

# A Web-Service-based Open-Systems Architecture for Achieving Heterogeneity in Synchronous Collaborative Editing Systems

Jon A Preston and Sushil K Prasad  
Georgia State University  
*jon.preston@acm.org and sprasad@gsu.edu*

## Abstract

*Motivated by the need to support concurrent, collaborative access to shared documents, we have designed and validated an architecture that integrates existing and familiar systems for client editing software and document repositories. Through Web-services, we achieve an open system wherein numerous clients can use varied editing tools to fit their preferences and access documents distributed on a heterogeneous collection of document repository systems (configuration management systems, or CMS). Our simulation results on numerous client/server configurations validate our architecture and demonstrate an increase in concurrent access to shared documents; by adding a Lock Manager to the server, our system achieves a 67% reduction in check-out failures. Additionally, we present a novel algorithm that avoids operational transformation (OT) by utilizing a dynamic, hierarchical locking scheme that is transparent to the user. This algorithm maximizes concurrent access and enables edit caching to minimize communication costs.*

**Key Words:** CSCW, Web Services, Collaborative Editing Systems, Simulation, Heterogeneity

## 1. Introduction

Collaborative editing systems (CES) offer much promise in increasing the ability for groups of people to work together to create and modify documents. These documents can range from research papers to computer source code to CAD drawings to any other computer-managed document type.

We note that many feature-rich editing systems such as OpenOffice, Microsoft Office, and various integrated development environments (IDEs) such as Borland's JBuilder, Microsoft Visual Studio, and Sun's NetBeans have a large existing user base. Likewise, many configuration management systems (CMS) and document repositories such as RCS, VSS, and CVS are currently implemented worldwide and store a large collection of documents.

Our work brings these existing client and server technologies together in an open-systems architecture that allows users to retain their favored tools and leverage on existing document servers through the use of Web-services. [1, 2, 3] discuss Web-service-based approaches similar to our system but their systems are coupled to specific tools (IDEs) whereas our approach allows for the integration of any IDE, CMS, and communication tools; consequently, our architecture is more flexible.

Central to our motivation is the need to allow users to synchronously and asynchronously edit documents. When accessing documents synchronously, users typically are made aware of other users in the system [4]. Our architecture handles the negotiation of awareness and concurrent access transparently to the users such that they can focus on the work at hand without being hindered by check-in and check-out level minutiae.

For this research, we assume that users are editing the document synchronously, thus we examine how we can increase concurrent access to a shared set of document while utilizing a pessimistic locking scheme. We recognize that other technologies such as Wikis allow for asynchronous editing of shared documents and collaborative editing systems (CES) may employ operational transformation (OT) algorithms to ensure convergence among replicated copies of a shared document; but our architecture supports synchronous editing without the computational and communication overhead of OT-based approaches.

Section 2 of this paper discusses existing concurrency schemes and our improvements. Section 3 discusses our system architecture and how we integrate existing systems. We provide an overview of a new dynamic locking algorithm developed for our system in Section 4. Section 5 goes into the details of the simulation used to validate our architecture and the results achieved, and we offer conclusions and future work in Section 6.

## 2. Concurrency Schemes

The two dominant concurrency schemes employed in CES are operational transformation (OT) and locking [5, 6]. While the OT approach increases concurrent access by avoiding locking, OT requires continuous multicast of changes to all users within the system and thus incurs a large communication cost. Alternatively, locking removes the high communication overhead, but since only one user can access each shared document at a time, the concurrency of collaboration may be inhibited. To minimize communication costs, our system employs a lock-based scheme that allows increasingly finer-grained locking for contending users to sustain concurrency.

Traditional lock-based systems lock at a file level [7], but there is much potential to be realized if a finer granularity is adopted [8, 9]. While others [8, 9] have advocated fine-grain locking, no existing systems implement this advantageous feature.

By utilizing a finer granularity on the lock, one can reduce the portion of the document placed into each atomic element within the repository. Since each document within the repository contains less content, the probability of two users requesting the same element may be reduced. This is akin to breaking up a large file into smaller files, each of which may be checked out concurrently without being inhibited by the pessimistic locking policy.

Additionally, by adopting a fine granularity for locking, the system may aggregate documents together to form virtual files and other versioned objects from collections of other objects [8]. Most CMS systems currently employ the concept of a project (or directory), which is itself an aggregate [10]. Unfortunately, most existing CMS systems do not make it easy to aggregate objects from different projects. But fine-grain CMS systems allow for heightened aggregation as the reusability of each element is increased; if elements are decoupled from other elements, then reuse should increase [10].

