Adaptive Consistency Guarantees for Large-Scale Replicated Services

Yijun Lu, Ying Lu, and Hong Jiang
Department of Computer Science and Engineering, University of Nebraska-Lincoln
{yijlu, ylu, jiang}@cse.unl.edu

Abstract

To maintain consistency, designers of replicated services have traditionally been forced to choose from either strong consistency guarantees or none at all. Realizing that a continuum between strong and optimistic consistencies is semantically meaningful for a broad range of network services, previous research has proposed a continuous consistency model for replicated services to support the tradeoff between the guaranteed consistency level, performance and availability. However, to meet changing application needs and to make the model useful for interactive users of large-scale replicated services, the adaptability and the swiftness of inconsistency resolution are important and challenging.

This paper presents IDEA (an Infrastructure for DEtection-based Adaptive consistency guarantees) for adaptive consistency guarantees of large-scale, Internet-based replicated services. The main functions enabled by IDEA include quick inconsistency detection and resolution, consistency adaptation and quantified consistency level guarantees. Through experimentation on the Planet-Lab, IDEA is evaluated from two aspects: its adaptive consistency guarantees and its performance for inconsistency resolution. Results show that IDEA is able to provide consistency guarantees adaptive to user's changing needs, and it achieves low delay for inconsistency resolution and incurs small communication overhead.

1. Introduction

Replicating data and services in distributed systems is an attractive strategy to increase availability and performance. For large-scale, Internet-based systems, replication may indeed be the only way to provide continuous services and to avoid data loss in the presence of unreliable Internet connections [3, 16]. In this environment, consistency control has become critical because poor consistency for replicated services results in poor QoS and even monetary losses for e-business applications. The dilemma is, on the one hand strong consistency [1] is very costly to maintain, while on the other hand optimistic consistency [4, 13, 14] can be too weak in certain scenarios [3]. Realizing that many applications are willing to sacrifice a certain degree of consistency for scalability, recent research has [16] proposed a continuous consistency model to support the tradeoff between a guaranteed consistency level and the desired scalability.

In this paper, we argue that it is equally, if not more, important to achieve *adaptability* in consistency control.

First, a system should be able to adjust the consistency levels for different objects on the fly, as opposed to maintaining a predefined consistency level for all objects. This is important because multiple applications with different consistency requirements can run simultaneously in a distributed system. While several consistency protocols can be deployed at the same time to cater to different applications, it would inevitably increase the complexity of the system design [15]. Besides, some application's requirement for consistency changes from time to time. Take online conference as an example. Users require higher consistency when an important speech is going on while they are willing to tolerate lower consistency for better performance otherwise [2]. In this scenario, a predefined protocol is incapable of capturing the semantic.

Second, users should have the control on how system adjusts the consistency levels for their objects. The system could start with maintaining a default consistency level for an object. Dynamically, users should be able to adjust the consistency when they are not satisfied or when their needs change. This way, although users may not be good at expressing their desired levels in quantitative terms, they could determine on the fly whether or not a given consistency level is adequate and accordingly control the system to get what they want.

To support interactive users of large-scale replicated services, it is also important that our system achieves high performance for the consistency control. That is, our system should be able to quickly detect inconsistencies and, when necessary, to resolve them in a timely manner. This is crucial because slow inconsistency detection and resolution will lead to poor QoS and cause user frustrations.

To this end, we present IDEA (an Infrastructure for DEtection-based Adaptive consistency guarantees) that achieves both adaptability and high-performance goals.

For the adaptability, IDEA adjusts the consistency levels on the fly through interactions with users. Upon the detection of inconsistencies, IDEA resolves them if the current consistency level does not satisfy user requirements; otherwise, inconsistencies will not be resolved unless the system is lightly loaded. For high performance, we extend our previous work [6, 7] and design fast inconsistency detection and resolution in IDEA.

To validate the design, we implement an IDEA prototype on the Planet-Lab [12]. On top of it, we emulate two applications, a distributed white board system and an airline ticket booking system. Collectively, they show that IDEA achieves the design goals of adaptability and high performance in consistency guarantees.

This paper has two contributions. First, we point out the importance of adaptability in consistency control and present IDEA to provide this adaptability. Second, we demonstrate the effectiveness of IDEA and its high performance by deploying a prototype on Planet-Lab.

