JPernLite: Extensible Transaction Services for WWW

Jingshuang Yang
Department of Computer Science
Columbia University
New York, NY 10027, USA
Tel: 1-212-939-7085
E-mail: jyang@cs.columbia.edu

Gail E. Kaiser
Department of Computer Science
Columbia University
New York, NY 10027, USA
Tel: 1-212-939-7081
E-mail: kaiser@cs.columbia.edu

CUCS-009-98

Abstract

Concurrency control is one of the key problems in design and implementation of collaborative systems such as hypertext/hypermedia systems, CAD/CAM systems and software development environments. Most existing systems store data in specialized databases with built-in concurrency control policies, usually implemented via locking.

It is desirable to construct such collaborative systems on top of the World Wide Web, but most web servers do not support even conventional transactions, let alone distributed (multi-website) transactions or flexible concurrency control mechanisms oriented towards teamwork – such as event notification, shared locks and fine granularity locks.

We present a transaction server that operates independently of web servers or the collaborative systems, to fill the concurrency control gap. The transaction server by default enforces the conventional atomic transaction model, where sets of operations are performed in an all-or-nothing fashion and isolated from concurrent users. The server can be tailored dynamically to apply more sophisticated concurrency control policies appropriate for collaboration. The transaction server also supports applications employing information resources other than web servers, such as legacy databases, CORBA objects, and other hypermedia systems. Our implementation permits a wide range of system architecture styles.

KEYWORDS: Distributed Transactions, Extended Transaction Models, WWW, Computer Supported Collaborative Work, Middleware

1 Introduction

Concurrency Control (CC) is an essential functionality of collaborative systems such as CAD/CAM systems, distributed authoring systems, software development environments and hypermedia systems. In such systems, users can simultaneously access the data objects, and their access must be coordinated in order to maintain data consistency and the correct semantics of user operations.

The concept of transactions [12] from database systems gives most of the features needed by the builders of these collaborative systems, such as atomic operations, locking, and crash recovery. But these systems should also be enhanced by extensible CC policies beyond the traditional transactions due to the long-duration, interactive and open-ended nature of the tasks performed. Extensions for multi-participant tasks can range from the persistent locks of the check-out/check-in model to shared locks for closely cooperating users, to a sophisticated implementation of one of the Extended Transaction Models (ETM) proposed in the literature [21]. Existing systems usually employ the CC features of the underlying storage facility, normally a database system [4, 14, 43]. The CC features built into the database might then be customized to the needs of the specific application.

The WWW is by far the largest and most popular distributed hypermedia system, and very attractive as an infrastructure for a variety of applications. Building Collaborative Hypermedia Systems [1], CSCW Systems [9], and Workflow Systems [28] on top of WWW have been research topics for several years and have recently moved into the commercial arena.

However, the CC features that are required by such collaborative systems are missing from most WWW servers. There is no standard underlying database system; indeed, most web servers store their pages directly in the file system. Further, most web servers are intended as merely read-only data sources: the hypermedia stored in them can be modified only via the underlying file system, not through the web interface, i.e., HTTP. Only a few web servers (such as jigsaw [45] from W3C) allow the PUT method, and thus support updates via HTTP, even though PUT is defined in the original HTTP specification [10].

There have been attempts to build CC capabilities such as locking, check-out/check-in, and even distributed transactions into the WWW infrastructure. There are two major approaches, categorized according to where the bulk of the functionality is placed, the server side or the client side. The server approach [40] introduces CC functionality into the web servers so that the clients can explicitly submit operations as part of a transaction; a collection of cooperating web servers can then realize distributed transactions. The client approach [27, 41] places CC policies into the clients, but still often requires the servers to provide very basic CC building blocks such as locks or versions.

Neither of these approaches addresses the problem of how to dynamically extend the range of CC policies available to applications. In the realm of collaborative systems, hard-wiring CC policies into either the server or client greatly limits scalability and flexibility. For example, let us consider a web server that implements a set of CC policies needed by a Software Development Environment (SDE). It might have specific rules such as whenever someone is changing a design document, other users should get a copy of the working draft when they try to read the document. However, when a CAD system shares the server with the SDE, it wants to let its users see only the stable version of the design document instead. The web server that has been built for the SDE would not allow such flexibility.

