Moderated group authoring system for campus-wide workgroups

Surendar Chandra

Abstract—This paper describes the design and implementation of a file system based distributed authoring system for campus-wide workgroups. We focus on documents for which changes by different group members are harder to automatically reconcile into a single version. Prior approaches relied on using group-aware editors. Others built collaborative middleware that allowed the group members to use traditional authoring tools. These approaches relied on an ability to automatically detect conflicting updates. They also operated on specific document types. Instead, our system relies on users to moderate and reconcile updates by other group members. Our file system based approach also allows group members to modify any document type. We maintain one updateable copy of the shared content on each group member's node. We also hoard read-only copies of each of these updateable copies in any interested group member's node. All these copies are propagated to other group members at a rate that is solely dictated by the wireless user availability. The various copies are reconciled using the moderation operation; each group member manually incorporates updates from all the other group members into their own copy. The various document versions eventually converge into a single version through successive moderation operations. The system assists with this convergence process by using the made-with knowledge of all causal file system reads of contents from other replicas. An analysis using a long term wireless user availability traces from a university shows the strength of our asynchronous and distributed update propagation mechanism. Our user space file system prototype exhibits acceptable file system performance. A subjective evaluation showed that the moderation operation was intuitive for students.

Index Terms—Collaborative editing of complex documents, moderated manual reconciliation, collaborative file system.

1 Introduction

Laptops are ubiquitous in university campuses [1]. We design a system that allows these campus users to collaboratively author documents. We focus on scenarios where any group member can jointly modify a variety of shared documents [2]. We consider scenarios in which a change in one section affects other document sections (that might be modified by another member) in an unpredictable fashion [3].

Consider a typical scenario: a group of students Alice, Bob, Emily and Tom running experiments, writing a Word report and preparing a Powerpoint presentation for a class project. Alice and Tom work on the Word report. Bob incorporates changes from the draft report into the Powerpoint presentation. Emily and Tom conduct experiments and store the results in a custom file format. These results are incorporated by Alice and Tom into the report and by Bob into the presentation. Emily and Tom conduct further experiments to validate the conclusions drawn in the Word report. Tom explains the significance of the results to Alice and Bob using MPEG4 video clips. Even with the division of labor, all group members require the ability to read and modify any document. Currently, students use emails to propagate their draft versions; usually with an unsatisfactory outcome [4].

Simple modifications such as trim, append etc. on the video clips may be automatically reconciled. However, the scope of the conflict caused by updates to the Word document may extend beyond the regions that were actually changed. Frequently, these conflict regions are hard to detect. For example, an adverse experimental outcome has far wider implications than in the *Results* section. Unless the scope is properly identified, one can create a document with conflicting arguments from different group members; another section may continue to report good results based on prior results. These conflicts require the skills of a human editor to resolve. The author who is responsible for the *Results* section may have to make changes to other sections and documents (e.g. Powerpoint) that are the responsibility of other group members. We target such authoring scenarios.

Recently, Marshall [5] analyzed the scholarly practices of researchers in a corporate lab. She observed a wide variety of document authoring practices. Some groups assigned ownership of parts of the document to different group members while others operated in a draft-passing mode. The behavior also changed depending on the closeness of the report to the submission deadline. Sometimes, the writing activity itself drove further research; triggering additional modifications when these results became available. She observed significant heterogeneity in content preparation tools and file formats (e.g. GnuPlot, JGraph, XFig and Visio for plotting graphs). Some authors were less involved in the writing process (especially while travelling) and yet were interested in keeping abreast of the evolving document (usually via emailed copies). Sometimes, users exercised editorial control over resolving conflicts. She highlighted the reasons behind users' reluctance to automatically merge

FX Palo Alto Laboratory, 3400 Hillview Avenue, Palo Alto, CA 94304, USA
e-mail: surendar@acm.org

updates to bibliographic databases. Finally, she noted a desire to manage a personal archive of the shared documents and the associated datasets. She offered a number of recommendations for group authoring systems.

We incorporate our observations on how students modify documents as well as many of Marshall's [5] recommendations in our system. We build a file system based approach in order to operate with a variety of document types. We introduce moderation operations to manually reconcile the updates. We use the made-with knowledge of file system meta-data operations, described by Novik et al. [6], to assist in the reconciliation.

Each *flockfs* member exclusively maintains their own updateable copy of the shared document. Developing drafts are not available to other group members until the author explicitly publishes them [3]. Published updates are automatically propagated for read-only hoarding by other interested group members. We designed the propagation mechanism based on empirical wireless user availability information from a university. A distributed implementation of *flockfs* provides ease of deployment and 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.

For a group of size n, each group member stores at most n copies of the shared document; a copy that they author and up to $n-1$ read-only replicas from other group members. Given the improvements in storage cost and capacity (a TB laptop drive retails for USD$90) extra storage is a reasonable overhead. Since the versions are similar, de-duplication mechanisms achieve excellent storage savings. Group members can further reduce the storage overhead by explicitly specifying the group members whose contents are replicated.

All these copies are presented seamlessly via the *flockfs* file system interface. Each author moderates and incorporates modifications from other group members using these read-only replicas. Each author copy will be editorially consistent even if it had not yet incorporated all the recent changes suggested by other group members. 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.

We built a *flockfs* prototype using fuse[1], a user space file system library. Rajgarhia et al. [7] showed that fuse provides acceptable performance. We used Git (http://git.or.cz/) for versioned update propagation. Git is designed to be fast and efficient. *flockfs* inherits these advantages. Benchmarks confirm that *flockfs* achieves performance similar to a fuse file system. *flockfs* is available at http://flockfs.sourceforge.net/.

1. http://fuse.sf.net/,http://code.google.com/p/macfuse/

Next, Section 2 discusses prior research. Section 4 analyzes prior sharing systems using empirical wireless user availability data described in Section 3. Section 5 describes *flockfs* in detail. We conclude in Section 6.

2 Related work

2.1 Structure of group authoring systems

Systems such as Google Docs[2], MS Office 11 for Mac and 365[3] are designed for co-authoring. They are aware of each modification to the shared document and can resolve any conflicts. However, traditional editors (e.g., MS Office 2010 and earlier) are not group-aware. They cache all updates, eventually writing the entire updated document to a temporary copy and then atomically exchanging the temporary copy for the shared document. Systems such as Docx2Go [8] extend Word for group authoring. On the other hand, file system based mechanisms such as AFS [9] and NFS [10] are application agnostic and allow the users to use shared documents in any format. They must rely on the limited interactions of editing applications with the file system to identify and resolve conflicts. We use a file system based approach in order to operate on any document type.

Docx2Go [8] was motivated by empirical observations [5] on publication workflow among researchers at their lab. When group members save their shared Word documents, Docx2Go detects modifications by parsing and comparing the XML document with a shadow copy. Partial changes are then asynchronously propagated to other nodes in a distributed fashion using Cimbiosys [11]. Cimbiosys reduces the synchronization overhead from these partial updates to be proportional to the number of devices rather than to the number of shared items. Docx2Go asynchronously communicates any unresolved conflicts to the user with inline Word annotations. The document eventually converges to a single authoritative version. We investigate the pair-wise convergence performance for campus wireless users in Section 4.2.

flockfs supports their observed document heterogeneity [5] by being agonistic to the document type. *flockfs* uses Git to implement versioned update propagation at the granularity of files. The reconciliation process in our system is manual, which better supports the observed reluctance of users to allow automated reconciliation of bibliographic databases [5]. We use the file system meta-data open operation on hoarded replicas from other group members as a made-with [6] knowledge for reconciliation purposes. Thus, our system can infer that a document is reconciled because the user opened the latest versions of documents from other group members; even if in response, they never modified their own copy. On the other hand, *flockfs* can benefit from the availability of file type specific applications similar to Docx2Go that can automatically detect changes in documents.

2. http://docs.google.com
3. http://office365.microsoft.com

2.2 Updates and conflict resolution

Prior systems assume that updates to the same contents are in conflict with the scope of changes required to resolve the conflict strictly limited to the changed section. For example, if two users modify the same sentence in the Word document, only that sentence is assumed to be in conflict. However, if those users added sentences describing similar concepts in two different paragraphs, they are considered to be acceptable even though the resulting document will be semantically repetitious.