2. Design

We assume that IDEA works with a general distributed file system that handles the ordinary read and write operations. As shown in Figure 1, IDEA is deployed in the middleware level. Applications on different nodes consult IDEA when they access files. Upon initiating an application, a user may choose to give a hint on his or her acceptable consistency level. The desired consistency level L_1 for an object will be set according to the hint or to a default value. (Note: throughout the paper, the terms object and file are used interchangeably.)

2.1. A Two-Layer Infrastructure

We build IDEA on top of a two-layer infrastructure. This infrastructure is first introduced by the authors in [6, 7]. It constructs a two-layer overlay for each object of the system, where the top layer includes those nodes that frequently update this object and the bottom layer consists of the remaining nodes.

Two reasons motivate us to adopt this two-layer architecture. First, it is very unlikely that participants in a large application are all interested in modifying the same object at the same time. Thus it is expected that all active writers form a much smaller subset of the whole participant group. Second, the small size of the top layer implies that it is much faster to detect and resolve inconsistencies among members of the top layer than that of the whole group.

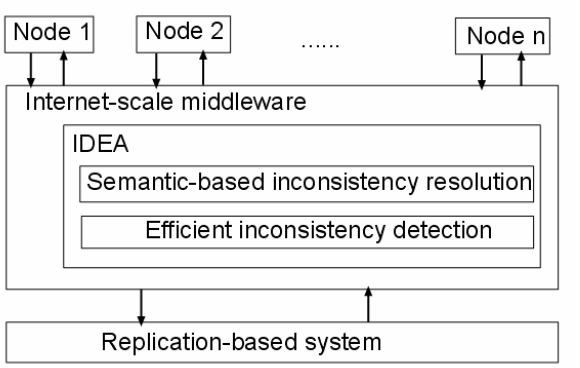

Figure 1: IDEA: a middleware service

2.2. Protocol

An outline of the IDEA protocol is depicted in Figure 2. It is triggered by two operations: write and certain read operations. Because the write operation, such as issuing an update to a white board, will cause inconsistency among replicas, it triggers the IDEA protocol. For read operations, the IDEA protocol is triggered when a user tries to retrieve a new file, such as a new snapshot of a white board, because in this case the system needs to make sure that the file retrieved is sufficiently consistent for the user. For other reads whether or not the protocol is triggered depends on the context: if the file is frequently updated locally, the reads will not trigger the IDEA protocol; otherwise they will.

After the protocol is triggered, it uses a detection mechanism to identify the inconsistency and then a customized mechanism to quantify the consistency level for a particular user. After the consistency level is calculated, IDEA checks whether it is acceptable, *i.e.,* above the level desired for the user. If acceptable, IDEA returns the file to the user; otherwise, IDEA resolves the inconsistency.

To achieve good responsiveness, IDEA first calculates the consistency level only among the top-layer nodes. This value may not be accurate because, albeit rather infrequently, the bottom layer nodes can also introduce inconsistencies. Hence, we deploy a rollback mechanism to solve this problem. After the file is returned from IDEA, users could proceed with their work. However, in the background IDEA continues to detect inconsistency in the bottom layer and calculates a new consistency level for the file. If the new value is sufficiently close to that obtained from the top layer, no action is needed; otherwise, IDEA alerts the user about the discrepancy and, upon the user's request, resolves the inconsistency and rolls back the user's operations.

