An Efficient, Low-Cost Inconsistency Detection Framework for Data and Service Sharing in an Internet-Scale System

Yijun Lu and Hong Jiang
Dept. of Computer Science and Engineering
University of Nebraska-Lincoln
Lincoln, Nebraska 68588-0115, USA
{yijlu, jiang}@cse.unl.edu

Dan Feng
Dept. of Computer Science and Engineering
Huazhong Univ. of Science and Technology
Wuhan,Hubei 430074, China
dfeng@hust.edu.cn

Abstract

In this paper, we argue that a broad range of Internet-scale distributed applications can benefit from an underlying low-cost consistency detection framework that is an alternative to inconsistency avoidance and can detect inconsistency among nodes sharing data or services in a timely manner.

This paper first presents an overview of the inconsistency detection framework. Then, it discusses the detailed design of the two-layer inconsistency detection module, the core component of this framework, which can detect inconsistency among nodes in a timely manner. The proposed two-layer inconsistency detection module is evaluated both analytically and via simulations, which shows that this module can significantly reduce the time to detect inconsistency among nodes without adding much maintenance cost. Finally, this paper outlines the possible mechanisms to discern the application semantics and to resolve the detected inconsistencies.

1. Introduction

Replicating data and services is an attractive strategy to increase availability and performance in distributed systems. In these systems, the importance of consistency control is well understood. In this paper, we are particularly interested in the consistency control problem in Internet-scale distributed systems in which the nodes span across the Internet. This includes a broad range of applications such as Grid, online collaboration, content distribution network, and large-scale e-business applications.

Conventionally, consistency control is designed to avoid the inconsistency up-front. Well-defined consistency protocols, such as strong consistency protocols [11] or optimistic consistency protocols that increase the availability while tolerate relaxed inconsistency among nodes [6, 12], are predefined and deployed before the system starts to run. In this paper, we refer to this scheme as *inconsistency avoidance*.

While inconsistency avoidance can be effective in a small-scale networked system, such as a small cluster, it has some drawbacks in an Internet-scale environment, such as Grid or large-scale distributed e-business applications.

More specifically, a strong consistency protocol can be very costly to maintain due to the membership maintenance and strict protocol enforcement cost. And because of the relatively unreliable network transmission in large-scale networks, it is impossible in most cases to maintain strong consistency [4].

While optimistic consistency protocol relieves the costly maintenance and strict enforcement burden associated with strong consistency protocols, it also does not suit the large-scale distributed system because it is *predefined*. In an environment where many applications are deployed, providing a predefined consistency protocol can be either overkill when an application does not need that strong consistency, or insufficient when an application needs stronger ones. While several consistency protocols can be deployed to cater different applications simultaneously, it would inevitably increase the complexity of the system design.

Besides, some application's requirement for consistency changes from time to time. In online conference, for example, users require higher consistency when an important speech is going on while are willing to tolerate lower consistency for better performance otherwise [3]. In this scenario, a predefined protocol is incapable of capturing that semantic.

This paper proposes a framework to detect inconsistency in a *timely* manner when it occurs

instead of avoiding it in the first place. We refer this as *inconsistency detection*.

Comparing to inconsistency avoidance, several advantages can be obtained from an inconsistency detection framework. First, it removes the costly membership management requirement that is used to enforce a consistency in the first place. Instead, it detects the inconsistency when it happens. That makes the system scalable. Besides, by ensuring that the potential inconsistent behavior be detected in a timely manner, a system can combine the results (if the results are combinable), break the tie or alert the users so that they can resolve the conflict as soon as possible using appropriate resolution protocols. This can at least prevent the conflicts from further damaging the system.

This mechanism can also support applications with high consistency requirement because, as long as this framework can detect the inconsistency, it can resolve it. In other words, application will not suffer inconsistency level when used this inconsistency detection mechanism.

Second, after the inconsistency is detected, the middleware can respond based on the application semantics. That is, it resolves the inconsistency when it is needed, while letting the detected inconsistency continue to exist when it is tolerable or even preferred. For the latter case, consider the air ticket booking system, an example for e-business, which requires a consistency control protocol allowing inconsistency – overselling – to exist within a certain threshold to cover the returned tickets.

In this aspect, this mechanism provides versatility as it can support several applications with different consistency requirements without deploying separate consistency protocols. Besides, it simplifies the system's design.

