Building Replicated Internet Services Using TACT: A Toolkit for Tunable Availability and Consistency Tradeoffs

Haifeng Yu Amin Vahdat
Computer Science Department
Duke University
Durham, NC 27708
{*yhf, vahdat*}*@cs.duke.edu*

Abstract

An ultimate goal for modern Internet services is the development of scalable, high-performance, highly-available and fault-tolerant systems. Replication is an important approach to achieve this goal. However, replication introduces the issue of consistency among replicas, which is further complicated by network partitions. Generally, higher consistency levels result in lower system availability in the presence of network partitions. Thus, there is a fundamental tradeoff between consistency and availability in building replicated Internet services.

In this paper, we argue that Internet services can benefit from dynamically choosing availability/consistency tradeoffs. With three consistency metrics, Unseen Writes, Uncommitted Writes *and* Staleness*, we show how consistency can be meaningfully quantified for many Internet services. We present the design of the TACT (Tunable Availability and Consistency Tradeoffs) toolkit that allows Internet services to flexibly and dynamically choose their own availability/consistency tradeoffs, enabling differentiated availability/consistency quality of service. Further, TACT makes it possible for Internet services to dynamically trade consistency for performance.*

1. Introduction

An ultimate goal for modern Internet services is the development of scalable, high-performance, highly-available and fault-tolerant systems. Replication is an important approach to achieving this goal. For example, Exodus[14] has replicas in nine metropolitan areas, while CNN[7] is replicated across several continents. However, replication introduces issues of consistency among replicas. The possibility of network partitions in wide-area networks further complicates this problem. Given the tremendous scale of the Internet, it is likely that a partition is present somewhere in the Internet at all times. Network congestion and node failures can also be considered less severe forms of network partitions. Generally, in the presence of network partitions, higher consistency levels result in lower system availability [10]. Thus, the tradeoff between consistency and availability is a fundamental issue that must be addressed by replicated Internet services.

Researchers have been concentrating on the two extreme end points of the availability/consistency tradeoff spectrum (Figure 1(a)): maximizing availability while maintaining strong consistency[1, 2, 10, 12, 16, 37], or sacrificing strong consistency in favor of one-copy availability [6, 19, 20, 24, 30, 31, 32, 35, 40]. As a result, in Figure 1(a), applications typically have only two choices: either strong consistency with whatever availability the system provides, or one-copy availability with whatever consistency the system provides.

In this paper, we argue that Internet services can benefit from dynamically choosing availability/consistency tradeoffs in response to current network, service and access characteristics. For instance, a replicated Internet stock quote service may provide differentiated availability/consistency Quality of Service (QoS) for different users. With the advent of mobile code and web hosting services, replicas of Internet services may be hosted by distinct administrative domains[42]. Autonomy requires that each domain be allowed to specify its own availability/consistency tradeoff.

The need to build adaptive systems also argues against static availability/consistency tradeoffs. Consider the case of an airline reservation system. The airline determines an optimal availability/consistency tradeoff that can maximize its profits without unduly sacrificing customer satisfaction. However, the optimal tradeoff depends on user access patterns, network performance and network partition scenarios. As these factors change, the reservation system should dynamically adjust system availability/consistency

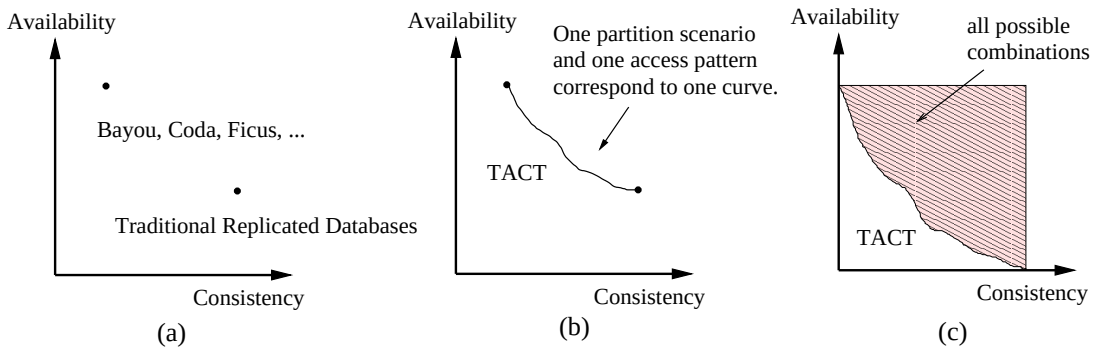

Figure 1. Availability/consistency tradeoff

to achieve the optimal tradeoff. For example, as a flight gets full, the airline may want to lower availability to tighten consistency. In the next section, we analyze the availability/consistency requirements of a number of Internet services in more detail.

