flockfs, a moderated group authoring system for wireless workgroups

Surendar Chandra (surendar@acm.org) and Nathan Regola (nregola@nd.edu)
University of Notre Dame, Notre Dame, IN 46556, USA

***Abstract*—This paper describes the design and implementation of a group authoring system for wireless users. Our analysis of the behavior of various groupware systems using wireless user availability traces showed that prior systems would have performed poorly, especially during peak availability durations when many group members were simultaneously available. These results motivate our design choices. *flockfs* maintains one updateable copy of the shared content on each group member's node. It also hoards read-only copies of each of these updateable copies in any interested group member's node. The various copies are reconciled using a moderation operation; each group member manually incorporates updates from all the other group members into their own copy. The various document versions will eventually converge into a single version through successive moderations. The system assists with this process by automatically logging the provenance of all causal reads of contents from other replicas into the author versions. A prototype userspace file system implementation of *flockfs* exhibits acceptable file system performance and update propagation latency.**

I. Introduction

We designed a group authoring system that is initially targeted towards university users. Recently we observed over 13K active wireless devices in our campus (as opposed to 1.3K wired desktops); wireless laptops are the primary computing platform in our campus as well as at other universities [1]. Hence, we design our system to operate among wireless users.

First, we describe an application scenario to motivate the need for our system. Consider a group of students Alice, Bob, Emily and Tom working on a class project. Together, they develop the source code for their experiments, write a Word report and create a Powerpoint presentation. Alice and Tom work on the Word report while Bob browses the latest draft of the Word report and incorporates any changes into the Powerpoint presentation. Emily and Tom conduct experiments and report their results that will be incorporated by Alice and Tom into the report and then by Bob into the presentation. Tom explains the significance of the results to Alice and Bob using video clips. Even though the group members divide the labor, all of them require the ability to access and modify the source code, report and the presentation. Note that our primary concern is on objects that are modified by the group and not on objects such as video clips that are created by a single user.

Many students in our campus use emails to modify shared documents. In the earlier example, after each update *session*, Alice emails her Word report to Bob, Emily and Tom; Bob, Emily and Tom store this email in some private location. Bob is free to either ignore Alice's report or incorporate components of the report into his Powerpoint presentation. Eventually, the group creates a definitive version. This version is loosely defined by consensus among the various group members; whether Bob incorporates Alice's changes into his Powerpoint presentation is not enforced by the system. Frequent session updates can potentially create a large number of email messages. Manually managing these email messages is tedious. However, as we will see in Section V, *flockfs* provides much of this functionality in a far more seamless fashion. Recently, our campus subscribed to the web based Google Docs thereby creating another venue for collaboration.

In general, shared modifications can be realized as follows:

— *same-time* mechanisms are synchronous; group members operate on the shared document at the same time. They require the participating group members to be simultaneously online and available. The collaboration can be implemented directly through peer-to-peer mechanisms or can be mediated through storage servers. For example, storage systems such as AFS [2] or NFS [3] can be used to store the contents in a shared folder. The sharing semantics depends on the file system: e.g., AFS uses *last writer wins* semantics. In order to avoid update conflicts, some systems lock the entire file (or the conflicting portions: either explicitly or by splitting the single document into multiple files) during a shared update. Other systems allow non-blocking edits of the same document by all the group members. The usability of such non-blocking edits depends on the structure of the document. A source code file can compromise of well designed functions, each of which may be independently modified. However, the project report document is complex; e.g., significant changes in the experimental results affects the entire document.

— *different-time* mechanisms are asynchronous and do not require the group members to be simultaneously available. The group members update their own local copies which are then reconciled. The reconciliation can either be mediated by servers (e.g., Coda [4], Ficus [5], Apple iDisk, Windows Live SkyDrive). Peer-to-peer (P2P) mechanisms (e.g., Bayou [6], Windows Live Sync) have also been used.

The expected performance of prior authoring systems depend on the characteristics of the shared document. In this work, we consider complex documents in which updates from different group members are not easily reconciled using automated mechanisms. The system performance also depends on the number of group members who simultaneously modify the shared document. In order to understand the group dynamics, first we collected the availability traces of wireless users in our university (campus as well as in the dormitories) in Sep. 2006 and Dec. 2007 through Aug. 2008. The data exhibited a diurnal pattern with more users available during the day times. The number of users decreased during the weekend even though the available time for such users was better over the weekends. Session durations were becoming shorter and duration between sessions were getting larger.

Using these access traces, we analyze the expected performance of prior sharing systems that was built among wireless users. For synchronous mechanisms, we show that a mandatory locking scheme rejects many updates because of simultaneous requests. On the other hand, lock-free mechanisms allow a large number of simultaneous modifications. Also, AFS [2] like *last writer wins* semantics causes a large number of inconsistent updates with an observable loss of update causality. On the other hand, asynchronous peer-to-peer propagation systems such as Bayou [6] experience a large number of update conflicts and transaction roll-backs. More importantly, the worst performance in all these systems was observed during the peak availability durations; these systems failed when all the group members were available. Our analysis highlights the inadequacy of prior systems that attempt to maintain a single copy of the shared document.

flockfs addresses the poor sharing performance of prior systems. *flockfs* exports a file system interface, allowing the users to use conventional tools. On the other hand, *flockfs* does not have the rich semantic knowledge that would be available if *flockfs* was implemented directly into a document editing program. *flockfs* avoids conflicts by requiring each group member (i) to exclusively maintain their own updateable copy of the shared document (S_i). Published updates are available for automatic read-only hoarding by other group members. For a group of size n, each group member maintains utmost n copies of the shared document; a copy that they author (S_i) and up to $n-1$ read-only replicas from other group members. Since the versions are similar, compression mechanisms achieve excellent storage size savings. Given the improvements in storage cost and capacity (a 500GB 2.5” hard disk retails for less than USD$100) extra storage is a reasonable overhead. Group members can reduce the replica maintenance overhead by explicitly specifying the group members whose contents are replicated. Each author manually moderates and incorporates modifications from other group members using these read-only replicas. Moderation operations are similar to manual reconciliation operations in Coda [7] and Ficus [8]. However, a *flockfs* user reconciles the changes from all the group members. Automatic moderation is the subject of future research. All these copies are presented seamlessly via the *flockfs* file system interface. The document versions will eventually converge through successive moderation operations. The system assists in the convergence process by automatically logging the provenance of causal reads in order to quantify whether updates from a particular user had been incorporated into the final version. The distributed implementation of *flockfs* provides performance comparable to a centralized system for large groups. Update propagation can be improved when members with good availability become a member of all groups regardless of whether they themselves were interested in the particular shared document.