Conflict resolution can either be manual or automatic [12], [13]. Any failure in automatic resolution required a manual reconciliation. Each update in Bayou [14] contained a programmable means to detect and respond to conflicting updates. *flockfs* relies on manual moderation operation for conflict detection and resolution.

Google Knol[4] is a moderated collaboration system that defines a single author. Everyone is allowed to comment on the articles written by the author. However, only the author decides on whether to incorporate any of these comments into the shared document. In *flockfs*, group members incorporate the suggestions into their own copy of the document; they are eventually incorporated into all copies by mutual consensus.

In a work-in-progress report, Howard et al. [15] introduce the notion of maintaining multiple autonomous versions that reconcile rarely with no single authoritative version. *flockfs* implements a similar model. We maintain n different versions which are reconciled using the moderation operation. Authoritative version is inferred using the file system made-with knowledge.

2.3 Temporal correlation

The collaboration itself can be synchronous or asynchronous. Synchronous mechanisms are optimized for concurrent modifications. Asynchronous mechanisms are suited for disconnected users. Users update their own local copies of the shared contents. Local updates are eventually reconciled with the updates from other group members. The reconciliation can be mediated by servers (e.g., Coda [16], Apple iDisk[5], Windows Live SkyDrive[6]) or through distributed mechanisms (e.g., Ficus [17], Bayou [14], Windows Live Sync[7], Docx2Go [8]).

2.4 Distribution mechanisms

Depending on the distribution mechanism, group authoring systems can be centralized or distributed.

2.4.1 Centralized

Centralized approaches offer good availability and easy location of the definitive copy of the shared document.

A centralized synchronous system allows all group members to simultaneously modify the shared document. The system performance depends on the network latency; as the latency to the server increases, it becomes difficult to coordinate the shared modifications.

Exclusively locking the contents can address the server latency by avoiding simultaneous modifications. The exclusivity can be limited to the duration when the group member is online or until the lock was explicitly released. Our availability analysis in Section 3.3 shows that the duration between user sessions was long; extending the exclusivity until explicit release can significantly reduce the system availability. Section 4.1.1 shows that restricting the exclusivity to the duration when the user was online also performs poorly.

Systems such as Google Docs and MoonEdit[8] allow non-locking and non-blocking access using operational transformation [18]. Google Docs orders the sequence of concurrent updates at the server thereby delaying updates from high latency users. These approaches are inappropriate for our documents where changes by individual members can have global consequences. Our students also preferred to not share incomplete drafts.

Client side caching also mitigates the effects of network latency. For example, NFS [19] uses limited client block caching. NFSv4 servers [10] can delegate cache consistency responsibilities to the clients. AFS [9] achieves server scalability by requiring full file caching at the clients. AFS also uses the *last writer wins* consistency model. In Section 4.1.2, 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.

Windows Live SkyDrive, Apple iDisk and DropBox provide disconnected operation through a cloud storage. Coda [20] provides disconnected access to the AFS file system. When they become online, each client individually reconciles updates made while disconnected with the server. Shared updates among disconnected clients still followed the *last writer wins* consistency model; updates while disconnected are reconciled on reconnection (Section 4.1.2). Instead of relying on the university to provide the servers, we prefer the ease of deployment of a distributed asynchronous approach.

2.4.2 Distributed

Distributed approaches do not require an infrastructure before collaboration systems can be deployed. However, they require mechanisms to locate the definitive version of the shared contents (e.g., to submit the definitive Word report that incorporates all the group members' contribution to the instructor for grading purposes).

Systems such SubEthaEdit[9] and UNA[10] use a synchronous approach. In Section 3.3, we show that users

4. http://knol.google.com
5. http://www.apple.com/mobileme/
6. http://skydrive.live.com
7. http://www.foldershare.com/
8. http://moonedit.com
9. http://www.codingmonkeys.de/subethaedit/
10. http://n-brain.net

exhibited extended offline durations making them unsuitable for distributed synchronous sharing.

In an asynchronous system, group members maintain a local copy of the shared contents. Local updates are asynchronously reconciled with updates from other users using a pairwise distributed protocol. Ficus [17] built a highly available system with NFS semantics using optimistic replication and *single-copy* availability. Nodes in Bayou [14] exchange updates using a pairwise anti-entropy protocol. Each update contained a programmable means to detect and respond to conflicting updates. Updates eventually reached all the participants. The system provides some bounds by using a primary commit protocol. In Section 4.2, we show that the campus Bayou users will likely experience a large number of transaction roll-backs.

3 Wireless user availability analysis

Ultimately, the design choice of collaboration and update distribution mechanisms depend on the user availability dynamics. Factors such as the number of simultaneously available group members and durations when they are available for collaboration plays a crucial role.

Measuring the user availability for collaborations can be complex. Unlike wired desktops [21], wireless devices may be battery operated and are likely in active use when they are online. However, the user might not be actively collaborating with other users. The availability of a user for synchronous applications depends on times when the user is actively using the particular authoring application. Asynchronous systems can participate in the collaboration and propagate updates anytime the wireless device is online even if the user was inactive.

Earlier [22], [23], we showed this complexity of user availability. Using empirical data on durations when devices were online, when the users opened the iTunes application and when they shared their iTunes collection, we showed that the availability behavior was different among these scenarios; users did not always use iTunes nor did they share their collection when they used iTunes. Since there are no availability traces from deployed group authoring systems, we assume that the network availability durations represent times when the user is potentially available for collaboration. We synthesize activity durations from these availability durations.

Earlier efforts had collected wireless device availability traces under a variety of scenarios. Tang et al. [24], Kotz et al. [1], [25] and Balazinska et al. [26] collected wireless traces in a university building, university campus and in a corporate lab by using SNMP probes of the access points, respectively. Tang et al. [27] analyzed users in a metropolitan-area network. These traces captured the low level user behavior. They were also collected before wireless access was ubiquitous. Hence, we collected application level information which better captured the behavior pertinent to a collaborative application. For example, link layer mobility was not captured; any user who migrated across access points were considered to be continuously online even though they associated and dis-associated with multiple access points. Also, the time taken to acquire IP addresses and authentication credentials were not included in our metric.

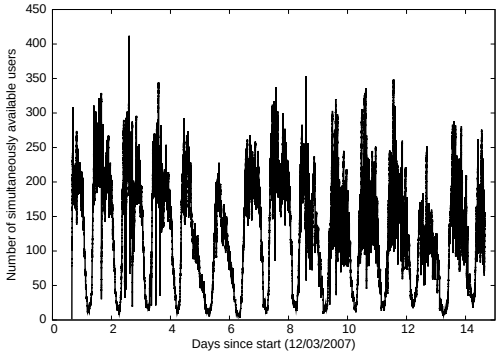


Fig. 1. Number of simultaneously available clients

3.1 Collection methodology

WLAN access in the university is ubiquitous and is available throughout the campus and in the dormitory using over 1,300 access points (AP). We collected application level availability using the Zeroconf[11] protocol. By default, clients running Mac OSX and Linux (with Avahi, http://avahi.org/) report the durations when they are available using the *_workstation._tcp* service. During our data collection interval, wireless users self-reported to be running Windows (8,977), Mac (3,319) and Linux (49) operating systems. Note that the *flockfs* prototype has also been primarily ported to Mac OSX. Zeroconf uses (non routed) link-local multicast for service discovery. We would require over 1,300 monitoring clients to discover all the wireless users in our campus. Hence, we configured all the APs in our campus to use a single wired VLAN; multicast traffic from all the 1,300 APs were routed to this Gigabit VLAN. We used appropriate packet filtering on the APs to reduce the amount of wireless traffic bridged back from the wired VLAN to the wireless network. We then installed a monitoring client on this wired VLAN to capture the user availability using the *dns-sd* tool. We collected traces from Dec. 3, 2007 to Aug. 25, 2008. For our experiments, we show the first fifteen 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. We observed that user behavior depended on the day of the week. Hence, we 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 students were not using *flockfs* or a similar system.