In this paper, we present the initial design of the TACT (Tunable Availability/Consistency Tradeoff) toolkit for replicated Internet services. The goal of our work is to provide flexible availability/consistency tradeoffs for modern Internet services. TACT is a library layer between the database replica and the application. It defines high-level *continuous* metrics for consistency, which are (linearly) related to the consistency observed by the Internet service. More specifically, the consistency of a replica in TACT is measured by three metrics: *Unseen Writes*, *Uncommitted Writes* and *Staleness*. Intuitively, *Unseen Writes* is the number of updates not seen by a replica, *Uncommitted Writes* is the number of unstable updates on a replica, while *Staleness* is the age of a replica relative to an up-to-date replica image. Using our metrics for consistency, Internet services can flexibly and dynamically specify their own requirements for the availability/consistency tradeoff. In Figure 1(b), given a partition scenario and workload, applications can choose arbitrary availability/consistency tradeoff on the spectrum. Figure 1(c) shows all possible availability/consistency tradeoffs in TACT. Internet services are allowed to adjust the tradeoff whenever deemed necessary, for example, when a network partition occurs or when it temporarily desires stronger consistency for more accurate information. TACT can also provide different consistency levels for each database replica and each database access (read or write), enabling differentiated availability/consistency QoS.

Consistency and availability can also be traded for performance. For example, achieving strong consistency in wide-are networks incurs significant communication overhead. Generally speaking, higher consistency levels require tighter synchronization and more communication on the critical path, which result in lower system throughput and latency. We believe that TACT can be readily used by Internet services to relax consistency for higher performance. However, for the purposes of this paper, we focus on availability/consistency tradeoffs.

This paper makes the following contributions:

- We describe the benefits of dynamically choosing availability/consistency tradeoffs and demonstrate how Internet services can utilize such support.
- We quantify consistency by assigning high-level continuous metrics for it. We explain how continuous consistency can be meaningful for many applications, although consistency is traditionally considered binary.
- The initial design of the TACT toolkit is presented, which allows Internet services to flexibly and dynamically choose availability/consistency tradeoffs, as well as to dynamically trade consistency for performance.
- Finally, our work demonstrates how consistency can be specified on both a per-replica and a per-user basis and how such flexibility enables dynamic QoS for Internet services.

In the next section, we analyze the availability/consistency requirements of several Internet services. We quantify consistency and assign continuous metrics for such services in Section 3. Section 4 presents the design of the TACT toolkit. Related work is described in Section 5. In Section 6, we conclude and describe future work.

2. Motivating applications

The TACT toolkit is designed to support replication in wide-area networks. Because replication is a primary approach for addressing performance, availability and fault-tolerance issues, we expect that a wide range of Internet services can benefit from the TACT toolkit. These Internet services include E-commerce systems, stock-trading systems,

web servers, web caching, wide-area resource allocation, bulletin boards, etc. We briefly describe how three particular Internet services can utilize TACT and how TACT addresses their requirements.

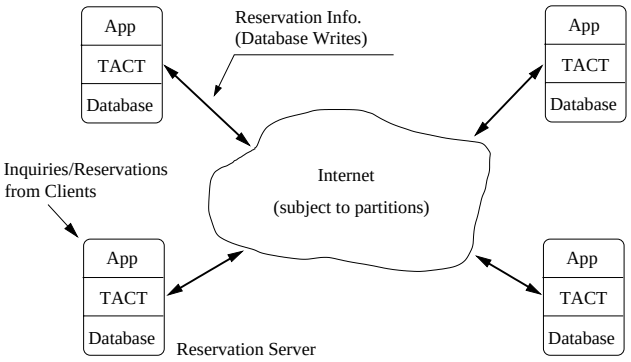

Figure 2. Replicated airline reservation system

Airline reservation system Replicated reservation servers (Figure 2) accept user reservations and inquiries about seat availability. Reservations can specify the **exact** seats desired. Inconsistency can result in reservation conflicts and the need to abort a reservation. Availability is defined as the percentage of time that the system is available, while consistency determines the percentage of reservations that are eventually aborted. Using the TACT toolkit, the airline can choose a point in the tradeoff spectrum of availability and consistency based on application-specific characteristics, such as user access pattern.

Stock trading system One popular model for performing stock trades is the market-maker paradigm. Here, a request to buy or sell a stock can be directed to any of the market-makers, who are responsible for matching buy and sell orders. Clients desire information about the trades performed (and requested) by market-makers distributed across the world. They can retrieve the trade information from any market-maker. Propagating trade information is essentially maintaining consistency among database replicas. This is an Internet service where differentiated availability/consistency QoS is crucial, because different market-makers and clients have different requirements for the freshness of their data.

Web caching In read-only web caching, the time-to-live (TTL) property has been used to bound the staleness of cached web pages. However, we argue that in many cases, bounding the staleness of a web page is inappropriate. For example, if a web page's TTL is set to one hour, a proxy cache has to contact the web server once per hour even if the page has not been modified. On the other hand, modifications can be bursty and the web page may be completely updated within 30 minutes. In the TACT toolkit, consistency is measured using three metrics and the above case can be handled gracefully by bounding one of the metrics (*Unseen Writes*). Furthermore, the TACT toolkit allows different availability/consistency QoS based on user/cache preferences.

3. Quantifying availability and consistency

In order to enable the Internet service to specify an arbitrary point in the availability/consistency tradeoff spectrum, the TACT toolkit assigns numerical values to availability and consistency. Availability can be well defined using a numerical value. Consistency, on the other hand, is traditionally considered binary. A system is either "consistent" or "inconsistent." Recent efforts attempting to define different consistency levels include [3, 13, 21, 22, 23, 33, 34, 36, 38, 39, 41]. Some approaches are restricted to the case of read-only caching, while others are either low-level or single-dimensional and cannot reflect applications' consistency requirements (see Section 5).