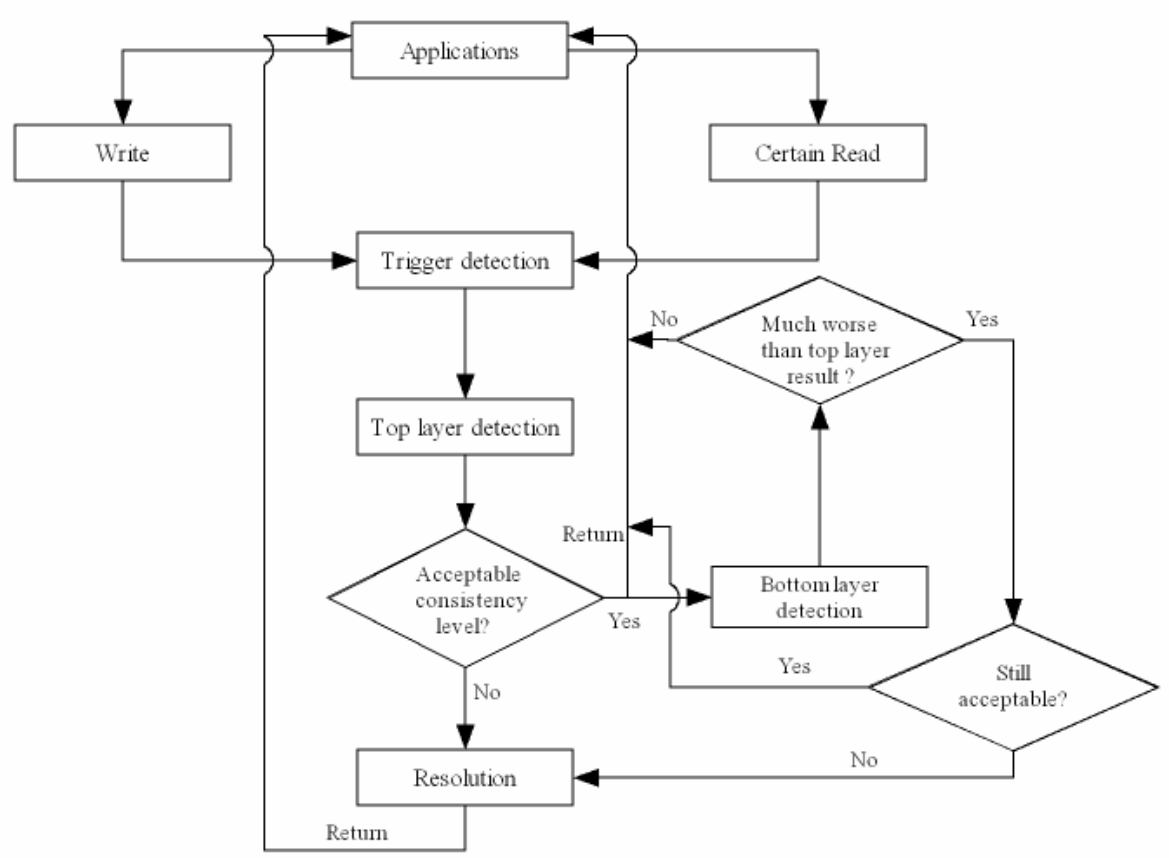

Figure 2: The IDEA protocol

2.3. Inconsistency Detection

To provide consistency guarantees, the inconsistency detection is the first component of IDEA. We have proposed in [6, 7] an efficient, low-cost inconsistency detection mechanism. IDEA adopts that for detecting inconsistencies.

The conflicts of two or more updates, *i.e.,* the inconsistencies among different replicas, are detected through exchanges of version vectors [11]. A version vector tracks the number of times a file is updated by certain users. For example, version vector (*A*:3 *B*:5) means that user *A* has modified the file three times and user *B* has modified it five times. So the replica represented by this version vector is considered more obsolete than that presented by version vector (*A*:4 *B*:7). Two replicas are inconsistent if their version vectors are different. By exchanging version vectors in the two-layer infrastructure [6, 7], most inconsistencies are detected in the top layer. Our previous research [8] has shown that 95% of inconsistencies are quickly detected in the top layer.

2.4. Consistency Level Quantifications

This section describes how IDEA quantifies the object consistency level for a user. We first extend the object version vector and then borrow the <*numerical error, order error, staleness*> metric from the TACT continuous consistency model [16] to quantify the consistency.

The current version vector [11] only tells the number of updates from individual writers. It is not sufficient for our purpose. Thus we design an extended version vector. As illustrated in Figure 3, the extended version vector includes several new items.

First, the extended version vector has timestamps associated with each update. For example, <*A*:2(1, 2)> means that the two updates from user *A* happen at time points 1 and 2. We assume that the differences among clocks of participating nodes are within seconds. This can be achieved by running a clock synchronization algorithm such as the one proposed in [5], or by letting all participating nodes use NTP (Network Time Protocol) [10] to synchronize with a time sever. This way the timestamps of different nodes become comparable.

