IDEA: An Infrastructure for Detection-based Adaptive Consistency Control in 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

In Internet-scale distributed systems, replication-based scheme has been widely deployed to increase the availability and efficiency of services. Hence, consistency maintenance among replicas becomes an important research issue because poor consistency results in poor QoS or even monetary loss. Recent research in this area focuses on enforcing a certain consistency level, instead of perfect consistency, to strike a balance between consistency guarantee and system's scalability.

In this paper, we argue that, besides balancing consistency and scalability, it is equally, if not more, important to achieve adaptability of consistency maintenance. I.e., the system adjusts its consistency level on the fly to suit applications' ongoing need. This paper then presents the design, implementation, and evaluation of IDEA (an Infrastructure for DEtection-based Adaptive consistency control), which adaptively controls consistency in replicated services by utilizing an inconsistency detection framework that detects inconsistency among nodes in a timely manner. Besides, IDEA achieves high performance of inconsistency resolution in terms of resolution delay.

Through two emulated distribution application on Planet-Lab, IDEA is evaluated from two aspects: its adaptive interface and its performance of inconsistency resolution. According the experimentation, IDEA achieves adaptability by adjusting the consistency level according to users' preference on-demand. As for performance, IDEA achieves low inconsistency resolution delay and communication cost.

1. Introduction

Replicating data and services is an attractive strategy to increase availability and performance in distributed systems; in an Internet-scale system, such as large-scale Grid, replication-based schemes may indeed be the only way to provide continuous service and prevent data loss in the presence of unreliable Internet connections [4, 26]. For this reason, replication-based systems become more and more popular. Consequently, the interest in consistency maintenance has also been revived because poor consistency in replication-based systems results in poor QoS or even monetary loss (in e-business applications). Consistency in this context measures the difference among snapshots of the application's status (such as the airline ticket booking record) in different replicas. Simply put, the smaller the difference, the higher the consistency level is.

Realizing that a large collection of applications, such as e-business, are willing to sacrifice a certain degree of consistency in order to scale their services [26], recent research has concentrated on striking a balance between consistency guarantee and system's scalability by enforcing a certain level of, rather than perfect consistency. TACT, for example, explores the continuum between strong and optimistic consistency and proposes a framework to limit inconsistency levels among different replicas according to the applications' tolerance to inconsistency [26]. Chang et. al. proposed an information updating framework in which different users choose different consistency levels and the system updates the data based on that information [3].

In this paper, we argue that it is equally, if not more, important to achieve *adaptability* of the consistency maintenance. Adaptability has two meanings here. First, the system should be able to adjust its consistency level on the fly, as opposed to a predefined consistency level. This is important because multiple applications with different consistency requirements can run simultaneously in a modern distributed computer system [11] and even one application's consistency requirement can change from time to time as elaborated below.

- **A system may run multiple applications with different requirements of consistency**. In this scenario, a predefined consistency level does not fit all applications. While it is possible to deploy multiple consistency protocols, it will definitely complicate the system design and drag down system's performance due to the cost associated with operating each consistency protocol.
- **For an application, the requirement of consistency may change from time to time**. Take a virtual white board, in which participants draw on a virtual white board to communicate and collaborate, as an example. In this scenario, a participant may have less consistency requirement in the first several minutes when the discussion is about the background, but can have stronger consistency requirement when an important topic arises. In this scenario, a predefined consistency level does not reflect participants' changing requirements of consistency over time.

Second, the end users should have the control on how to adjust the consistency level (or requirement) on the fly. That is, the users first give a hint about what kind of consistency level (or requirement) they prefer and then adjust that preference when the need arises. The rationale behind this is that, although the users themselves may know what they want, they may not be good at expressing it in concrete and/or quantitative terms. Instead, they know whether a given consistency level is enough or not only when they see it.

While previous work by other researchers has attempted to address these two issues, none of them has solved them completely. For example, TACT [26] proposes a framework to let servers adjust the total consistency level for applications. Also, Chang et. al.'s work [3], which is specially developed for online conference applications, the users specify their desired consistency level before the system runs. Unfortunately, these frameworks do not have interactions with end users for them to specify the desired consistency level once the system starts running.

Beyond the adaptive interface, it is equally important that the consistency maintenance achieve high performance. That is, to find inconsistencies and when necessary to resolve them in a timely manner. This is crucial because slower detection and resolution can lead to poor QoS.

To this end, we present IDEA (an Infrastructure for DEtection-based Adaptive consistency control) that achieves both the adaptability and high-performance goals. To achieve adaptability, IDEA adjusts the consistency level on the fly through interaction with users. Upon the detection of inconsistencies, IDEA resolves them if the current consistency level doesn't satisfy applications' requirement; otherwise, IDEA will not resolve the inconsistencies except when the system is lightly loaded. The advantages of this approach are two folds: it can adjust the consistency level on the fly by resolving inconsistencies on demand; and more importantly, it gives the users the ability to control their perceived consistency level. It is worth mentioning that, because higher consistency level means lower response time, we do not expect users to abuse the system by overstating their consistency requirement because that will ultimately hurt them (lower response time).

To achieve high performance, IDEA utilizes an efficient Inconsistency Detection mechanism proposed in [14, 15] by the authors. Our previous work has shown that the detection can be done in a timely manner, not least because it divides the system nodes into two layers (top/bottom layers) and is able to capture the majority of inconsistencies in a relatively small top layer that includes the most active writers. As shown in the evaluation section, this ability to capture most inconsistencies in a small top layer is also crucial to guarantee the efficiency of the resolution.

To validate the design, we have implemented an IDEA prototype on Planet-Lab [20] and emulated two distributed applications, a distributed white board system and an airline ticket booking system, on top of IDEA. Collectively, they have shown that IDEA has achieved the design goal of adaptability and efficient inconsistency resolution (with small resolution delay and minimal communication cost).

This paper hence has made two contributions. First, we point out the importance of adaptability in consistency maintenance and present a new protocol IDEA to provide this adaptability. Second, we validate and evaluate IDEA by deploying a prototype on Planet-Lab. Results demonstrate that IDEA achieves high performance in inconsistency resolution.

The rest of the paper is organized as follows. Section 2 presents an overview of IDEA and Section 3 introduces the targeted applications. Section 4 and Section 5 present the design of IDEA and how IDEA can be applied to its targeted application. In Section 6, IDEA is evaluated through the emulation of two real applications on Planet-Lab, respectively. Related work is described in section 7. Finally, Section 8 concludes this paper and discusses future work.

2. The Overview of IDEA

IDEA is motivated by the observation that conventional consistency control approaches are in-flexible because they deploy pre-defined consistency protocols before systems start to run, which may not be appropriate for an Internet-scale distributed system where increased complexity calls for a more flexible approach. We propose a new framework called IDEA that, instead of enforcing a predefined consistency level, detects inconsistencies when they arise and subsequently resolves them based on applications' consistency requirements that may be hinted by users or derived dynamically from applications' semantics.

IDEA is assumed to work with a general distributed file system that handles the ordinary read/write operations. The general distributed file system is assumed to ensure the correctness of read/write functionalities, while IDEA detects inconsistencies among nodes and resolves them based on applications' changing requirements. That is, IDEA provides consistency control to this general file system.

Figure 1 illustrates the vision of IDEA. IDEA is deployed in the middleware level and applications on different nodes consult IDEA when they access files. Upon a request, IDEA retrieves a copy of the file from the underlying replication-based system and returns it to the application. At the same time, IDEA derives a consistency level for the returned replica. Then IDEA checks whether the inconsistency level is acceptable based on either users' predefined tolerance levels or the interaction with users in real time. If the inconsistency level is acceptable, IDEA does nothing; otherwise, IDEA will resolve this inconsistency upon the request from the user. As discussed in Section 3.1, users can communicate with IDEA about why the current consistency level is not sufficient and IDEA will learn from this to prevent annoying users again.

More specifically, upon initiating an application, users have the option to predefine or hint on their acceptable consistency levels. Or they could just respond to IDEA interactively. If there is an initial hint level, we denote it as L_1. Upon receiving the response, IDEA will not invoke the inconsistency resolution module unless the consistency level is below L_1. When a user is not satisfied with the result, IDEA will increase the consistency level by Δ. $L_1 + \Delta$ will then become the new desired consistency level for the user and IDEA will keep the application's consistency above this new level to avoid annoying the user again in the future. This way, IDEA makes the consistency control adaptive and gives users great flexibility to adjust consistency level themselves.

Comparing with conventional consistency control protocols, the benefit of detection-based IDEA lies in the following tradeoff: it achieves faster detection and resolution (thus stronger consistency guarantee) than that of optimistic consistency control [8, 23, 24], the de facto consistency protocol in large distributed systems, with a slightly higher cost; its overhead is much smaller than other protocols, such as strong consistency [1], with slower detection speed (thus relaxed consistency guarantee). Conceptually, the tradeoff is depicted in Figure 2. In fact, using relaxed consistency to trade for lower overhead is a common practice, such as that used in web caching [7].