We built a prototype of *flockfs* using Git[1], an open source distributed version control system and Fuse[2], a userspace file system. Git is designed to be fast and efficient (space and time). *flockfs* inherits these advantages. Note that Git itself does not directly support the *flockfs* functionality. Benchmarks show that *flockfs* achieves performance similar to a fuse file system. *flockfs* is in regular use within our group and is being released to a larger audience.

For the rest of the paper: Section II discusses prior research. Section IV analyzes prior sharing systems using empirical wireless user availability data (Section III). Section V describes *flockfs* with concluding remarks in Section VI.

II. Related work

In our application scenario, students modify the shared objects from anywhere on campus or the dormitory using ubiquitous high speed wireless LAN network. Groupware systems can then be broadly classified by whether the modifications are synchronous (same-time) or asynchronous (different-time).

A. Synchronous: same-time, different-place groupware

Synchronous systems allow the group members to simultaneously modify the shared object. These systems can either be implemented using a server based approach or a peer-to-peer based approach. Storage systems such as AFS [2] and NFS [3] as well as applications such as MoonEdit[3] and GoogleDocs[4] use a server based approach while SubEthaEdit[5] and UNA[6] use a peer-to-peer based approach.

The performance of these systems depend on the network delay among the group members; as the latency increases, it becomes difficult to coordinate the shared modifications. One approach to deal with the latency is to avoid simultaneous modifications by exclusively locking the shared contents. On the other hand, systems such as GoogleDocs and SubEthaEdit allow non-locking and non-blocking access to the shared documents; group members can modify anywhere in a document at any time. Non-blocking systems might be inappropriate for complex documents (the focus of this paper), where changes by individual members can have global consequences in the document. For example, a different experimental outcome (like in Section V-D) showing poor file system performance of *flockfs* would require changes in the Abstract, Introduction as well as the Conclusion sections. Group members who are simultaneously modifying the Conclusion section might not realize the global change triggered by their modification. Our performance analysis of systems that require exclusive access to shared documents show that exclusive modifications

[1] http://git.or.cz/

[2] http://fuse.sourceforge.net/

[3] http://moonedit.com

[4] http://docs.google.com

[5] http://www.codingmonkeys.de/subethaedit/

[6] http://n-brain.net

can cause inordinate amounts of delay in modifying shared contents. Conversely, allowing simultaneous modification can cause observable conflicts, especially since user availability behavior was observed to be temporally consistent.

Another approach to mitigate the effects of network latency is to use client side caching and use cache consistency protocols. The performance of this approach depends on the periodicity of the cache consistency algorithms. For example, NFS [9] uses limited client block caching. NFSv4 servers [3] can delegate cache consistency responsibilities to the clients. AFS [2] achieves server scalability by requiring full file caching at the clients. AFS also uses the *last writer wins* consistency model. In Section IV-A2, we show that the number of write conflicts in our application scenario adversely affects AFS style collaboration mechanisms, especially when many group members are available.

B. Asynchronous: different-time, different-place groupware

When the group availability is poor, asynchronous systems can allow the group members to maintain a local copy of the shared contents; local updates are then reconciled with the shared object. Conflict resolution of the hoarded contents with the server contents can either be manual or automatic [8], [7]. When the automatic reconciliation fails, many systems (including Windows Live Sync) allow the user to manually reconcile the updates. The reconciliation can be mediated using servers or directly in a peer-to-peer fashion.

Applications such as Windows Live SkyDrive[7] and Apple iDisk[8] reconcile the local updates of each group member with a global storage. Coda [10] extends AFS to support disconnected access to the distributed storage system; wireless users hoard contents that were required for proper functioning while disconnected. Mummert et al. [11] reduced the reconciliation overhead of Coda. Based on the Berkeley file usage study [12], Ficus and Coda assumed that conflicting shared writes were rare. However, shared updates among disconnected clients still followed the *last writer wins* consistency model; updates while disconnected are reconciled on reconnection.

Leonard et al. [13] described a replicated document management system that formed the basis for the Lotus notes system. Their system was optimized for rarely connected clients and used asynchronous communication mechanisms to propagate updates to the shared databases. Ficus [5] built a highly available system with NFS semantics using optimistic replication. Similarly, Bayou [6] uses asynchronous communication for collaborative applications. Nodes exchange updates using a pair-wise anti-entropy protocol. Each update contained a programmable means to detect and respond to conflicting updates. Updates eventually reach all the participants. The system provides some bounds by using a primary commit protocol. In Section IV-B2, we show that the campus users will likely experience a large number of transaction roll-backs.

In a work-in-progress report, Howard et al. [14] introduce the notion of maintaining multiple autonomous versions that reconcile rarely with no single authoritative version. *flockfs* also maintains n different versions which are reconciled using the moderation option. Recently, Google released its moderated collaboration system called Knol[9]. Knols defines a single author; everyone is allowed to comment on the articles written by the author. The authors decide on whether to incorporate any of these comments. Each user in our system also performs moderated collaboration to their copy of the shared document.

III. Wireless user availability analysis

flockfs is designed to be practical; the design decisions are driven by the observed behavior of wireless users. First, we collected application level wireless user availability behavior in our campus where wireless access is provided using over 1,300 access points (AP), both in the campus and in dormitories. During our data collection interval, users self-reported 8,977 Windows, 3,319 Mac and 49 Linux clients. We collected application level availability using the Zeroconf[10] protocol. By default, clients running Mac OSX and the Linux (with Avahi) report the durations when they are available using the *_workstation._tcp* service. Note that the *flockfs* prototype has been ported to Mac OSX and Linux platforms. Zeroconf uses (non routed) link-local multicast for service discovery; a single wireless monitoring client cannot discover all the wireless users in our campus. Hence, we configured all the APs in our campus to use a single wired VLAN. We used appropriate packet filtering on the APs to reduce the amount of wireless traffic bridged onto the wired VLAN. We then installed a monitoring client on this wired VLAN to capture the user availability using the *dns-sd* tool. Our recent data collection lasted from Dec. 3, 2007 to Aug. 25, 2008. For our experiments, we show the first fifteens days worth of data when we observed 2,716 unique users. During this end-of-fall-semester duration, users were likely busy collaborating with other colleagues while preparing for final course projects and exams. Note that they would not be using *flockfs* or a similar collaboration system. We observed that user behavior depended on the day of the week. Hence, we especially focus on the two days from Dec. 6, 2007 (Thu.) to Dec. 8, 2007 (Sat.) to highlight the behavior on weekdays and on weekends. Note that it is possible that some devices used either the wired and wireless infrastructure. Any user who is offline (based on the wireless study) might actually be available using a wired connection. We could not correlate the MAC addresses or Zeroconf names of wired and wireless interfaces of the same laptop. Given the smaller number of wired devices sighted during the data collection interval (12,825 wireless devices vs 1,445 wired student VLAN), such a behavior was unlikely.

