Composable Consistency for Large-scale Peer Replication

Sai Susarla and John Carter
{*sai, retrac*}*@cs.utah.edu*

UUCS-03-025

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

November 14, 2003

Abstract

The lack of a flexible consistency management solution hinders P2P implementation of applications involving updates, such as directory services, online auctions and collaboration. Managing shared data in a P2P setting requires a consistency solution that can operate in a heterogenous network, support pervasive replication for scaling, and give peers autonomy to tune consistency to their sharing needs and resource constraints. Existing solutions lack one or more of these features.

In this paper, we propose a new way to structure consistency management for P2P sharing of mutable data called *composable consistency*. It lets applications compose a rich variety of consistency solutions appropriate for their sharing needs, out of a small set of primitive options. Our approach splits consistency management into design choices along five orthogonal aspects, namely, concurrency, consistency, availability, update visibility and isolation. Various combinations of these choices can be employed to yield numerous consistency semantics and to fine-tune resource use at each replica. Our experience with a prototype implementation suggests that composable consistency can effectively support diverse P2P applications.

1 Introduction

Existing P2P systems largely focus on locating and sharing read-only content. The availability, failure resilience and incremental scalability of the P2P organization can also benefit applications that involve updates, such as directory services, online auctions, and collaborative work environments (e.g., Lotus Notes). However P2P sharing of mutable data raises issues not present in existing centralized implementations of these services, namely, replication and consistency management. A dynamic wide-area P2P environment poses new challenges to replication algorithms, such as diverse network characteristics, a large but floating population of peers and their diverse capabilities and consistency needs. Maintaining a replica at a peer consumes compute and network resources to keep it up-to-date with other replicas. Peers vary in their resource availability and willingness to handle load from other peers. A P2P consistency management system must therefore be able to handle pervasive replication (e.g., creating and destroying replicas when needed) to scale with load, and enable peers to individually balance consistency and availability against performance and resource use.

A number of previous efforts [10, 3] have viewed consistency semantics as a continuous linear spectrum ranging from strong consistency to best-effort eventual consistency. However, an application's consistency needs are not always the same for all types of access (e.g., read or write). For example, an auction service might need strong consistency for updates to serialize purchases, but could tolerate a weaker consistency level for queries to improve performance at the risk of supplying stale data to users. This argues for expressing consistency requirements as a two-dimensional space. In fact, we have surveyed the data sharing needs of a variety of distributed services [7] and found that they could be described along five major dimensions that are largely orthogonal:

- *concurrency* - the degree to which conflicting (read/write) accesses can be tolerated,
- *consistency* - the tolerance to stale data and the update dependencies to be preserved,
- *availability* - how data access should be handled when some replicas are unreachable,
- *visibility* - the time at which local modifications to replicated data must be made visible globally, and
- *isolation* - the time at which remote updates must be made visible locally.

There are multiple reasonable options along each of these dimensions, that create a multi-dimensional space for expressing consistency requirements of applications. When these

options are combined in various ways, they yield a rich collection of consistency semantics for reads and updates to shared data, covering the needs of a broad variety of applications. For instance, our approach lets a P2P auction service employ strong consistency for updates across all peers, while enabling different peers to answer queries with different levels of accuracy by relaxing consistency for reads to limit synchronization cost. A user can still get an accurate answer incurring higher latency by specifying a stronger consistency requirement for her query.

In this paper, we describe *composable consistency*, our approach to consistency management based on the above classification. We believe that letting applications express consistency requirements individually along these dimensions on a per-access basis instead of supporting a few packaged combinations gives them more flexibility in balancing consistency, availability and performance. We have developed a wide-area P2P data store called *Khazana* that provides aggressive replication and composable consistency behind a simple file system-like interface. We developed several P2P applications on top of Khazana including a prototype auction service that employs peer proxies via shared objects, a file system that supports collaborative document sharing, and a replicated BerkeleyDB database to serve as a shared directory service. We found that Khazana effectively exploits locality in these applications while precisely meeting their consistency requirements [8].

In Section 2 we describe our composable consistency scheme, and outline several useful semantics it enables in section 3. In Sections 4 and 5 we compare our approach with related work and conclude.

2 Composable Consistency Options

Composable consistency lets applications express data sharing requirements along five dimensions: *concurrency*, *consistency*, *availability*, *visibility*, and *isolation*. These dimensions reflect common decisions during consistency management. Table 1 lists several useful design options along each dimension. These options can be combined several useful ways as described in section 3, although not all of them make sense.

Runtime Model

We assume the following runtime data access model for the rest of our discussion. Application instances (hereafter referred to as *clients* of the consistency system) access their local

Concur. control		Consistency guarantee			Update Visibility	Reader Isolation	Availability
rd	wr	Strength	Timeliness	Data Deps.			
RD	WR	hard (pull)	latest	none	session	session	optimistic
			time-based	total order	per-access	per-access	
RDLK	WRLK	soft (push)	mod-based	causality	manual	manual	pessimistic
				atomicity			

Table 1: Composable Consistency Options.