In this section, we first describe the basic replication system model we assume, then discuss the guidelines we use in quantifying consistency. After analyzing the consistency requirements of two Internet services, we present our consistency metrics.

3.1. System model

For simplicity, in this paper we refer to the application data as databases, though the data can actually be stored in other formats such as files, persistent objects, etc. The database is replicated across multiple *server*s. Each server maintains a full copy of the database. Servers accept read/write requests from users. In our model, a read or write can contain multiple primitive database read/write operations. (In some sense, a read in our model is the analog of a read-only transaction in traditional database terminology, while a write is an update transaction.) In the airline reservation example, an inquiry about flight information is a read, and a reservation is a write. To perform a read/write on the database, a server accesses its local copy of the database image through the TACT layer. On a server, a read or write is isolated from other reads or writes during execution. Depending on the consistency requirements specified, a server may or may not contact other servers when processing read/write requests.

A server may reject a read/write request to the database. This can occur when the network is partitioned and the server is unable to contact some other servers to attain the desired level of consistency. In such a case, availability is traded for consistency as required by the application.

Although we are exploiting weak consistency, we believe that eventual consistency should be preserved by the system. In the absence of newly-introduced writes, all replicas in an eventually consistent system will converge to the same "final image." If a system is not eventually consistent, inconsistency is allowed to accumulate as time goes by, ultimately making the replicas useless for many Internet-based applications. Eventual consistency requires that all writes ultimately be propagated to each server. A method is also required for servers to determine a total *commit order* on all writes in the system, so that writes can eventually be applied to the database in the same order across the system.

There are two kinds of writes in the system, *uncommitted writes* and *committed writes*. When a server first accepts a write, the write is *uncommitted*, which means other servers may or may not honor this write and the position of this write in the commit order is not determined. The local server applies the uncommitted writes to its database image once they are accepted, possibly before those writes are actually propagated to other replicas. A write gets committed when its position in the commit order is fixed.

3.2. Guidelines for quantifying consistency

In defining metrics for consistency, we believe it is crucial to make them:

1. **High-level.** Metrics should be directly related to the application's view of consistency. It is especially important that a numerical value be assigned to consistency that is proportional to the consistency observed by the application. For instance, in the airline example, the consistency observed by the application is the rate that reservations are later aborted. Thus, if we define some numerical value for consistency, relaxing consistency from 0.9 to 0.8 should have the same effect on the aborted reservation rate as relaxing consistency from 0.2 to 0.1. Furthermore, to provide flexibility in implementation and to make the metrics widely applicable, they should be implementation-independent. The system model in which the metrics are defined should not introduce additional restrictions.

2. **Full-coverage.** Metrics should measure and specify the consistency level at any point on the availability/consistency spectrum (from strong consistency in traditional databases to weak consistency in optimistic replication systems). This is necessary for TACT to export arbitrary tradeoffs on the spectrum to the application.

3. **Multi-dimensional.** Consistency has multiple dimensions and cannot be captured using a single metric. For example, in the web caching example, using a single TTL metric to measure consistency is inaccurate in many cases. Taking the number of modifications on a web page into account will make the metrics much more accurate. Thus, consistency should be quantified across multiple dimensions and these metrics should be both necessary and sufficient to capture the requirements of a broad range of applications.

3.3. Defining consistency metrics that meet applications' needs

One goal of our work is to define a set of metrics to quantify the consistency requirements of a wide range of real Internet services. In this section, we derive the consistency metrics used in TACT by analyzing the consistency of several typical Internet services.

In the airline example, inconsistency comes from two sources. First, it can come from unseen reservations. For example, someone reserves a *particular* seat on a flight on server A, but the reservation information has not been propagated to server B. If another user makes a reservation for the same seat based on inconsistent data at B, the second reservation will be aborted later. The more unseen reservations, the higher the probability that a new reservation will conflict with an unseen reservation, and the lower the observed consistency level of this application. In fact, the probability that a new reservation will be later aborted because of an unseen reservation grows linearly with the number of unseen reservations for a particular replica.

The second source of inconsistency is the "uncommitted reservations" applied to the local database image. When first accepted by a server, a reservation is "uncommitted" because it may or may not be honored by other servers. However, that server still applies the database update (reservation) to its database image in the hope that the update will finally get "committed" without conflicts from other reservations. If later some server cannot honor this reservation because of an existing reservation for the seat, the new reservation will be aborted.

Thus, the effects of uncommitted reservations on the database may change when the system commits them. If users base their decisions on a database image with uncommitted reservations, the validity of the decision depends on these uncommitted reservations. For example, if Jerry reserves the only two remaining seats of a flight, the reservation is initially uncommitted on the accepting server. Users accessing this server will see the flight full. However, sometime later, it is found that this reservation cannot be honored

by another server because Mary has already reserved one of the two seats. Jerry's purchase is then aborted since Jerry wants to reserve two seats together. But the users accessing Jerry's server lost the opportunity to reserve one of the two seats because they saw Jerry's uncommitted reservation. Thus, the more uncommitted writes a database replica has applied, the larger the probability that the database image will change later and the less consistent (or stable) it is.