Figure 1 plots the number of simultaneously online users which gives an indication of the behavior of synchronous collaboration mechanisms. From Figure 1, we note that the user availability exhibits a diurnal pattern with the total number of users varying between ten and four hundred. Its important

[7] http://skydrive.live.com

[8] http://www.apple.com/mobileme/

[9] http://knol.google.com

[10] http://www.zeroconf.org/

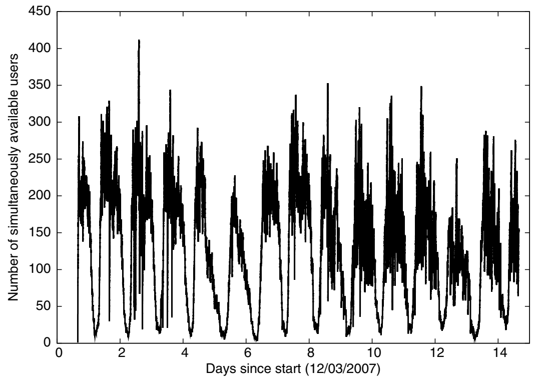


Fig. 1. Number of simultaneously available clients

to note that, during the time when the number of users was small (early morning hours), the demand for collaboration is also minimal (i.e., not enough members to request services).

Next, we analyze the session lengths (time that an user was online) as well as the time between two consecutive available sessions (not illustrated for lack of space). 50% of the sessions were under 20 minutes and 95% of the sessions were less than 75 minutes while the duration between sessions can be as high as four days. Earlier analysis of our campus users in 2006 ([15] showed that 50% of the sessions were under 1 hour with 95% of the sessions were under 6.7 hours. Even though the number of devices had increased (from 2,036 in 2006 to 2,730 devices 2007), the session durations had decreased. The systems exhibited constant node churn which places heavy load on epidemic propagation systems that need to transmit prior updates to newer nodes.

IV. Analysis of prior collaboration systems

Next, we investigate the expected performance of prior synchronous and asynchronous group authoring systems using our wireless availability traces (Section III). *flockfs* will address any shortcomings. For our analysis, we consider shared updates on a single object. Since wireless networks are ubiquitous and *free* in our campus, we assume that users only modify their shared documents when they are online. This assumption might not hold in wide area scenarios where wireless access is neither ubiquitous nor inexpensive. Also, we define *session* as the duration between when an author started to modify a document and when they are ready to share the draft with other members. In our motivating example, Alice goes through multiple *open*, *update* and *close* phases. However, Alice does not want others to see her drafts until the document reaches an acceptable form (as defined by Alice). The end of a *session* is explicitly defined by the author; file system *close()* does not mark the end of a *session*. *Sessions* are longer during the early stages of document creation; later *sessions* are shorter as users only make minor changes.

Ideally, we prefer empirical data on when users update shared contents. However, there are no such empirical data as systems similar to our target system are not widely deployed. Hence, we synthetically create update *sessions* using the wireless availability traces. We randomly selected groups of five, ten, twenty and thirty users. While online, each of these users randomly waited for some duration before starting a *session* that lasted for 0.5, one and two hours. Section III showed that duration when users were unavailable was long. Hence, users were assumed to end a *session* before going offline. Hence the average *session* lengths can be shorter than the target. When online, the users did not always modify the shared object, we considered cases where the user created *sessions* (on average) every one, two, three or four times that they were available. Next, we illustrate our session creation mechanism. Consider Tom who is online from 1:00-2:15, 4:00-4:15 and 9:00-10:00 (available from our wireless user availability traces). Assume that the *session* length is 30 minutes long and that the user requests an update *session* (on average) once every three times that they are online. Our random traces might create update sessions from 1:55-2:15 and from 9:00-9:30. Note that some sessions can be less than the thirty minutes. For brevity, we present the results from two setups: Busy (session length: 2 hours, group size: 30, update frequency: every time) and Light (session length: 30 mins, group size: 10, update frequency: every four times that user was available). We repeated each experiment with 1,000 randomly selected user groups and present the average values across the groups.

Note that it is possible that collaborating groups will tailor their availability behavior in order to facilitate the editing task. Our analysis in Sections IV-A and IV-B will show that the ideal coordination mechanism that is appropriate for existing systems is to stagger their availability durations and operate on the documents sequentially (and not use a more natural model of all users being simultaneously available).

A. Synchronous: same-time, different-place groupware

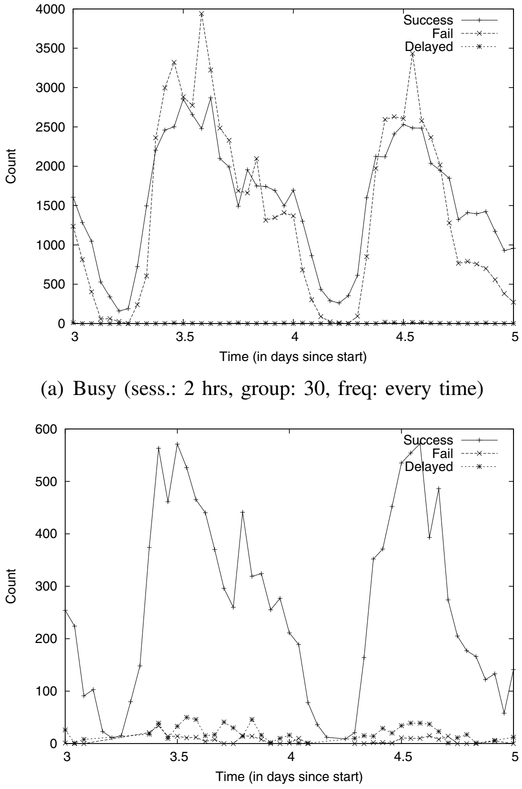
We investigated the behavior of systems that exclusively lock the shared objects, as well as a *last writer wins* scheme.

1) Exclusive access: One way to avoid update conflicts is to exclusively lock the object during the entire *session*. Other group members continue to read the prior version of the document until the exclusive update *session* is completed (they can then read the new updated version). Conflicting attempts to modify the object are handled as follows:

— *user waits:* if the new user was still available when the current exclusive *session* completed, then the new user is allowed to exclusively acquire the object for updating. Note that the actual *session* length achieved can be smaller than the originally requested *session* length if the user became unavailable sooner (users end a *session* before going offline).