Second, we use a numerical value in square brackets, for instance <… [5] …>, to represent some critical metadata for the application. It is used to

characterize the value difference of different replicas. For examples, in the case of a distributed white board, the metadata could be the sum of the ASCII values of the few recent local updates; while in an airline ticket booking system, the metadata could be the total sale price. The differences in these metadata indicate the effect of the conflicts.

Third, the <*numerical error, order error, staleness*> triple is attached at the end of the extended version vector. The numerical error is derived by comparing the metadata values; the order error counts the difference between numbers of updates and the staleness is calculated from the timestamps.

Because IDEA adopts this extended version vector, for brevity we will simply use the term "version vector" to refer to the "extended version vector" in the remaining of the paper.

We now illustrate how to quantify the consistency level. Assume that an object has two active writers (*A* and *B*) and two associated replicas (*a* and *b*). And after several updates, *a* and *b*'s version vectors are *<A:2(1,2) B:0 [5] 0 0 0 >* and *<A:0 B:1(3) [2] 0 0 0 >* respectively.

IDEA first derives a *reference consistent state*. It is a state that is considered as the basis for the consistency level evaluation. In Section 2.5 we will show that there are multiple ways to derive the *reference consistent state*. For now, let us assume that if two replicas are in conflict with each other, the replica with higher ID value becomes the reference. Thus, for the above example *b* is considered in the *reference consistent state* and is used to calculate the consistency levels for replicas *a* and *b*.

To quantify the consistency levels, the <*numerical error, order error, staleness*> triples are first computed, where the numerical error is derived from the metadata values; the order error counts the difference between numbers of updates and the staleness is defined as the time difference between the most recent update in the *reference consistent state* and the last time when the replica is consistent with the reference. For replica *a*, since its metadata differs in 3 units from that of *b* (the reference state), its numerical error is 3; replica *a* misses one and has two extra updates, so its order error is 3; finally, the last time when *a* is consistent with *b* is at time point 1, the most recent update at *b* is at time point 3, and their difference is 2, so the staleness of replica *a* is 2. After the calculation, *a*'s version vector becomes *<A:2(1,2) B:0 [5] 3 3 2 >*. The version vector remains *<A:0 B:1(3) [2] 0 0 0 >* for replica *b* because *b* is in the *reference consistent state*.

IDEA quantifies the consistency levels using the maximum values and customized weights of the <*numerical error, order error, staleness*> triple. For example, if the order error is always less than 10, then

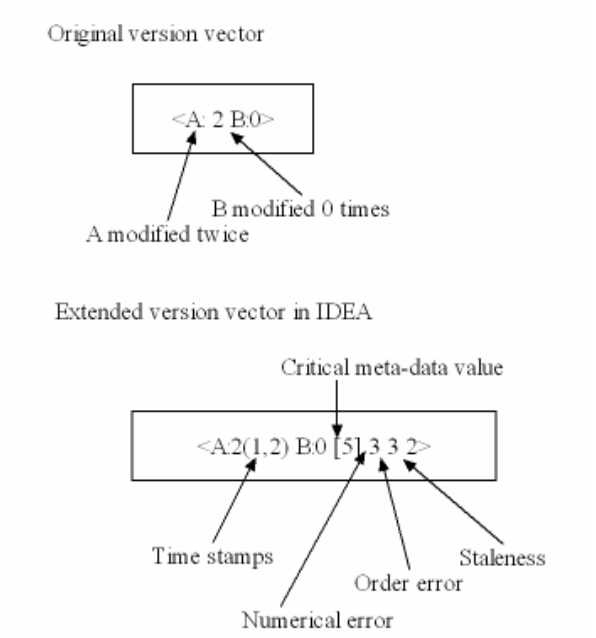

Figure 3: Extended version vector

the maximum value of the order error (*max_order*) can be set to 10. The weights (*num_weight*, *order_weight*, and *stale_weight*) on *numerical error, order error,* and *staleness* are customized for individual user of the object and they can be adapted according to the user's changing needs (Section 2.8). For example, if a user weighs the three types of errors equally, *num_weight*, *order_weight*, and *stale_weight* will all be 33.3%. For a user, the consistency level of a replica is quantified by the following equation:

$$Consisntecy = \frac{\max\_num - num\_error}{\max\_num} \times num\_weight + \frac{\max\_order - order\_error}{\max\_order} \times order\_weight + \frac{\max\_staleness - staleness}{\max\_staleness} \times stale\_weight \quad \ldots (1)$$

2.5. Inconsistency Resolution Policies

After detecting inconsistencies and quantifying the consistency level, IDEA checks whether or not the consistency is above the user's desired level. If not, IDEA invokes the inconsistency resolution module to provide the user with the consistency guarantee.