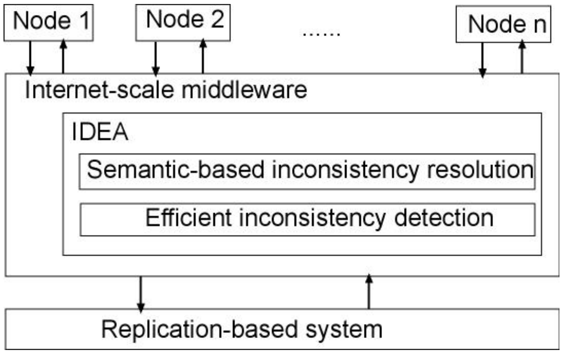

Node 1
Node 2
......
Node n
Internet-scale middleware
IDEA
Semantic-based inconsistency resolution
Efficient inconsistency detection
Replication-based system

Figure 1: The overview of IDEA

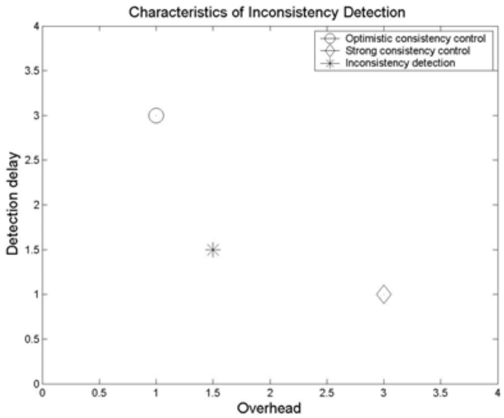


Figure 2: The trade-off of inconsistency detection-based IDEA

3. Targeted Applications

IDEA is designed to support a wide range of distributed, replication-based applications that are willing to trade certain consistency requirement for the ability to scale to an Internet-scale distributed system, particularly distributed online collaboration applications.

Previous research has indicated that there are two types of distributed online collaborations: synchronous collaboration in which the participants appear online at the same time and asynchronous collaboration in which the participants do not necessarily appear online at the same time [2].

In this section, we list two representative applications—one is synchronous collaboration and the other is asynchronous collaboration—and discuss their working flow. How consistency levels can be measured and how adaptability is achieved through IDEA for both applications will be discussed in Section 5 after presenting the design of IDEA is presented in Section 4.

3.1. Distributed white board system

A distributed white board system allows participants to draw or write on the same virtual white board so that these participants can interact and collaborate with one another while working on a single project or task. Because all participants usually appear online at the same time, this is a synchronous application. We assume that, in a distributed white board system, each user/participant has a white board system locally. Thus, due to network delays, the message a user reads may not be the most up-to-date information on other users' view (or vice versa), which results in inconsistency.

3.2. Airline ticket booking system

An airline ticket booking system is an example of e-business applications. Because not all participants are necessarily to appear online at the same time, this is an asynchronous application. In this system, we assume the existence of several booking servers, which are distributed in a wide area environment. To improve the efficiency of booking and avoid underselling, each server tracks its booking record independently. However, this may cause inconsistency—one server does not necessarily know the booking record of other servers in a timely manner—and hence overselling. Certainly, both underselling and overselling will hurt the company economically. From the company's point of view, there is a clear trade-off between the efficiency of booking—to avoid underselling—and the chance of overselling. Essentially, overselling is fine as long as the amount is within a certain range, which can be treated as a cost of avoiding underselling.

4. The Design of IDEA

There are two important features of IDEA: first, its ability to adaptively resolve the detected inconsistency based on applications' changing requirements and users' preference; and second, the high performance of both inconsistency detection and resolution in terms of delay.

In this section, we first present a two-layer (top/bottom layer) infrastructure adopted by IDEA which is essential to both fast inconsistency detection and resolution by capturing the majority of inconsistencies in the top layer. The workflow of the IDEA protocol is then presented. After that, we discuss an efficient inconsistency detection mechanism that provides IDEA a powerful API, *detect (update),* which, given an update, will return "success" when there is no inconsistency or "fail" when there is conflict (thus inconsistency) detected.

However, this detection API does not quantitatively measure how inconsistent a conflict is and thus cannot tell the system whether a detected inconsistency is acceptable or not. To solve this problem, IDEA extends the original detection messages and uses a single formula to quantify consistency level based on the information provided by the return value of the detection API. This formula is applicable to a variety of applications and will be presented.

In terms of inconsistency resolution, we discuss two mechanisms—background and active resolution—that serve different purposes: background resolution improves consistency in the system from time to time and the latter is triggered when a user explicitly requests a resolution operation.

Different applications naturally have different meanings of adaptability and that issue is discussed after. Finally, we discuss the interface provided by IDEA for application developers to configure IDEA.

4.1. The two-layer infrastructure

IDEA utilizes a two-layer (top/bottom layer) infrastructure to detect and resolve inconsistency for each shared file or object. In the case of a white board application, for example, the shared object is the virtual white board itself. This two layer infrastructure is first presented in [14, 15] and the top layer for a given file, also referred to as a "temperature overlay", is constructed by leveraging the RanSub protocol [9] to include nodes that update this file sufficiently frequently and/or recently (hence the term updating "temperature"). The remaining nodes form the bottom layer.

Comparing with a flat architecture in which all the nodes are in the same layer, there are two advantages of utilizing this two-layer architecture. First, it is unlikely that all the nodes in a large network will be interested in the same file at the same time, thus it is possible to capture all the active writers with a much smaller subset of the whole network to form a top layer. Second, due to the top-layer's relatively small size, it is much faster to detect and resolve inconsistency among its members than the whole network. In the background, however, IDEA always visits the bottom layer, which covers all the nodes in the network, to catch the possible, although somewhat unlikely, missed detections or resolutions by the top layer.

We also need to mention that, because consistency is associated with a single file, the concept of top/bottom layer is also associated with a given shared file—different files may have different top layers—and different top layers do not interfere with one another. For example, if a user joins multiple virtual white boards, each white board is treated separately and independently.

4.2. Overview of the IDEA protocol

An overview of the IDEA protocol is depicted in Figure 3. From the figure, we can see that the IDEA protocol is triggered by two operations: write and certain read operations. The write operation, such as issuing an update in a white board, triggers the IDEA protocol because it is essentially an update operation that will surely cause inconsistency among replicas. For read operations, IDEA is triggered when a reader tries to retrieve a new file (such as a new snapshot of a white board) because, in this case, the user needs to make sure that the file retrieved is sufficiently consistent for the user's purpose. For other reads, IDEA is triggered according to the context: if the file is locally updated frequently, the read will not trigger IDEA; if the file hasn't been locally updated for a long time and the user is afraid that the file may be inconsistent, IDEA can be triggered.

After IDEA is triggered, it will use a detection-based mechanism to check the inconsistency level, represented by a single percentage number, such as 90%. Here, we assume that this number can be obtained appropriately either from interpreting users' view of QoS or by (analysis of) the nature of an application; the mechanism to properly quantify this parameter will be discussed in the Section 4.4. After the inconsistency level is returned, IDEA checks whether the inconsistency level is acceptable based on either users' predefined tolerance levels or the interaction with users real time. If the inconsistency level is acceptable, IDEA does nothing; otherwise, IDEA will resolve this inconsistency upon the request from the user. As discussed in Section 3.1, users can communicate with IDEA about why the current consistency level is not sufficient and IDEA will learn from this to prevent annoying users again.

For efficient inconsistency detection, the inconsistency level is initially detected only among the top-layer nodes to improve the response time. Hence, this value may not be accurate because the nodes in the bottom layer can cause inconsistencies too, albeit rather infrequently. To cope with this issue, we deploy a rollback mechanism. More specifically, IDEA lets users continue their work when they indicate that the initially returned consistency level (from top layer nodes) is acceptable. In the background, however, IDEA continues to detect inconsistency in the bottom layer and returns a new value. If the new value is sufficiently close to the previous one obtained from the top layer, IDEA keeps silent; otherwise, IDEA alerts the user about the discrepancy and resolves the inconsistency if the users so demand.

4.3. Efficient inconsistency detection

In [14, 15], we presented the design and evaluation of an efficient, low cost inconsistency detection mechanism, which is responsible for the inconsistency detection module in IDEA. The basic idea of this mechanism is to rely on the top layer of the two-layer infrastructure for timely detection. In the bottom layer, it uses gossip-based protocol [6] to check in the background any missed inconsistency by the top-layer. Essentially, the detection module provides a powerful API (Application Programming Interface) to IDEA: *detect (update)*. Given an update, this operation will return "success" when there is no inconsistency or "fail" when there is conflict (thus inconsistency) detected. Theoretically speaking, this detection mechanism, as a rather independent component in IDEA, can be used by other consistency control mechanism (other than IDEA) as well.