In the electronic stock-trading example, consistency determines what percentage of performed trades are observed by a client. Inconsistency arises from new trades executed at other market-makers, not seen by this client's market-maker. The "amount" of inconsistency is the number of unseen trades. Thus consistency is continuous in this example and can be quantified as the number of unseen trades. Another dimension of consistency in this application is the staleness of a replica. For example, one replica has all transaction information up to the last minute, while another replica has transaction information up to the last five minutes. The less stale a replica, the stronger the consistency.

From these and other Internet services, such as E-commerce, web caching, wide-area resource allocation, electronic bulletin boards and email, we define consistency to be three-dimensional:

$$consistency\ of\ a\ replica = (Unseen\,Writes, Uncommitted\,Writes, Staleness)$$

Unseen Writes The number of writes that have been accepted by the system but have not been seen by a replica at time t. These are the unseen reservations in the airline reservation example and unseen trades in the stock-trading example. Intuitively, *Unseen Writes* is the "distance" between the current database image and the "final image," when the system reaches eventual consistency. It is usually expensive for a replica to know its *Unseen Writes* exactly, since this requires global information.

Uncommitted Writes The number of uncommitted writes that this replica has applied to its current database image before time t. These are the uncommitted reservations in the airline reservation example. *Uncommitted Writes* measures how unstable a database replica is. Since the effects of uncommitted writes may change, the more uncommitted writes a replica has applied, the less stable the database image.

Staleness The difference between the current time and the acceptance time of the oldest write that has not seen by this replica. This is the staleness of a market-maker's database replica in the stock-trading example.

By specifying limits for *Unseen Writes*, *Uncommitted Writes* and *Staleness*, each database replica can select its desired consistency level. If all three metrics are limited to zero, the system is one-copy serializable. When none of the metrics is bounded, TACT achieves the other end of the availability/consistency spectrum. In such a case, the system provides the consistency level same as that in optimistic replication systems.

It should be noted that end users do not deal with the three metrics directly. Application programmers express the application-specific consistency using these metrics and export the consistency level to end users. For example, in the airline case, a client will see the consistency level as "your reservation has at most 1% probability of being aborted".

4. TACT toolkit design

In this section, we discuss how the TACT toolkit achieves tunable availability/consistency tradeoffs, leveraging several existing techniques in optimistic replication systems[18, 19, 20, 24, 31, 40] and interactive groupware[5].

4.1. Replica reconciliation and write commitment

Each server maintains a write log for the server's database image, which is the "update history" of the image. An update or a write in our system is essentially a procedure, which checks the database state before performing a set of database writes [21, 22, 31, 40].

We use anti-entropy sessions[11, 18, 31] to reconcile replicas and to ensure eventual consistency. During anti-entropy sessions, the writes, rather than the whole database image, are propagated among servers. Anti-entropy sessions are categorized into *voluntary anti-entropy session*s and *compulsory anti-entropy session*s. A server may perform voluntary anti-entropy sessions with other servers as often as it desires. Voluntary anti-entropy sessions have no effect on the correctness of our system, but they do affect performance. Compulsory anti-entropy sessions are performed by a server when it desires to achieve a particular consistency level. If a compulsory anti-entropy session fails because of network partitions, accesses to the replica will be denied.

Each server maintains a Lamport logical clock[25]. For each client write operation, a server assigns its current logical clock time to the write as its *accept stamp*. A write also carries the identification of the accepting server. The tuple (accept stamp, server-id) determines the *commit order*, for all writes in the system. Write W1 precedes Write W2 in the commit order if and only if W1's accept stamp is smaller than W2's accept stamp; or W1 and W2's accept stamps are the same but W1's server-id is smaller than W2's server-id. Figure 3 illustrates a simplified replicated Internet service with two servers A and B. The writes in the

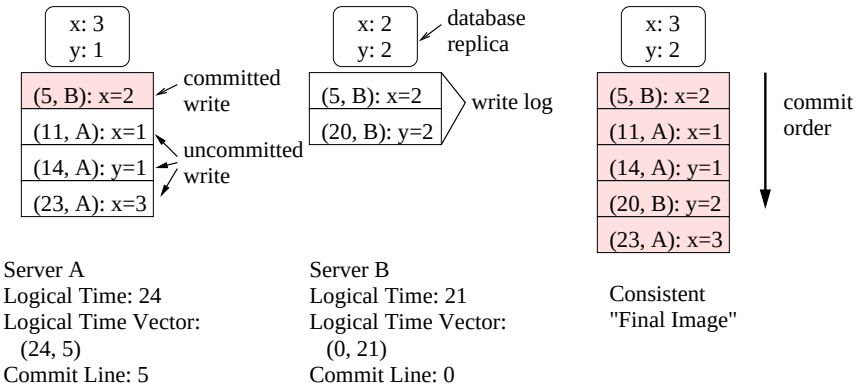

Figure 3. An Internet service with two replicas

write log are denoted by their accept stamps and accepting servers. Server A has accepted three writes. Before these three locally-accepted writes, server B propagates a write (5, B) to A, so there are four writes in A's write log. The commit order is the write log order in the consistent "final image"(Figure 3), i.e., the database image when the system reaches eventual consistency.