## 3. Architecture

### 3.1. Achieving Heterogeneity

Given that many users have their own favorite editing software on the client side and there are many existing server-side repositories that contain documents, it is advantageous to create a system that can support the use of these existing technologies. Users are often hesitant to adopt new collaborative tools that don't have the same feature set or familiarity of their current tools [13]; as a result, we strive to provide a means by which a heterogeneous collection of existing client and server side technologies may be interconnected within a Collaborative Editing

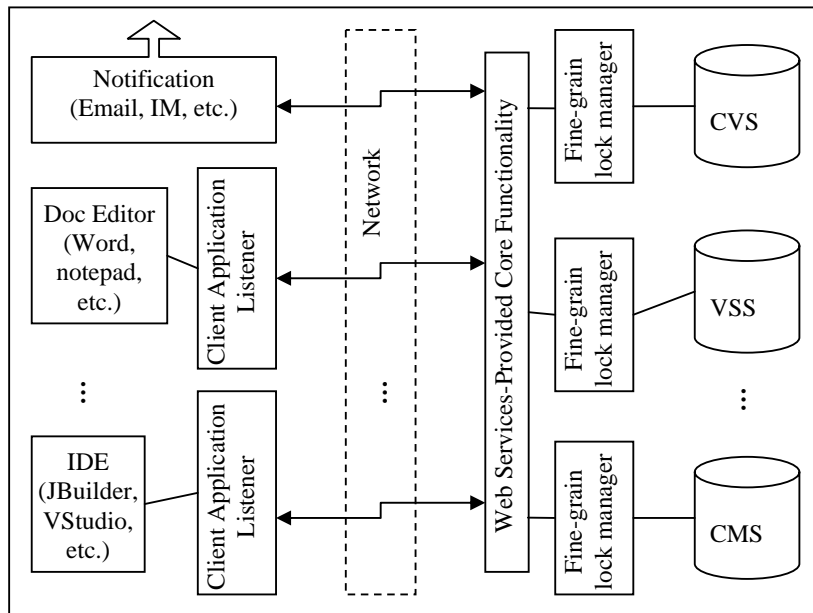


Figure 1. Web services collaborative architecture

System such that user can retain the use of their favored tools and connect to the plethora of existing server repositories.

Figure 1 demonstrates the approach of our architecture in allowing varied technologies to connect and work together in a CES. On the client side, different document editors such as Microsoft Word, notepad, Open Office, etc. can be used by different clients within the CES, yet each has a listener entity that translates local changes to the shared document to be replayed by other clients on their chosen applications. Similarly, the Web services API provides a consistent interface by which clients may request files for check-in and check-out; the specific server technology remains hidden, so it does not matter if CVS, VSS, or another CMS technology is adopted. To achieve heterogeneity among the clients, it is necessary that a client application listener be employed that can detect changes to the document, translate these changed into an application-independent format, and then send these changes to other clients via the server-side coordination Web service.

### 3.2. Overview and Components

As seen in Figure 2, our architecture consists of four components that connect with each other and to existing client applications and server repositories.

First, the **Client Application Listener** component connects to existing client applications such as MS Word and IDEs like JavaBeans so that users may use their current, preferred methods of editing. The role of this component is to listen to change events that occur within the application (edits to the document) and cache (if desired) and send on these changes to the server coordinating the collaborative editing among other users. This component also receives update notifications from the server and sends the changes to the client application, thus maintaining consistency among all users collaborating together.

Second, the **Web Service** component provides an API for traditional CMS systems (check-in and check-out, etc.) as well as an API for managing changes among the users that are collaborating together (insert, delete, move, etc.). This component also provides an API by which users can subscribe to receive synchronous and asynchronous notification when a document has been changed.