The conflict of two or more updates, and hence the inconsistencies of different replicas, is detected through exchanging version vectors [19] among replicas. A version vector tracks the number of times a file is updated by a certain user and uses that to detect conflict. 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 earlier in time (or more obsolete) than that presented by version vector (*A*:4 *B*:7). With version vector, two replicas are inconsistent if their version vectors are different. As measured before, with this two-layer inconsistency detection framework, most inconsistencies can be caught in the top layer with a very high probability (more than 95% in a variety of scenarios) [16] without much maintenance cost and, most importantly, in a timely manner [14, 15].

However, this detection API does not quantitatively measure how inconsistent a conflict is and thus cannot tell the system whether a detected inconsistency is acceptable or not. To solve this problem, IDEA uses a

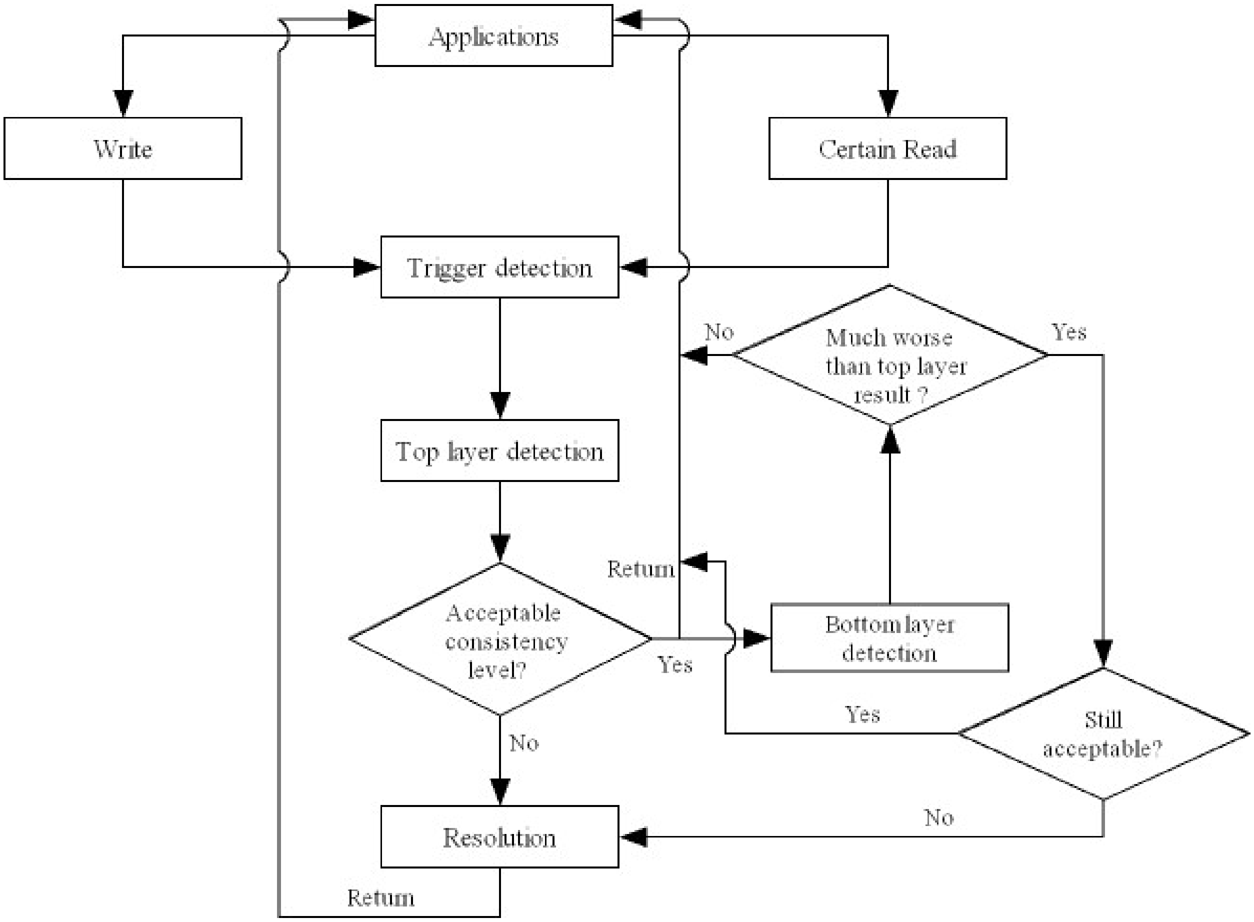


Figure 3: Overview of the IDEA protocol

single formula to quantify consistency level based on the information provided by the return value of the detection API. This formula is applicable to a variety of applications and will be presented in the following subsection.

4.4. Quantifications of consistency level

Basically, the inconsistencies among nodes is quantitatively measured by a metric adopted from the TACT measurement [26] where a <*numerical error, order error, staleness*> triple is used to indicate the inconsistency level, as developed by TACT. Now we use an example to illustrate how this is achieved and how it can be applied to a variety of applications.

4.4.1. A scenario

First of all, we assume two replicas (*a* and *b*) and two active users/writers (*A* and *B*), as depicted in Figure 4(a). User *A* resides in replica *a* and user *B* in replica *b*.

Now we let each of them have some updating activities and, by the end of these activities, their version vectors are shown in Figure 4(b). Because the version vector here is actually an extended version of original version vector that only tells us the number of updates from each writers (in the form of <*A*:2 *B*:0>, for example), we now take a closer look of replica *a*'s version vector to show the difference.

First, the extended version vector has time stamps associated with each update, such as <*A*:2(1, 2)> that means the two updates from A happens in time point of 1 and 2, respectively. To make the timestamp comparable among different sites, we assume that the gap among time clocks of participating nodes in the system is within seconds, which is small enough to neglect in a globally distributed system. Practically, there are two mechanisms to achieve this precision. First, the system can run a globally synchronizing clock algorithm, such as that proposed in [12]. If it is too troublesome to run such a clock synchronizing algorithm, another choice is to let each node to keep their time accurate by synchronizing with a time sever using NTP (Network Time Protocol) [17], which can be easily achieved in both Windows or Linux operating systems by enabling the corresponding modules [17].

Second, as we can see, there is a numerical value in square brackets (the <[5]> column in the extended version vector). We use this value to represent some critical meta-data of applications to characterize the difference of different versions, as explained below. In the case of distributed white board, for example, the meta-data can be the sum of the ASCII value of the last several updates; in an airline ticket booking, it can be the total sale price. These meta-data can give a quick sense of what the effect of the conflict would be, which is easier to understand in the airline booking example—the data tells the total sale that has significant business value.

Third and finally, the <*numerical error, order error, staleness*> triple is attached at the end to conclude the extended version vector. The numerical error is deduced by comparing the value of critical data; the order error counts for the difference between number of updates (an example of calculation will be given shortly after); and staleness error is calculated from the time stamps (an example will be given later too). In Figure 4(b), because replica *a* is not aware of any conflict in the system, all the errors are set to zero.

Figure 5 visually depicts the difference between the original version vector and the extended one used in IDEA. Because IDEA uses this extended version vector instead of the original one, we will simply use the term "version vector" to denote the extended version vector for brevity in the rest of this paper.

Suppose now that the detection process is started, let's assume that the version vector of replica is traversed from replica *b* to replica *a*, as shown in Figure 4(c).

Then, after comparing with that of replica *a*, the modified version vector of replica *a* and *b* will be changed and the new ones are shown in Figure 4(d). The calculation is carried out as follows.

First, IDEA derives a *reference consistent state* which is the state chosen by IDEA that is regarded as the basis for consistency level calculation. As we will show later, there are several ways to derive the reference consistent state. For now, let's assume that the replica with higher ID value becomes the reference consistent state, which means that, if version vector from *a* and that from *b* conflict with each other, IDEA will choose *b* (*b* > *a*) as the reference consistent state and then use it to calculate *a* and *b*'s consistency levels.