11. http://www.zeroconf.org/

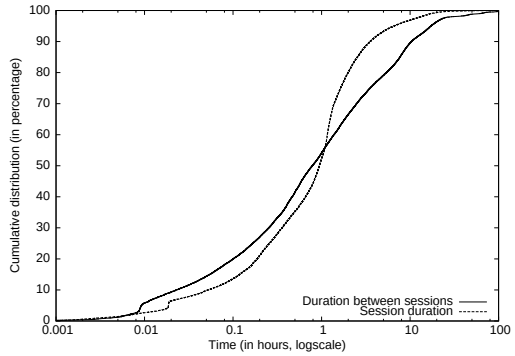


Fig. 2. Session duration and the time between sessions

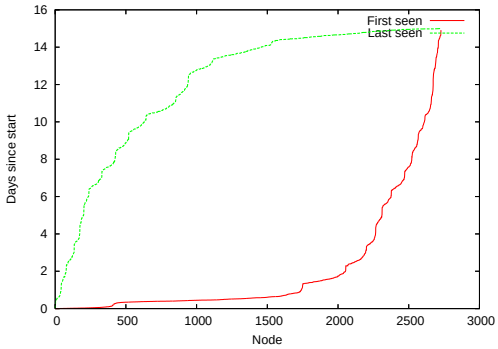


Fig. 3. Node churn

3.2 Number of simultaneously available users

The number of simultaneously online users affects the performance of synchronous collaboration mechanisms. From Figure 1, we note that similar to other scenarios [1], [25], [26], our user availability exhibits a diurnal variation with the total number of users varying between ten and four hundred. Thus, during the early morning hours, the demand for synchronous collaboration is also minimal. As we will see in Section 4, this observation has a profound effect on prior systems.

3.3 Session duration

Next, we plot the session lengths (time that a user was online) as well as the time between consecutive sessions over the entire trace duration in Figure 2. The session length measures the duration when communications with other group members are possible. Depending on the collaboration mechanism, the users can also operate on their local hoarded copy while offline.

We note that 50% of the sessions were under 20 minutes and 95% of the sessions were less than 75 minutes. Also, 50% of the duration between user sessions were less than 1.2 hours while 15% were longer than ten hours. Earlier analysis of our campus users in 2006 [22] showed that 50% of the sessions were under one 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. Next, we analyzed the duration between successive arrivals of a particular user in order to understand whether the shortening session durations equaled the increase in duration between sessions. We observed that the median values were 1.78 hours while 75% of users were online every 5.5 hours. In 2006, the median values were 2.52 hours with 75% of users online every 6.9 hours. The trend is for users to be online often but for shorter durations. We show the adverse effects of this reduction in session durations in Section 4.

3.4 Node churn

We plot the time when a node was first seen as well as when it was finally observed in Figure 3. Without node churn, we expect all nodes to appear on the first day and last till the last day. We observed constant node churn throughout the observation interval. This behavior was unexpected, as the traces were collected during the end-of-semester when students were expected to wait until the exams are over before introducing new nodes.

4 Analysis of Prior Systems

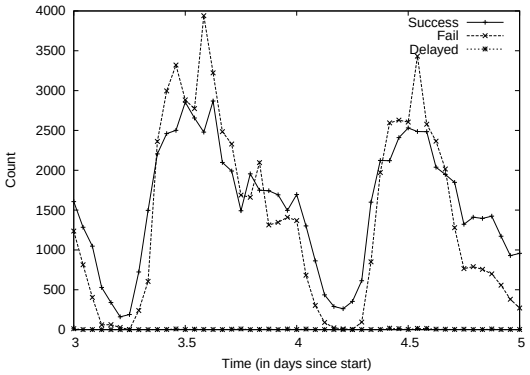
Next, we analyze prior group authoring systems using our wireless availability traces.

First, we define *fsession* as the duration between when an author started to modify a document and when they were ready to share the draft with other members. The end of a *fsession* is explicitly defined by the author and is not implicitly defined by a file system *close* operation.

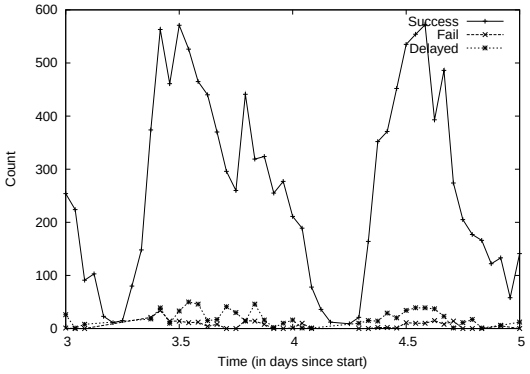
We prefer empirical data on *fsession* durations. However, we lack this data because groupware systems are not yet widely deployed. Hence, we synthesize *fsessions*. Section 3.3 showed that duration when users were unavailable was long. Wireless networks are ubiquitous and *free* in our campus. Hence, we assume that users are never disconnected while modifying the shared documents; users are assumed to end a *fsession* before going offline. Note that this assumption need not hold in wide area scenarios where wireless access is achieved through cellular networks which are neither ubiquitous nor inexpensive. While online, each user randomly waits for some duration before starting a *fsession* that lasts for 0.5, one and two hours. The average *fsession* 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 *fsessions* (on average) every one, two, three or four times that they were available.

Consider a user who is online from 1:00-2:15, 4:00-4:15 and 9:00-10:00 from our wireless trace. Assume that the *fsession* length is 30 minutes long and that the user creates a *fsession* (on average) once every three times that they are online. We might create *fsessions* from 1:55-2:15 and from 9:00-9:30. Note that some *fsession* lengths are less than the requested thirty minutes.

We selected groups of five, ten, twenty and thirty users according to a uniform distribution. For brevity, we present the results from two setups: Busy (session length:

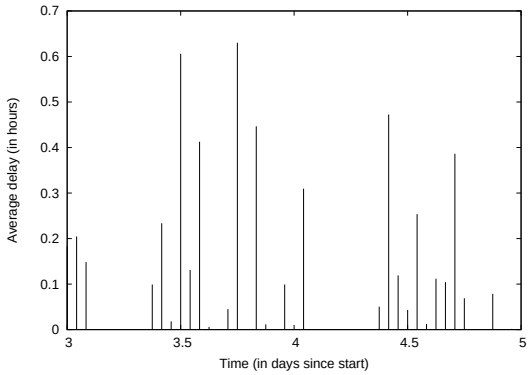


(a) Busy (sess.: 2 hrs, grp: 30, freq: every)

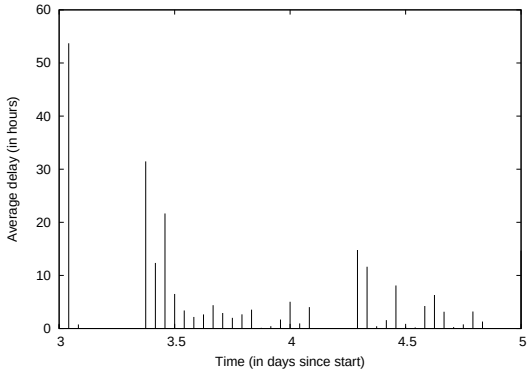


(b) Light (sess.: 30 min, grp: 10, freq: every 4)

Fig. 4. *Fsession* success for exclusive access



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



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

Fig. 5. Average delay for exclusive access

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 analyze shared updates on a single object. We repeated each experiment with 1,000 different user groups and present the average values across the groups.

4.1 Centralized approach

First, we investigate the behavior of synchronous systems that exclusively lock the shared objects as well as an optimistic *last writer wins* scheme. We also discuss the performance of an asynchronous approach.