This paper presents an overview of an inconsistency detection framework, i.e. its main structure and main components. Then it presents the design of the inconsistency detection module – the core component in the framework, which detects inconsistency in a *timely* manner, as wells its evaluation. The mechanisms to discern the application semantics and to resolve the inconsistencies are outlined subsequently.

The rest of the paper is organized as follows. Section 2 discusses the overview of the inconsistency detection framework. Section 3 presents the enabling technologies of the proposed two-layer inconsistency detection mechanism whose design is presented in section 4. Section 5 evaluates the framework by both analyses and simulations. A discussion about inconsistency resolution is discussed in Section 6. Related work is presented in section 7. Finally, section 8 concludes this paper and discusses future work.

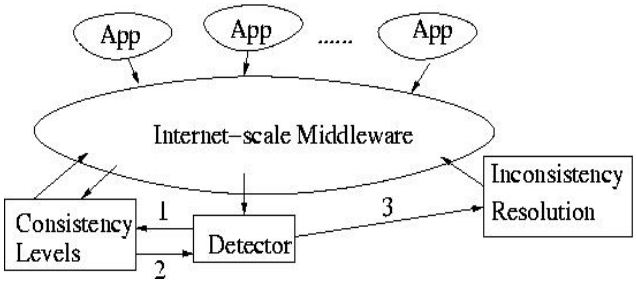

Figure1. Architecture of the Inconsistency Detection Framework

2. Overview of the Inconsistency Detection Framework

As an alternative to inconsistency avoidance, the inconsistency detection framework detects inconsistency among nodes in a timely manner. A logical diagram of this framework is shown in Figure 1.

In this framework, multiple applications share data and services through the support of the Internet-scale middleware and the inconsistencies among them are detected by the detector. Upon detection, the detector consults with the inconsistency level monitor (step 1 and step 2) before reaction is initiated. Based on the applications' semantics, if the inconsistency is tolerable, the detector does not react; otherwise, the detector informs the inconsistency resolution model to resolve this inconsistency (step 3).

The arrows from the middleware to the detector module means that the detector gets information from the middleware, and the arrow from the inconsistency resolution module to the middleware means that the module can influence the middleware. The two arrows between the consistency level module and the middleware means that it can get the consistency levels for applications from the middleware and, it can potentially help the applications to adjust their consistency levels.

As we can see, the core module of this framework is the *timely* inconsistency detection mechanism, which will be discussed in the next section.

3. Timely Inconsistency Detection: Basic Idea and Enabling Technologies

The basic idea is to build an overlay on top of the underlying network based on nodes' updating history. As the top layer is based on nodes' updating history, or updating temperature, it is referred as "temperature

overlay". The bottom layer of gossip-based inconsistency detection is used as a backup and only triggered when the top layer does not find any inconsistency. The architecture of the framework is illustrated in Figure 2.

In the temperature overlay, each node tracks its own updating history and exchanges this information with others through the RanSub [7] protocol periodically. When a node commits an update, this update is propagated in the temperature overlay in such a way that the nodes that update this file most frequently are visited first. The rationale behind this design is that a user usually works on a file for a certain period of time. For example, he/she may edit a report for 10 minutes, then debugs, thus updates, a C++ file for 20 minutes.

The hypothesis of this design is that, through this two-layer framework, most inconsistencies can be detected in the top layer quickly, probabilistically speaking. The two enabling technologies and their roles in timely inconsistency detection are discussed below.

3.1. RanSub

RanSub [7] is proposed to address the challenge of locating disjoint content within a system. RanSub distributes random subsets of nodes' information through a tree by executing the *collect* and *distribute* processes. The *collect* process starts from the leaves and goes all the way up to the root. In this process, each node informs its parent about the information it has about its sub-tree by constructing a representative subset of all the nodes in it and then delivers the information all the way up to the root. The *distribute* process then starts from the root and delivers the information from a subset of nodes to each of its child. The child then distributes a subset information determined based on its own information and the subset information received from its parent to pick a subset of nodes and distribute the information further down the tree. Using the *collect* and *distribute* processes, RanSub delivers information from a random subset of nodes to each node per *epoch*.

A key operation in RanSub is the *compact* operation. In the *collect* process, *compact* constructs a fixed size subset to randomly and uniformly represent its sub-tree members. In the *distribute* process, *compact* constructs a fixed size subset to randomly and uniformly represent the global information for each of the current node's children. There are several flavors of the *compact* operation, and we choose the *RanSub-all-non-identical* option to deliver the update information. It distributes a random subset of nodes among the whole system, thus is suitable for the inconsistency detection purpose.