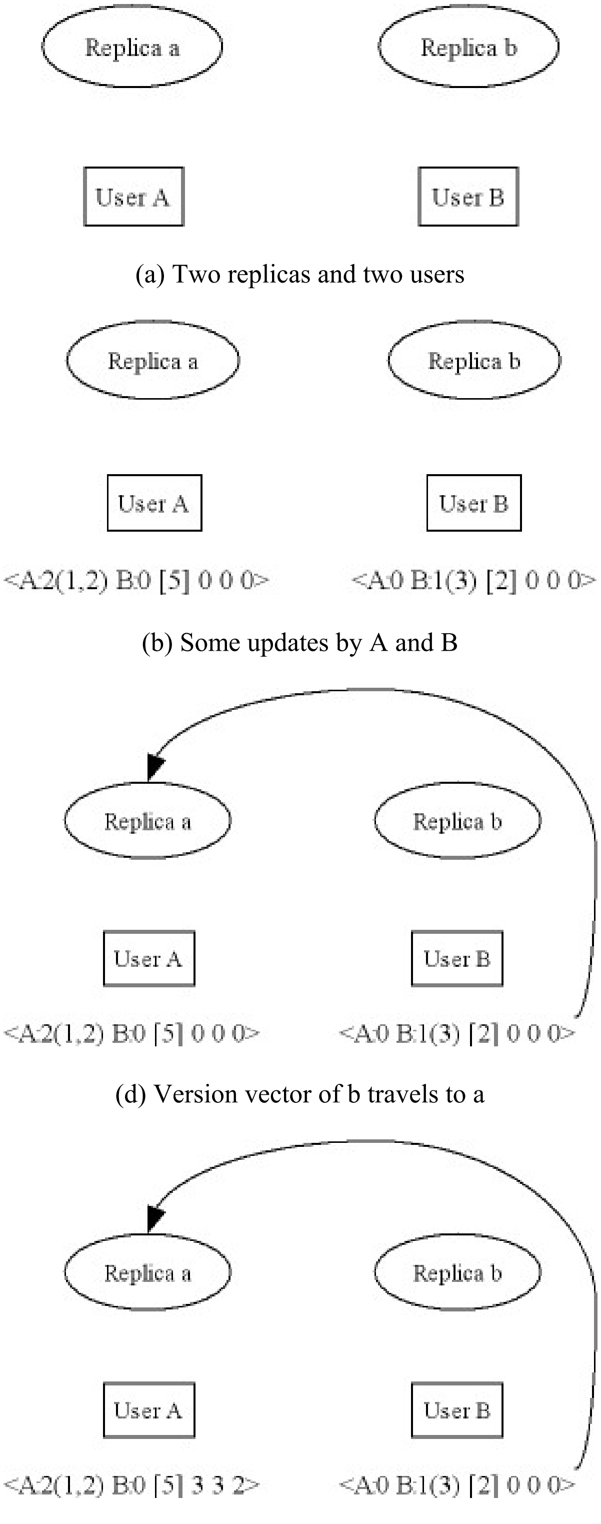
But first, we need to use *b* (the reference consistent state) to calculate <*numerical error, order error, staleness*> triple that will be used to derives a numerical consistency level as follows: the replica *a*'s final value of its meta data has a gap of 3 with that of *b* (the reference one), so the numerical error is 3; replica *a* misses one update and has two extra ones, so the order error is 3 too; finally, the last time point when a is consistent is time 1, and that has a gap of 2 with the most recent update at *b* (time 3), so the staleness is 2. Generally, staleness of one replica is defined as the time difference between the most recent update in the *reference consistent state* and the last time point when it is consistent.

Then, IDEA calculates the consistency level as follows. First, IDEA predefines a maximum value for each member of the triple. For example, if in practice, the order error is very unlikely to be larger than 10, then the maximum value for order error can be set as 10. Then IDEA gets input from users and sets weight for the three members respectively. For example, if users treat the three members equally, their weight will be equal and 33.3%. Then the consistency level can be quantified as in Formula 1:

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

The calculation of consistency level of version vector of replica *a* and *b* according to Formula 1are presented in Figure 4 (e) by assuming that the maximum error for all three metrics are 10.

One may wonder that, if the consistency state is easy to be figured out, why don't we resolve it immediately? And that is because of two things: communication overhead for the system and its potential to block updating operations for users. First, if we resolve every conflict, the huge communication cost (copying remote updates to local sites) will be huge. For this reason, we prefer to defer this resolution whenever possible. Second, once we decide to resolve the inconsistency, all future updates will be blocked until the resolution is finished (to prevent invalid updates that based on an inconsistent copy). Thus, to improve system's responsiveness, we prefer not to run the resolution unless the inconsistency is unacceptable (or based on periodical running).

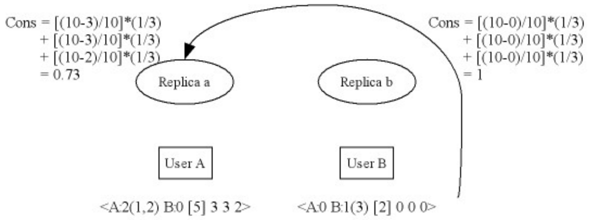


(a) Two replicas and two users

(b) Some updates by A and B

(d) Version vector of b travels to a

(d) Comparing two version vectors



(e) Calculate consistency level for replica a and b

Figure 4: An example of consistency level quantification

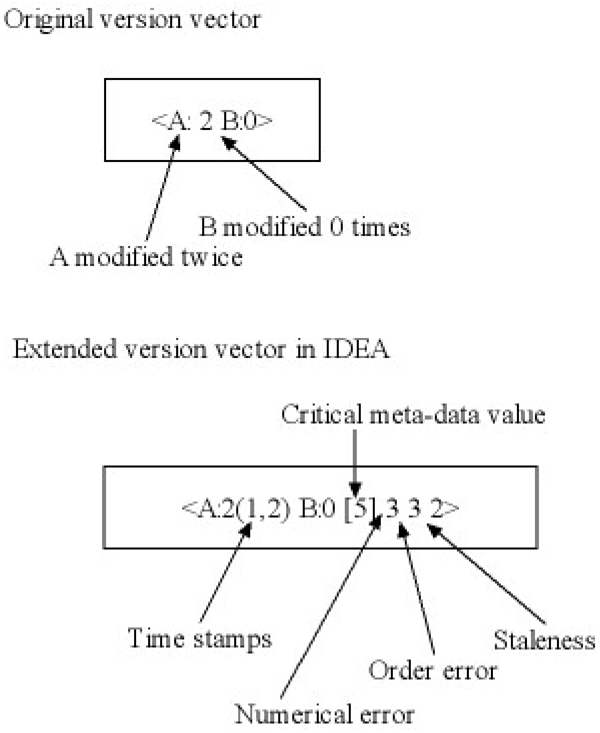


Figure 5: Comparison between original version vector and the extended version vector in IDEA

4.4.2. Accuracy of the calculation

We have to admit that this calculation of consistency level may not be 100% accurate because it does not include the replicas in the bottom layer. Nonetheless, as explained in the IDEA protocol, inconsistency detection will be carried out in the bottom layer after that in the top layer is done. After a certain period of time, the result of the bottom layer will be returned. Then, if the new result is sufficiently close to the one returned from the top layer (e.g., 78%

vs. 80%), the top-layer result remains intact; if the results from the two layers are not close enough, the top-layer result needs to be modified and the operations during this period should be rolled back if the new consistency level is not acceptable according to the user's preference that IDEA has learn so far.

There are two things that we need to point out about this potential rollback operation. First, to void annoying users, IDEA will handle the rollback in the background and return the result to the users afterwards. Second, the impact of rollback should not be overstated. According to our previous analysis [16], it is very rare (less than 5% in a variety of scenarios) that the top layer will leave an inconsistency undetected. Thus, we treat the rollback mechanism as a back-up and do not expect it to slow down the performance of IDEA.

Due to the potentially large number of nodes in the bottom layer, which covers all nodes in the system, a critical question is how long the detection in the bottom layer would take. Intuitively, the longer the delay, the larger number of states will potentially need to be rolled back, which causes more overhead and frustrates users more. Currently, we use TTL (Time to Live) to control the traversal of the bottom-layer detection messages, thus bound the delay. Clearly, this is a trade-off between accuracy and responsiveness. We believe that, in an Internet-scale system like Gird, this trade-off is reasonable and necessary. Other mechanisms to tackle this problem certainly exist and we plan to investigate this issue further in the future.

4.5. Inconsistency resolution

Up to this point, we have discussed how consistency level can be derived, and now move on to discuss how an inconsistency can be resolved when needed. We discuss the inconsistency resolution process in two steps. First, we discuss the mechanisms to resolve an inconsistency. Second, we discuss two ways to initiate the resolution: background resolution and active resolution.

We mentioned in the beginning of this section that one feature of IDEA is that it can resolve an inconsistency in a timely manner. This claim holds for both background and active resolution, and it originates from the relatively small size of top layer. This claim will be evaluated in Section 6.

4.5.1. Resolution mechanisms

As mentioned earlier in Section 4.3, inconsistencies are detected by comparing version vectors. Given two version vectors u and v from two replicas, the replicas are inconsistent if their version vectors are different. Further, as defined in [19], two vectors are comparable if and only if $u < v$, $u = v$ or $u > v$. If not, they are not comparable with each other. For example, (A:5, B:3) is not comparable with (A:3, B:6).

Now, if the two version vectors are comparable, the resolution is relatively easy: just let the smaller one learn from the larger one.

However, if the two vectors are not comparable, resolving the inconsistency between them is not that easy. For example, if two sentences are written, how can a system determine which one should come first? In practice, a variety of options can be adopted. Here we list three possible policies, as well as their target applications. These policies are briefly described for illustration purposes only and are not meant to fully and solely rely on for inconsistency resolution. In practice, other policies are also possible.