— *request fails:* If the user could not acquire the object for exclusive access, then the user tries the next time when the user becomes available. When the exclusive *session* could not acquired till the next update *session* for this particular user, the *session* is considered to have failed.

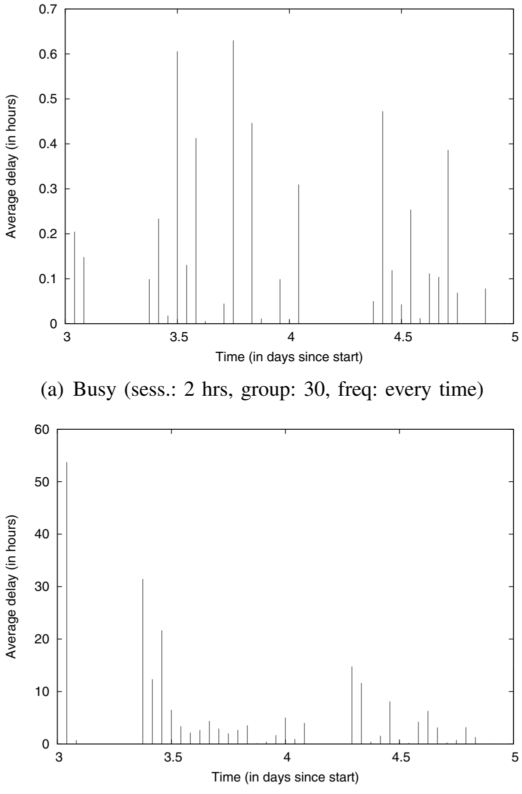
Consider two users: Alice who is online from 1:00-3:00 and Tom who is online from 1:00-2:15, 4:00-4:15 and 9:00-10:00 (gleaned from the user availability traces). Assume that the



(a) Busy (sess.: 2 hrs, group: 30, freq: every time)

(b) Light (sess.: 30 min, group: 10, freq: every 4 times)

Fig. 2. *Session* success for exclusive access (cumulative from 1000 groups)



(a) Busy (sess.: 2 hrs, group: 30, freq: every time)

(b) Light (sess.: 30 min, group: 10, freq: every 4 times)

Fig. 3. Average delay for exclusive access (cumulative from 1000 groups)

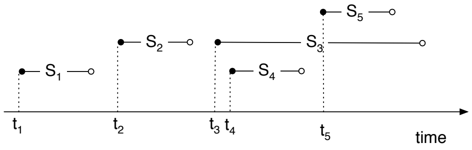


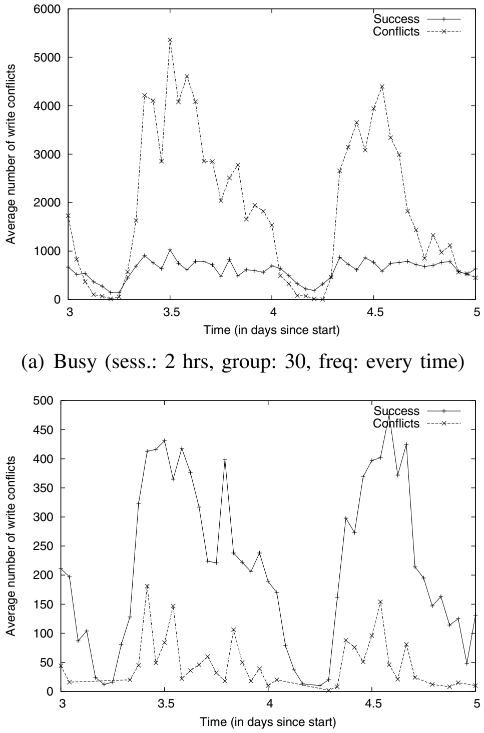
Fig. 4. *last writer wins* consistency model

session lengths are 30 minutes long and that the user requests an update *session* (on average) once every three times that they are online. Now suppose that Alice requests an update *session* at 1:50 and Tom requests update *sessions* at 1:55 and at 9:00. Alice's request will succeed at 1:50 and the document will be exclusively available to her until 2:20. However, Tom's request will be delayed and rescheduled at 4:00, a potential delay of 2:05. Had Tom been available at 2:20, the delay would have only been 25 minutes. Now, if the document was exclusively used by some other user at 4:00, Tom's request will fail because of the new update *session* requested by Tom at 9:00. Note that if the update succeeded at 4:00, Tom will only achieve 15 minutes of *session* length.

We plot the number of successful, delayed and failed *sessions* for the Busy and Light scenarios with the time of day in Figure 2. We also plot the actual amount of delay experienced by the delayed *sessions* in Figure 3. We prefer the system to require no delayed or failed transactions. From Figure 2(b), we note that the users experience minimal delay under lightly loaded scenarios (session: 30 min., group size: 10, frequency: once every four times). Since the user only requests exclusive *sessions* once in every four times that they are available and the *session* durations were small, most *sessions* can be rescheduled to a later time. On the other hand, the delays incurred can be quite large: from Figure 3(b), we note that the delays can be as high as 55 hours. Busy scenarios (Figures 2(a) and 3(a)) show that more transactions fail as compared to succeed. Few transactions are delayed with a delay duration of up to 0.6 hours. The delay is small because most transactions cannot be rescheduled (*session* length is long and users initiate an exclusive *session* every time they come online).

More importantly, the system performance is worse during times when all the users are available (daytime). For good performance, users will have to be uniformly available throughout the day including durations when they are not currently online (night times). Our system, *flockfs*, addresses this concern by not requiring exclusive access to contents; only the author is allowed to modify their own copy of the shared document.