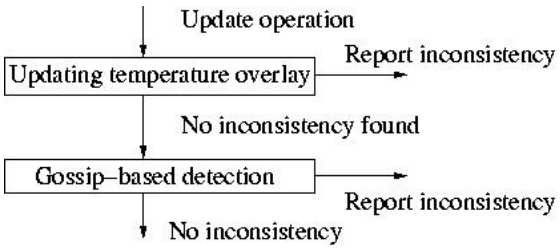

Figure2. Architecture of the two-layer Inconsistency Detection Module

While we use RanSub as an underlying protocol to exchange nodes' information among one another, we advance RanSub by proposing an interest-group based collect/distribute process.

3.2. Gossip-based Data Dissemination

To alleviate the scalability bottleneck of information dissemination, gossip based data dissemination has been proposed. The Lightweight Probabilistic Broadcast (lpbcast) [5] scheme advances the gossip-based scheme by eliminating the requirement of global view of the nodes. Instead, a node maintains a fixed size of a random subset of this system. Then a node disseminates non-duplicate packets to a randomly chosen subset of neighbors in its local view every T seconds. To minimize bandwidth cost, each message only travels a certain hops.

In the context of the two-layer inconsistency detection module presented in the next section, the bottom layer uses this gossip-based dissemination to distribute updates it receives to other members periodically.

4. Design of a Two-layer Inconsistency Detection Module

4.1. Measure the Updating Patterns

An important operation in the framework is to measure the updating patterns of nodes. Basically, we let each node track the number of its updates operations with regard to a particular file in a certain period of time (in the current study, we use 30 seconds as the default). Straightforwardly, the higher the number of updates on a file is, the higher the updating temperature of this file is. Because there could potentially be many files in a node's machine, there is actually a temperature vector in each machine, with each file having an entry in this vector.

While this scheme works, it is obviously not scalable or network bandwidth efficient. For example, a node may have 10,000 files in its machine but only modifies less than 10 files in a certain period of time. In this case, there is really no point of keeping a vector in which 99.9% entries are 0 (no updates in the past). To solve this problem, we introduce the notion of interest–group based temperature collection and distribution, which is described in Section 4.3. But before we proceed to that optimization, we first discuss the mechanism by which the nodes learn updating temperatures from each other, thus laying the background for future discussions.

4.2. Learning the Updating Patterns

Assume that each node is tracking its own updating patterns and has prepared its temperature vector. Then the temperature information is propagated via RanSub. Recall that RanSub assumes the existence of a multicast tree that covers all the nodes and use that to collect and distribute the nodes' information.

The only concern we have about RanSub is that it is based on a single-tree structure and thus can not tolerate even a single interior node failure. As identified in its original paper, possible ways to work around this include the use of multiple trees to substitute the single-tree structure. This is an interesting question and we are currently working on deploying a multi-tree based multicast, such as SplitStream [1], as the underlying communication mechanism of RanSub. We expect such mechanism to dramatically increase the resilience to node failures of the detection framework.

4.3. Interest-Group Based Temperature Collection / Distribution

One critical question about utilizing RanSub to propagate temperature information is how to minimize the network bandwidth cost. Without optimization, a huge amount of data (updating temperature information in this case) could be sent across the network, and that could put significant strain on the network. In this section, we propose an interest-group based temperature collection and distribution scheme to minimize the network bandwidth cost.

More specifically, we let the nodes only report the updating temperature of the files that they are interested in the *collect* process and every interior node tracks the interested files of its sub-tree. In the *distribute* process, an interior node only accept and forward the updating temperatures of files which are of interest to the nodes within its sub-tree.

To discuss this mechanism and analyze the bandwidth cost of it more formally, we define the parameters as follows.

Assume that the total number of nodes in the system is n and each node has k_i number of files in total. Each node is interested in p_i files within a certain period of time. Suppose that there are q exchanges involved in propagating the temperature information. Then we assume that each updating temperature entry has a size of s.

Because RanSub collects information through a tree structure, for the purpose of analysis, we assume that an interior node maintains m neighbors on average. If we assume a balanced tree, then the height of tree, h, is the smallest number larger than $log_m(n)$.

