with three different workloads and two different query distributions. The first query distribution distributes queries and updates uniformly in the dataspace while the second distribution distributes queries and updates normally with mean 50000 (center of the data space) and standard deviation of 10000 (10% of the data space). In the first workload readonly transactions execute 10 range queries (of width 100) each while update

transactions execute 10 updates each. The second workload is twice as heavy as the first workload while the third workload is thrice as heavy.

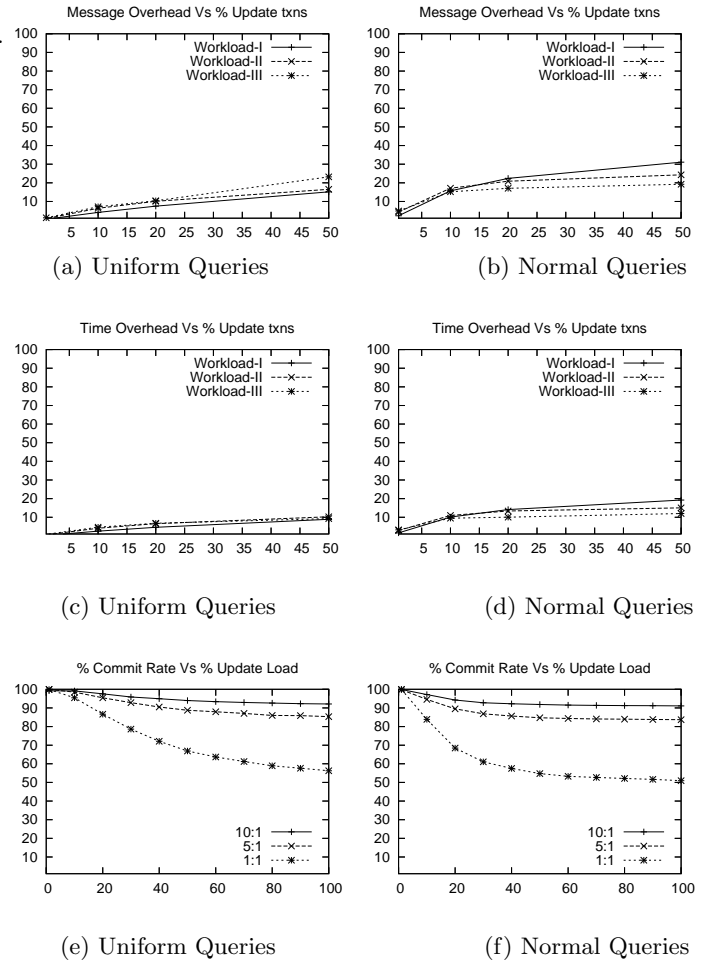


Figure 3: Simulation Results

In a purely read only environment the overhead imposed by LSTP is 0. Hence here we examine the over-head as the percentage of update transactions in the system increases. Figure 3(a) and (b) show the percentage overhead of LSTP in terms of number of messages against the percentage mix between read-only transactions and update transactions. Figures 3(c) and (d) show the overhead in terms of time which is different from that of messages because messages are often sent in parallel. The graphs show that the overhead depends largely on the proportion of update transactions in the system and to a much lesser extent on the load per transaction. The overhead in both time and number of messages show similar behavior but the overhead is lower in terms of time. In the uniform workloads the overhead increases linearly while the behavior is more irregular for the skewed workloads. In no case does the overhead rise above 30% thereby validating our claim that LSTP is a low overhead protocol.

Figures 3 (e) and (f) show the percentage of commit rate when the amount of contention in the system increases. We use the total number of items updated by the workload as a

measure of the update load. The commit rate declines heavily in the worst of all cases: when the workload is skewed, update transactions to read-only transactions ratio is 1:1 and the number of items updated in total is equal to the size of the index. But we note that this is an extreme case wherein the system is under extremely high load and contention. Under more reasonable parameters such as when workload updates 20% of the index, the commit rate is at acceptable values. The conclusion is that LSTP provides acceptable commit rates for read-heavy environments but is unsuitable for heavy contention, high update-load environments.

A few heuristics are possible to reduce the rate of aborts among update transactions. For example, when a transaction T crashes in the middle of a commit phase some of its written items are in the uncommitted state which prevents write attempts by another transaction T1 on the same items being rejected. Instead of moving on to its abort phase T1 can use the lookup(*tid*) and check if T completed its precommit phase. If the lookup returns a positive result, then the transaction can commit by proxy by sending a commit(*tid*) even though the user process of *tid* is different. But given that the protocol is fundamentally biased towards read heavy environments we do not consider this issue any further.