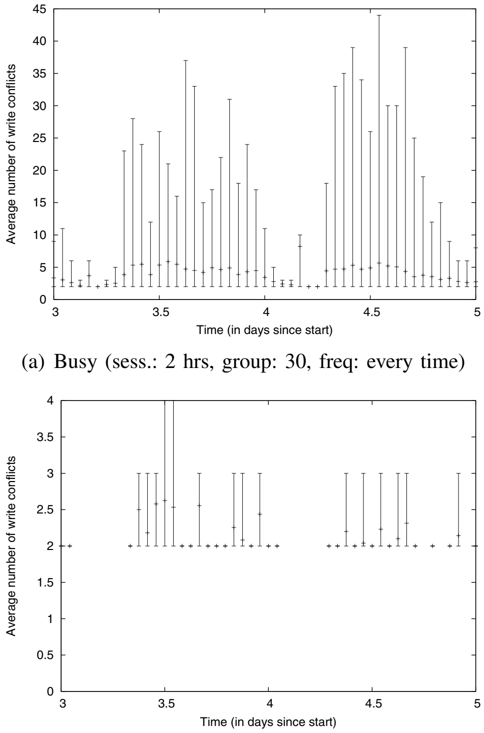
2) last writer wins: Next, we analyze optimistic mechanisms that allowed concurrent *sessions* and resolved any conflicts. For example, AFS [2] used a *last writer wins* policy in which simultaneous updates are allowed with the latest update becoming persistent and replacing all prior updates regardless of when the update *session* actually started. Consider an illustration of several *sessions* (S_i) in Figure 4. *Sessions* S_1 and S_2 produce consistent results because they do not overlap. However, *sessions* S_3, S_4 and S_5 can lead to inconsistent results because updates created by S_4 and S_5 are superseded



(a) Busy (sess.: 2 hrs, group: 30, freq: every time)

(b) Light (sess.: 30 min, group: 10, freq: every 4 times)

Fig. 5. Session success for *last writer wins* (cumulative from 1000 groups)



(a) Busy (sess.: 2 hrs, group: 30, freq: every time)

(b) Light (sess.: 30 min, group: 10, freq: every 4 times)

Fig. 6. conflicting updates (max, avg and min) (cumulative from 1000 groups)

by S_3 even though S_3 was concurrent with S_4 and S_5. The inconsistent state of the system is observable by other users. For example, the document view changes from S_2 to S_4 to S_5 before finally changing to S_3. Update S_3 will not incorporate any of the changes created in updates S_4 and S_5. S_3, S_4 and S_5 are in conflict with a count of three.

Next, we analyzed the behavior of this optimistic mechanism for the various session lengths, group sizes and update frequencies. We illustrate the number of conflicting and successful updates for the Busy and Light scenarios in Figure 5. For conflicting updates, we also plot the number of conflicting *sessions* in Figure 6. Figure 6 shows the average, maximum and the minimum number of *sessions* that cause the conflict (we need at least two overlapping *sessions* to cause a conflict).

Figure 5 shows high conflict rates. As compared to a mechanism that required the *sessions* to be exclusive (Section IV-A1), the *last writer wins* protocol allows more sessions to proceed even though the resulting system with its write conflicts can make the collaboration mechanism unusable. For example, the Busy scenario showed that the conflicting *sessions* can be over 5,500 (in 1,000 experiment runs) as compared to less than 1,000 that succeed at the same time. Even for Light session, the system exhibits conflicts. From Figure 6, we note that the number of *sessions* that participate in a single conflicting update can be as high as forty five.

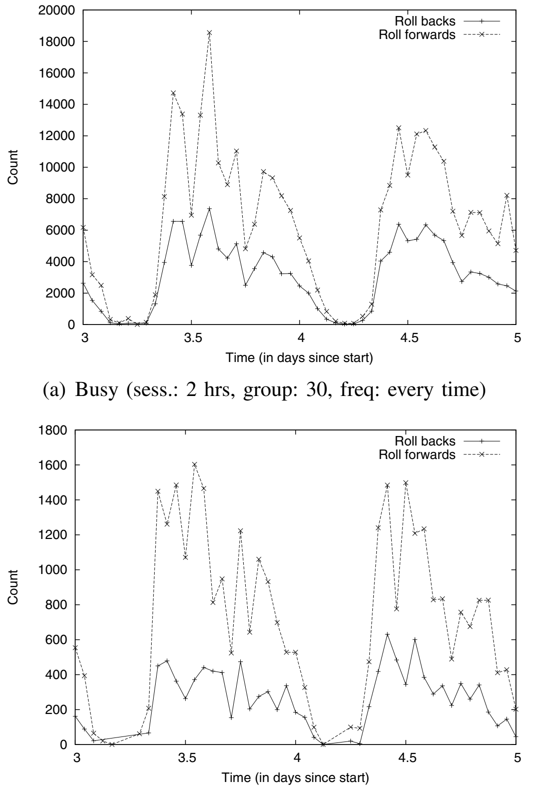
As we observed in Section IV-A1, the performance follows a diurnal pattern with higher conflicts during the times when more users are available (daytimes). A possible solution is to require user availability to spread uniformly throughout the day, though such a requirement is impractical. *flockfs* should provide a consistent view of the documents.

B. Asynchronous sharing

Asynchronous protocols update local copies of shared documents. At the end of a *session*, the local updates are either reconciled with a server copy or with each of the local copies on other users using a P2P pairwise reconciliation process.

1) Server mediated: Coda [10] extends AFS to support disconnected access by hoarding contents. On reconnection, each user reconciled their hoarded updates with the server. Shared updates among disconnected clients still followed the *last writer wins* consistency model; updates while disconnected are reconciled on reconnection. Similar to AFS (Section IV-A2), conflicting updates are hard to reconcile using pair-wise reconciliation because concurrent updates require reconciliation by both the users. The number of conflicting updates (not illustrated for lack of space) also exhibit a diurnal pattern with more conflicts during times when many users are available.

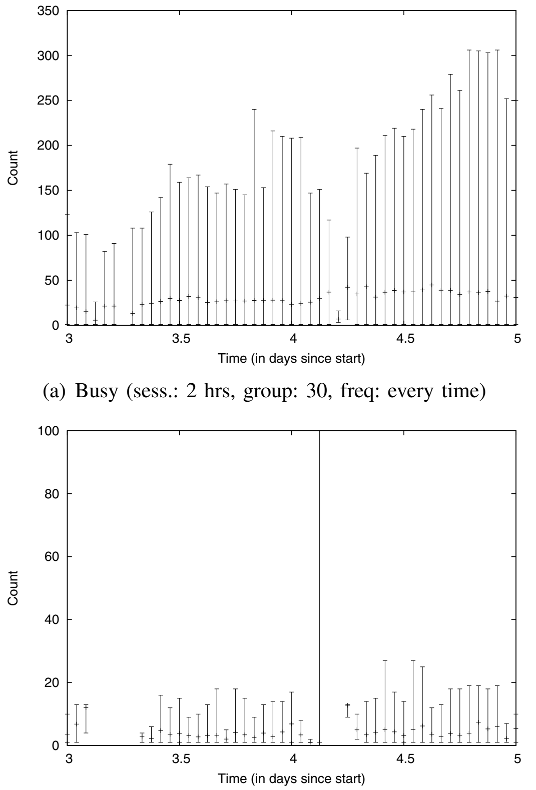
2) Peer-to-peer: Epidemic algorithms [16] are a popular P2P mechanism to propagate local updates. For example, Bayou [6] uses an epidemic based pair-wise anti-entropy protocol to reconcile updates; out of order updates require rolling back the local state in order to apply them in the correct order. High values of roll backs and roll forwards are not preferable; high roll forwards imply that the user was operating using an older version while high roll backs affect the causality relationships. In our motivating scenario, suppose Alice had