Third, the **Fine-Grain Lock Manager** component acts as a proxy that checks-out and checks-in documents from the existing server repository (such as CVS, VSS, etc.). This component receives check-in and check-out events from the Web Service component and processes and executes these requests via the existing server repository. This component provides the ability to manage artifacts at a finer granularity (viewing an artifact as a collection of sub-artifacts); as an example, a user can edit page one of a shared artifact at the same time another user is editing page two. This component tracks who is currently working on each artifact in the server

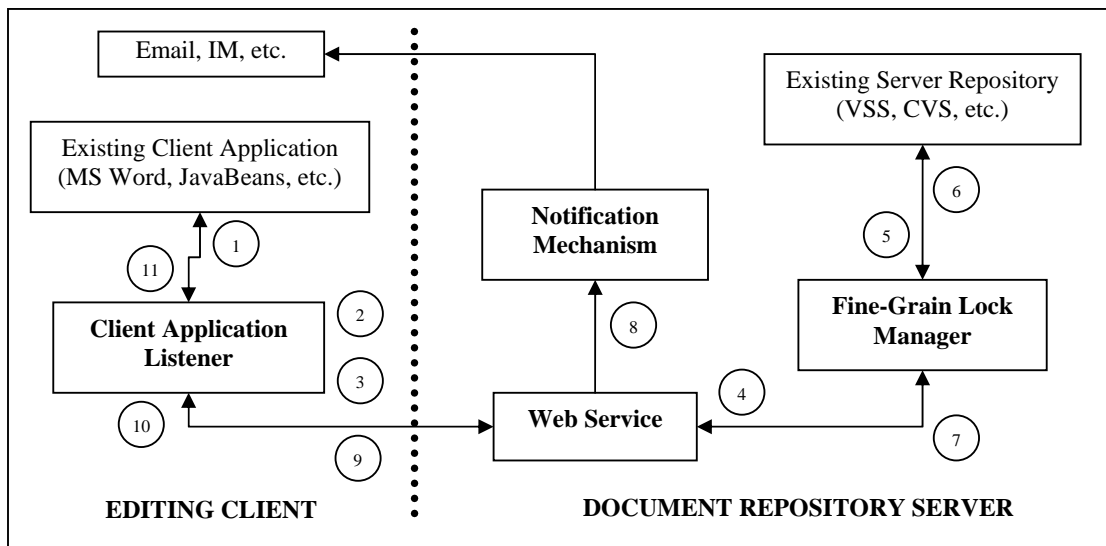


Figure 2. Open-system architecture for distributed repositories

repository and is thus able to “push” these changes to the necessary clients. A more detailed description of this component is provided in Section 3.2.

Fourth, the **Notification Mechanism** component is responsible for passing on any events that the user has requested notification of (document change, check-out, etc.) to the users’ preferred email, IM, etc. This component receives the event from the Web Service component and sends the notification to the client. Clients may subscribe for notification when changes are made (even if they are not currently editing the document); thus the system supports synchronous and asynchronous collaboration.

In summary, heterogeneous editors are able to coordinate by sending messages to the server via an established API. Since the server provides the common API, any client IDE can connect if it utilizes this API. The server propagates changes to other users and maintains consistency among all users’ copies of the artifact as needed. The system tracks who is currently working on each artifact in the server repository and is thus able to “push” these changes to the necessary clients.

### 3.3. Adding a Lock Manager to the CMS Server

The addition of the fine-grain lock manager proxy to the server machine allows for the addition of fine-grain check in and check out of artifacts. This lock manager intercepts messages from the network and processes them accordingly. The lock manager maintains a set of artifacts that have been checked out from the server; this stored database of artifacts also contains information about subsections within the artifacts. This subsection management allows a client to check out only a subsection of an artifact and allows other clients to check out other subsections. Consequently, the lock manager will only check in an artifact if there are no clients accessing the artifact. Assuming pessimistic locking, a check out request is only passed to the server from the lock manager if there are no other clients currently accessing the subsection being requested.

The result of this additional lock manager is that each artifact may be checked out simultaneously by different clients so long as the clients are accessing disjoint subsections of the artifact. Notice in this scheme, no change is required to the existing CMS system; the addition of multi-granular locking is transparent to the existing CMS system. Furthermore, if the existing configuration management system does not support replication of the files among multiple clients, then our approach adds this capability by checking the files out and in via lock manager; thus the existing CMS is only aware of one user (the lock manager) and the lock manager is then responsible for coordination among the clients.