In a pair-wise anti-entropy session, a server B propagates to another server A all writes in its write log not seen by A. Writes are propagated according to the partial order defined by accept stamps. Sending writes in this way enables us to use a *logical time vector* to summarize the writes seen by each server. The logical time vector has an entry for each server in the system. For server A, the following *coverage property* is preserved between its logical time vector and the writes it has seen. If the logical time vector entry corresponding to server B is t, then A has seen all writes accepted by server B with accept stamp less than t. For example, in Figure 3, server A's logical time vector is (24, 5), which means it has seen all writes accepted by A with time stamp less than or equal to 24 and all writes accepted by B with time stamp smaller than or equal to 5.

At the start of an anti-entropy session, B obtains A's logical time vector to learn the writes seen by A. Server B can then scan its own write log and send all writes not covered by A's logical time vector. At the end of the anti-entropy session, B sends its logical time vector to A, allowing A to update its own logical time vector by taking a pair-wise maximum of the two logical time vectors.

We use a *write commit algorithm* to enable each server to independently commit writes on its own. Each server may have its own policy about when to invoke this algorithm and commit writes in its write log. Intuitively, a server can commit an uncommitted write if it is certain that no preceding writes are missing from its log. Our commit order is the total order determined by the tuple (accept stamp, server id). Thus if a server can determine that it has seen all writes that bear accept stamps less than a particular logical time, it can safely commit those writes. The minimal entry in a server's logical time vector is called the *commit line*. With the coverage property, a server is sure that it has seen all writes with accept stamps less than or equal to a commit line t. For example, in Figure 3, A's commit line is 5, meaning that A can commit all writes with accept stamps less than or equal to 5. A server commits all such writes in its write log by reordering them according to the commit order. During the reordering process, rollback of the write log may be necessary. Note that the commit order preserves causality, since it is determined by logical time.

4.2. Dynamically tunable consistency levels

We achieve application-desired levels of consistency by bounding the inconsistency in the system. The application specifies the desired consistency level by setting limits for the three consistency metrics, allowing TACT to bound inconsistency in terms of these metrics.

We use a "push" approach to bound *Unseen Writes*, requiring cooperation among all servers in the system. Each server (e.g., A) maintains an *unseen writes bound vector*. The vector has an entry for every server (e.g., B) in the system. Each entry is an upper bound on the number of writes that A can accept without B seeing these writes. We bound *Unseen Writes* by requiring each server to check these upper bounds before accepting a new write. If a new write will violate the upper bound, the server must first perform compulsory anti-entropy sessions with other servers to reduce their *Unseen Writes*. In Figure 4, server A has accepted three writes unseen by server B. If server A's *unseen writes bound vector* (not shown in the figure) entry for B is three, then before A can accept another new write, it must perform compulsory anti-entropy with B to bound B's *Unseen Writes*.

This *unseen writes bound vector* provides flexibility

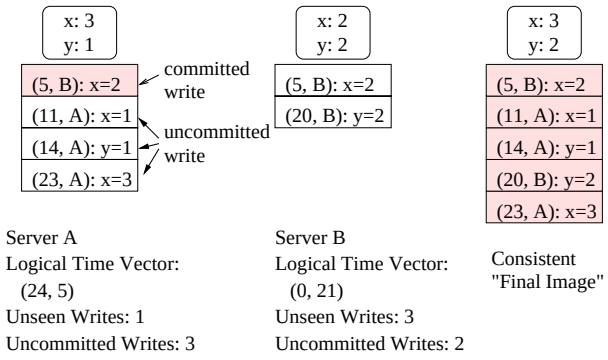

Figure 4. *Unseen Writes* **and** *Uncommitted Writes* **of replica A and replica B**

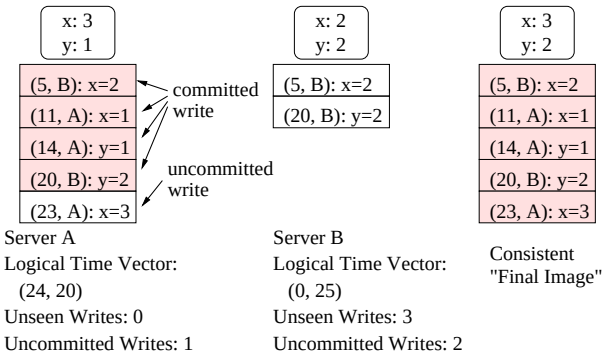

Figure 5. *Unseen Writes* **and** *Uncommitted Writes* **after an compulsory anti-entropy session from B to A**

because the entries on different replicas for a particular server do not have to be identical. A server may believe updates from a particular replica to be particularly important and set a smaller *unseen writes bound vector* entry on that server. Since bounding *Unseen Writes* requires cooperation from other servers, a replica cannot adjust *Unseen Writes* by itself. To enforce a smaller limit for *Unseen Writes*, a replica must use a consensus algorithm to ensure that other servers agree to enforce this new limit for this replica.

The metric *Uncommitted Writes* can be bound using our write commit algorithm. Whenever deemed necessary, a server may invoke this algorithm to reduce the number of uncommitted writes in its write log. If necessary, the server will pull writes from other servers by performing compulsory anti-entropy sessions to advance its logical time vector and commit line. For the example depicted in Figure 4, a read request to server A might specify a consistency level of $Uncommitted\ Writes \leq 2$. In order to process this request, server A performs anti-entropy session with server B to pull writes. In this way, server A can commit some writes and *Uncommitted Writes* is reduced to one. The result of such an anti-entropy session is illustrated in Figure 5.