7. CONCLUSION

We argued that the use of structured P2P indexes should be adopted in cluster computing scenarios. We identified the lack of consistency guarantees to be one of the major stumbling block in such an adoption and hence studied the problem of providing transactional semantics to P-Ring a P2P system that efficiently answers range queries. We developed LSTP a scalable transactional protocol that guarantees no missed dependencies and provides read-only transactions with basic consistency when update transactions are serializable. We believe that the design and preliminary implementation of LSTP in this paper acts as a proof of concept that notions of consistency can indeed be added to structured P2P systems in cluster environments.

8. REFERENCES

- [1] A. Adya. *Weak Isolation: A generalized theory and optimistic implementations for distributed transactions*. PhD thesis, Massachusetts Institute of Technology, March 1999.
- [2] Amazon simple storage service. <http://www.amazon.com/gp/browse.html?node=16427261>.
- [3] L. Barraso, J. Dean, and U.Hölzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23:22–28, 2003.
- [4] M. Bawa, A. Gionis, H. Garcia-Molina, and R. Motwani. The price of validity in dynamic networks. In *SIGMOD*, 2004.
- [5] H. Berenson, H. Bernstein, J. Gray, J. Melton, E. O’Neil, and P.O’Neil. A critique of the ansi sql isolation levels. In *SIGMOD*, 1994.
- [6] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wessley Publishing Company, 1987.
- [7] A. Bosworth. Database issues for the 21st century. In *SIGMOD*, 2005.
- [8] A. Chan and R. Gray. Implementing distributed read-only transactions. *IEEE transactions on Software Engineering*, 11:205–212, 1985.
- [9] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram. P-ring: An efficient and robust p2p range index structure. In *SIGMOD*, 2007.
- [10] F. Dabek. *Cooperative File System*. PhD thesis, Massachusetts Institute of Technology, September 2001.
- [11] A. Fekete, D. Liarokopis, E. O’Neil, P. O’Neil, and D. Shasha. Making snapshot isolation serializable. *ACM transactions on Database Systems*, 30:492–528, 2005.
- [12] B. Gedik and L. Liu. Peercq: A decentralized and self-configuring peer-to-peer information monitoring system. In *ICDCS*, 2003.
- [13] Google alerts. <http://www.google.com/alerts>.
- [14] A. Gupta, O. Sahin, D. Agrawal, and A. E. Abbadi. Meghdoot: Content-based publish/subscribe over p2p networks. In *Middleware*, 2004.
- [15] R. Huebsch, B. Chun, J. Hellerstein, B. Loo, P. Maniatis, T. Roscoe, S. Shenker, I. Stoica, and A. Yumerefendi. The architecture of pier: an internet-scale query processor. In *CIDR*, 2005.
- [16] J. Kubiawicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *ASPLOS*, 2000.
- [17] P. Linga, A. Crainiceanu, J. Gehrke, and J. Shanmugasundaram. Guaranteeing correctness and availability in p2p range indices. In *SIGMOD*, 2005.
- [18] G. Lomet. Replicated indexes for distributed data. In *PDIS*, 1996.
- [19] N. Lynch, D. Malkhi, and D. Ratajczak. Atomic data access in distributed hash tables. In *IPTPS*, 2002.
- [20] D. Narayanan, A. Donnelly, R. Mortier, and A. Rowstron. Delay aware querying with seaweed. In *VLDB*, 2006.
- [21] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM*, 2001.
- [22] S. Rhea, D. Geels, T. Roscoe, and J. Kubiawicz. Handling churn in a dht. In *USENIX*, 2004.
- [23] J. Shanmugasundaram, A. Nithrakashyap, R.Sivasankaran, and K. Ramamritham. Efficient concurrency control for broadcast environments. In *SIGMOD*, 1999.
- [24] M. Srivatsa and L. Liu. Vulnerabilities and security threats in structured overlay networks: A quantitative analysis. In *ACSAC*, 2004.
- [25] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, 2001.
- [26] C. Türker, K. Haller, C. Schuler, and H. Schek. How can we support grid transactions? towards peer-to-peer transaction processing. In *CIDR*, 2005.
- [27] W. E. Weihl. Distributed version management for read-only actions. *IEEE transactions on Software Engineering*, SE-13:55–64, 1987.