### 3.4. Event Flow in the System

The following 11 events are illustrated in Figure 2. When a change event occurs in the client’s document editing application, a state update message (user edit of artifact) is sent (1) to the Client Application Listener. The Client Listener receives the update message and caches the change (2). When the cache must be flushed (when the cache is full or when another user enters the document as a reader), changes are sent (3) to the Web Service on the server via the network. The Web Service receives the updates and sends (4) them to the Fine-Grain Lock Manager to be processed. Upon receipt of a check-out or check-in message, the Fine-Grain Lock Manager updates its data store of users that must be notified of the change and may also send (5) the check-out or check-in message to the existing Server Repository. The Server Repository (an existing CMS or document server) processes the check-in or check-out and confirms (6) update of the artifact to the Fine-Grain Lock Manager. The Fine-Grain Lock Manager notifies (7) the Web Service component that the change has been committed (the check-in or check-out has succeeded). For each client subscribed for notification concerning this document being changed, the Web Service component sends (8) a message to the Notification Mechanism (which will notify the client). Additionally, the Web Service component selectively broadcasts (9) via the network change notifications to each client interested in the change (and client currently reading the document being modified). The Client Application Listener will receive the update notification (10) and cache it if the user is not currently viewing the updated section. When the client views the changed section of the document, the Client Application Listener flushes the update cache to the Client Application (11); this maintains consistency as the user views the content of the shared document.

## 4. Hierarchical Dynamic Locking

We have developed suitable data structure and algorithms for the Fine-Grain Lock Manager component that allow users multi-granular shared access to a document with no write conflicts [12]. All users see the same version of the document in real time, although this apparent synchronized view employs lazy updates to minimize client-server communications. The shared access to the document is transparent to the user in that the user is not required to explicitly request a lock for a section of the document; the system handles the lock request automatically. Similarly, locks are released automatically as necessary.

Based upon the semantic structure of the document, the document may be broken up into sections, subsections, paragraphs, sentences, words, etc. If the document being shared is a CAD drawing, it may be broken into layers, objects, etc. If the document is programming source code, it may be broken into classes, components, methods, blocks, etc. Thus we do not have any preconceived notion of what the sections of the document contain, nor do we require any specific depth/level of decomposition. Our approach works well with a variety of document structures. Note that the document tree consists of internal nodes that represent structure, and all document content resides at leaf nodes. We refer to sections (and subsections) of a document based upon their spatial relationships – denoting disparate sections based upon locality. Our approach is generalizable to define disparate elements of a document based upon some other, non-spatial criterion; for example, layers within a CAD document or drawing objects of different structure/types could be viewed as distinct and disparate regardless of the physical locality of the entities within the same locality of the document. Thus we use the term “section” to denote any non-conflicting section.

Our algorithm avoids the problem of merging two versions of a document by providing exclusive write access. Traditionally, lack of concurrency is a key limitation of systems that employ such exclusive write access to a shared document, but our system overcomes this lack of concurrency by using a multi-granular (i.e., multi-level) locking scheme that locks sub-hierarchies of the shared document.

Furthermore, each user gets exclusive write access to the largest sub-hierarchy possible to enable infrequent communication to server through relatively large messages delivering a bunch of local updates. Our system is novel in that it avoids the merge problem associated with systems that employ optimistic locking and improves concurrent access and throughput when compared to systems that employ pessimistic locking. The principle contribution of our work is an algorithm that manages these multi-granular locks and automatically increases and decreases the lock level in the document-tree hierarchy to maximize exclusive access to the shared document while minimizing communication costs.

Two principles guide the behavior of our system:

- When you are writing to the document, you have exclusive write access to that section of the document that you are modifying (i.e., you have a lock on the node that represents the section of the document).
- When you are reading a section of the document, you always have the most recent (fresh) copy of the content at the node that represents that section.

Because of the first principle, we must provide users with mutual exclusion and the ability to lock a node in the structure. Because of the second principle, we must provide updates to all interested users that are viewing a given node (i.e. when the content at that node is changed, the change is broadcast to the users viewing that node’s content). Alternatively, if a user  $u_i$  updates a section of the document at node  $n_i$  but no other user is viewing that node’s content, then the changes may remain local in cache on the machine of user  $u_i$ . This “dirty cache” must be flushed to the server when another user request access to the node  $n_i$  – either through a write (lock) request or a read request.

It is advantageous to maintain a lock on the largest sub-tree that is permissible; a lock on a sub-tree rooted at node  $n_i$  is permissible for user  $u_i$  so long as no other user has a lock on any node within the tree rooted at node  $n_i$ . By maximizing the sub-tree that any user owns, we minimize the communication costs of the system with regard to cache updates.