Given two replicas, if their version vectors *u* and *v* are different, they are inconsistent. In the presence of inconsistencies, IDEA will derive a *reference consistent state* for the object. There are multiple ways to derive the *reference consistent state*. Which method

to apply is determined by the inconsistency resolution policy. Several typical policies are presented in this section.

Two version vectors are comparable if and only if $u < v$, $u = v$, or $u > v$. When comparable, it is straightforward — we should always use the latest replica, *i.e.,* the replica with the bigger version vector. That replica therefore defines the *reference consistent state*. The decision becomes difficult when the version vectors are incomparable. For example, *<A:2(1,2) B:0 [5] 0 0 0 >* is not comparable with *<A:0 B:1(3) [2] 0 0 0 >* because the first item, the number of updates by A, of the first vector is bigger but its second item, the number of updates by B is smaller. In this case, the best choice of the *reference consistent state* is not obvious. To solve this problem, three different policies are listed and their applications are discussed as follows.

- **Invalidate Both**. Following this inconsistency resolution policy, both of the conflicting versions are considered invalid and they will be rolled back to a previous consistent state. That state is thus defined as the *reference consistent state*. For example, in a distributed white board system, two simultaneous updates at the same spot can be both cleared to prevent ambiguity and to ensure fairness (*i.e.,* no one is more important than the other).
- **ID-Based**. In this policy, each node is assigned a random ID, like a hash value of their IP address via MD5 (a commonly used hash function in Peer-to-Peer systems). When detecting a conflict of version vectors, the system chooses the node with a larger ID and considers its replica to be in the *reference consistent state*. This approach can be used in both a distributed white board system and an airline ticket booking system when users prefer progresses to fairness. Unlike the **invalidate both** policy where updates are rolled back, this approach always proceeds with the discussion in a white board system and sells more tickets in an airline ticket booking system.
- **Priority-Based**. This policy assigns different priorities to users. When a conflict arises, the higher-priority user wins and his or her replica is chosen as the reference. For instance, a supervisor could have a higher priority than employees of a company. Thus the supervisor's updates on a white board are always preserved. In an airline ticket booking system, higher priorities should be given to preferred customer groups.

2.6. Background & Active Resolutions

IDEA invokes the consistency resolution in two ways, referred to as background and active resolutions.

To improve the consistency level, the background resolution resolves inconsistencies periodically. It invokes resolution without the user's intervention.

Unlike the background resolution, the active inconsistency resolution is triggered by a user or by IDEA when the consistency level is not acceptable. Upon the resolution request, the nearest replica acts as the initiator and starts a two-phase protocol. The initiator first sends a request to all top layer nodes to call for attention to the upcoming resolution process. If no one else is initiating the same process, the initiator will get positive acknowledgements from all top layer nodes, and start the resolution procedure. However, if someone else has sent out the same request, this initiator will back-off and retry after a random period. Here back-off is used to avoid redundant resolutions and to save bandwidth. During the back-off period, if the initiator receives a notice that the requested resolution has been started by someone else, this initiator will simply cancel the retry process.

Once the resolution is started, the procedure is the same for the background and active approaches. IDEA sequentially visits all top layer nodes and collects the replica version vectors. By following a resolution policy presented in Section 2.5, IDEA derives the *reference consistency state* and determines updates that are needed for building the consistent replica. It then sends the information to top layer nodes and other involved members to let them construct or retrieve the consistent replica.

2.7. Control the Adaptation

Broadly speaking, IDEA provides three mechanisms for controlling the adaptation that cater to different application semantics.

- **Hint-Based**. Following this scheme, users could specify the consistency levels that might satisfy their needs in advance. IDEA then guarantees their perceived consistencies are always above the desired levels.
- **On-Demand**. This scheme provides users with a way to dynamically adjust the consistency when they are not satisfied or when their needs change. This way, although users may not be good at expressing their desired levels in quantitative terms (*i.e.,* hints), they could still determine on the fly whether or not the guaranteed consistency is adequate and control the system accordingly.