(a) Busy (sess.: 2 hrs, group: 30, freq: every time)

(b) Light (sess.: 30 min, group: 10, freq: every 4 times)

Fig. 7. Asynchronous update propagation (cumulative from 1000 groups)



(a) Busy (sess.: 2 hrs, group: 30, freq: every time)

(b) Light (sess.: 30 min, group: 10, freq: every 4 times)

Fig. 8. Number of roll backs per *session* (cumulative from 1000 groups)

created updates [1, 4, 5] (in logical clock order) and Tom had created updates [2, 3]. If Bob first received updates from Alice, he will incorporate this version [5] of the report into his presentation. Now, if Bob received updates from Tom, he will roll back by 3, and then roll forward by 5 by applying updates [2, 3, 4, 5]. Bob will now need to revisit the presentation to keep it consistent with the new state of the report.

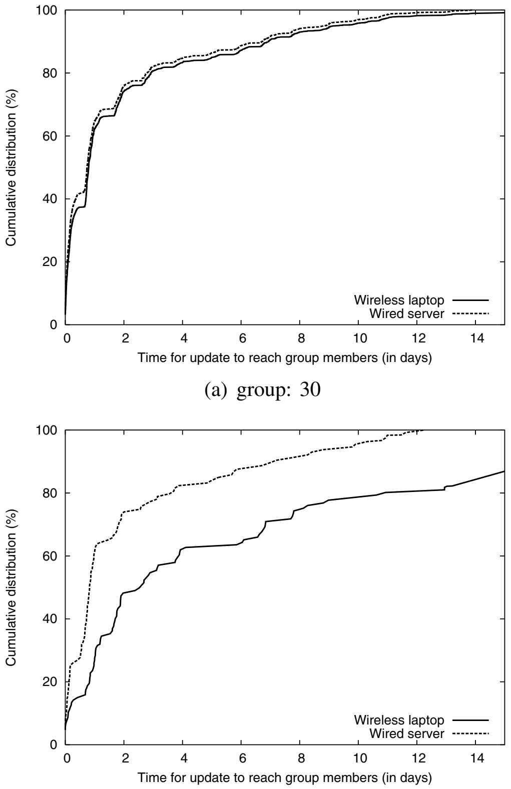
Next, we investigated the behavior of an epidemic propagation mechanism using our wireless user traces and measured the number of roll backs and roll forwards incurred by 1,000 random groups. We plotted the results for the Busy and Light scenarios in Figure 7. For updates that required a roll-back, we also plotted the minimum, average and maximum number of actual roll-backs per conflicted operation in Figure 8. The user will incur large roll backs immediately after they were offline for long durations as compared to other group members. We prefer the roll backs and roll forwards to be small.

We observe significant roll backs for both the Busy and Light scenarios in Figure 7. Using the cumulative results from 1,000 different groups, we note that Busy scenario required as much as 7,000 roll backs and about 19,000 roll forwards. For the Light scenario, we still required up to 500 roll backs and 1,600 roll forwards. From Figure 8, we note that the maximum roll back for a single *session* can be as high as 300 updates for the Busy scenario. Such a large number of *sessions* would lead to unacceptable behavior. As was observed in earlier sections, the worst system behavior was observed during the intervals when the users were highly available. The high roll backs during the daytime was caused by night times when the users were unavailable for longer durations.

V. Our system: *Flockfs*

We showed that *operating on a single shared copy is untenable* for wireless users: both in synchronous and asynchronous scenarios (Sections IV-A and IV-B). The worst performance was observed when many users were simultaneously available. Hence, we designed *flockfs* to address this limitation. As articulated by Howard et al. [14], *flockfs* does not maintain a single shared copy. Each user is responsible for updating and maintaining their own copy. The system provides read-only access to other group member's versions.

flockfs can be structured as a centralized or a P2P mechanism. Centralized approaches provide good management control and availability of the shared contents to the group members. Distributed approaches do not require the university to provide the necessary storage infrastructure before *flockfs* can be deployed. However, distributed mechanisms increase the storage requirements in each of the users laptops; each user may need to hold $n*S$ where n is the group size and S is the size of the shared document. Given the vast improvements in laptop storage cost and capacity (a 500GB 5400 RPM 2.5" hard disk retails for less than USD$100) extra copies are a reasonable overhead. Also, since the document versions are similar, deduplication can achieve good storage savings (our prototype uses Git which reduces storage requirements using



(a) group: 30

(b) group: 5

Fig. 9. Time to propagate single update to group (ave. of 1,000 groups)

similar techniques). Besides, we do not require that all group members must maintain copies of contents from other users.

We prefer the P2P approach for the ease of deployment. Next, we investigate the performance loss of a distributed approach. Using our campus wireless user availability traces, we analyzed the time it took for an update created by one group member to be available to the other group members using both these approaches. Contents are available when a particular group member is online; P2P approaches also require the message to be propagated to the user through other group members. Suppose Alice created an update at 1:00 AM; in a centralized approach, if Bob came online at 11:00 AM, then this new content will be available to Bob at 11:00 AM. Assume that no other group member was simultaneously available with Bob at 11:00 AM. If Tom (who has a read-only copy of Alice's contents) came online at 11:30 AM, then Bob has access to the contents at 11:30 AM.

For this experiment, we chose groups of size five, ten, twenty and thirty, injected an update into a random node. We measured the time it took for the content to be available to every other group member using a centralized and distributed approaches and plot the results in Figure 9. We note that the system requires as much as fifteen days to reach all the group members. For large group sizes (Figure 9(a): thirty users), the centralized approach performs nearly identical to a distributed approach. For small groups (Figure 9(b): five users), the server drastically improves the availability; from requiring over two days to reach 50% of the group members in about one day. We can incrementally improve performance by creating a dummy user who operates from a machine with good availability (e.g. wired desktop) and subscribes to all the group members even though he himself does not create any contents.