For example, if a user  $u_i$  owns the entire tree (the entire document), then all changes to the document can be stored locally in the user’s cache. If another user  $u_j$  enters the system and requests a section of the document, then the section of the tree owned by user  $u_i$  is reduced to accommodate the insertion of user  $u_j$  (if possible). Only that portion of the tree that has been modified (marked dirty cache) by  $u_i$  that are part of the sub-tree now

owned by  $u_j$  must be sent to  $u_i$ ; the other portion of  $u_i$ 's cache remains local to  $u_i$ . The result is a minimization of messaging within the system by reducing cache updates/flushing.

OT can be integrated into our algorithm, and communication costs remain minimized by avoiding multicasting in favor of select broadcasting. For example, if only three users are within a section  $S_i$  of the document and another 15 users are in other sections, the OT-based updates related to  $S_i$  need only be sent to the 3 users within the section; the other 15 users can be updated later (and only if they are interested) to achieve consistency among all users. In this way, users maintain consistency on an as-needed basis.

## 5. Simulation and Results

To validate our architecture and experimentally determine whether this lock manager approach could improve concurrency, we simulated two configurations of our architecture – one in which the middleware was absent (as in a traditional distributed repository) and one in which the lock manager was present (as serving to implement fine-granular locking). We utilized the discrete event DEVS Java simulation framework for this study [11].

Both simulations connected numerous clients to a set of servers hosting CMS (document repositories) through a network. Clients would simulate users requesting documents, editing a document once owned, and returning the document to the repository when the edits were completed (checking the document back in).

The second simulation configuration was identical to the first except that this system added a lock manager component to the server that intercepted document requests from clients and processed these requests as a proxy to the server; this component is shown as a dashed box in Figure 3 to denote that it was not present in the original simulation configuration. These simulation configurations are illustrated in Figure 3; note that if the lock manager was not present, the Web Services API would communicate directly with the document repository (CMS).

Figure 4 shows the architecture implemented in the DEVS Java simulation package. The simulation was designed so that client users and servers could be added easily upon initial configuration; the lines connecting the components denote discrete event message paths within the simulation (i.e. requests for check in and check out, success or fail messages from the server, etc.), thus that the entire collaborative editing system was modeled accurately. In Figure 4, the lock manager component is shown and labeled as “middleware.”

On the left side in Figure 4, you see a set of clients that represent the users in the CES; we did not specify which editing software/applications the clients were using – we simply send the check-out and check-in requests denoting that the clients desire to edit (check-out) and are done with editing (check-in). The network

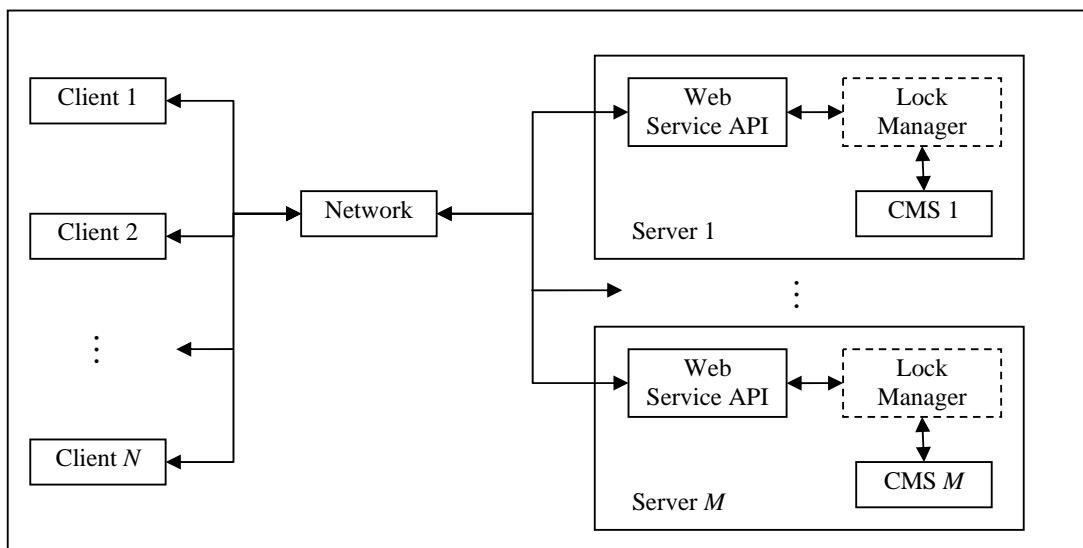


Figure 3. Simulation configuration (shown with middleware proxy)

entity connects the clients to the servers. On the right side of Figure 4, you see a set of servers; we allow the set of documents to be spread over a heterogeneous set of servers, thus each server publishes a Web Service API that standardizes how clients may request check-ins and check-outs of documents. Notice that it is transparent to the clients as to whether the server is running any particular configuration management software (RCS, CVS, VSS, etc.). The connecting lines in Figure 4 denote the message paths from clients through the network, from the network to the servers, and internally within the servers' Web Services API to the lock manager/proxy and then to the existing/legacy CMS. Also note that the lock manager/proxy was only present in the section version of the simulation; it was left out in the first version of the simulation to see if the addition of this proxy improved check-out fail rates.