To bound the *Staleness* of a replica, each server maintains a *real time vector*. This vector is similar to the logical time vector, except that real time instead of logical time is used. A similar coverage property is preserved between the writes a server has seen and the real time vector. If A's real time vector entry corresponding to B is t, then A has seen all writes accepted by B before real time t. To bound *Staleness* within l, a server checks whether $current\ time - t < l$ holds for each entry in the real time vector. (We assume that server clocks are loosely synchronized.) If the inequality does not hold for some entries, the server performs compulsory anti-entropy session with the corresponding servers and advances its real time vector.

A benefit of our design is that replicas can provide different consistency QoS for different clients/accesses. The consistency level of a read or write access is defined to be the consistency level of the replica at the time of access. If a replica desires stronger consistency than its current consistency level for some access, it can temporarily enforce smaller limits for *Unseen Writes*, *Uncommitted Writes* and *Staleness*.

4.3. Trading consistency with availability (performance)

In our algorithm, a server maintains consistency by performing compulsory anti-entropy sessions with other servers. The set of servers it needs to contact upon receiving each user request is directly determined by the consistency level. The larger the limits for *Unseen Writes*, *Uncommitted Writes* and *Staleness*, the less frequent the compulsory anti-entropy sessions and the fewer servers that need contacting upon each access. Thus, given a network partition with fixed scope and duration, the weaker the consistency level, the larger the probability that a server is able to perform the required compulsory anti-entropy sessions and an access will be accepted, which means higher system availability.

At the two extremes of the availability/consistency spectrum, TACT demonstrates the behavior of the voting algorithm[16] (with read quorum being 1 and write quorum being n) and the behavior of the optimistic replication systems[31, 40], respectively. (As Figure 1 shows, the end points of the curve in (b) corresponds to the two points in (a).)

As another effect of our consistency algorithm, when an application relaxes its consistency requirements, the frequency of compulsory anti-entropy sessions decreases. The

latency perceived by an access is decreased through reducing required communication on the critical path.

5. Related work

We first describe previous research on the two extreme end points of the availability/consistency tradeoff spectrum. In the traditional replicated transactional database model, strong consistency (one-copy serializability) is considered necessary for correctness. In this context, voting algorithms[16], missing writes algorithm[12], virtual partition algorithms[1, 2] and class conflict analysis[37] have been proposed to increase availability in the presence of network partitions. Coan et. al.[9] proved that the achievable database availability under network partitions has a tight upper bound. At the other end of the spectrum are optimistic systems such as Bayou[31, 40], Ficus[19], Rumer[20] and Coda[35]. In these systems, availability is explicitly favored over strong consistency. Bayou provides limited support (session guarantees[13, 38, 39]) for applications to increase the consistency level observed by users, while Ficus, Rumer and Coda have no such mechanism. However, Bayou's session guarantees are effective largely when a client switches from one server A to another server B, ensuring that B's consistency level is, in some predefined sense, no "weaker" than A's consistency. Session guarantees provide little assurance about the consistency level of a server itself. For example, a user cannot limit the staleness of the local copy of the meeting room schedule.

Researchers have also tried to define numerical metrics for consistency. The time-to-live (TTL) property in web caching is probably the simplest metric to bound inconsistency. Timed consistency[41] and delta consistency[36] explore weakened consistency along the dimension of staleness. Both consistency models were proposed to address the lack of timing in traditional consistency models such as sequential consistency, while in TACT, we concentrate on the consistency requirements of real Internet services and summarize consistency using three concrete metrics.

Pu et. al.[33, 34] use low-level consistency metrics, which are informally the number of conflicting reads and writes. These metrics measure the "mutual inconsistency" observed by multiple reads in a query transaction, while we concentrate on the consistency level observed by each read. For example, a replica with three unseen writes may actually execute a query transaction with zero conflict with update transactions. In *quasi-copy* caching[3], Alonso et.al. proposed four "coherency conditions": delay condition, frequency condition, arithmetic condition and version condition. Version condition is very similar to our *Unseen Writes* metric. However, the coherency conditions and system design are limited to the context of read-only caching rather than general-purpose read/write replication.

The notion of *Unseen Writes* is also related to the *k-completeness* concept in the SHARD system[26]. Krishnakumar and Bernstein[21, 22] propose the concept of an "N-ignorant" system, where a transaction runs in parallel with at most N conflicting transactions. This looks similar to our *Unseen Writes* metric, but there are actually subtle differences between bounding *Unseen Writes* within N and providing N-ignorance (the latter is stronger). Furthermore, the definition of N-ignorance does not allow different N's for different replicas. The authors concentrate on possible database final states, while we investigate how Internet services' consistency requirements are continuous and can be quantified with our metrics. Availability is not explicitly addressed in [21, 22]. In two recent papers[8, 29], metrics similar to *Unseen Writes* and *Staleness* are used to measure database freshness. However, no design is proposed to provide guaranteed freshness levels by bounding the metrics.