Access (rd & wr) semantics	Strength of guarantee	Check for updates	Wr Visible	Availability in partition	Use/ Provider
strong (exclusive)	hard(pull)	on open	on close	no	serializability
close-to-open	hard(pull)	on open	on close	app. choice	collaboration, AFS
close-to-rd	hard	on access	on close	app. choice	read stable data
wr-to-open	hard	on open	on write	app. choice	NFS
wr-to-rd	hard	on access	on write	app. choice	log monitoring
eventual close-to-rd	soft(push)	never	on close	yes	Pangaea
eventual wr-to-rd	soft	never	on write	yes	chat, streaming

Table 2: Major flavors of access semantics possible.

replicas of shared data via sessions bracketed by open() and close() calls. Before accessing a local replica, a client opens a session, optionally specifying the desired consistency semantics as a vector of consistency options along the dimensions below. In response, the system brings the local replica to the desired consistency. The client then reads or writes the local replica directly and later closes the session [1]. Consistency options can either be set on the data globally affecting all replicas, individually on a per-replica basis, or when opening a session to hold good for the rest of that session. Local options override global ones. However, weakly consistent access at a replica cannot compromise stronger consistency guaranteed elsewhere.

Concurrency Options

Concurrency control refers to the parallelism allowed among reads and writes at various replicas. We found that two distinct flavors of access mode for each of reads and writes (specified at open() time) are adequate for a variety of applications: *concurrent* (RD, WR) and *exclusive* (RDLK, WRLK) access. Concurrent modes allow other writes to proceed

[1] Clients can also make updates by supplying an update procedure to be applied at all replicas.

in parallel, at the expense of operating on stale data or generating write conflicts. Exclusive modes enforce the traditional concurrent-read-exclusive-write (CREW) semantics and globally serialize writes for strong consistency. Resolving write conflicts may require application intervention unless last-writer-wins [9] rule suffices.

Consistency Options

Consistency refers to the degree to which stale data can be tolerated. Applications can specify this in terms of *timeliness* guarantees and *data interdependencies*.

Timeliness refers to how close data read must be to the most current global version. There are three useful choices for timeliness: *most current*, *time-bounded* and *modification-bounded*. Our evaluation presented elsewhere [8] shows that even small time bounds (e.g., 10 millisecs) can significantly improve performance of *most current* under high speed workloads. Modification bound specifies the number of unseen remote writes tolerated by a replica before it needs to synchronize.

Clients can specify whether their timeliness requirements are *hard*, meaning they must be checked before every access, or *soft*, indicating that best effort suffices. For example, close-to-open consistency for files can be achieved by choosing hard 'most-current' guarantee at open() time. Soft guarantees are adequate for applications like bulletins boards and shopping queries, and can lead to significantly better performance and scalability, as our study [8] indicates. Hence application choice over hardness is valuable.

Data interdependence refers to whether an application requires that multiple updates be seen in a particular order by replicas. We identified four useful options. Updates may need to be seen (i) in no particular order, (ii) in the same order at all replicas, (iii) in causal order [2], meaning that if replica A sees B's update and makes an update, B's update must be seen before A's everywhere, or (iv) atomically, meaning that updates by sessions grouped as atomic must be seen together everywhere.

Unordered and totally ordered updates suffice for many popular applications including file updates, chat sessions, and updates to replicated directory services. Lack of causal ordering could cause confusion to mobile users if updates made by them do not propagate as fast as they move. Databases typically require atomicity to preserve data integrity. However, preserving causality and atomicity often incur significant bookkeeping and hence applications must be able to disable them when not needed.

Availability Options: When a consistency or concurrency control guarantee cannot be

met due to node or network failures, applications may be *optimistic* i.e., continue with the available data in their partition, or be *pessimistic* i.e., abort. When connected, both options behave identically. Applications can choose between availability and consistency on a per-session basis.

Visibility and Isolation Options

Visibility refers to the time at which updates are made visible to remote sessions. *Isolation* refers to the time when a replica must check for and apply remote updates. We identified three useful flavors of visibility and isolation: *session*, *per-access* and *manual*. They imply that updates are made visible by writers or checked by readers (i) only at a session boundary, (ii) eagerly on every access, or (iii) only when the application requests it. Session semantics ensures that readers never see intermediate writes of remote sessions. This is important for document updates. Eager semantics allows long-lived sessions at replicas to quickly disseminate or monitor each other's updates and can be used for stock quote updates, Internet chat and real-time data multicast. Manual semantics lets an application completely hand-tune its update propagation strategy using application knowledge to balance timeliness against CPU and bandwidth usage.

3 Discussion

The above options allow a variety of applications to compose consistency semantics appropriate for their sharing needs. Table 2 lists some common sets of options that correspond to well known "hard wired" consistency semantics. The first column denotes a particular consistency semantics, and the other fields denote the set of options that an application should specify to achieve this semantics. For example, "close-to-rd" semantics means that a session reads only writes by completed sessions, not the intermediate writes of still open sessions. If an application's updates preserve data integrity only at session boundaries, this set of options ensures that reads always return stable (i.e., internally "consistent") data regardless of ongoing update activity. In contrast, "wr-to-rd" semantics does not provide this guarantee, but is useful when up-to-date data is preferred over stable data, or when write sessions are long-lived as in the case of distributed data logging, or live multimedia streaming. Finally, the rightmost column gives an example situation or application where that choice of consistency options is appropriate.