There are three types of clients in the simulation: random, clustered, and hybrid. These clients represent the broadest range of edit patterns among users/editors within a collaborative editing session.

- The random client has a high probability (90%) of selecting a new random artifact from the repositories from the full range of all of the documents.
- The clustered client is programmed to exhibit a localization policy in that it remains within a close proximity to a single document. We achieve this by sequentially numbering the documents, so this client would check out documents numerically close to its currently preferred document.
- The hybrid client is programmed as a mixture of the clustered and random client behaviors – behaving like each of them 50% of the time.

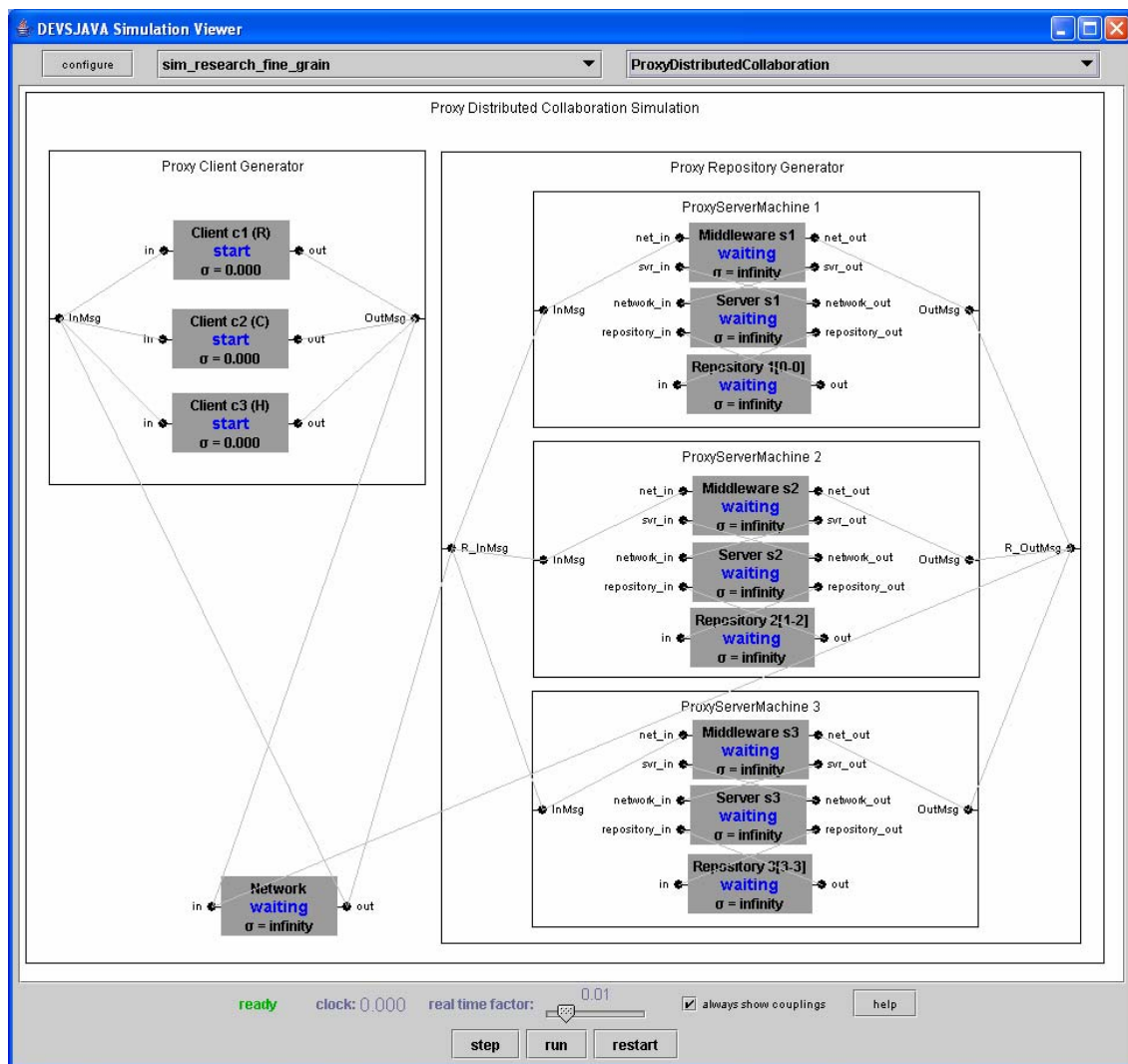


Figure 4. DEVS Java simulation (shown with “middleware” lock manager)

The simulation was run in nine configurations for each of the two versions of the simulation (for a total of 18 runs). Table 1 shows the various configurations. The number of iterations is defined by the number of iterations for which the simulation was run (all time advances). The client distributions denote how many of each type of client (random, clustered, and hybrid) were in the system when the simulation was run; for example, for test 1, there was one client of each of the three types. The repository distributions denote how many artifacts existed at each server and how many servers existed in the system; for example, in tests 1-4, there was one artifact at server 1, two artifacts at server 2, and one artifact at server 3.

**Table 1. Simulation test configurations**

Test	Iterations	Client Distribution (# per type)			Repository Distribution (# Artifacts at each Server)								
		Random	Clustered	Hybrid	S1	S2	S3	S4	S5	S6	S7	S8	S9
1	500	1	1	1	1	2	1						
2	500	3	0	0	1	2	1						
3	500	0	3	0	1	2	1						
4	500	0	0	3	1	2	1						
5	500*	1	1	1	10	20	10						
6	500	10	10	10	30	50	80	30	30	40	40	100	100
7	5000	10	10	10	30	50	80	30	30	40	40	100	100
8	2500	10	10	10	15	25	40	15	15	20	20	50	50
9	5000	1	1	1	1	2	1						

\* Test 5 for the fine-grain version was run to 5000 iterations to obtain lock failures

**Table 2. Simulation results**

Test	Check-out Fail Rate		Improvement
	Without Fine-grain Locking	With Fine-grain Locking	
1	32.75%	7.27%	78%
2	23.33%	11.67%	50%
3	26.92%	6.38%	76%
4	19.64%	7.02%	64%
5	2.00%	0.75%	63%
6	16.39%	5.81%	65%
7	7.91%	2.62%	67%
8	9.08%	2.99%	67%