Thus, the number of total messages exchanged among the tree nodes in the *collect* process is:

$$N \le m + m^2 + \ldots + m^h$$

Each node only submits $k_i \times s$ bytes of data in the *collect* process. In the *distribute* process, a fixed number of nodes' information is distributed. Suppose the fixed number is b, then the message is of size $b \times k_i \times s$. Let k_a denote the average number of interested files. If the length of an epoch is L seconds, then the total bandwidth cost is:

$$\begin{aligned} Total_{BW} &= (N \times k_a \times s + N \times b \times k_a \times s) / L \\ &= (b+1) \times N \times k_a \times s / L \end{aligned}$$

There are N links in this RanSub tree, so on average, the bandwidth cost is:

$$\begin{aligned} Avg_{BW} &= Total_{BW} / N \\ &= (b+1) \times k_a \times s / L \end{aligned}$$

Given a network with parameters b of 100, k_a of 5, s of 10 and L of 30, the bandwidth cost is 183 bytes per second, which is small enough that can be supported by a dial-up connection.

To further reduce the bandwidth cost, we use a threshold to control the reporting of updating temperatures. If a node has a temperature less than a threshold t for a file and an interior node has already had at least k entries for this file, then the lowest temperature information will be dropped.

4.4. Two-layer Inconsistency Detection

When a module is updating a file, it consults the two-layer inconsistency detection module to detect any possible conflicts as follows. First, the node checks its local cache to carry out the top-layer detection, where

it chooses the node with the highest updating temperature on the file being updated and forwards the update to that node. If the receiving node has an update which conflicts with the one it receives, it notifies the sender directly. Otherwise, the receiving node chooses another node in its local cache that has the highest updating temperature on this file and relays the update to it. In addition, the traveling path is attached with the update to prevent the same update from traveling back to a previously visited hop.

If there is no conflict, then the update will stop eventually at a node that has no nodes in its local cache and has not been visited before. At this point, the bottom-layer inconsistency detection is triggered. The update is then sent to the last hop's friend list and its friend sends it out to the friend's friends again. To control flooding, each update only travels up to a predefined number of hops. This process is illustrated in Figure 3.

In Figure 3, the solid line represents the top layer (updating-temperature based) and the dotted line represents the bottom layer (gossip based). In the figure, when node *A* commits an update, it first traverses the top layer to check with *B* and *C* to detect any inconsistency. If either of them conflicts with *A*'s update, *C* (the last hop in the top layer) starts the detection from the bottom layer. In this case, if *E* happens to have conflict with *A*, then this inconsistency will be detected in the bottom layer.

Version vectors [10] are used to detect conflicts among updates. A version vector tracks the number of times a file is updated by a certain user and uses that to detect inconsistency. For example, version vector (A:3 B:5) is earlier in time than version vector (A:4 B:7). Two version vectors u and v are comparable if and only if u < v, u = v or u > v. If not, they conflict with each other. For example, (A:5 B:3) conflicts with (A:3 B:6).

In the ideal case, if all the nodes never change their interested files, then all the inconsistency can be resolved in the top layer. However, this is not the case in practice. The analysis about the case where the nodes change their interested files with a rate *r* is conducted in Section 5.1.

4.5. Caching and Garbage Collection

Two forms of caching are considered here. First, in caching of the temperature information, when a node receives the information about other nodes, it saves that information into its local cache. When new information arrives, it updates its local cache if the information is already in the cache. Otherwise, the new information is added to the cache.

Garbage collection is used to keep the temperature

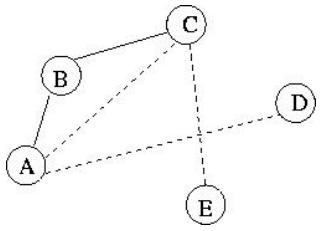

Figure3. Inconsistency Detection

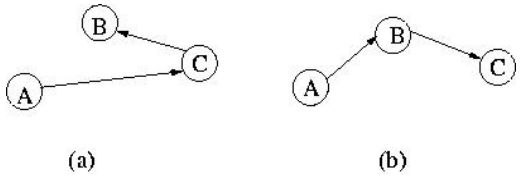

Figure 4. Issues with update propagation path

history fresh. To check the freshness, each entry in the local cache is assigned a time stamp, which basically tracks when that entry was last updated. If an entry is updated again, then the update time is reset to the new time. All the entries in the local cache are sorted according to its freshness. Periodically (the current study uses a period of 3 minutes) the garbage collection scans the list and removes all the entries that are older than that period.