Fox and Brewer[15] argue that strong consistency (one-copy serializability[4]) and one-copy availability[32] cannot be achieved simultaneously in the presence of network partitions. In the context of the Inktomi search engine, they show how to trade harvest for yield. Harvest measures the fraction of the data reflected in the response, while yield is the probability of completing a request. In TACT, we concentrate on consistency among service replicas, but a similar "harvest" concept can also be defined using our consistency metrics. For example, bounding *Unseen Writes* has similar effects as guaranteeing a particular harvest.

Olston and Widom[28] address tunable performance/precision tradeoff issue in the context of aggregation inquiries over numerical database records. As compared to their systems, our consistency metrics and design are presented in a more general context and are applicable to non-numerical data and accesses other than aggregation inquiries.

In Fluid Replication[27], clients are allowed to dynamically create service replicas to improve performance. Their study on when and where to create a service replica is complementary to our study on availability and consistency issues among replicas. Similar to Ladin's system[24], Fluid Replication supports three discrete consistency levels: last-writer, optimistic and pessimistic. TACT can achieve both optimistic and pessimistic consistency levels, as in fluid replication. Varying frequency of reconciliation in fluid replication allows applications to adjust the strength of last-writer and optimistic consistency. Bounding staleness in TACT has similar effects. However, as discussed earlier, staleness alone cannot fully capture application-specific consistency requirements.

6. Conclusions

In this paper, we argue for the importance of dynamically choosing availability/consistency tradeoffs for replicated Internet services. We show that consistency is continuous rather than binary for many Internet services. We derive three high-level consistency metrics, *Unseen Writes*, *Uncommitted Writes* and *Staleness*, directly from real Internet services. These metrics are (linearly) related to the consistency observed by end users. We present the initial design of the TACT toolkit, which allows Internet services to flexibly and dynamically choose their availability/consistency tradeoffs.

We are currently working on a simulator to further validate our consistency metrics and toolkit design. Next we plan to build a prototype of TACT, and several sample Internet services that utilize our toolkit. With these sample services, we intend to show:

1. Our consistency metrics capture the consistency requirements of Internet services.

2. Applications can utilize TACT to dynamically trade consistency for availability (performance) based on changing client, network and service characteristics.

7. Acknowledgments

We thank Jeff Chase for his insightful comments during the initial phase of the TACT project. We also thank the anonymous referees for their careful review of this paper.

References

[1] Amr El Abbadi, Dale Skeen and Flaviu Cristian. An Efficient, Fault-Tolerant Protocol for Replicated Data Management. *Proceedings of the 4th ACM SIGACT/SIGMOD Conference on Principles of Database Systems*, 1985.

[2] Amr El Abbadi, Dale Skeen and Flaviu Cristian. Maintaining Availability in Partitioned Replicated Databases. *Proceedings of the 5th ACM SIGACT/SIGMOD Conference on Principles of Database Systems*, 1986.

[3] Rafael Alonso, Daniel Barbara and Hector Garcia-Molina. Data Caching Issues in an Information Retrieval System. *ACM Transactions on Database Systems*, September 1990.

[4] P. Bernstein and N. Goodman. The Failure and Recovery Problem for Replicated Distributed Databases. *ACM Transactions on Database Systems*, December 1984.

[5] Sumeer Bhola and Mustaque Ahamad. The Design Space for Data Replication Algorithms in Interactive Groupware. Technical report GIT-CC-98-15, 1998. College of Computing, Georgia Institute of Technology.

[6] A. Birrell, R. Levin, R. Needham and M. Schroeder. Grapevine: An Exercise in Distributed Computing. *Communications of the ACM* 25(4):260-274, April 1982.

[7] Cable News Network. `http://www.cnn.com`, 1999.

[8] Junghoo Cho and Hector Garcia-Molina. Synchronizing a Database to Improve Freshness. Technical Report 1999. Computer Science Department, Stanford University. http://www-db.stanford.edu/pub/papers/cho-synch.ps

[9] Brian Coan, Brian Oki and Elliot Kolodner. Limitations on Database Availability When Networks Partition. *Proceedings of the 5th ACM Symposium on Principle of Distributed Computing*, August 1986.

[10] Susan Davidson, Hector Garcia-Molina and Dale Skeen. Consistency in Partitioned Networks. *Computing Survey*, Vol. 17, No. 3, September 1985.

[11] A. Demers, D. Greene, C. Hauser, W. Irish, J, Larson, S. Shenker, H. Sturgis, D. Swinehart and D. Terry. Epidemic Algorithms for Replicated Database Maintenance. *Proceedings of 6th Symposium on Principle of Distributed Computing*, August 1987.

[12] Derek Eager and Kenneth Sevcik. Achieving Robustness in Distributed Database Systems. *ACM Transactions on Database Systems*, 8(3):354-381, September 1983.

[13] W. Keith Edwards, Elizabeth Mynatt, Karin Petersen, Mike Spreitzer, Douglas Terry and Marvin Theimer. Designing and Implementing Asynchronous Collaborative Applications with Bayou. *Proceedings of 10th ACM Symposium on User Interface Software and Technology*, October 1997.

[14] Exodus Communications. Internet Data Centers. `http://www.exodus.com/idcs`, 1999.