- **Fully Automatic**. IDEA also provides a mechanism to balance between the consistency and the overhead, where best-effort consistencies are maintained without violating the system overhead constraints. In this scheme, IDEA applies background resolution, whose invocation frequency is determined by the allowed overhead. A potential application of this scheme is the airline ticket booking system. For example, if the consistency overhead is deemed not to exceed 20% of the system capacity so that enough resources will be used to process customer requests, then the frequency of IDEA background resolution should be adjusted accordingly.

2.8. User Interface

We provide users with an interface to control how IDEA adjusts the consistency levels for their objects. Before starting the application, users could give hints on their required consistency. As explained in Section 2.4, IDEA quantifies the consistency level using the values and customized weights of the <*numerical error, order error, staleness*> triple. Since different users may weigh the three types of errors differently for different applications, the interface allows users to customize the weights (*num_weight*, *order_weight*, and *stale_weight*) as well as the desired consistency level.

However, users may not know how to quantify their desired levels and weights beforehand. Moreover their needs could change dynamically. Therefore, IDEA also provides users with an interface to adjust their preferences on demand. Via the interface, users could communicate with IDEA and provide feedback on their satisfaction with the consistency and the system responsiveness. If a user prefers better responsiveness to consistency, IDEA will lower his or her desired consistency level and thus improve the system responsiveness. If it is the consistency that is not acceptable, a user could control the guaranteed consistency level by adjusting the weights or boosting the desired level.

Take the distributed white board system as an example. In the system, numerical error denotes the metadata difference of replicas; order error measures the update sequence error; and staleness reflects the replica's up-to-dateness. Among them, the order error is a common and the most confusing error because white board writes make sense only if they are placed in order. Thus, if white board users prefer order preservation to reduced staleness, IDEA could give more weight to the order error such as assigning 70% to *order_weight* and 10% to *stale_weight*.

3. Evaluation

Three metrics—consistency level, delay, and communication overhead—are used to evaluate the adaptability and performance of IDEA.

3.1. Adaptive Guarantees

We emulate a distributed white board application to show the effectiveness of IDEA in achieving the consistency guarantee and adaptability. In this application, a participant uses the IDEA interface to indicate a desired consistency level. We assume when consistency is above the specified level, the user is satisfied. Thus, IDEA should provide the quantified consistency level guarantee. In addition, when needs change and the user is no longer satisfied with the guaranteed consistency level, he or she should be able to control IDEA and adjust the consistency.

We run the experiments on forty Planet-Lab [12] nodes, among which four of them are concurrent writers of a given file. After the system warms up, the four nodes become the top layer of the file. To carry out experiments in an Internet scale, the forty nodes are chosen from all over US and Canada. In addition, the concurrent writers are carefully chosen to be far apart from each other.

In this group of experiments, each of the four writers issues an update every 5 seconds. Thus during a 100-second experiment period, there are totally 20 updates per writer. For the first two experiments, the desired consistency levels are set at 95% and 85% respectively. Therefore, IDEA begins to resolve inconsistencies when the consistency level is below 95% or 85%. In Figures 4(a) and 4(b), we show the experiment results, where the "view from the user" curve presents the consistency perceived by the first writer who triggers the IDEA active inconsistency resolution and the "system average" curve is the average consistency level of all writers. As demonstrated by the curves, the consistency levels improve right after IDEA invokes the active resolution. After the resolution, the consistency is guaranteed to be above the desired level (95% or 85%).

In both scenarios, IDEA is able to resolve inconsistencies fairly quickly. Since we only sample the system once every five seconds, all we can tell from the two figures is that the consistency is brought back in less than five seconds. As measured in Section 3.2, it actually takes only 315ms to actively resolve the inconsistencies.

In the third experiment, we evaluate IDEA for its consistency adaptability. The experiment runs for 200 seconds, where initially the desired consistency level is set at 95% and after 100 seconds it is changed to 90%. From Figure 5, we can see that once the consistency

falls below the *current* desired level, IDEA resolves inconsistencies to provide the consistency guarantee. Clearly the adaptation of the guaranteed consistency level is achieved.