The second caching scheme is to help minimize the update routing cost. Here, a node caches the propagation path along which the update traverses within the top layer to detect any inconsistency.

Figure 4 illustrates the process through an example. In (a), the update from *A* is forwarded to *C*, then *B*. The rationale is that the temperature of the updated file in *C* is higher than that of *B*, thus there is a better chance to resolve inconsistency by visiting *C* first than by visiting *B*. However, there is a tradeoff between high probability of resolving inconsistency and low routing cost. Consider (b), *A* visits *B* first, then *C* without sacrificing much routing delay. In general, (b) is a better update propagation scheme than (a).

We use a simple heuristic scheme to deal with the tradeoff. First, each node picks three nodes with the highest temperatures on the file being updated and compares their routing delays, based on information cached locally. It then chooses the closest node and forwards the update to it.

There are certainly other options available. For example, more complicated schemes could be developed to derive a formula and assign different weights to the two parameters. However, to accurately choose the right weights, extensive empirical studies need to be conducted to investigate the issue and we leave this to future work.

4.6. Discussion

In practice, there are several forms of updates, such as creating, modifying, and deleting operations. We believe that the proposed two-layer inconsistency detection module can benefit all the cases by minimizing the delay of inconsistency detection.

5. Evaluation

The two-layer inconsistency detection module is evaluated by both probabilistic analysis and simulation. In order to best evaluate the system performance, we choose the Transit-Stub model [14] to simulate a physical network. In the following simulations, the Transit-Stub model generates 1452 routers that are arranged hierarchically, like the current Internet structure. Then we generate 1,000 end nodes and attach them to routers randomly with uniform probabilities. Each end node was directly attached by an LAN link to its assigned router.

In all the simulations, we run each simulation 5 times and calculate the mean value.

5.1. Probability that the Top-Layer Fails to Detect an Inconsistent State

One of our goals is to determine the probability that a conflict is missed by the top layer and thus the bottom layer has to be triggered. Here, we assume that each node changes their interested files with a rate of *r*, which is defined as the ratio of the number of its newly interested files with the total number of its interested files in an epoch of RanSub. Thus *r* represents how fast a node changes its interest. We further assume that all the users have the same rate *r* and, given the collection of the files, the new interest of users is uniformly distributed across the whole collection.

One concern of this workload assumption is whether it can capture the different popularity patterns of the files, such as bursts. Here we argue that this workload is able to capture these different patterns by adjusting the total number of files in the system because, in bursty accessing patterns, what we really care is the worst case scenario, which is when its popularity is the highest, in which case we can decrease the total number of files in the collection to make each file more popular (given a fixed number of nodes, the smaller size the collection, the more popular each file is).

If a node becomes interested in a file for the first time, we let the node report its interest of new files one epoch before its updating. In practice, this can be done when a user first opens a file. This assumption is valid as long as the open operation is at least an epoch ahead of the real updating operation. We believe that this assumption is reasonable.

Hence the worst-case scenario happens when a node, *A,* just becomes interested in file *f* and then keeps updating that file. In this case, the only chance that the conflicts from *A* can be detected in the top layer is that other updaters can somehow find the information about this node in the top layer.

Suppose that there are *n* nodes in the system and each node receives *b* other nodes' information during an epoch. Thus after an epoch, each node receives *b* nodes' information, and in total there are $n \times b$ pieces of information exchanged across the system. Because this information is uniformly distributed, node *A*'s newest interest can be received by *b* other nodes on average. However, because each receiver changes its interest at rate *r*, only $(1 - r) * 100\%$ of the *b* nodes will be still interested in this file after an epoch. So after the first epoch, the number of nodes that are still interested in and maintain a link to node *A* with regard to the file *f* is:

$$N_{new} = b \times (1-r) \qquad (1)$$

Then we assume that there are N_{exist} nodes already interested in this file in this system before node *A* becomes interested in it and they have already formed a top-layer overlay and are maintaining it.

Suppose another node *B*, different from *A*, commits an update on file *f*. Here we assume that *B* is among the N_{exist} nodes which are already in the top layer with regard to file *f*. The case that *B* is not among the N_{exist} is discussed later.