We present a third, middleware approach: CC policies are realized in a new component, the external transaction server. The clients make requests to the transaction server, which carries out the concurrency control policies in the process of obtaining the data from the web servers on behalf of the clients. The web servers are not required to provide any CC support at all, although any support they do happen to provide can also be exploited. The major advantages of this approach are:

- The transaction server can easily be tailored to apply the desired CC policies of specific client applications.
- The approach does not require any changes to either the servers or the clients in order to support the standard transaction model, but it can exploit any CC features the servers or clients do provide to accomplish more sophisticated CC policies. The transaction boundary delimitation could either be done automatically by a third-party system (such as a proxy, as we discuss in section 7.1.2) or by the client.
- Coordination among clients that share data but have different CC policies is possible if all of the clients use the same transaction server (or, in planned future work, allied transaction servers).

This approach has its own challenges and limitations, though: It must operate together with different web servers supporting different CC capabilities and/or different variants of HTTP, and may need to "speak" non-HTTP communication protocols to encompass information resources other than standard web servers. The transaction server must have the capability of being customized dynamically to support the CC policies needed by the applications. And, most severe, since the transaction server is outside of and independent from the web servers and other information resources, the transaction server has no firm control over the data and thus data consistency guarantee is hard to achieve (see Section 7.2.1).

We present a prototype transaction server, called **JPernLite**, which solves all except the last problem – which is impossible to solve in the general case without changing to the servers. In the following sections, we first analyze the need for extensible CC policies and independent CC components for web-based collaborative hypermedia systems. Then we introduce the external transaction service model, and our design and implementation of JPernLite, followed by a discussion of some issues identified by our research.

2 Requirements Analysis

We analyze the requirements from two sides: the features needed by collaborative systems in general, and the specific capabilities needed to exploit WWW as the primary information resource.

2.1 CC Features Required by Collaborative Systems

The concurrency control features required for collaborative systems, especially those for hypermedia systems and software development environments, have been identified in [5, 42]:

- *Event notification:* notifying collaborating users of data operations that may interest them.

- *Fine-grained notification:* the ability to distinguish operations on individual attributes of a datum.
- *User-controlled locking:* locking is done explicitly by the users or applications, not implicitly by the data sources. In such systems, the semantics of an operation (e.g. check-out a document) is only understood by the client or even the human user. Therefore the backend data source wouldn't know what locks to put on when such an operation is requested.
- *Fine-grained locking:* locking part of a datum.
- *Shared locks:* allowing collaborating users to hold locks that are normally considered conflicting (e.g., two write locks, or a read and a write lock).
- *Persistency:* collaboration information such as the state of on-going transactions should be persistent, i.e., survive failures, so that the clients can recover from server crashes without losing valuable work (most transaction managers roll back in-progress work to its initial state after a failure).
- *Synergistic cooperation:* several users might have to exchange knowledge (i.e., share it collectively) in order to be able to continue their work. These users may pass shared objects back and forth in a way that cannot be achieved by a serial schedule.

2.2 Requirements Imposed by the WWW

In addition to these requirements, extensibility of CC policies is also very important for web-based collaborative systems. When a traditional database is used as the storage, it is likely to be dedicated to one or a few known applications; therefore one could hardwire the desired CC policy(ies) into them. However, a web server is likely to be utilized by multiple unrelated applications, which might be unknown to the website administrators. In fact, the applications themselves might not know in advance which websites will be accessed. So the CC policies these applications might require cannot be known to the requisite web servers a priori. Thus it is necessary to be able to extend the supported CC policies dynamically, while collaborative systems are operating.

Building collaborative systems on top of WWW (as opposed to dedicated databases) also brings the following requirements because of the peculiarities of the WWW infrastructure:

2.2.1 Large Scale and Heterogeneity

With more than two million websites [32] and hundreds of millions of web pages, WWW is orders of magnitude larger than any other existing distributed database in both the number of objects and the number of sites. Therefore, the ability to scale up to a huge number of participating sites is critical to a collaborative hypermedia system that intends to apply to WWW in general, beyond a few specialized web servers.

Although all web servers speak HTTP, the kind of CC support they provide, if any, can be very different. Furthermore, some legacy database systems have been modified to respond to HTTP requests [23], making the set of

apparent web servers more diverse. Because the data an application accesses may be stored in many different kinds of repositories, an application should not be hard-wired to assume any specific capabilities from its data sources, in particular, communication protocols and CC features.

Also, web servers are distributed around the world and the global network is far less stable and reliable than a local area network in terms of round-trip delay and accessibility. The traditional recovery algorithms might not work well under such circumstances. For example, rolling back operations by rewriting the value of each modified object would be much more expensive than in a local database case, and an application should consider using any "undo" or other relevant facilities offered by the data sources.