9	26.55%	7.24%	73%

As shown in Table 2, check-out fail rates for the simulation configuration without the fine-grain locking ranged from 2% (test 5) up to 32.75% (test 1). Check-out fail rates for the simulation configuration with the fine-grain locking ranged from 0.75% (test 5) up to 11.67% (test 2).

In all configurations, the version of the simulation that contained the fine-grain lock manager significantly outperformed the other version (without the lock manager) in reducing the number of check-out failures (collisions). The minimum improvement when adding the fine-grain locking in reducing check-out failures occurred in test 2 (50% improvement), and the maximum improvement occurred in test 1 (78% improvement). The average improvement in reducing check-out failure as a result of adding the fine-grain locking was 67%.

This work has shown that the hypothesis behind adding middleware to existing repository management systems is sound and that fine-grain management of artifacts via proxy does improve the reduction of failed check-outs (collisions) among multiple users in a distributed collaborative system.

In all test scenarios, dramatically fewer check-out failures occurred in the fine-grain locking version of the simulation as compared to the initial version of the simulation without fine-grain locking. This is as expected as the middleware, fine-grain version of the simulation effectively increases the number of artifacts (via subsections of the artifacts) that clients are able to simultaneously check out; this is due to the fact that checking out a subsection of an artifact does not preclude another client from checking out a different subsection of the same artifact.

Additionally, this work shows that the number of failed check-outs is related to the relative density of clients when compared with artifacts; note that test 1 and 5 differ only in the number of artifacts stored in the server machines (by a factor of 10). The check-out fail rate decreases dramatically as the number of artifacts is increased. This is as expected since the clients have a wider range of artifacts from which they may select.

The results also indicate that the improvement in moving from the initial simulation to the fine-grain enabled simulation is comparable regardless of the number of iterations to which the simulation is run. This claim is supported by examining the comparable improvements between test 6 and test 7 (in which only the iterations was changed).