Then the only case where *B* cannot reach *A* is that the N_{new} nodes have no overlap with the N_{exist} nodes, with a probability of:

$$p = \left(\frac{n - Nnew}{n} \right)^{Nexist} \qquad (2)$$

As with the analysis in Section 3.3, we still assume a network of 1000 nodes with an exchange size *b* of 100. Suppose that the rate *r* is 0.2, hence $N_{new} = 80$ from formula (1). If file *f* is a hot file with 20 nodes interested in it, then $N_{exist} = 20$. Hence from formula (2), the probability that the conflict on *f* cannot be resolved in top layer is 18.9%.

Now we come back to the case where *B* is not in N_{exist}. In this case, *B* is a new comer for file *f* as is for *A*. In this case, the probability that the conflicts from *B* and *A* cannot be resolved is conditioned on two events.

First, the N_{new} from A and B cannot overlap. And, second, either N_{new} from A or N_{new} from B is not overlapped with N_{exist}. And this probability can be represented as:

$$p' = \left(\frac{n - Nnew}{n} \right)^{Nnew} * 2p \qquad (3)$$

Using the same set of parameters, we calculate p' to be 0.04%, which is much smaller than the earlier case of 18.9%.

In summary, the probability that the top layer fails to detect conflicts is quite low when the file is hot. This result fits well with our design goal, which is to minimize the delay of inconsistency detection. The proposed scheme is especially effective when there are a lot of spontaneous updates, an indication that a file is becoming hot. In this case, as presented in formula (3), the top layer can detect the inconsistency among them with a probability very close to 1. For the two cases, case two (thus formula 3) happens when a file becomes hot suddenly and many users access it the first time, while case one happens in other cases.

5.2. Maintenance Cost

The number of messages received by each node during the maintenance process is used to evaluate the maintenance cost of the temperature overlay.

We run the two-layer inconsistency detection module for 800 seconds. Because the RanSub process starts at the end of 30 seconds, there are 26 epochs involved in total. At the end of the simulation, we collect the number of messages received by each node and the result is illustrated in Table 1.

Although the Max (which comes from the root of the tree which RanSub uses) is much higher than the mean value, it must be pointed out that it is accumulated over 26 epochs. Thus within each epoch, it receives 180 messages which equals to 6 messages per second (one epoch runs every 30 seconds). Even if the size of a message is 1KB, the network bandwidth cost is only 6KB/s for the root. From that we can see that the maintenance cost will not overwhelm the root.

This maintenance cost can be further reduced by utilizing multiple tree based RanSub as follows. If there are multiple trees, and they are configured with different root, each epoch can then use a different root to run the RanSub procedure, in which all the roots share the maintenance cost.

6. Inconsistencies Resolution

Table 1. Maintenance cost

Max	Mean	Median
4680	51.9	26

As discussed before, one important advantage of timely inconsistency detection is the opportunity to enforce different consistency levels according to the application semantics. In this section, we outline the mechanisms to capture the application semantics and resolve the detected inconsistencies.

In practice, there are two ways to get information about application semantics. First, the middleware can ask the users to specify their preferences before they use the system. For example, in an online conference, the users can specify the most important speaker to them. Thus, when inconsistency about this particular speaker has been detected, the middleware will resolve it as soon as possible. However, the middleware will not resolve the inconsistency associated with other non-important speakers because the users do not care and are willing to sacrifice consistency for better performance, such as low transmission delay (inconsistency resolution takes time).

Still, another method to discern the semantics is to monitor the systems behavior and use feedback control to modify the middleware's response. Deployed on the users' side, these monitors track the users' response to these inconsistencies and increase the inconsistencies priority when users indicate their dissatisfaction with them.

Several schemes can be used to resolve the inconsistency. The middleware can, for example, send the correct version to all the replicas, if the inconsistency can be resolved in the middleware level. Otherwise, the middleware may flag an inconsistency alert to the system administrator for human intervention [8]. Therefore, inconsistency resolution can be made versatile to different applications.

7. Related Work

TACT [13] recognizes the inherent tradeoff between consistency level and performance, as well as the rich semantics of this trade-off, It proposed a set of parameters to measure the consistency level of applications and developed algorithms to bound the inconsistency within a certain level. However, it is still in realm of inconsistency avoidance while this paper promotes an inconsistency detection framework.

DENO [2] is a decentralized, peer-to-peer object-replication system for a loosely connected environment. The novelty of DENO lies in its combination of weighted voting and pair-wise,

epidemic information flow. While DENO improves voting scheme to provide weak consistency, we believe that it is still very hard for voting scheme to deal with the dynamism and security issues in a large-scale network. We believe that probabilistic schemes, such as the two-layer inconsistency detection mechanism proposed in this paper, are more scalable and robust.