3.2. Responsiveness

This section uses the distributed white board application to evaluate the delay of inconsistency resolution, *i.e.,* the responsiveness of IDEA. Assume that a user adapts the desired consistency level on-demand and triggers the active resolution because he or she is not satisfied with the current consistency level. In this scenario, it is important that IDEA quickly resolves inconsistencies and guarantees the user the desired consistency level.

To measure the average delay, we run the active inconsistency resolution four times and each time we pick a different writer to initiate the request. The final result reflects the average resolution delay of the four runs. Table 1 shows the average delay per active inconsistency resolution. It breaks down into two phases: the call-for-attention as phase one and the actual resolution as phase two (Section 2.6). As confirmed by the data, phase one is executed much faster than phase two. There are two reasons. First, the operation in phase one is only a call-for-attention, thus it causes little computation overhead, compared to the actual resolution process in phase two, which involves collecting and analyzing the information of replicas, deriving a consistent state and constructing a consistent replica. Second, the call-for-attentions to different nodes are executed in parallel, which further improves its efficiency; while for the second phase, our current implementation involves sequential visits of all top layer members.

Based on the data collected, we extrapolate the performance of IDEA in a more dynamic environment where there are more simultaneous writers. This is used to analyze the scalability of the IDEA inconsistency resolution. Because phase one is executed in parallel, its performance does not change significantly with the top layer size. On the other hand, since phase two operations are executed sequentially, its delay is estimated to increase linearly with the top layer size. The results shown in Table 1 are based on a top-layer of four nodes, where, when resolving inconsistencies, one node initiates the operation and the other three nodes are contacted sequentially. Thus, on average it takes about 104.747 (*i.e.,* 314.141 / 3) ms to process one top layer node. Consequently, the response time of active resolution for an *n*-node top layer is extrapolated as follows:

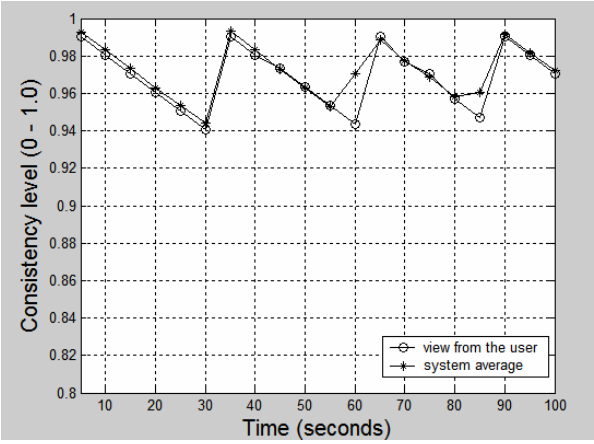

Figure 4(a): Resolution guarantees consistency level ≥ 95%

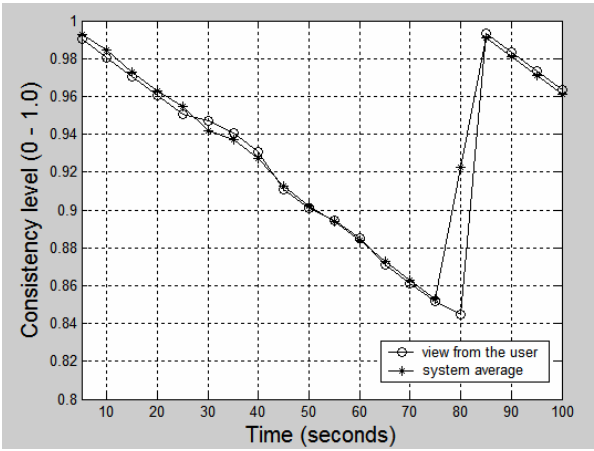

Figure 4(b): Resolution guarantees consistency level ≥ 85%

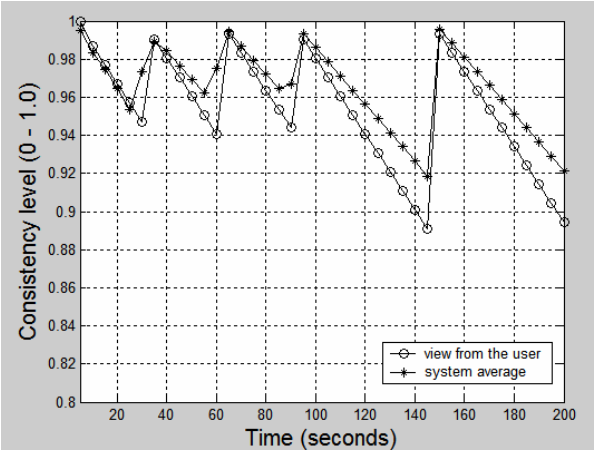

Figure 5: Adaptive guarantees

	Average delay in one round of active resolution
Phase1	0.46825 ms
Phase2	314.241 ms