Additionally, the results indicate that the concurrency is maximized at some number of artifacts relative to the number of clients. Examining the difference between test 7 and test 8, the decrease in the number of artifacts by 50% does not show any appreciable difference in the improvement rate. Consequently, we may infer that both of these tests had a sufficiently large set of artifacts from which the clients could make use that the check-out failure rate was not affected by the reduction in the number of artifacts. It is interesting to note that the improvement rate is still significant when the lock manager is added, even though the number of artifacts is large enough to handle the client requests well in both simulation configurations.

## 6. Conclusions and Future Work

We have presented a novel architecture and simulation results demonstrating that our approach of adding a fine-grain lock manager to existing configuration management systems reduces the check-out fail rate within a collaborative editing session.

The architecture presented allows heterogeneous client and server technologies to connect and support collaborative editing. The simulation results obtained validate the architecture, and our hierarchical locking mechanism ensures maximum concurrent access while minimizing communication costs.

We now focus on implementation of the client listener hooks, the Web-services API, and the dynamic locking manager. Once implemented, we intend to connect to a text editor as a prototype and then to an Microsoft Office editor to demonstrate the viability of the client listener. Additionally, our Web-service and lock manager proxy will be deployed to an existing Visual Source Safe (VSS) document repository server to which our client hook can connect. We will perform load testing to further validate the architecture and hierarchical locking algorithms are robust under real-world constraints.

## 7. References

- [1] A. Mehra, J. Grundy, and J. Hosking, "Supporting Collaborative Software Design with a Plug-in, Web Services-based Architecture", *Workshop on Directions in Software Engineering Environments (WoDiSEE) from Proceedings of the 26<sup>th</sup> International Conference on Software Engineering (ICSE'04)*, IEEE, Edinburgh, Scotland, 2004.
- [2] M. Younas and R. Iqbal, "Developing Collaborative Editing Applications using Web Services", *Proceedings of the 5<sup>th</sup> International Workshop on Collaborative Editing*, Helsinki, Finland, September 115, 2003.
- [3] V. Bharadwaj and Y. V. R. Reddy, "A Framework to Support Collaboration in Heterogeneous Environments", *SIGGROUP Bulletin*, (24) 3, Dec. 2003, pp. 103-116.
- [4] C. Gutwin and S. Greenberg, "The Importance of Awareness for Team Cognition in Distributed Collaboration", *Team Cognition: Understanding the Factors that Drive Process and Performance*, APA Press, Washington, pp. 177-201.

- [5] C. Sun, X. Jia, Y. Zhang, , Y. Tang, and C. Chen. "Achieving convergence, causality-preservation, and intention-preservation in real-time cooperative editing systems", *ACM Transactions on Computer-Human Interaction*, (5) 1, Mar 1998, pp 63-108.
- [6] C. Ellis and S. Gibbs. "Concurrency Control in Groupware Systems", *ACM SIGMOD Conference on Management of Data*, May 1989, pp. 399-407.
- [7] Cederqvist, P. "Version Management with CVS", Available from info@signum.se, 1993.
- [8] M.C. Chu-Carroll, J. Wright, and D. Shields, "Supporting Aggregation in Fine Grain Software Configuration Management", *SIGSOFT 2002/FSE-10*, Charleston, SC, USA, November 18-22, 2002, pp. 99-108.
- [9] B. Magnusson, U. Asklund, and S. Minör, "Fine-Grained Revision Control for Collaborative Software Development", *Proceedings of ACM SIGSOFT'93 – Symposium on the Foundations of Software Engineering*, Los Angeles, California, December 1993.
- [10] Microsoft Corporation. Visual Source Safe. Available at <http://visualsourcesafe.com>, February 2005.
- [11] B. P. Zeigler and H.S. Sarjoughian. "Introduction to DEVS Modeling & Simulation with JAVA: Developing Component-based Simulation Models", Technical Document, University of Arizona. 2003.
- [12] J. Preston and S. Prasad. "A Deadlock-Free Multi-Granular, Hierarchical Locking Scheme for Real-time Collaborative Editing", *The Seventh International Workshop on Collaborative Editing Systems*, ACM, Sanibel Island, Florida, USA, November 2005.
- [13] B. Korel et al. "Version Management in Distributed Network Environment", *Proceedings of the 3rd International Workshop on Software Configuration Management*, pp. 161-166, May 1991.