- **Invalidate both**. In this case, the two conflicting versions are both invalidated and they will roll back to a previous consistent version. In a distributed white board, for example, two simultaneous updates at the same spot can be both cleared to prevent ambiguity and ensure fairness (so that no one is more important than the other).
- **User ID based**. To ensure fairness, each node can be assigned a randomly chosen ID, such as the hash value of their IP address via MD5, which is commonly used in Peer-to-Peer systems. When a conflict arises, the user with the larger ID wins. This approach can be used in both a distributed white board and an airline ticket booking system where certain progress is preferred (if both updates to be invalidated, no progress can be made in a white-board-based discussion and no ticket will be sold in an airline ticket booking system). In this case, it is desirable to treat its members equally (ensured by using randomized user IDs).
- **Priority based**. In this policy, different levels of priorities are assigned to users. For example, the supervisor of a company will have a higher priority than ordinary workers. When the conflict arises, the version created by higher priority user wins. In a distributed white board, a supervisor can have a higher priority and other employees; in an airline ticket booking system, giving preferred customers, such as those who have traveled the most with this airline, higher priority is a sensible choice.

4.5.2. Background and active resolution

Here we discuss two inconsistency resolution mechanisms—background and active resolution—that serve different purposes: background resolution improves consistency in the system from time to time and the latter is triggered when a user explicitly requests a resolution operation.

The necessity of the two mechanisms is explained as follows. Active resolution is needed because we expect that end users will explicitly request an inconsistency to be resolved when it becomes unacceptable. However, if we only resolve inconsistencies when they become unacceptable, it will unavoidably annoy users from time to time: once a while, the system's consistency will become really bad. Even IDEA can avoid annoying users by resolve the inconsistency right before it becomes unacceptable, it still does not prevent the consistency from dropping continuously. So, IDEA also periodically resolves inconsistency in the system to improve the consistency level, which is called background resolution.

Now we illustrate the process of background resolution, followed by that of active resolution.

First of all, the background resolution process is started by IDEA periodically to improve the consistency among replicas on a regular and continuous basis without users' intervention. Once it is started, one replica (chosen by IDEA) in the top layer for a certain file acts as the initiator and collects all the version information of the members in the top layer by sequentially visiting them and then determines a consistency replica, by following the resolution polices discussed in the Section 4.5.1. It then informs all the members of information about the new consistent replica and the members will update their copies by acquiring any missing updates to reflect this change.

Active resolution, unlike the background consistency resolution, is triggered when a user explicitly requests an inconsistency to be resolved. That is, active consistency resolution is a backup of the background consistency resolution and only kicks in when the periodical background consistency resolution still cannot satisfy some users' needs.

When active consistency resolution is triggered, the nearest replica (including the user's local copy) takes the responsibility of initiating inconsistency resolution. More specifically, we use a two-phase protocol. First, the initiator sends a request to all the members in the top layer in parallel to call for attention to the upcoming resolution process. Second, only after it gets all positive acknowledgement (i.e., no one else is initiating the same process), it starts the resolution procedures; if someone else has already sent the same request out, they will back-off and retry after a random amount of time. Here, the back-off process is used to suppress redundant resolution process to save bandwidth: in the retry period, if one receives another's notice before it tries, it will simply cancel its own resolution process.

When this first phase succeeds, the resolution process is the same with that of the background resolution process.

4.6. Adaptive consistency control

Different applications naturally have different meanings of adaptability and here we discuss how adaptive consistency control works from an application's point of view. That is, how IDEA caters to application semantics in practice. Here we list three possible application types that can benefit from IDEA and explain how IDEA works for them based on their semantics, respectively. Our hope is that, through these three examples, we can give practitioners some hints on and insight into applying IDEA in a real environment.

- **On-demand**. In this scheme, users explicitly request consistency resolution when they are not satisfied with the current consistency level. Otherwise, they depend on the background consistency resolution. One possible application is the distributed white board system in which each newly posted message will contain its consistency level generated by IDEA. Then, when the users feel that the consistency level is unacceptable, they tell IDEA to adjust the weights of the three metrics, or to keep the same weights but boosting the overall consistency, or to do both.
- **Hint-based**. This scheme asks users to give hints about their approximate consistency requirements. When a consistency level is derived, IDEA only triggers the active consistency control when the consistency level drops below that hinted by the user. In this mode, users in a distributed white board system indicate their tolerance levels and IDEA will keep the consistency level above that. However, if users later feel that the pre-set hint level is not high enough, they can communicate with IDEA and IDEA will change the hint level to a higher one.
- **Fully automatic**. This scheme improves consistency with best effort, by adjusting the

frequency of background resolution, under certain constrains. Possible applications include e-business applications such as an airline ticket booking system. For example, if the consistency overhead is deemed not to be over 20% of available system capacity (to save enough network bandwidth for customers' requests), then, based on the system's current total available capacity, the frequency of background resolution needs to be adjusted accordingly. At the same time, as explained in Section 3.2 earlier, such a system should also not cause either underselling or overselling, which has undesirable economical consequences. Thus, IDEA first needs to learn these two bounds of the frequency of background resolution that causes underselling and overselling. When IDEA adjusts the frequency of background resolution based on the current system load (for example, to consume less than 20% of the total available bandwidth), the adjustment will obey the two bounds: it will not be above the higher bound in order to prevent underselling and not be under the lower bound to prevent overselling.

4.7. IDEA APIs

IDEA has two interfaces, one is to the developers and the other is to the end users, and they serve different purposes. On the one hand, the develop interface is to let them use IDEA to serve their particular applications, be it distributed white board or others. On the other hand, the end user interface lets users to interact with IDEA during the runtime of IDEA. For services other than consistency, end users are supposed to interact with applications directly. The difference between the two interfaces is illustrated in Figure 6 as follows.

Because we have discussed IDEA's interface to end user extensively in previous sections, we devote this sub-section to discuss IDEA's interface to application developers. This interface, in the form of APIs (Application Programming Interface), is for application developers to interact with IDEA. Currently supported APIs are listed in Table 1 and we explain how they are used as follows.

- **Cast applications to IDEA's consistency metric**. While we use the triple consistency<*numerical error, order error, staleness*> as a generic form to derive a consistency level, the system administrators need to explicitly define the meaning of the three metrics in the application's context. For example, he or she needs to define the granularity of application's objects, to define what kind of error is considered, etc. This is to cast applications to IDEA infrastructure. This is done through the set_consistency_metric function.

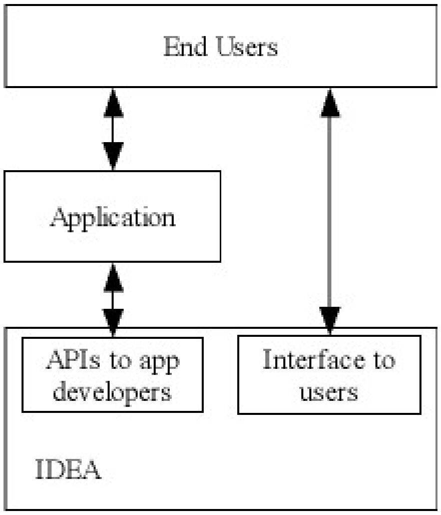


Figure 6: Two interfaces of IDEA

Functions
set_consistency_metric (a, b, c): cast applications to IDEA infrastructure
set_weight (a, b, c): set weights for the three metrics for calculating consistency level.
set_resolution (r): set the resolution strategy
set_hint (h): set the initial hint level
demand_active_resolution ( ): call for active inconsistency resolution
set_background_freq (f): set the frequency for background inconsistency resolution

Table 1: APIs for configuring IDEA

- **Setting weights of metrics**. This is done through the *set_weight* function. To derive a single value, the system administrators need to define the weight of each metric in the quantification using a triple weight<*numerical error, order error, staleness*>. For example, to treat each metric equally, they can indicate weight<0.33, 0.33, 0.33>. If one metric, such as order error, is not suitable for one particular application, it can be marked by indicating its weight as 0, such as weight<0.4, 0, 0.6> in this case.

- **Setting resolution strategy**. This is done through the *set_resolution* function. The parameter is a single integer number that indicates the preferred inconsistency resolution policy. Suppose there are four policies, as explained in Section 4.5, then the possible value will be 1, 2, 3, or 4.
- **Setting hint for hint-based applications**. This is done through the *set_hint* function and is only used in hint-based applications, one type of applications discussed in Section 4.6. A valid parameter should be between 0 and 1, such as 0.85. In particular, by setting this value to 0, the administrator indicates that this is not a hint-based system; setting this value to 1 means that the user does not tolerate any inconsistency.
- **Demand active inconsistency resolution**. Applications use function *demand_active_resolution* to explicitly ask IDEA to actively resolve the conflicts through a resolution strategy defined through "Resolution strategy" API.
- **Setting frequency for background resolution**. Applications set the frequency for background resolution performed by IDEA through the function *set_background_freq*.

5. Apply IDEA to Applications

In this section, we discuss how consistency levels can be measured and how adaptability is achieved through IDEA for distributed white board system and airline ticket booking system.

5.1. Distributed white board system

As stated in the design of IDEA, IDEA uses the <*numerical error, order error, staleness*> triple to indicate the consistency level. In the case of a distributed white board system, numerical error denotes the gap of some meta data between two replicas (such as the sum of the ASCII value of the last several updates); order error measures the degree of the wrong sequence of updates that appear in one node and, in white board, this is the most confusing for users because these updates make sense only when they are read in order; finally, staleness represents the gap between now and the last time a replica is consistent.