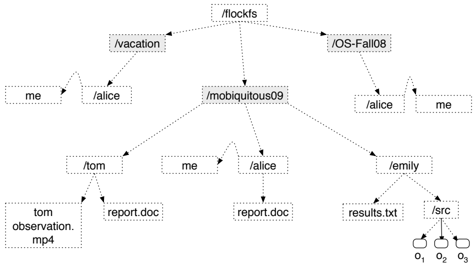


Fig. 10. Alice's view of her *flockfs* space

A. System Architecture

We designed *flockfs* as a distributed, moderated document sharing system. Each user updates and maintains their own copy. Allowing updates exclusively by the author has the added benefit that updates are always propagated in-order, obviating the need for roll-backs. The system provides read-only access to other group members versions. *flockfs* allows the user to participate in many workgroups and is agnostic to the content data types. Each user individually chooses the group members for a particular workgroup; *flockfs* does not maintain global group membership lists. All contents are shared with everyone though one can imagine a white list mechanism that can be used to limit the group members who are allowed to directly interact with the local copy of *flockfs*. Secure collaboration using *flockfs* is a topic for future enhancement.

Users interact with *flockfs* using the POSIX file system interface. By convention, *flockfs* is mounted under the user home directory as `~/flockfs`. The workgroups are available as directories at the root of the `~/flockfs` directory. Within each workgroup, *flockfs* maintains the shared contents from each group member in a separate tree. For convenience, we create a soft-link from the current user to a directory called `me`. We illustrate *flockfs* name space with an example. Suppose Alice belongs to workgroups `vacation`, `mobiquitous09` and `OS-Fall08`. As discussed earlier, Alice collaborates with Emily and Tom for the `mobiquitous09` workgroup. The file system view depends on the particular user (each user works with different workgroups and group members); Alice's view of the collaboration system is illustrated in Figure 10. Alice can modify her copy of the report; while Alice is in the middle of an update *session*, she operates on a local copy of report.doc. At the end of the update *session*, the local copy becomes the authoritative copy. Alice has access to the shared contents from Tom (`tom observation.mp4` and `report.doc`) and Emily (`src` and `results.txt`). *flockfs* behaves like a traditional POSIX file system; Alice is not notified of new files from Emily or Tom - instead, she is required to poll the file system for new content. Users operate on shared contents using their own applications; Alice will likely use MS Word to operate on her `report.doc`.

1) Special file system interaction: flockfs treats POSIX *mkdir()* and *rmdir()* operations differently. We performs three different operations for a *mkdir* request. A *mkdir()* operation on the */flockfs* root directory creates a new workgroup. This directory becomes a shared workgroup when another user also creates the same workgroup and adds this user as a group member. Within a workgroup, a *mkdir()* in the top level directory signified the users intent to collaborate with the particular user. When the requested user becomes available, either directly or via other users, the contents for the user's project are downloaded and presented at a future time. When the user creates a new project, *flockfs* automatically creates a directory for the local user (with a soft link to a special directory name of *me*). A *mkdir* inside the user's directory performs the traditional directory creation operation. These operations are entirely local. A project name collision is harmless; if two projects chose the same name, then users can access group members from either projects as their collaborators. Mixing members from two different groups makes the moderation operation harder. Similarly, non existent user names do not generate any errors because the system does not differentiate between a non-existent user and user who is exhibiting a long idle time. The corresponding *rmdir* operation undoes the *mkdir* operations. A *rmdir* of an user means that the local *flockfs* will no longer keep track of the contents from this user. An user leaves a workgroup using a *rmdir* of the project directory.

2) Session Maintenance: flockfs uses three system programs to support *sessions*:

1) `publish [<project>] [<comment>]`: marks the end of a *session*; by default, operating on the project belonging to the local directory. Users can name sessions using *comments*
2) `retract [<project>]`: retracts the local modifications to the project since the last `publish`. This operation is entirely local; other group members can only request the authoritative copy of the contents.
3) `status [<project>] [<user>]`: prints the comments associated with the last authoritative *session*; can be applied to the owner and collaborators' replicas.

B. Moderation operation

We depend on the user's ability to manually incorporate updates from other group members into their own version of the shared document. Moderation operation is already popular; many users use email to send their versions to other group members who perform an ad hoc moderation operation. Consider an user u_i moderating the v_{ia} version of the shared document using updates (v_{jb}, v_{kc}) from other users (where v_{jb} is the version b of document from user j). Note that a *flockfs* user is unaware of versions other than by using the *comment* field of the publish operation. Given the asynchronous nature of our epidemic propagation, user i need not have the latest versions from user j. User i is expected to identify the changes between the documents v_{ia}, v_{jb} and v_{kc} and incorporate the appropriate changes into his own v_{ia}. Using empirical evidence from a wider deployment of *flockfs*, we intend to investigate automated tools to assist in the moderation operation. Once version v_{ia} is published, the other users will incorporate v_{ia} into their own documents. Eventually, the various versions converge.

1) Provenance logging to assist in convergence: To identify whether a document had converged, we automatically log all file *open()* system calls of files from the current project before a *publish* operation. We assume that if a file is opened by an user, then they incorporated the changes suggested by that file. In the above example, the *publish* operation for v_{ia} logs the fact that user i opened files v_{jb} and v_{kc} before *publishing* his version. Eventually, user j can check the provenance comments on their local replica of v_{ia} to infer that v_{ia} had incorporated their latest changes (v_{jb}). Successively applying this mechanism allows any user to verify whether their own updates had been incorporated by every other users.

C. Implementation details

We implemented our update propagation using the Git version control system. Git is distributed, fast and efficient. Git is designed for well connected scenarios; Git users are expected to know the IP address of the group partners. Git users typically contact the lead project developer to obtain authoritative versions (e.g., request the latest version of Linux from Linus Torvalds). However, our system differs from the functionality provided by Git in a number of ways:

1) *flockfs* operates among weakly connected wireless users. It is not practical to request shared documents from other group members only when required in a lazy fashion (i.e., the other user might not be available). Hence, we maintain local copies of all group contents.
2) Git does not use any automated mechanism to propagate changes across sites; the users are expected to request newer updates directly from the content creator. However, our system automatically propagates the updates among the group members using an epidemic algorithm (including pulling contents from other group members).
3) *flockfs* defines the notion of group members and present their contents under an unified name space. Git users download contents into any directory of their choosing.
4) Git requires a priori knowledge of group membership and node location (IP addresses). *flockfs* uses wide area DNS to update its location. *flockfs* uses this location update to contact group members for update propagation.