4.1.1 Synchronous: Exclusive access

Exclusively locking the entire object during the *fsession* can avoid conflicts. While the object is locked, other group members continue to read the prior version of the document. Other authors *wait* to modify the locked object. If the new author was still available when the exclusive *fsession* completed, then they are allowed to lock the object. The actual *fsession* length achieved by the new *fsession* can be smaller than the originally requested *fsession* length if the author became unavailable sooner (authors always end a *fsession* before going offline). If the new author could not lock the object while they were online, they can retry the next time when they are available. When the exclusive *fsession* could not be acquired until the next scheduled *fsession*, the *fsession* 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. Assume that the *fsession* lengths are 30 minutes long and that the user requests a *fsession* once every three times that they are online. Now suppose that Alice requests a *fsession* at 1:50 and Tom requests *fsessions* 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 the update succeeded at 4:00, Tom will only achieve 15 minutes of *fsession* length. 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 *fsession* requested by Tom at 9:00.

We plot the number of successful, delayed and failed *fsessions* for the Busy and Light scenarios with the time of day in Figure 4. We also plot the actual amount of delay experienced by the delayed *fsessions* in Figure 5. We prefer minimal delayed or failed transactions.

From Figure 4(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 *fsessions* once in every four times that they are available and

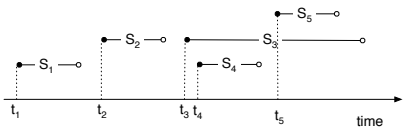


Fig. 6. *last writer wins* consistency model

the *fsession* durations were small, most *fsessions* can be rescheduled to a later time. On the other hand, the delays incurred can be quite large: from Figure 5(b), we note that the delays can be as high as 55 hours.

Busy scenarios (Figures 4(a) and 5(a)) show that more transactions fail. Few transactions are delayed with a delay duration of up to 0.6 hours. The delay is small because most transactions could not be rescheduled.

More importantly, the system performance is worse during times when all the users are available (daytime). *flockfs*, addresses this concern by not requiring exclusive access; only the author is allowed to modify their own copy of the shared document.

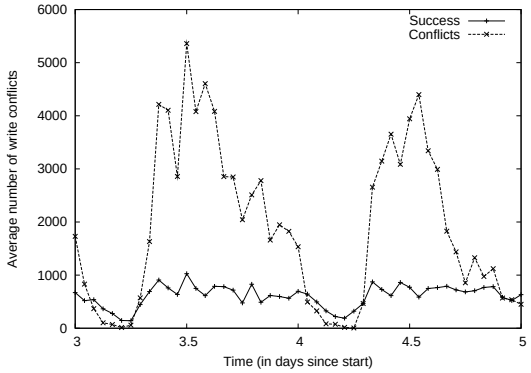
4.1.2 Synchronous: optimistic last writer wins policy

Next, we analyze optimistic mechanisms that allow concurrent *fsessions*; conflicts are resolved separately. For example, AFS [9] 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 *fsession* actually started.

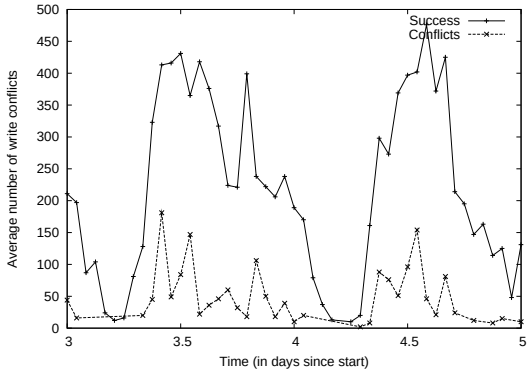
Consider an illustration of several *fsessions* (S_i) in Figure 6. *fsessions* S_1 and S_2 produce consistent results because they do not overlap. However, *fsessions* S_3, S_4 and S_5 can lead to inconsistent results because updates created by S_4 and S_5 are superseded by S_3 even though S_3 was concurrent with S_4 and S_5. The inconsistent system state is observable by other users as well. For example, the document 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 7. For conflicting updates, we also plot the number of conflicting *fsessions* in Figure 8. Figure 8 shows the average, maximum and the minimum number of *fsessions* that cause the conflict (we need at least two overlapping *fsessions* to cause a conflict).

Figure 7 shows high conflict rates. For the Busy scenario, the conflicting *fsessions* can be over 5,500 (in 1,000 experiment runs) as compared to less than 1,000 that succeed at the same time. From Figure 8, we note that even for Light sessions, the number of *fsessions* that participate in a single conflicting update can be as high as forty four. As compared to exclusive *fsessions* (Section 4.1.1), the *last writer wins* protocol allows more sessions to proceed even though the resulting system with its write conflicts can make the collaboration impossible.

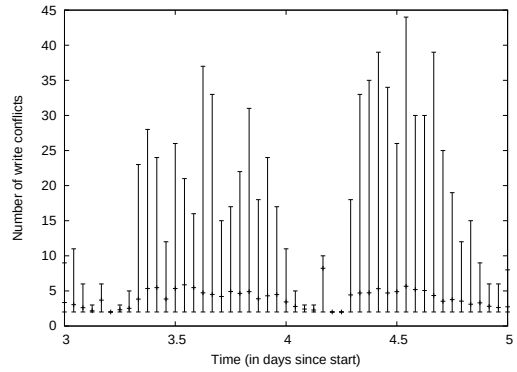


(a) Busy (sess.: 2 hrs, grp: 30, freq: every)

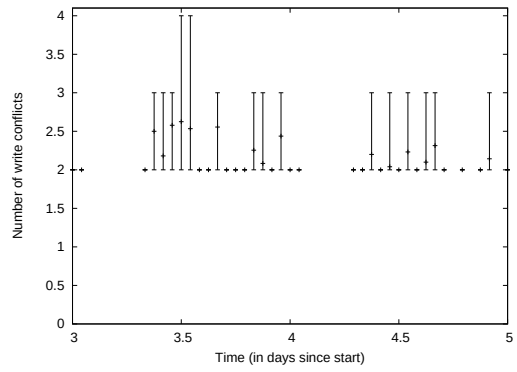


(b) Light (sess.: 30 min, grp: 10, freq: every 4)

Fig. 7. Session success for *last writer wins*

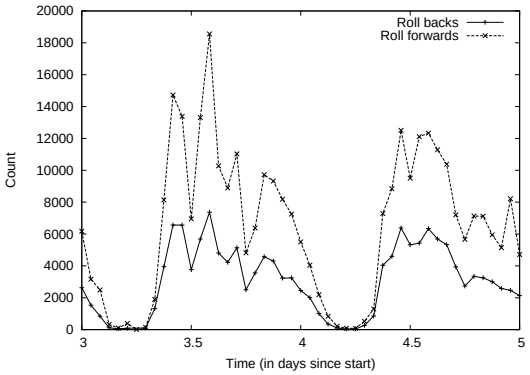


(a) Busy (sess.: 2 hrs, grp: 30, freq: every)

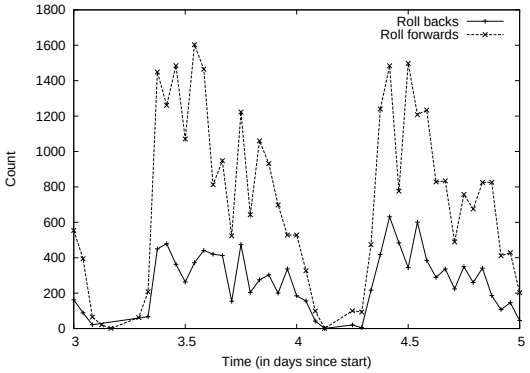


(b) Light (sess.: 30 min, grp: 10, freq: every 4)

Fig. 8. Conflicting updates for *last writer wins*

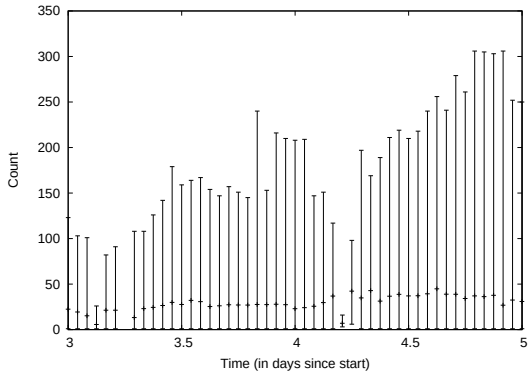


(a) Busy (sess.: 2 hrs, grp: 30, freq: every)