[15] Armando Fox and Eric Brewer. Harvest, Yield and Scalable Tolerant Systems. *Proceedings of HotOS-VII*, March 1999.

[16] David K. Gifford. Weighted Voting for Replicated Data. *Proceedings of the 7th ACM Symposium on Operating Systems Principles*, 1979.

[17] D. K. Giford. Information Storage in a Decentralized Computer System. Technical Report CSL-81-8, Xerox Parc, 1983.

[18] Richard Golding. Weak-Consistency Group Communication and Membership. Ph.D. Thesis, Computer and Information Science Board, University of California at Santa Cruz, December 1992.

[19] R. Guy, J. Heidemann, W. Mak, T. Page, Jr., G. Popek and D. Rothmeier. Implementation of the Ficus Replicated File System. *Proceedings Summer USENIX Conference*, June 1990.

[20] R. G. Guy, P. Reiher, D. Ratner, M. Gunter, W. Ma and G. J. Popek. Rumor: Mobile Data Access Through Optimistic Peer-to-Peer Replication. *Proceedings of the 17th International Conference on Conceptual Modeling (ER'98)*, November 1998.

[21] Narayanan Krishnakumar and Arthur Bernstein. Bounded Ignorance in Replicated Systems. *Proceedings of the 10th ACM Symposium on Principles of Database Systems*, May 1991.

[22] Narayanan Krishnakumar and Arthur Bernstein. Bounded Ignorance: A Technique for Increasing Concurrency in a Replicated System. *ACM Transactions on Database Systems*, Vol. 19, No. 4, December 1994.

[23] Geoffrey Kuenning, Rajive Bagrodia, Richard Guy, Gerald Popek, Peter Reiher and An-I Wang. Measuring the Quality of Service of Optimistic Replication. *Proceedings of the ECOOP Workshop on Mobility and Replication*, July 1998.

[24] R. Ladin, B. Liskov, L. Shrira and S. Ghemawat. Providing High Availability Using Lazy Replication. *ACM Transactions on Computer Systems* 10(4):360-39, November 1992.

[25] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558-565, July 1978.

[26] Nancy Lynch, Barbara Blaustein and Michael Siegel. Correctness Conditions for Highly Available Replicated Databases. Technical report MIT/LCS/TR-364, June 1986. MIT Laboratory for Computer Science.

[27] Brian Noble, Ben Fleis and Minkyong Kim. A Case for Fluid Replication. *Proceedings of the 1999 Network Storage Symposium (Netstore)*, October 1999.

[28] Chris Olston and Jennifer Widom. Bounded Aggregation: Offering a Precision-Performance Tradeoff in Replicated Systems. Technical Report 1999. Computer Science Department, Stanford University. http://www-db.stanford.edu/pub/papers/trapp-ag.ps

[29] Esther Pacitti, Eric Simon and Rubens Melo. Improving Data Freshness in Lazy Master Schemes. *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems*, May 1998.

[30] T. Page, R. Guy, J. Heidemann, D. Ratner, P. Reiher, A. Goel, G. Kuenning and G. Popek. Perspectives on Optimistically Replicated Peer-to-peer Filing. *Software-Practice and Experience* 28(2):155-180, February 1998.

[31] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible Update Propagation for Weakly Consistent Replication. *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

[32] Gerald Popek, Richard Guy and Thomas Page, Jr. Replication in Ficus Distributed File Systems. *Proceedings of Workshop on Management of Replicated Data*, November 1990.

[33] Calton Pu and Avraham Leff. Replication Control in Distributed System: an Asynchronous Approach. Technical report CUCS-053-090, January 1991. Department of Computer Science, Columbia University.

[34] Calton Pu and Avraham Leff. Epsilon-Serializability. Technical report CUCS-054-90, January 1991. Department of Computer Science, Columbia University.

[35] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel and D. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transaction on Computers* 39(4):447-459, April 1990.

[36] Aman Singla, Umakishore Ramachandran and Jessica Hodgins. Temporal Notions of Synchronization and Consistency in Beehive. *Proceedings of the 9th ACM Symposium on Parallel Algorithms and Architectures*, June 1997.

[37] D. Skeen and D. Wright. Increasing Availability in Partitioned Networks. *Proceedings of the 3rd ACM Symposium on Principles of Database Systems*, April 1984.

[38] D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer and B. Welch. Session Guarantees for Weekly Consistent Replicated Data. *Proceedings 3rd International Conference on Parallel and Distributed Information System*, September 1994.

[39] D. Terry, K. Petersen, M. Spreitzer and M. Theimer. The Case for Non-transparent Replication: Examples from Bayou. *IEEE Data Engineering*, December 1998, pages 12-20.

[40] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer and C. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. *Proceedings 15th Symposium on Operating Systems Principles*, December 1995.

[41] Francisco Torres-Rojas, Mustaque Ahamad and Michel Raynal. Timed Consistency for Shared Distributed Objects. *Proceedings of the 18th ACM Symposium on Principle of Distributed Computing*, May 1999.

[42] Amin Vahdat, Thomas Anderson, Michael Dahlin, Eshwar Belani, David Culler, Paul Eastham and Chad Yoshikawa. WebOS: Operating System Services for Wide-Area Applications. *Proceedings of the 7th IEEE Symposium on High Performance Distributed Systems*, July 1998.