2.2.2 Distributed CC Support

Web servers are moving towards supporting updates (i.e., the HTTP PUT method) in addition to reads and posts (i.e., form submission for processing by CGI scripts). However, most mainstream servers still do not support distributed CC features such as:

- *Distributed commitment protocol*, e.g., the conventional two-phase commit, to support transactions that include accesses to multiple websites;
- *Failure recovery algorithms*, especially client-side recovery to restore consistent state in the client application;
- *Web page locking*, to prevent unintended overwrites and dirty reads, and *versioning*, to permit other users to access an older version, e.g., while the latest version is locked.

2.2.3 Application Semantics

The data model of WWW is very simple: everything is a web document that can be read and written, with the main "typing" embedded in filename extensions and MIME types indicating the appropriate viewer plug-ins. However, for a web-based collaborative system, a semantically richer data model, including a broader range of operations that can be performed on the data, is desired. For example, a web-based software development environment probably wants to model web documents as source code, design document, test report, code inspection report, etc., even though these may all be the same "type" from the web browser's perspective (e.g., text or HTML).

In the following section, we describe related work that addresses part of the concurrency control problem by satisfying a few of the above requirements. Our own work concentrates on extensible CC policies and application semantics, while the scaling and heterogeneity problems are addressed to a lesser extent.

3 Related Work

3.1 Three-Tier Architecture Transaction Servers

In a Three-Tier Architecture, client, data source and application logic are separated and implemented as independent components of a system. The application logic includes Transaction Processing (TP) monitors in some cases. Separating the transaction manager from the clients and the data sources makes it more flexible and scalable, which is the same in our approach. However, these TP monitors do not address the need for ETMs.

3.1.1 Encina and Tuxedo

Encina, which is now part of IBM TXSeries4.2 [27], is based on Open Software Foundation's (OSF) Distributed Computing Environment (DCE) standard. TXSeries provides concurrency control, transaction management, data modeling and application distribution through its toolkits. The Encina Monitor is a full-featured TP monitor that controls and monitors applications in a distributed environment. It allows servers to be replicated to increase availability and performance and supports load balancing and automatic restart of failed application servers.

BEA Tuxedo [7], another TP monitor, offers both library-based and language-based programming interfaces to its clients. The library-based API, the Application to Transaction Manager Interface (ATMI), is a superset of X/Open XATMI [35] provided in C or COBOL libraries; the language-based interface is an RPC facility called TxRPC. Both interfaces provide three major transaction primitives: `tpbegin`, `tpcommit` and `tpabort`. The API set also allows one to use synchronous, asynchronous, pipelined, forwarded, queued, and conversational mechanisms to structure the distributed applications. In addition to transaction management, BEA Tuxedo also integrates various essential services, such as name service, event service, queuing service, security management and load balancing.

These TP monitors qualify as "External Transaction Manager" as we define later in Section 4.1, in that they are third-party components that operates independent of data source and clients. However, because they do not support ETMs, the applications built on top of them is limited to use the atomic transaction models, and thus enjoy less flexibility in transaction management.

3.1.2 Object Management Group OTS and Implementations

The Object Transaction Service (OTS) specification [33] is one of the CORBA [34] services defined by OMG. Transactional objects that know how to participate in distributed two-phase commit and enforce ACID transactions are the building blocks of the service to the transactional clients. The transaction service issues transaction context objects that are used in further transaction operations to identify transactions, and separate transaction boundaries (i.e., delimit operations that belong to different transactions). The transactional objects use Recoverable Resource Objects (RRO) that know how to rollback changes in case of a transaction abort. If the data source is remote or does not know how to rollback, the RRO must implement the rollback. The OTS service provides an easy to use interface

to transactional client programmers who want to exercise simple atomic flat transactions, with support for a simple form of nested rollback. It also provides transparent distributed transaction support, assuming distributed computing communication through a CORBA ORB and/or IIOP.

There have been several implementations of OTS; here we describe two major ones, from Sun Microsystems and from Microsoft.

Enterprise Java Beans™ (EJB) [37] of Sun Microsystems defines a set of APIs that target the creation of component-based, transaction-oriented applications. Java™ Transaction API (JTA) [38] is required by EJB as the client-side API, which consists of methods to explicitly start, commit and abort transactions. The Java™ Transaction Service (JTS) [39] is a lower-level API intended to be used by server component builders to provide transaction-aware system infrastructure. JTS is a Java™ mapping of OMG OTS.

MTS [30] is the transaction service option package of Microsoft Windows NT 4.0 that realizes a three-tier architecture. The transaction service is applied to client requests in an implicit way, called automatic transactions. Each back end component, which implements a module of application logic, is marked as different types of transaction support, such as **Requires New, Required, Supported,** or **Not Supported**. Based on the type, when a back end component is accessed, MTS automatically starts, switches, or temporarily goes out of transactions. For example, if a component is marked **Required**, MTS would start a new transaction if the current execution is not already encompassed in a transaction. Through the use of the component type makers, one can build hierarchical transaction models such as nested transactions.