Table 1: Delay of active resolution

$$\text{Delay} = 0.46825 + 104.747 * (n-1) \quad \ldots (2)$$

Besides the active resolution, we have also introduced the background resolution in Section 4.6. One round of background resolution essentially consumes the same amount of time as the phase two of the active resolution. Thus its delay is approximated by the following equation:

$$\text{Delay} = 104.747 * (n-1) \quad \ldots (3)$$

We can clearly see that even with ten simultaneous writers, which is highly unlikely in practice, the cost of active or background resolution is still below one second. We believe that this is a reasonably good performance because in a large-scale distributed system it is not uncommon that a message is delayed for seconds or more [3], thus offsetting the impact of the delay caused by IDEA. Nonetheless, if performance is a concern, to further improve the responsiveness of IDEA, a parallel resolution mechanism can be easily implemented and deployed (*i.e.,* to let the initiator contact all top layer nodes in parallel).

3.3. Communication Overhead

To study the communication overhead, we emulate on IDEA an airline ticket booking system, which mainly applies the background inconsistency resolution. Due to page limit, the detailed experimentation is omitted, but the main result is that IDEA incurs minimal communication overhead. The details of this experiment and how users can set the resolution frequency so as to satisfy the system overhead constraint can be found in [9].

4. Conclusion

In this paper, we made two contributions. First, we point out the importance of adaptive consistency guarantees and design IDEA to provide this service. Second, we implement and deploy an IDEA prototype on the Planet-Lab to show its adaptability.

References

[1] P. Bober and M. Carey, Multiversion Query Locking, in Proc. of 18th Conference on Very Large Databases, San Francisco, CA, USA, 1992. pp. 497-510.

[2] T. Chang, G. Popsecu, and C. Codella, Scalable and Efficient Update Dissemination for Interactive Distributed Applications, in Proc. of ICDCS 2002, Viena, Austria, July, 2002.

[3] 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

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

[5] C. Liao, M. Martonosi, and D. W. Clark, "Experience with an adaptive globally-synchronizing clock algorithm," in Proc. of ACM Symposium on Parallel Algorithms and Architectures, Saint Malo, France, 1999, pp. 106-114.

[6] Y. Lu and H. Jiang, A Framework for Efficient Inconsistency Detection in a Grid and Internet-Scale Distributed Environment, In Proc. of HPDC-14. Research Triangle Park, NC, pp. 318-319.

[7] Y. Lu, H. Jiang, and D. Feng, An Efficient, Low-Cost Inconsistency Detection Framework for Data and Service Sharing in an Internet-Scale System. In Proc. of IEEE ICEBE 2005, pp. 373-380.

[8] Y. Lu, X. Li, and H. Jiang, Accurate Performance Modeling and Guidance to the Adoption of an Inconsistency Detection Framework, *2008 IEEE International Conference on Networking, Architecture and Storage (NAS 2008),* Chongqing, China, June 2008.

[9] Y. Lu, Y. Lu, and H. Jiang, IDEA: An Infrastructure for Detection-based Adaptive Consistency Control in Replicated Services, *Technical Report TR-UNL-CSE-2007-0001,* University of Nebraska-Lincoln, Jan. 2007.

[10] D. L. Mills, A brief history of NTP time: memoirs of an Internet timekeeper, ACM SIGCOMM Computer Communication Review, 33 (2), April 2003. pp. 9-21.

[11] 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

[12] L. Perterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet, In Proc. of ACM HotNets-1 workshop.

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

[14] 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 15th ACM SOSP, 1995

[15] Y. Yang and D. Li, Separating Data and Control: Support for Adaptable Consistency Protocols in Collaborative Systems, ACM CSCW 2004, Chicago, Illinois, Nov. 2004, pp. 11-20.

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