It is worth mentioning that staleness is different from response time—a performance metric we will use later to evaluate IDEA. The key difference is that staleness denotes how long the replica has been in an inconsistent state, while response time is the transmission delay for a *requested consistent image* (*i.e.* content of a shared file/object) to arrive. So, even if staleness equals a long delay, the response time of IDEA as a whole can still be minimized because they evaluate different processes.

Given the triple value, the consistency level can be then quantified, as in the formula 1 in Section 4.4. By adjusting the weight given to each member of the triple, IDEA can reflect applications' different characteristics. For example, users in a white board scenario may prefer more order preservation (all messages appear in the same order at different nodes) than staleness, so IDEA will give more weight to order error, such as 0.7 to order error and 0.1 to staleness.

After discussing how to evaluate consistency level, now we briefly talk about how users can interact with IDEA to achieve adaptability.

First of all, with IDEA, the inconsistency among different sites can be detected and IDEA derives a consistency level for a given replica in a timely manner. Then IDEA checks whether the inconsistency level is acceptable based on either users' predefined tolerance levels or the interaction with users in real time. If the participant considers the current consistency level tolerable (for example, the order preservation is good enough), he or she needs not do anything. Otherwise (for example, the order preservation is bad and annoys him or her), he or she can explicitly ask the inconsistency to be resolved.

There are three ways users can communicate with IDEA about why the consistency is unacceptable: change the weight, boost overall consistency level without changing the weights, or do both. More concretely, the users change the weight when they feel frustrated about one particular metric, but not others. For example, they may feel that order preservation is fine but the staleness is too high. They can then ask an increase of the weight for staleness. Alternatively, they can simply ask IDEA to boost the overall consistency level if they are satisfied with the assignment of weights. Finally, the users can ask IDEA to do the two at the same time: first changing the weight assignment and then boosting the overall consistency level.

If users demand inconsistency resolution, IDEA will do so and return a consistent result afterwards. As explained in Section 2, during the same time, IDEA will also learn the new acceptable consistency level and try to avoid annoying users again by keeping the consistency level above this new one in the future.

Overall, by periodically detecting inconsistency with sufficient frequency behind the scene, but only resolving them when users demand, IDEA keeps the system running smoothly without interruption to the application. However, when the need arises, IDEA is able to bring the consistency level back to acceptable states in a timely manner as well as dynamically adapts the consistency measurements parameters to prevent annoying users again.

5.2. Airline ticket booking system

As in a white board scenario, the consistency level of an airline ticket booking system can be measured by the weighted sum of the triple values. In airline ticket online booking, however, order preservation may not be the sole focus because staleness and numerical can potentially affect profits too. In this scenario, order error means the wrong sequence of the booking order from users and that can cause conflicts when the order matters, such as assigning seats when clients purchase tickets; staleness denotes the delay of a booking record that appears on other nodes and it cause conflict too because a replica may decide the sale without the full knowledge; and numerical error can represents the gap of the system's overall sale price on different web server. Hence, the weights given to the three members of the triple should reflect this. For example, we can give the weight of 0.33 to each of them. As in the white board application, the weights can be dynamically adapted during runtime.

In terms of adaptability, IDEA may not directly interact with the application's clients because it is the booking servers that ultimately commit updates. However, it is difficult to decide the preference of each booking server because it is the overall system's performance that matters.

For this reason, IDEA runs a background inconsistency resolution protocol among the booking servers periodically to improve the consistency from time to time. Clearly, there is a tradeoff between the frequency of background inconsistency resolution and the overhead of consistency control: the more frequently the resolution protocol runs, the better consistency the system can achieve; at the same time, it will incur high overhead and increase the likelihood of underselling—the system is kind of locked when the inconsistency is being resolved.

In an e-business environment, neither underselling nor overselling is desirable. Thus the consistency level is not always the higher the better. Hence, the frequency of background resolution cannot be too high even if the system can sustain it. In practice, an ideal frequency can possibly be deduced or learned (e.g., through machine learning techniques) from a long period of running in the following manner. First, IDEA sets an initial frequency and adjusts it on the fly based on system's load. Second, when the frequency is too low (and the consistency level is low too) and causes overselling, IDEA will increase the frequency beyond the current level and keep the frequency above this one to avoid overselling; similarly, when the frequency is too high (and the consistency level is high too) and causes underselling, IDEA will decrease the frequency below the current level and keep the frequency under this one to avoid underselling. Overtime, IDEA will learn the two boundaries within which it can adjust the frequency.

6. Evaluation

To evaluate the adaptability and performance of IDEA, we implemented IDEA and two emulated real applications (a virtual white board and an online airline ticket booking application) that run on top of IDEA, and deployed them on the Planet-Lab [20].

The applications are emulated by following their operational sequences. In the case of a distributed white board application, we abstract the distributed white board as a set of objects that are replicated on each participating node. Then, we treat each update on the white board as a write operation on its local replica. Similarly, for an airline ticket booking application, each booking server has a replica of it and each update is considered as a write operation in its local replica. Due to the lack of available traces, we use a synthetic workload that assumes uniform distribution of the updating frequency for both applications. After updates are issued, IDEA works to maintain the overall consistency level of the virtual white board according to the protocol. Because our purpose of the experiments is to evaluate the performance and effectiveness of IDEA, we assume that these updates are all conflicting with one another (otherwise, IDEA needs not to care about them). While the two applications look similar in this abstract level, they differ in how the consistency is maintained: a participant in a distributed white board either gives a hint about their consistency requirement or interacts with IDEA on-demand; booking servers in an airline ticket booking application, however, can only depends on automatic consistency resolution whose frequency can be adjusted because, unlike participants in a white board, each booking server does not care about its view of consistency—instead, it is the overall consistency that affecting the business goal that matters.

We use three metrics—namely, delay, consistency level, and incurred overhead—to measure the performance of IDEA. Delay information is important because it determines the performance of IDEA.

Consistency level is also a metric because it controls the QoS perceived by participants. Finally, we evaluated the incurred communication overhead, measured in number of protocol messages, by IDEA to demonstrate its scalability (the lower the overhead, the more scalable IDEA is).

As mentioned earlier, we focus on two aspects of IDEA: its adaptive interface and its performance. To evaluate the adaptive interface, we use an emulated distributed white board application and let users interact with it in an on-demand fashion. To evaluate the performance, we first investigate the response time of consistency resolution in a distributed white board scenario and then evaluate the communication overhead in an emulated airline ticket booking system. As for correctly re-order conflicting updates, we simply choose the one with higher ID as the perfect image, one of three policies discussed in Section 4.6.

Also, in Section 4.4.2, we discussed the rollback mechanism that is triggered when the detection in the bottom layer returns an actual consistency value much worse than the one returned from the top layer. In this evaluation part, however, we do not consider the rollback mechanism for two reasons. First, according to our previous analysis [16], the possibility that the top layer fails to detect an inconsistency is indeed very small (less than 5% in a variety of scenarios and as small as 0.04% in certain cases). Second, this evaluation serves the purpose to validate the design of IDEA and the rollback mechanism is not essential for this purpose because the rollback mechanism uses TTL to control the detection delay in the bottom layer and we do not expect the rollback mechanism to be a performance bottleneck.

6.1. The adaptive interface of IDEA

Here we use a hint-based application to show the effectiveness of the adaptive interface of IDEA. In this application, each user indicates a certain tolerance level to the inconsistency level, which is the *hint*. The assumption is that, when the system's consistency level is above the hint level, the user is satisfied. Thus, IDEA only resolves inconsistency when the consistency level drops below the hint.

However, this scheme will cause the user to suffer from at least a short period of time during which the user is in an inconsistent state, an undesirable event. To cope with this, the user can set a hint level slightly above its real acceptable consistency level. In this way, IDEA starts to resolve any inconsistency early enough to keep the system's consistency level above the user's *real* hint level all the time. As shown in the following experimentation, IDEA can bring the system's consistency level back to a satisfactory level in a timely manner.

The experimentation is run over 40 Planet-Lab nodes, in which four of them are assumed to be concurrent writers of a given file. After warming up, the four writers form a top layer of four nodes that includes all of them. Because these 40 nodes span US and Canada, we believe it is representative of an Internet-scale distributed system. While a top layer of four nodes is not a large one, it is sufficient for our investigation purpose because they are carefully chosen so that they are far apart from each other. Also, based on data collected from this setup, we will later extrapolate the result to predict the performance of IDEA in a more dynamic system (with more simultaneous writers).

After that, the four nodes start to update the same file every 5 seconds during a 100- second period, which amounts to a total of 20 updates. This experiment is run with two different hint levels. First, we set a user's hint level to 95%, which allows IDEA to kick in when the user's consistency level is lower than 95%. Second, we set the user's hint level to 85%, where IDEA kicks in when consistency level is below 85%. The results are summarized in Figure 7(a) and Figure 7(b), respectively, in which the "view from the user" is the consistency level of the writer with the worst consistency and the "system average" is the average value of the consistency level of the four writers.