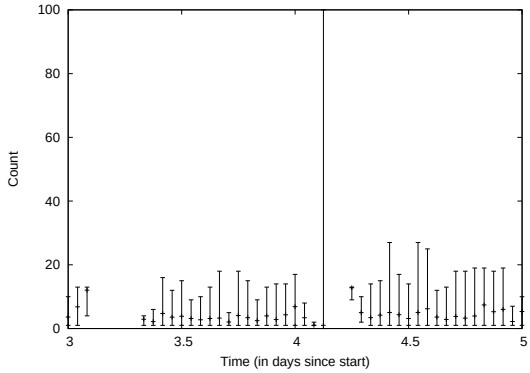


(b) Light (sess.: 30 min, grp: 10, freq: every 4)

Fig. 9. Distributed, asynchronous update propagation



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



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

Fig. 10. Number of roll backs per *fsession*

As we observed in Section 4.1.1, the performance follows a diurnal pattern with higher conflicts during the times when more users are available (daytimes). For the observed availability, *flockfs* provides a consistent view.

4.1.3 Asynchronous

Users hoard documents from server and operate on them while disconnected. Upon reconnection, each user independently reconcile their hoarded updates with the server. For example, Coda [20] extends AFS to support disconnected access. However, shared updates in Coda still followed the *last writer wins* consistency; the *fsession* started when the user hoarded the contents in preparation for disconnection and lasted during the entire duration when the user was unavailable. Since our wireless users exhibited long unavailable durations (Section 3), the *fsessions* in Coda were much longer than synchronous *fsessions*. Hence, the number of conflicting updates (not illustrated) was significantly higher than what was observed for the *last writer wins* mechanism (Section 4.1.2). Note that Coda users can use out-of-band coordination to reduce conflicting updates.

4.2 Distributed approach

In this approach, group members modify their copy of the shared document. Epidemic algorithms [28] are a popular mechanism to propagate updates; each local update is reconciled with those of other group members using a pairwise reconciliation process. Consensus protocols are used to identify the definitive version. We require at least a pair of group members to be simultaneously available in order to propagate the updates. Practical policies [29] choose propagation frequencies that trades off propagation rate with the network overhead.

For example, Bayou [14] uses a pair-wise anti-entropy protocol to optimistically reconcile updates; out of order updates roll 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 show that the user was operating using an older version of the document while high roll backs affect causality relationships. In our motivating scenario, suppose Alice had 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 the updates (**2**, **3**, **4**, **5**). Bob will now need to revisit the presentation to keep it consistent with the new state of the report. A pessimistic approach [30] could delay applying updates until they are committed and thus obviating the need for roll backs. In Section 5.3.1, we show that the time to propagate and commit updates on all group members can be large.

Next, we investigated the behavior of an epidemic

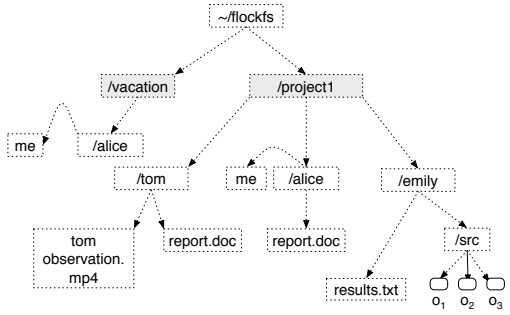


Fig. 11. Alice's view of her *flockfs* space

propagation mechanism using our wireless user traces and measured the number of roll backs and roll forwards incurred by 1,000 random groups. For our analysis, updates are instantaneously transmitted to group members who are also currently online. Typical policies lazily delay the durations when they perform a gossip operation, further reducing the system performance. We plotted the results for the Busy and Light scenarios in Figure 9. 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 10. 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 9. 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 10, we note that the maximum roll back for a single *fsession* can be as high as 300 updates for the Busy scenario. Such a large number of roll backs 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.

5 *flockfs*

Prior approaches manage the full spectrum of detection and application of updates from each group member in order to create a definitive version of the shared document. They assume that the scope of changes are localizable to the section where the user actually changed the shared document. Even with this assumption, we showed (Section 4) that these systems will experience poor performance. Next, we describe our approach.

5.1 *flockfs* user interface

Users interact with *flockfs* using the file system interface; *flockfs* is mounted as `~/flockfs`. Users join new workgroups and subsequently leaves them using the POSIX *mkdir* and *rmdir* system calls, respectively. For each workgroup, users add group members from whom they require read-only replicas using the *mkdir* system call. A *rmdir* for a user means that the local *flockfs* will no longer keep track of the contents from this user. While creating a project, *flockfs* automatically creates the authoritative copy that is modifiable by the particular user. For convenience, we create a soft-link from the current user to a directory called '`me`'. Within the author directory, *mkdir* and *rmdir* follow POSIX semantics.

Users indicate the end of a *fsession* by creating a special file called '.commit'. Users of the Mac OSX Finder can use the graphical user interface to indicate the end by attempting to *Lock* the project directory (which sets the *UF_IMMUTABLE* attribute).

Each user collaborates with different workgroups and hoards copies from different group members; the file system view depends on the particular user. Figure 11 illustrates the name space for Alice. Alice belongs to workgroups `vacation` and `project1`. Alice accesses the read-only copies of Emily and Tom's objects for the `project1` workgroup in directories $\sim/flockfs/project1/emily/$ and $\sim/flockfs/project1/tom/$, respectively. While Alice is in the middle of a *fsession*, she operates on a local copy of report.doc. At the end of the *fsession*, the local copy becomes the authoritative copy. Alice has access to shared contents from Tom (`tom observation.mp4` and `report.doc`) and Emily (`src` and `results.txt`). *flockfs* behaves like a POSIX file system; Alice is not notified of new files from Emily or Tom. Instead, she polls the file system for new content. Users operate on shared contents using their own applications; Alice will likely use Word to operate on her `report.doc`.

5.2 Moderation operation to incorporate updates

We depend on the user's ability to manually incorporate updates from other group members into their own version of the shared document; the feasibility of automated moderation will be investigated in the future. Consider a user u_i moderating his v_{ia} version of the shared document using updates v_{jb} and v_{kc} from other users (v_{jb}: version b of document from user j). Note that a *flockfs* user is implicitly aware of document versions through sessions. Depending on the network conditions, user i need not have the latest document version 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}. Once version v_{ia} is published, the other users will incorporate v_{ia} into their own documents. Eventually, the various versions converge. Even before convergence, each document is editorially consistent. Unlike automatic reconciliation, moderation will not cause semantic duplication.

5.2.1 Provenance logging to assist in convergence

Next, we describe our provenance logging mechanism by which every group member knows whether their own modifications had been incorporated by other group members. For making this deduction, we do not require

the rich provenance mechanisms described by Reddy et al. [31]. When all the group members' updates had been incorporated by every other user, the system had converged to a definitive version. The time required for convergence depends on when the users moderate updates from other users.

As a file system based approach, update granularity is limited by interactions of the authoring application with *flockfs*. Typical applications (e.g., Powerpoint) do not edit in place. Instead, they write the updated version into a temporary file and then switch the shared file with the temporary file. Hence, we cannot reliably know whether specific updates (i.e., byte ranges) made by one user to their author copy were read and incorporated by others into their own author copy. Prior approaches [32] require this ability to provide causality mapping. Instead, we assume that if a file from another user's replica was *read* (based on file system *read()* requests) during a *fsession*, then all changes suggested by that file are incorporated into the local author copy. This assumption is stronger than current practice where students assume that a particular group member had incorporated all their changes if they receive a confirmation email. Note that when multiple documents were emailed, the confirmation does not always specify the document version that was read.