These transaction servers share the same architecture as our approach. However, they support primarily the vanilla, atomic transaction model. The only enhancement is that the transactions can be grouped together as in nested transactions, and thus can be rolled back in a nested fashion. Options of nested transaction model such as Intra-transaction parallelism are not supported. Another common characteristic of these transaction servers is that they assume the back end data source to be transaction-aware. For example, recovery (undo) and two-phase commit capability are required. With such requirements, these transaction servers can give stronger guarantees of data consistency. Our approach, on the other hand, does not require the back end data to provide any particular support. Of course, if we were to impose the same requirements, we can achieve the same guarantees.

Another important issue is transaction awareness of the clients. In both EJB and the current implementation of our approach, the clients need to explicitly make requests such as begin transaction and commit transaction. But in MTS, the clients simply access the back end components via COM and the MTS automatically encompass the access in transactions. The transaction structure and context, as we described above, are totally defined by the back end components. Our approach can be enhanced by a client side helper (such as an applet or a browser plug-in) that

makes the transaction requests for a client that is not transaction aware. In this case, the client side helper would need to intercept the client's data requests and wrap them in transactions.

3.2 WWW standards and protocols

There are various Web standards and protocols about concurrency control and distributed transactions. In our approach, we do not try to modify or replace any of them, but rather follow the standards and work with the clients and servers that speak such protocols. In this section we mention two very important ones: WEBDAV that deals with concurrency control in web servers, and TIP that deals with distributed transactions.

3.2.1 WEBDAV

Web Distributed Authoring and Versioning (WEBDAV) [41] is an Internet draft standard for web servers that intend to support locking (and versioning, which is under-construction) of their web pages. WEBDAV defines an HTTP extension that the web servers should speak in order to provide such services. A transaction server can leverage these features: when a transaction locks a web page, the transaction server not only locks it internally, but can also place a real lock at the web server. This way, even clients that do not go through the transaction server also share the concurrency control states.

As of the date of writing, there exists only experimental implementations of WEBDAV, and some of them are obsolete according to the latest WEBDAV specification [19]. However, several companies are developing WEBDAV-capable products such as WebSite Director from CyberTeams [17]. We expect to see more web servers supporting WEBDAV, therefore how to utilize the CC features these servers provide is an important issue (see section 6.1).

3.2.2 Transaction Internet Protocol (TIP)

TIP [40] is another Internet draft standard. It proposes a simple two-phase commit protocol to support atomic transactions across distributed nodes (the TIP term for servers) on the Internet. The transaction server can leverage the commit protocol implemented by a TIP-compliant web server to implement its own form of distributed transactions (which might encompass non-standard CC policies, not addressed by TIP). A sample implementation of TIP, the Internet Travel Bureau [16], has been developed in Tandem, a division of Compaq.

3.3 Extended Transaction Modeling Formalisms

If one could define a concise, powerful and executable ETM definition language that can express all possible ETMs, the implementation of transaction managers would be simplified to writing a program in such language and execute

in such a language machine. However, the two most famous works we describe in this section, ACTA and Asset, did not prove to satisfy all of these needs. Both of them are not implemented and not proved to be extensible enough. In our approach, we let the ETM designer to define ETMs in Java, a general purpose programming language, with a helpful transaction engine API.

3.3.1 ACTA

ACTA [15] is a framework that allows one to specify the effects of one transaction on other transactions (how the transactions are structured) or data items (how the data objects are modified). It is a comprehensive framework used for formally defining the properties of ETMs. ACTA defines transaction models through axioms, or invariants, that make assertions about the possible histories of a particular transaction model. It uses the notions of history, dependency, view, conflict and delegation as basic constructors to describe the transaction effects.

The authors of ACTA have used it to describe several ETMs, including nested transactions [20], Sagas [22] and Split/Join transactions [36]. Although ACTA is useful for validating transaction models, because it defines transactions in the form of axioms, it does not help much in producing an operational transaction model that supports the model.

3.3.2 Asset

Asset (A System for Supporting Extended Transactions) [13] proposes a set of transaction primitives that can be used in writing application programs. The transaction primitives, such as `commit, abort, delegate` and `permit,` allow one to create transactions, delegate objects between transactions and permit dirty reads among them. It has been shown in [13] that several kinds of ETMs can be implemented with this set of primitives.