As shown in the two figures, the consistency level is improved right after IDEA kicks in, by evoking the active resolution scheme. In both scenarios, IDEA was able to bring the consistency level back to satisfactory states fairly quickly. Please note that IDEA actually brings the system's consistency level back to acceptable states in less than one second, as discussed in the previous section. The reason for why these two figures show that the consistency level is brought back to acceptable states after five seconds is because we sample the system's consistency level every five seconds in this experiment.

The lowest consistency levels for users in the two experiments are 94% and 84% respectively. Thus, if a user's real hint level is 94% or 84%, he/she can set the hint level to 95% or 85%, respectively, to avoid suffering from being in inconsistency states all together.

Then we combine the two settings by running the experimentation for 200 seconds. Same as above, the four writers update the same file every 5 seconds, which amounts to a total of 40 updates per writer. We initially set the users' hint levels to 95% and reset the hint levels to 90% after 100 seconds. The result is summarized in Figure 8. The achieved lowest consistency level for writers (even for the one with the worst consistency) in the experiment is about 95% in the first 100 seconds and 90% in the second 100 seconds.

Collectively, these experiments and results clearly show the feasibility and effectiveness of the IDEA's adaptive interface.

6.2. IDEA's response time

To evaluate the performance of IDEA's active consistency resolution scheme in terms of response time, we consider a simple distributed white board application in which four concurrent writers form the top layer. Because we treat distributed white board as an on-demand application, a node triggers active resolution when it feels that the consistency level is not satisfactory. We run the consistency resolution scheme four times, and each time we pick a different writer to initiate the request for active consistency resolution. We use the average of the four runs as the final result.

Table 2 shows the response time breakdown for the two phases involved in an active consistency resolution. As elaborated before, phase one is a call-for-attention and phase two resolves inconsistency among the top-layer nodes by visiting them sequentially.

The result shows that phase one is much shorter than phase two. This is due to two reasons. First, the operation in phase one is only a call-for-attention, thus there is little computing overhead involved; on the other side, phase two involves collecting replicas' information (such as comparing version vectors) and resolving the potential inconsistencies, which has higher communication as well as computation overhead. Second, the call-for-attention operations for different nodes are executed in parallel, which further improves its speed; for the second phase, though, it traverses all the top layer members sequentially to resolve the inconsistencies one by one. In this design, we choose to run the second phase sequentially because it simplifies the active writer's job—just need to communicate with one other active writer at a time. However, if performance is a concern, it is not difficult to exploit parallelism for the second phase (letting an active writer contact all the other active writers at once).

Now we use this result to estimate the scalability of active resolution as follows. Because phase one is

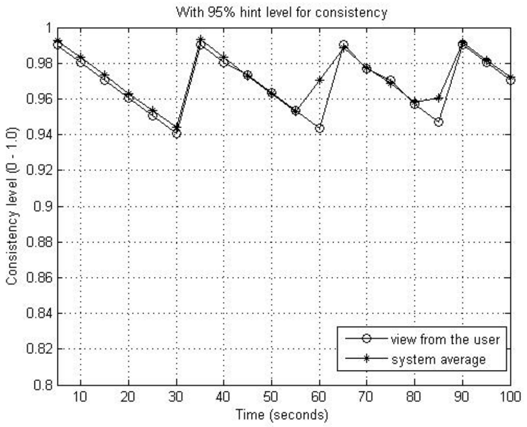


Figure 7(a): Setting hint level at 95%

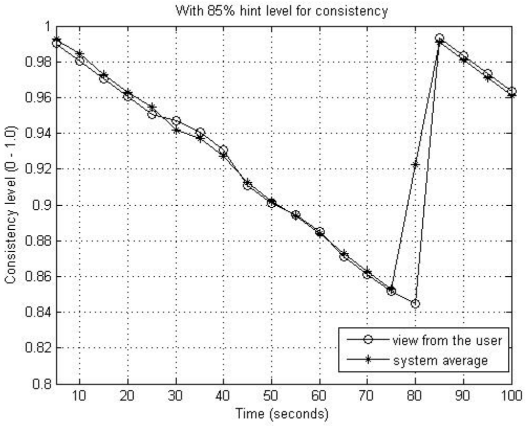


Figure 7(b): Setting hint level at 85%

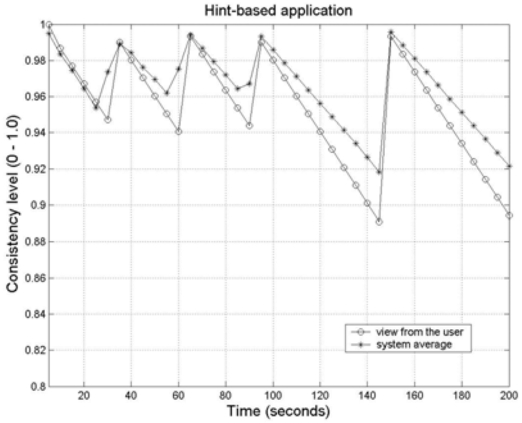


Figure 8: Hint-based application

	Delay for 1 round of active resolution
Phase 1	0.46825 ms
Phase 2	314.241 ms

Table 2: A breakdown of two phases involved in active resolution

executed in parallel, its performance does not change significantly with the top layer size. Since phase two is executed sequentially, its response time increases approximately linearly with the top layer size. The result in Table 2 is from a top-layer of size four where there are three nodes in top layer that the initiator needs to contact, thus on average, the cost for each additional member in the top layer is roughly 104.747 ms (314.141 / 3, because there are only three nodes need to be traversed). Thus the response time of active resolution for a top layer of size *n* is extrapolated as in Formula 2:

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

We depict the cost for active consistency resolution with top layer size up to ten in Figure 9. From the figure we can clearly see that, even with ten simultaneous writers, which is highly unlikely in a short period of time (in order of seconds) in practice, the cost of active resolution is still below one second. In an Internet-scale system, we believe that this is a reasonably good performance because it is not uncommon that, in a large-scale distributed system, a message is to be delayed for seconds or even more, thus offsetting the impact of delay caused by IDEA. Nonetheless, as explained earlier, a parallelism mechanism can be easily deployed to further improve the responsiveness of IDEA, which is useful in a scenario where the number of active writers is rather large.

We elaborated in Section 4.5.2 that background resolution essentially consumes the time incurred by the phase two of active resolution (first to collect all the updating information; second to send the consistent replica image information back), the delay of background resolution can thus be presented approximately as in Formula 3.

$$Delay = 104.747 * (n-1) \qquad \text{… (3)}$$

Clearly, the cost is even smaller than the one of active resolution. Together, the two measurements indicate that neither the active nor the background consistency resolution schemes in IDEA slows down the system even with a relatively large number of simultaneous writers.

6.3. IDEA's communication overhead

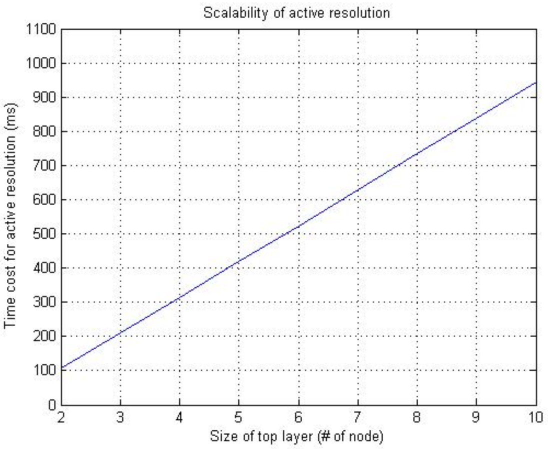


Figure 9: Scalability of active resolution scheme

Frequency	Overhead (# of exchanged messages)
20 seconds	168
40 seconds	96

Table 3: Overhead

To measure the communication cost in an appropriate context, we deploy IDEA in an automatic airline ticket booking system that, as stated in Section 4.6, mainly depends on the background resolution scheme to maintain consistency among nodes. Running periodically, the background resolution scheme brings the system's consistency level back to satisfactory states periodically.

Naturally, consistency resolution implies communication overhead, which is what we are going to measure here. In this application, however, the frequency of running the background resolution scheme is also a design tradeoff: the more frequently it is run, the better the system's average consistency level, but the overhead could become formidable. Thus, as stated in Section 4.7, there is a need to control the total overhead of IDEA below a certain ratio of the currently available bandwidth.

Hence, after evaluating the absolute communication cost, we will further explore the derivation of an optimal rate of running background resolution based on system's total capacity here.

6.3.1. The communication overhead

We run this experimentation with the same environment as in the previous section with two settings: first, we allow the background resolution scheme to kick in every 20 seconds; second, we allow the background resolution scheme to kick in every 40 seconds. The results are shown in Figure 10 in which the consistency level is the one perceived by all the top layer nodes. The incurred overhead, in terms of exchanged messages, is summarized in Table 3.