For each *read* system call, *flockfs* automatically logs the current time as well as the file version, file name and the user's name whose replica was read. The file versions are unique to each author's copy and is a monotonically increasing number that is automatically created during the end of a *fsession*. The log records are stored as part of the *fsession* and propagated to other users' read-only replicas. In the above example, the *fsession* for v_{ia} logs that user i read files v_{jb} and v_{kc}. We provide a system program which lists which version of the local copy were incorporated by other group members into their own shared document. For example, consider Alice who is operating on her own version 3 of the shared documents. When Bob had incorporated the most recent version into his copy and Emily had incorporated (read) the prior version of README.txt. Our program describes this scenario as follows:

```
Current version: 3
File: README.txt
     User Emily incorporated version 2
          at 01/25/10 13:12:16
File: report.ppt
     User Bob is current
```

Note that Bob might be operating on his own version 3 of the report.ppt file (which is unrelated to Alice's version 3). Also, the usefulness of our mechanism depends on the application. For example, Apple iWork 8 uses multiple files to represent a single document. A new blank document used over 24 individual files while our typical Keynote presentations used more than 100 internal files. Users will see the status of each individual internal file; *flockfs* does not support such document bundles. The newer version of iWork uses a single file to represent the shared document.

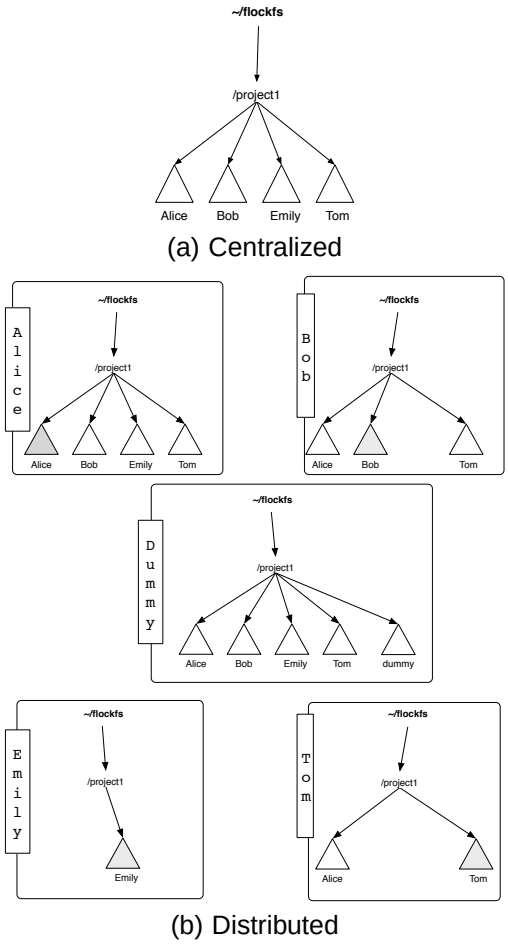


Fig. 12. *flockfs* moderated collaboration system

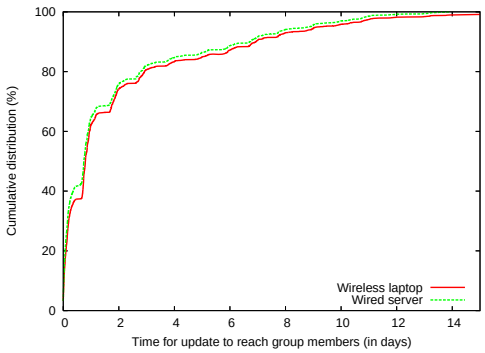
5.3 Update propagation

flockfs propagates a read-only version of the author copy to all other group members.

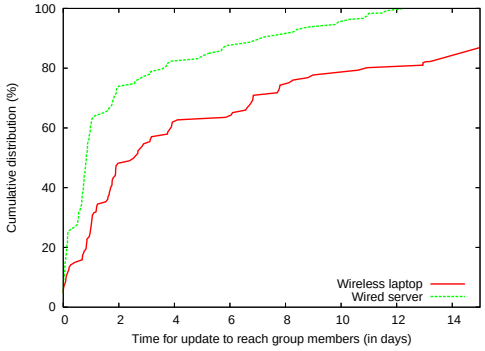
5.3.1 Distributed or centralized mechanism

flockfs can be structured as a centralized or a distributed mechanism. In the centralized approach (Figure 12(a)), the shared copies for Alice, Bob, Emily and Tom are stored on the server. Alice is only allowed to modify contents under her directory `~/flockfs/project1/alice`, Bob under his own directory name, etc., say using ACLs. All the copies are available for read-only access by other group members. The storage cost at the server increases by the group size. Since only one user updates a particular copy, there is no need for concurrency management protocols.

Distributed approaches do not require the university to provide the server storage before *flockfs* can be deployed. Each user (Figure 12(b): ignore the user Dummy for now) maintains their own copy while also hosting read-only copies of others' contents and thereby



(a) group: 30



(b) group: 5

Fig. 13. Time to propagate single update to group

increasing the storage requirements in each of the users' laptops. Each group member manages the local storage overhead by selectively maintaining copies from specific members. For example, Emily only hosts her personal copy, while Alice also has a read-only copy of Emily's contents. Given the vast improvements in laptop storage cost and capacity (a TB laptop hard disk retails for USD$90), 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 similar techniques). Since only a single author updates each copy, propagated updates are always in-order.

We prefer a distributed approach for its ease of deployment. Using the wireless user availability traces, we evaluate the time taken for the update from a group member to reach all the 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. In a distributed approach, the update is unavailable to Bob if no other group member was simultaneously available with him 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.

We chose random groups of size five, ten, twenty and thirty, injected an update into a node uniformly at random and measured the time taken for the update to be available to every other group member using a centralized and distributed approach and plot the results in Figure 13. We note that the system requires as much as fifteen days to reach all the group members. For large group sizes (Figure 13(a): thirty users), the distributed approach performs nearly identical to a centralized approach. For small groups (Figure 13(b): five users), the server improves the availability; from requiring over two days to reach 50% of the group members to about one day. The distributed approach is competitive. Users can improve performance of the distributed approach by creating a dummy user who operates from a machine with good availability (e.g. wired desktop in the dormitory) and subscribe to all the group members even though he himself does not create any contents (Figure 12(b)).