The table shows three broad categories of semantics. We hereafter refer to them as *strong*,

eager and *best-effort/eventual* consistency semantics for brevity. Strong semantics are achieved by exclusive access modes (RDLK, WRLK), and others by concurrent modes (RD, WR) with hard and soft timeliness requirements respectively. The eager and best-effort semantics can be further qualified by a timeliness bound to achieve continuously variable control over replica divergence both in terms of time and number of unseen updates. These semantics can be achieved independently for reads and writes, further enhancing the options available to applications. Combining atomic updates with the above options yields a rich variety of weak consistency flavors for databases, all of which ensure transactional atomicity. Similarly, combining eager read and strong write semantics with atomicity ensures that a reader always sees an internally consistent snapshot of data without blocking ongoing writes elsewhere. This allows a reader to safely navigate shared objects and data structures being actively modified elsewhere (analogous to Objectstore's MVCC reads [4]).

As an example, consider a P2P implementation of an online marketplace such as EBay where peers (i) publish and browse items for sale, and (ii) place bids and make sales. The latter require exclusive access to an item (e.g., to prevent selling it to two people). However, the item catalog itself can be widely replicated with weak consistency as shoppers are used to fluctuating content. To reduce coherence traffic further, updates can be forwarded to a few 'master' sites, while all others handle queries and resynch with masters only periodically or when a query fails.

In summary, our combinatorial approach covers a large space of consistency semantics individually provided by many applications in a single system. Each of our consistency options can be implemented independently of others and contributes to a portion of the overall semantics. Hence we hypothesized that a consistency solution composed out of these options is also likely to perform well. Our experience with a prototype implementation confirms our hypothesis.

4 Related Work

Several solutions exist to manage sharing of aggressively replicated data among wide-area peers. However, most previous work has targetted read-only data such as multimedia files (PAST [6], KaZaa) or access to personal files [9], but not frequent write-sharing. Numerous consistency schemes have been developed individually to handle the data coherence needs of specific services such as file systems, directory services [5], databases and persistent object systems [4], Distributed file systems such as NFS, Pangaea, Sprite, AFS, Coda and Ficus target traditional file access with low write-sharing among multiple users. Com-

posable consistency adopts a novel approach to support many of their consistency schemes efficiently in a P2P setting.

Fluid replication [1] provides multiple selectable consistency policies. In contrast, our approach offers primitive options that can be combined to yield a variety of policies, offering more customizability.

Many previous efforts have explored a continuous consistency model [10, 3]. Of those, the TACT toolkit comes closest to our approach. TACT provides continuously tunable consistency along three dimensions similar to those covered by our timeliness and concurrency control aspects. We provide two additional dimensions, namely update visibility and isolation, that cater to a wider variety of application needs as described in Section 3. TACT's order error offers continuous control over the number of update conflicts. In contrast, our concurrency options provide a binary choice between zero and unlimited number of conflicts. However, we believe that for many real-world applications, a binary choice such as ours is adequate and reduces bookkeeping overhead.

5 Conclusions

In this paper we advocated a new way to structure consistency management for P2P sharing of mutable data, called composable consistency. It splits consistency management into design choices along several orthogonal dimensions and lets applications express their consistency requirements as a vector of these choices on a per-access basis. Our design choices are chosen such that they can be combined in various ways to yield a rich collection of semantics, while enabling an efficient implementation. We described composable consistency options and outlined how a variety of popular consistency schemes can be easily achieved with them.

References

[1] L. Cox and B. Noble. Fast reconciliations in Fluid Replication. In *Proc. 21st Intl. Conference on Distributed Conputing Systems*, Apr. 2001.

[2] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch. The Bayou architecture: Support for data sharing among mobile users. In *Proc. Workshop on Mobile Computing Systems and Applications*, Dec. 1994.

[3] N. Krishnakumar and A. Bernstein. Bounded ignorance: A technique for increasing concurrency in a replicated system. *ACM Transactions on Data Base Systems*, 19(4), Dec. 1994.

[4] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The Objectstore database system. *CACM*, Oct. 1991.

[5] Microsoft Corp. Windows 2000 server resource kit. Microsoft Press, 2000.

[6] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. 18th Symposium on Operating Systems Principles*, 2001.

[7] S. Susarla. A survey of implementation techniques for distributed applications. http://www.cs.utah.edu/~sai/papers/app-survey.ps, Mar. 2000.

[8] S. Susarla and J. Carter. Khazana: A flexible wide-area data store. Technical Report UUCS-03-020, University of Utah School of Computer Science, Oct. 2003.

[9] Y. Saito et al. Taming aggressive replication in the Pangaea wide-area file system. In *Proc. 5th Symposium on Operating System Design and Implementation*, 2002.

[10] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proc. 4th Symposium on Operating System Design and Implementation*, Oct. 2000.