If we assume that each packet has size of 1KB (this is a reasonable assumption because the version vector only needs several bits to store its information), the total overhead of the first run (every 20 second) is 168KB and, after deriving it by the 100 second of running time, equals to 1.68KB/s, or 13.5bps, which is a very minimal bandwidth cost even for dial-up connections.

6.3.2. The trade-off

This experiment also clearly shows the tradeoff between overhead and achieved consistency level. That is, with the increased frequency of background checking and resolution—thus the increased overhead, the average system's consistency level becomes higher, at the expense of higher overhead (Table 3). Here we try to derive a formula to determine an optimal rate of running background resolution as follows.

First we assume the existence of a monitoring program on the server side to monitor the current total available bandwidth and we believe that this is a reasonable assumption. Then all that is needed in order to control the overhead of IDEA under a certain percentage of the current total available bandwidth is the communication cost of one round of IDEA background resolution. For example, if the current total available bandwidth is *b Mbps*, the maximal percentage of the bandwidth that can be used by IDEA is *x%*, and the one round communication cost is *c Mb*, the optimal rate of the background resolution can be presented as:

$$Optimal\_rate = \frac{b \times x\%}{c} \quad \ldots (4)$$

To derive an optimal rate of background resolution according to Formula 4, we need to know the communication cost of one round *c*. From Table 3, we have total six runs in these two experiments and we can approximate one round of background resolution as:

$$\#\_of\_messages = \frac{Total\_number}{rounds} \quad \ldots (5)$$

and the final value is (168+96)/6, which is 44.

Second, because the average size of exchanged messages varies from application to application, we use a parameter *s* to denote it and practitioners should substitute it with any real value they have. Thus the one round communication cost of background resolution in the experimentation setup is *c* = 44**s*. At this point, practitioners can use the derived *c* value to derive an optimal rate based on system's ongoing load by following Formula 4.

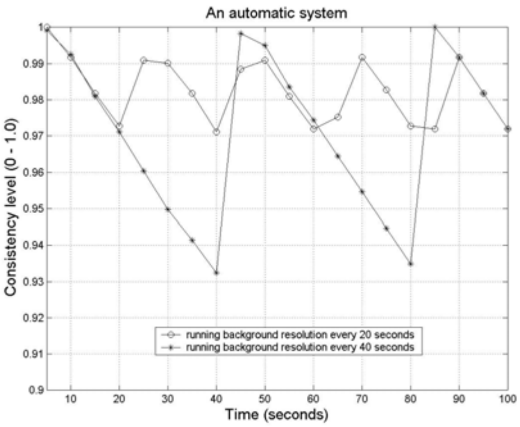


Figure 10: An automatic system

Finally, because the communication cost scales linearly with the size of top layer, the communication cost (thus the optimal rate of background resolution) for a particular application can be extrapolated according to its typical top layer size.

In addition to the trade-off, there is also an issue with respect to preventing underselling and overselling, which is unique to this airline ticket booking application. Detailed discussion about this issue and possible solutions were presented previously in Section 5.2.

7. Related Work

We discuss related works of IDEA from three aspects: (1) the tradeoff between consistency level and data availability; (2) the mechanisms to achieve adaptive control of consistency in distributed systems; and (3) the systems that IDEA can potentially work with to improve their consistency control.

7.1. Tradeoff between consistency level and data availability

In terms of the tradeoff between consistency level and data availability, probably the most closely related work to this paper is TACT [26]. Recognizing the inherent tradeoff between consistency level and performance and the rich semantics of this tradeoff, TACT proposed a set of parameters to measure the consistency level of an application and developed algorithms to bound the inconsistency within the system in a certain level. While IDEA uses TACT's definition to quantitatively define consistency level, it is significantly different with TACT because it is a detection-based consistency control scheme. Instead of tightly bound a system's predefined consistency level as was the case in TACT, IDEA recognizes that different applications may have different requirements for consistency and that one application's requirement for consistency can change from time to time, and explores the design space of efficient inconsistency detection to adaptively maintain acceptable consistency level based on applications' semantics.

7.2. Adaptive consistency control

To achieve the adaptability of consistency control, Yang and Li [25] have proposed a framework that puts a set of existing consistency protocols in a central module and the, based on the current application's characteristics, attaches the right consistency protocol dynamically and adaptively. While their work has the benefit of accommodating existing protocols, our work is advantageous in the sense that there is only one protocol that is needed to be deployed, which greatly simplifies the system design.

In the perspective of consistency resolution, Om [27] maintains consistency among replicas by automatically generating a consistent replica from a quorum system. Unlike Om, which focuses on the automatic generation of consistent copies, IDEA caters to applications' requirement by generating consistent copies on demand. More specifically, while Om imposes a two-layer replication scheme to enforce strong consistency; IDEA, targeting a wide range of applications that can benefit from the trade-off between consistency level and performance, enforces consistency control based on applications' semantics.

7.3. Systems that IDEA can work with

A number of Peer-to-Peer file systems [5, 10, 18, 22] use replication-based scheme to prevent data loss. However, they either assume that the files are read only, or use optimistic consistency control as a default option. Designed as an infrastructure to enforce consistency based on applications' ongoing requirement of consistency, IDEA focuses primarily on consistency issue and complements these Peer-to-Peer file systems. In theory, IDEA can work perfectly with these replication-based systems to improve their usability and performance from the perspective of consistency control.

This work can also be broadly put into the realm of autonomic computing because IDEA makes effort to meet applications' ever-changing requirement. However, previous work in autonomic computing does not consider applications' consistency requirements [13, 21], which is indeed important for replication-based distributed systems. In this sense, IDEA also makes contribution in the autonomic computing arena from the perspective of consistency control.

8. Conclusions and Future Work

In this paper, we presented the design, implementation, and evaluation of IDEA, an infrastructure for detection-based adaptive consistency control. As an alternative to conventional consistency control approaches, IDEA achieves adaptable, yet efficient, consistency control by detecting inconsistency among nodes in a timely manner and resolving the inconsistencies based on applications' ongoing requirement of consistency. Detailed design of IDEA, including its interaction with applications, is discussed.

A prototype of IDEA was deployed on Planet-Lab and two emulated applications, a distributed white board and an airline ticket booking application, are used to evaluate the adaptability interface and resolution efficiency of IDEA. Through the experimentation, we validated the adaptive interface of IDEA and showed the performance of IDEA in terms of low resolution delay and low communication cost it incurred.

In the future, we plan to investigate the implications of IDEA by deploying it to other distributed applications and use IDEA as a building block to improve the applications' usability in terms of consistency.

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] H. Chandler, The Complexity of Online Groups: A Case Study of Asynchronous Distributed Collaborations, ACM Journal of Computer Documentation, 2001, 25, (1): 17-24.

[3] T. Chang, G. Popsecu, and C. Codella, Scalable and Efficient Update Dissemination for Interactive Distributed Applications, in Proc. ICDCS 2002, Viena, Austria, July, 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] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area Cooperative Storage with CFS. In Proc. of the 18th ACM Symposium on Operating Systems Principles, October 2001.

[6] P. T. Eugster, R. Guerraoui, S. B. Handurukande, etc. Lightweight Probabilistic Broadcast, In DSN 2001.

[7] C. Huang, S. Sebastine, and T. Abdelzaher, An Architecture for On-Demand Active Web Content Replication, 16th Euromicro Conf. on Real-Time System, July 2004.

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

[9] D. Kostic, A. Rodriguez, J. Albrecht, A. Bhirud, and A. Vahdat. Using Random Subset to Build Scalable Network Services, In Proc. of USENIX USITS 2003.

[10] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-scale Persistent Storage, In Proc. of ACM ASPLOS, Nov. 2000.

[11] R. Levy, J. Nagarajarao, G. Pacifici, M. Spreitzer, A. Tantawi, and A. Youssef, "Performance management for cluster based web services," in Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management, (Colorado Springs, CO), March 2003.

[12] 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.

[13] H. Liu, and M. Parashar, Enabling Self-management of Component-based High-Performance Scientific Applications, In Proc. of HPDC-14, Research Triangle Park, NC, July 24-27. pp. 59-68.

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

[15] 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.

[16] Y. Lu, X. Li, and H. Jiang, IDF: an Inconsistency Detection Framework – Performance Modeling and Guide to Its Design, *Technical Report TR-UNL-CSE-2006-0003,* University of Nebraska-Lincoln, March 2006

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

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

[19] 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

[20] 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.

[21] R. Ribler, J. Vetter, H. Simitci, and D. Reed. Autopilot: Adaptive Control of Distributed Applications. In Proc. of HPDC-7, July 28-31, 1998. pp. 172-179.

[22] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, a Large-scale, Persistent Peer-to-peer Storage Utility. In Proc. of the 18th ACM Symposium on Operating Systems Principles, October 2001.

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

[24] 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

[25] 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.

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

[27] H. Yu and A. Vahdat, Consistent and Automatic Replica Regeneration, In. Proc. NSDI 2004.