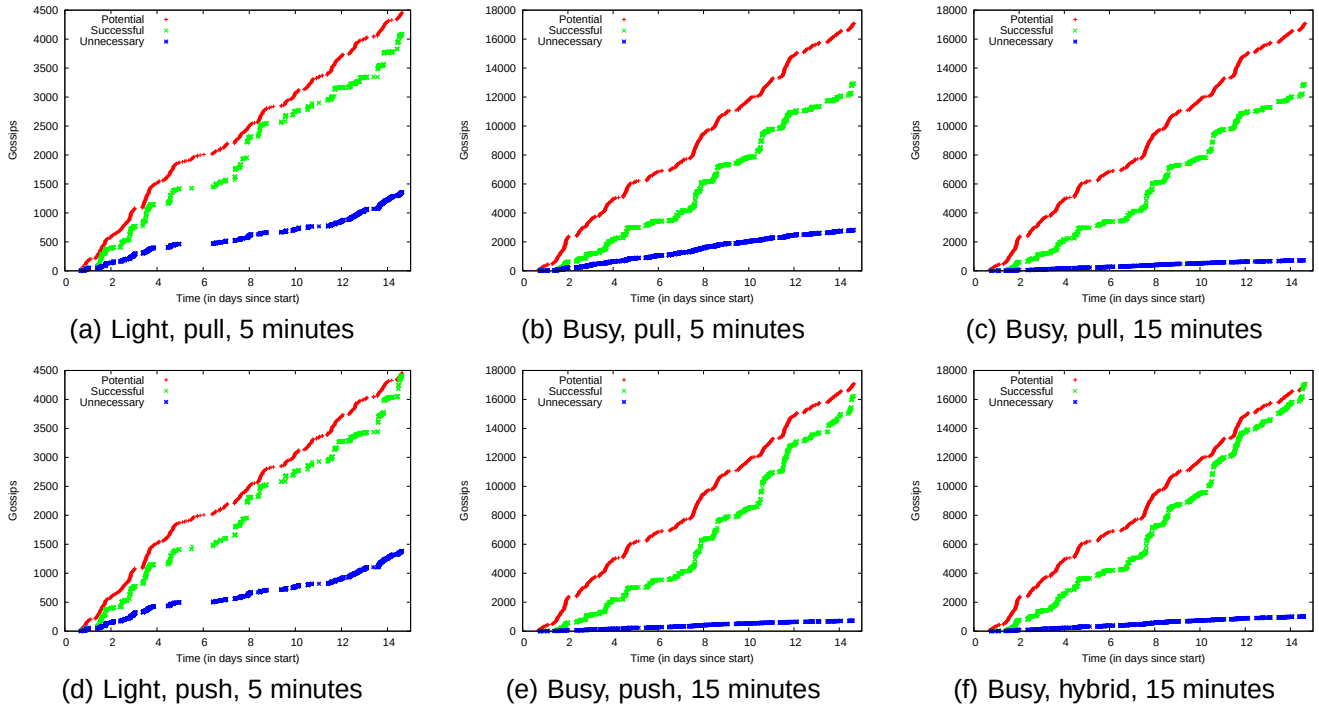
5.3.2 Practical propagation parameters

flockfs propagates all the local updates to the author's copy of the shared document to all the other read-only replicas. For example, for a n member group, if group member i created u_i updates, then *flockfs* propagates $\sum_{i=1}^{n} u_i * (n-1)$ updates amongst the group members.

We use a pair-wise epidemic algorithm [28] to periodically forward updates through other online read-only replicas. Each gossip transmits all the new updates that were available among the participating user pair. Frequent gossips propagates updates quickly while gossips which did not propagate any new update (because there were none since the last gossip) wastes network resources. The goal is to choose the propagation frequency while avoiding unnecessary gossips.

In *flockfs*, users propagate updates whenever they become online. Updates can either be periodically pushed or pulled from another member. Simultaneous pushing and pulling behaved similar to a push or pull at twice the propagation frequency [29]. Initially, few nodes have the update and hence we require aggressive propagation of the update from the originating node before going offline. Before a gossip is scheduled, the pushing node does not a priori know whether the receiver requires any new updates while a pulling node does not know whether the sender has any new updates. Our earlier analysis [29] suggested that push and pull policies are not complementary. In general, pull randomized the times when messages were propagated and hence can achieve better update propagation than push based mechanisms.

Next, we analyze push and pull based schemes using the wireless user availability traces and collaboration groups described in Section 4. We analyzed update propagation frequencies of five, fifteen, thirty and sixty minutes. For a given group, we measured the cumulative number of potential updates that needed to be propagated by *flockfs* (each *fsession* corresponds to $n-1$ potential updates), the number of successful updates that were already propagated by a particular policy as well as the number of unnecessary gossips. Ideally, we prefer the number of potential and successful updates to be the same with no unnecessary gossips. Note that each



(a) Light, pull, 5 minutes (b) Busy, pull, 5 minutes (c) Busy, pull, 15 minutes

(d) Light, push, 5 minutes (e) Busy, push, 15 minutes (f) Busy, hybrid, 15 minutes

Fig. 14. *flockfs* propagation policies

gossip can send multiple updates, especially after a user was unavailable for long durations.

We plot the results for a representative Light (sess.: 30 min, group: 10, freq: every 4 times) and Busy scenario (sess.: 2 hrs, group: 30, freq: every time) using the *push* and *pull* policies in Figure 14. We observe that unnecessary gossips were higher for propagation frequency of five minutes; the updates were being aggressively propagated. For the Busy scenario (Figures 14(b) and 14(c)), on the fifteenth day, the unnecessary gossips reduced from about 2,800 to 800. The values were lower for larger propagation durations. We consider 15 minutes to be an adequate balance. For the Light scenario (Figures 14(a) and 14(d)), the push and pull policies exhibited similar amount of successful update propagation; the system had propagated almost all the outstanding updates by the fifteenth day. On the other hand, for the Busy scenario (Figures 14(c) and 14(e)), we note that the push policy performed better. By the fifteenth day, the push policy had almost propagated all updates while the pull policy had only propagated 13,000 of the 17,000 updates. The long propagation durations is an artifact of choosing random group members. Our wireless traces do not contain information about actual collaborators who might exhibit better availability amongst the group.

Further analysis (not illustrated) showed that many sessions ended right when the user was going offline before the system had a chance to pull and propagate these updates. Hence, we designed a hybrid policy that periodically performed a pull operation while using a push operation when a *fsession* completed. We observed better performance (Figure 14(f)) than either the push or pull policies; almost all the updates were successfully delivered to all the group members while incurring slightly more unnecessary gossips (1,000 rather than 800 at the end of the fifteenth day). We support this hybrid approach for our *flockfs* prototype. Next, we discuss the security implications of the hybrid policy.

5.4 Open architecture and its security implications

Group management is fully distributed. Students can use their university issued userid to avoid name conflicts. Any user can create a new project by creating a directory. This directory becomes a shared workgroup when another user also creates the same directory and adds this user as a group member. *flockfs* does not maintain global group membership lists. Users are allowed to add unknown (or potentially non-existent) group members. 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.

This openness can lead to group membership conflicts. When group members are disjoint, two different projects choosing the same name are not aware of each other. For example, if Alice and Emily as well as Bob and David created a project called *project-A*, there is no conflict unless Alice inadvertently requested a read-only replica of Bob's documents (from his collaborations with David). Alice is still required to incorporate Bob's document into her own document. Alice could cheat on her project report by copying the outcomes from Bob's report; *flockfs* does not provide secure provenance verification [33].

Malicious users can corrupt and propagate the read-only replicas of other group members. The push opera-

tion in the hybrid propagation policy (Section 5.3.2) also introduces a security threat in that a malicious node can push random updates into the author copy; push operation is more suitable for trusted LAN networks. In our prototype, users need to explicitly allow the push component of the hybrid policy. By default, the system behaves like the pull policy. Secure collaboration using *flockfs* is a topic for future enhancement.

5.5 Implementation details

We implemented the *flockfs* prototype using a 1,250 line C program (available at http://flockfs.sourceforge.net/). We use the FUSE library to implement a user space file system. *flockfs* implements all the special file system attributes necessary to use the Finder on Mac OSX Snow Leopard. We built a custom location server that maps the user name with their current IP address; production versions can use wide area DNS services. Our prototype does not operate through NAT firewalls and requires all the group members to be directly accessible.

We used the Git version control system for propagating updates among the group members. Each replica and the authoritative copy are maintained as a separate Git repository. *flockfs* uses the ability of Git to manage various document versions and propagate them manually on demand. We also utilize the capability for cryptographic authentication of commits in order to identify malicious updates by group members. However, *flockfs* does not use the ability of Git to create document branches. Since a single user updates the author copy, we also do not require Git's functionality for merging versions.

Git itself does not provide the *flockfs* functionality. Git is designed for distributed scenarios where many users can download the source repository into an arbitrary location without a unified name space. Git users manually contact a peer repository and obtain the latest versions of the source code. Updates are automatically merged by Git; the merge procedures are optimized for source code repositories. By convention, certain peers are authoritative (e.g., Linus Torvald's repository for the Linux kernel). Git users are expected to know the IP address of the trusted partners (e.g., git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git for Linux kernel). Also, Git is designed for well connected scenarios; clients expect the repository to be available on demand.

flockfs operates among weakly connected users; it is not practical to request shared documents from other group members only when required. Hence, we maintain local copies of all group contents. Unlike Git, our system automatically propagates the updates among the group members using an epidemic algorithm (Section 5.3.2). *flockfs* defines the notion of group members and presents their contents under an unified name space.

5.6 System performance

First, we report the objective performance of *flockfs*. *flockfs* performs update propagation in the background which does not affect the file system performance. The network overhead for an unnecessary gossip was about 1,500 bytes. Successful gossips used compressed data to transfer the documents and were thus network efficient.