We use the tools provided by Git as the basis for our system; we internally maintain, propagate changes and manage the contents from group members. Each replica and the authoritative copy is maintained as a separate Git repository. Git explicitly exposes the document versions; even though we maintain the versions of update *sessions* internally, we do not (yet) expose them using the file system interface.

Version	Data	Size (MB)	Time (sec.)
1	Random	10	13.35
2	Random (delta)	10	4.12
1	C File	10	4.97
2	C File (delta)	10	3.05

TABLE I
TIME TO PROPAGATE DOCUMENT VERSIONS

op.	Laptop (MB/s)			Desktop (MB/s)		
	native	fuse	*flock*	native	fuse	*flock*
write	22.74	16.65	16.28	57.63	41.35	40.27
rewrite	22.59	16.55	16.29	49.82	58.37	55.05
fwrite	22.18	16.57	16.37	56.50	52.27	58.21
frewrite	21.95	16.70	16.20	56.83	58.16	54.64
random write	22.03	16.21	15.85	55.72	62.65	56.44
read	22.46	16.12	15.67	55.06	52.31	56.30
reread	22.38	16.10	15.65	56.81	44.18	53.49
fread	22.38	15.84	15.58	56.50	54.52	53.93
freread	22.29	15.99	15.01	56.30	54.27	47.80
random write	26.96	13.18	13.15	58.14	43.23	43.33

TABLE II
IOZONE BENCHMARK (RECORD SIZE=16K)

D. System performance

Our system uses Git to distribute the contents and a fuse file system to access the various copies of the shared document. Our system performance closely matches the performance of Fuse and Git. First, we conducted experiments and measured the time taken to propagate session contents between a wireless laptop and desktop (both using IEEE 802.11g). We tabulated the results in Table I. We investigated the performance for a 10MB random file as well as a 10MB C program file. We then slightly changed these two documents (first few bytes). The system took 13.35 seconds to propagate the random file while taking only 4.97 seconds for the C file (Git compresses updates). For the updates, the system took 4.12 and 3.05 seconds, respectively. *flockfs* performs these propagation operations in the background.

We also benchmarked the *flockfs* using the IOZone benchmark[11] and tabulate the performance comparison between the native file system, *flockfs* and a vanilla *fuse* file system. We performed these experiments on a Macbook and an iMac desktop running Mac OSX 10.5.4. The laptop used a G4 processor, 1.5 GB memory with 60 GB (5400RPM) hard disk while the desktop used a X86 Core2Duo processor, 1 GB memory and 250GB (7200RPM) hard drive. We tabulate the results for various file system operations in Table II. In general, the performance of *flockfs* file system is slower than the native file system with performance similar to a fuse file system.

VI. STATUS AND DISCUSSION

We analyzed the behavior of our campus wireless users and showed the cost to maintain a single copy of the shared contents. We relaxed on this requirement and designed a system that used a moderation operation to allow users to maintain consistent versions of documents. Our implementation benefits from using two mature tools: fuse and git. *flockfs* works in Mac OSX and Linux. *flockfs* is deployed within our group. We are in the process of distributing *flockfs* to student workgroups within our department. *flockfs* will be freely distributed to a wide audience. Empirical usage data from a wider audience will be used to investigate automated moderation mechanisms.

ACKNOWLEDGMENT

This work was supported in part by the U.S. National Science Foundation (CNS-0447671).

REFERENCES

[1] T. Henderson, D. Kotz, and I. Abyzov, “The changing usage of a mature campus-wide wireless network,” in *Mobicom ’04*, 2004, pp. 187–201.

[2] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. Rosenthal, and F. D. Smith, “Andrew: a distributed personal computing environment,” *Commun. ACM*, vol. 29, no. 3, pp. 184–201, 1986.

[3] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck, “Network file system (nfs) version 4 protocol,” RFC 3530, Apr. 2003.

[4] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere, “Coda: A highly available file system for a distributed workstation environment,” *IEEE Transactions on Computers*, vol. 39, no. 4, Apr. 1990.

[5] T. W. Page, R. G. Guy, J. S. Heidemann, D. Ratner, P. Reiher, A. Goel, G. H. Kuenning, and G. J. Popek, “Perspectives on optimistically replicated peer-to-peer filing,” *Software—Practice and Experience*, vol. 28, no. 2, pp. 155–180, February 1998.

[6] A. Demers, K. Petersen, M. J. Spreitzer, D. Terry, M. Theimer, and B. Welch, “The bayou architecture: support for data sharing among mobile users,” in *Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, Dec. 1994, pp. 2–7.

[7] P. Kumar and M. Satyanarayanan, “Flexible and safe resolution of file conflicts,” in *USENIX 1995 Technical Conference*. New Orleans, Louisiana: USENIX Association, 1995, pp. 8–8.

[8] P. Reiher, J. S. Heidemann, D. Ratner, G. Skinner, and G. J. Popek, “Resolving file conflicts in the Ficus file system,” in *USENIX Conference Proceedings*, Boston, MA, Jun. 1994, pp. 183–195.

[9] B. Nowicki, “NFS: Network file system protocol specification,” RFC 1094, Mar. 1989.

[10] J. J. Kistler and M. Satyanarayanan, “Disconnected operation in the coda file system,” vol. 10, no. 1, pp. 3–25, Feb. 1992.

[11] L. B. Mummert, M. R. Ebling, and M. Satyanarayanan, “Exploiting weak connectivity for mobile file access,” *SIGOPS Oper. Syst. Rev.*, vol. 29, no. 5, pp. 143–155, 1995.

[12] J. K. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson, “A trace-driven analysis of the unix 4.2 bsd file system,” in *SOSP ’85*. Orcas Island, WA: ACM, 1985, pp. 15–24.

[13] J. Leonard Kawell, S. Beckhardt, T. Halvorsen, R. Ozzie, and I. Greif, “Replicated document management in a group communication system,” in *1988 ACM conference on Computer-supported cooperative work (CSCW ’88)*, 1988, p. 395.

[14] J. H. Howard, “Using reconciliation to share files between occasionally connected computers,” in *Fourth Workshop on Workstation Operating Systems*, Oct. 1993, pp. 56–60.

[15] X. Yu and S. Chandra, “Campus-wide asynchronous lecture distribution using wireless laptops (short paper),” in *ACM/SPIE: Multimedia Computing and Networking (MMCN’08)*, vol. 6818, San Jose, CA, Jan. 2008, pp. 68 180M–1 – 68 180M–8.

[16] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, “Epidemic algorithms for replicated database maintenance,” in *PODC ’87*, Aug. 1987, pp. 1–12.

[11] http://www.iozone.org