Lpbcast [5] is a gossip based broadcast protocol and a similar scheme is deployed in the bottom layer of our two-layer inconsistency detection framework as a backup protocol. The difference between lpbcast and our work is that lpbcast is a pure gossip based broadcast protocol while ours is designed to further minimize the delay of inconsistency detection.

Quorum system [9] is a widely deployed scheme to maintain consistency in distributed systems. Depending on the structure of the system, a certain number of nodes can organize a quorum which promises that no others can organize another quorum at the same time. However, the quorum could possible fail in the presence of node failures. Unlike quorum system, our two-layer inconsistency detection framework is robust because if a node fails, the messages can always be route the update to other candidates, in both layers.

8. Conclusions and Future Work

We presented an inconsistency detection framework, as an alternative to inconsistency avoidance, in Internet-scale distributed systems. The detailed design and evaluation of a two-layer inconsistency detection module was elaborated and evaluated by both analysis and simulations. Results show that, with this inconsistency detection module, most inconsistency of hot files can be detected in the top layer with a high possibility. Further, this framework is bandwidth efficient and with low maintenance cost.

In the future, we plan to use this inconsistency detection framework to support consistency-conscious applications, such as e-business applications.

Acknowledgements

This work is partially supported by the National Basic Research Program of China (973 Program) under Grant No. 2004CB318201.

References

[1] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth multicast in cooperative environment. In *Proc. of the SOSP*, Bolton Landing, New York, USA, October 2003.

[2] U. Cetintemel, P. J. Keleher, B. Bhattacharjee, and M. J. Franklin, Deno: A Decentralized, Peer-to-Peer Object-Replication System for Weakly-Connected Environments, IEEE Transactions on Computers, 52(7), 2003

[3] T. Chang, G. Popsecu, and C. Codella, Scalable and Efficient Update Dissemination for Interactive Distributed Applications, In Proc of the International Conference of Distributed Computing System (ICDCS), Viena, Austria, July 2-5, 2002

[4] D. Dullmann, W. Hoschek, J. Jaen-Martinez, B. Segal, A. Samar, H. Stockinger, and K. Stockinger, Models for Replica Synchronization and Consistency in Data Grid, In Proc. of 10th IEEE International Symposium on High Performance Distributed Computing (HPDC), Aug. 7-9, pp. 67-75, 2001

[5] P.T. Eugster, R. Guerraoui, S. B. Handurukande, A. M. Kermarrec, P. Kouznetsov. Lightweight Probabilistic Broadcast, In Proc of the International Conference on Dependable Systems and Networks (DSN 2001), July, 2001

[6] James J. Kistler and M. Satyanarayanan, Disconnected Operation in the Coda File System, ACM Transactions on Computer Systems, 10(1) pp. 3-25, February 1992

[7] D. Kostic, A. Rodriguez, J. Albrecht, A. Bhirud, and A. Vahdat. Using Random Subsets to Build Scalable Network Servioces, In Proc. of 4th USENIX Symposium on Internet Technologies and Systems. March 2003

[8] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen, Ivy: A Read/Write Peer-to-Peer File System, OSDI 2002

[9] M. Naor, U. Wieder, Scalable and Dynamic Quorum Systems, In Proc. PODC 2003

[10] D. Parker, G. Popek, G. Rudisin, A. Stoughton, B. Walker, E. Walton, J. Chow, D. Edwards, S. Kiser, and C. Kline, Detection of mutual inconsistency in distributed systems. In IEEE Transactions on Software Engineering, 9(3), pp. 240-247, 1983

[11] M. Stonebraker, Concurrency Control and Consistency of Multiple copies of Data in Distributed INGRES, IEEE Transactions on Software Engineering, 5(3), May 1979

[12] D. B. Terry, M. M. Theimer, K,. Petersen, A. J. Demers, M. J, Spreitezer, and C. H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System, In Proc. of the Fifteenth ACM SOSP, 1995

[13] H. Yu and A. Vahdat, Design and Evaluation of a Continuous Consistency Model for Replicated Services, In. Proc. OSDI 2000

[14] E. Zegura, K. Calvert, and S. Bhattacharjee. How to model an internetwork. In *INFOCOMM*, San Francisco, California, 1996.