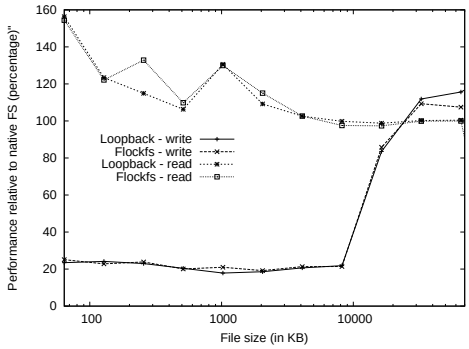


Fig. 15. Iozone file system benchmark performance

Next, we measured the performance of *flockfs* using the IOZone (www.iozone.org) benchmark. Rajgarhia et al. [7] showed that the performance of fuse was adequate for several scenarios. Hence, we compare the performance of *flockfs* with that of the *lookback*[12] reference file system. *lookback* redirects file system calls through fuse and hence provides the baseline for comparing the additional overhead imposed by *flockfs*. For our experiments, we used a MacBook pro running Mac OSX 10.6.2. This laptop had 4 GB of memory and a 320GB, 5400 RPM hard disk. We plot the read and write performance for various files sizes (average performance using various block sizes) relative to accessing the native HFS+ file system in Figure 15. Note that we used a logarithmic scale for the x-axis. We note that the fuse write performance was poorer than the read performance, especially for small files (less than 75MB). For small files, fuse achieved better read performance than from the native file system due caching; we did not disable the default fuse caching options. In general, the *flockfs* system achieved performance similar to the fuse file system.

5.6.1 Subjective performance evaluation

The objective system performance was adequate. The real challenge was in understanding the subjective effectiveness of the moderated collaboration model. Moderation operation is already popular. Contemporary campus users use email to share documents in an ad hoc fashion. In our application scenario, Alice emails her Word report to Bob, Emily and Tom. Bob uses these emails and manually incorporates Alice's report into his Powerpoint presentation. Bob is responsible for keeping track of all the email versions, there is no system support to maintain the update order. The definitive version is loosely defined by consensus among the various group members with no system support to ensure that Bob has seen the latest Word report. Earlier work [4], [34], [35] had highlighted the difficulties in using emails for group

12. http:code.google.com/p/macfuse/wiki/REFERENCE_FILE_SYSTEM

coordination. *flockfs* automates many of these operations. Our sample of students found the moderation operation to be intuitive. They felt that *flockfs* was mimicking many of the operations that they were already performing manually while providing additional support to deduce whether the document had converged. Further study is needed to evaluate the usability of moderation operation for general campus users.

6 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 editorially consistent versions of documents. Our implementation benefits from using two mature tools: fuse and git. *flockfs* is deployed within our group and the source code is freely available. Empirical usage data from a wider audience will be used to investigate automated moderation mechanisms.

Acknowledgment

Kevin Smyth helped us in collecting the availability traces. Nathan Regola implemented an earlier version of *flockfs* prototype. 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] R. M. Baecker, D. Nastos, I. R. Posner, and K. L. Mawby, "The user-centered iterative design of collaborative writing software," in *CHI '93*, 1993, pp. 399–405.

[3] C. M. Neuwirth, D. S. Kaufer, R. Chandhok, and J. H. Morris, "Computer support for distributed collaborative writing: defining parameters of interaction," in *CSCW '94*, 1994, pp. 145–152.

[4] V. Bellotti, N. Ducheneaut, M. Howard, I. Smith, and R. E. Grinter, "Quality versus quantity: e-mail-centric task management and its relation with overload," *Hum.-Comput. Interact.*, vol. 20, pp. 89–138, Jun. 2005.

[5] C. C. Marshall, "From writing and analysis to the repository: taking the scholars' perspective on scholarly archiving," in *JCDL '08*, 2008, pp. 251–260.

[6] L. Novik, I. Hudis, D. B. Terry, S. Anand, V. Jhaveri, A. Shah, and Y. Wu, "Peer-to-peer replication in winfs," MSR, Tech. Rep. MSR-TR-2006-78, Jun. 2006.

[7] A. Rajgarhia and A. Gehani, "Performance and extension of user space file systems," in *25th ACM Symposium on Applied Computing (SAC)*, Sierre, Switzerland, Mar. 2010.

[8] K. P. Puttaswamy, C. C. Marshall, V. Ramasubramanian, P. Stuedi, D. B. Terry, and T. Wobber, "Docx2go: collaborative editing of fidelity reduced documents on mobile devices," in *MobiSys '10*, 2010, pp. 345–356.

[9] 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.

[10] 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.

[11] V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, C. C. Marshall, and A. Vahdat, "Cimbiosys: a platform for content-based partial replication," in *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, 2009, pp. 261–276.

[12] 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: USENIX, Jun. 1994, pp. 183–195. [Online]. Available: http://www.isi.edu/~johnh/PAPERS/Reiher94a.html

[13] P. Kumar and M. Satyanarayanan, "Flexible and safe resolution of file conflicts," in *USENIX 1995 Technical Conference*. Berkeley, CA, USA: USENIX Association, 1995.

[14] 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.

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

[16] 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.

[17] 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. [Online]. Available: http://www.isi.edu/~johnh/PAPERS/Page98a.html

[18] C. A. Ellis and S. J. Gibbs, "Concurrency control in groupware systems," *SIGMOD Rec.*, vol. 18, no. 2, pp. 399–407, 1989.

[19] B. Nowicki, "NFS: Network file system protocol specification," RFC 1094, Mar. 1989.

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

[21] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer, "Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs," in *ACM SIGMETRICS*, 2000, pp. 34–43.

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

[23] S. Chandra and X. Yu, "An empirical analysis of serendipitous media sharing among campus-wide wireless users," *ACM Transactions on Multimedia Computing, Communications and Applications (ACM TOMCCAP)*, vol. 7, no. 1, p. 23, Jan. 2011.

[24] D. Tang and M. Baker, "Analysis of a local-area wireless network," in *ACM Mobicom '00*, 2000, pp. 1–10.

[25] D. Kotz and K. Essien, "Analysis of a campus-wide wireless network," in *ACM MobiCom '02*, 2002, pp. 107–118.

[26] M. Balazinska and P. Castro, "Characterizing mobility and network usage in a corporate wireless local-area network," in *MobiSys '03*, 2003, pp. 303–316.

[27] D. Tang and M. Baker, "Analysis of a metropolitan-area wireless network," *Wirel. Netw.*, vol. 8, no. 2/3, pp. 107–120, 2002.

[28] 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*, Aug. 1987, pp. 1–12.

[29] X. Yu and S. Chandra, "Designing an asynchronous group communication middleware for wireless users," in *MSWiM '09: Proceedings of the 12th ACM international conference on Modeling, analysis and simulation of wireless and mobile systems*. New York, NY, USA: ACM, 2009, pp. 274–279.

[30] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch, "Session guarantees for weakly consistent replicated data," in *Proceedings of the third international conference on on Parallel and distributed information systems*. IEEE Computer Society Press, 1994, pp. 140–150.

[31] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer, "Provenance-aware storage systems," in *USENIX '06*.

[32] K.-K. Muniswamy-Reddy and D. A. Holland, "Causality-based versioning," *Trans. Storage*, vol. 5, pp. 13:1–13:28, Dec. 2009.

[33] A. Gehani and U. Lindqvist, "Bonsai: Balanced lineage authentication," in *IEEE ACSAC*, Miami, FL, Dec. 2007, pp. 363–373.

[34] R. B. Segal and J. O. Kephart, "Mailcat: an intelligent assistant for organizing e-mail," in *AGENTS '99*, 1999, pp. 276–282.

[35] R. Boardman and M. A. Sasse, ""stuff goes into the computer and doesn't come out": a cross-tool study of personal information management," in *CHI '04*, 2004, pp. 583–590.

Surendar Chandra received the PhD degree in Computer Science from Duke University. His research interests are in experimental systems topics in multimedia, storage, security, networks and sensor systems. He held positions in academia at the University of Georgia and Notre Dame and in industry at the FX Palo Alto Laboratory. He was the recipient of a US National Science Foundation CAREER award and is a senior member of the ACM.