In the Asset approach, transaction models are coded into the applications rather than bundled with the data sources. The policies concerning data sharing among applications are defined by the individual applications. Therefore, an application has to know in advance (when the application is coded, using the transaction primitives) what other applications it might be sharing data with, code specifically for those collaborative partners, so cannot dynamically form collaboration teams. Also, the system was never implemented[1], and therefore evaluation is rather difficult.

---

[1] Personal communication between the second author and the authors of Asset shows that Asset was not implemented

```
From: nhg@allegra.att.com (Narain Gehani)
To: kaiser@westend.psl.cs.columbia.edu
Subject: Re: ASSET

asset was not really implemented.
i can give you Ode though.
Narain
```

3.4 Extended Transaction Model Realization

3.4.1 Hyperform

Hyperform [44] is a toolkit that assists development of dynamic, open and distributed multi-user hypermedia systems. A few hypertext systems, including HyperDisco [43], have been built using Hyperform. Hyperform allows programmers to extend the data model and data management policies by subclassing the `Object` class of its OODB. Limited extensibility in CC policies is also permitted by subclassing the `CC Object` class. As hinted by [44], the possible extensions apparently do not include shared locks, since `the CC Object` provides short database-style ACID transactions that are directly integrated with the lower-level OODB. Also, it is not clear how this approach could be applied to WWW, where the web servers may not be under the control of the hypermedia system developer -- meaning users cannot extend the CC policies by changing the code of the web servers.

3.4.2 PJama

PJama [3] is an on-going research and development project at Sun Labs. It provides orthogonal persistency for Java™ programmers by transparently making all objects and classes reachable from a given root object persistent. The transaction support in PJama has not yet been fully designed and implemented, but through the first author's knowledge by taking part in the project during summer 1998, PJama is intended to support ETMs.

The goal of PJama's transaction support [18] is to provide type orthogonal, code independent, dynamically customizable transaction modeling support, with high performance, in Java™ VM. It separates transaction modeling and application development by defining two classes of programmers: The transaction programmers use the transaction facilities provided by the VM to design and implement ETMs by extending a system class `TransactionShell`. The application programmers simply instantiate the extended `TransactionShell` classes and run their tasks in the transactions by passing the executable (a `Method` object that represents a portion or the whole body of the transaction) to `TransactionShell` objects.

PJama is certainly a very promising technology, and it shares a lot of common goals with our research. However, PJama's transaction support is designed for the objects inside a Java™ VM-which are in much finer granularity and larger numbers, while our target transactions consist of objects such as documents and web pages, and the frequency of data access is often limited by the human user operation. This difference leads to different design choices. For example, as we think distributed transaction support is essential, PJama does not have it built-in: it is the transaction programmer's responsibility to manage data sources from multiple sites and provide a distributed commit protocol.

3.4.3 Pern

Pern [25, 26] is a transaction management component previously developed in our lab. Its transaction management features are extensible via plug-ins invoked before and after each of its exported operations (begin, commit, lock, etc.) [25]. Pern can also be customized by CC policy rules (written in a rule-based language called Cord, developed under the inspiration of CRL [6] another previous work in our lab) specifying how to handle lock conflicts [26]. Pern has been integrated with several systems (Oz, PCTE, and Process Weaver [24, 25]) to provide transaction functionality. The work presented in this paper inherits its plug-in concept from Pern, but does not incorporate an extended transaction-modeling notation like Cord.

We extended the research in several directions. First, we want to support heterogeneous data sources without changing either the clients or the server. Second, we define a clear transaction engine API for the ETM designers. Pern does not define such an API and writing ETM plug-in requires sophisticated knowledge of the internals. Third, we are dealing the problem in a distributed environment, where the plug-in code can be sent to the transaction server, while it is running, from the Internet.

4 Our Approach

Our middleware approach separates the information resources that store the hypermedia (the web servers), the applications that access the hypermedia for collaborative work (the clients), and the transaction server that provides the CC capability. The transaction server is independent of both the data sources and the applications. When an application needs to access a datum, the request is routed through the transaction server, and includes CC information such as which transaction this access is part of and thus which set of CC policies should be applied. The actual enforcement of the CC policies is the job of the transaction server. The transaction server may choose to exploit any CC support provided by the data source, such as locking or versioning, or not.

The transaction server should be extensible in several ways, including the range of CC policies it can support, its mechanisms for communication with backend data sources, and other application-specific customization such as event notifications.

Let us consider a simple joint authoring scenario: Alice and Bob are two co-authors of some articles stored and published by two web servers S1 and S2. They use a collaborative authoring tool that reads and writes the articles on S1 and S2. David, who is a fan of Alice and Bob, also accesses the web servers and attempts to read the article. It is conceivable that Alice and Bob do not want David (or anyone else) to see any of the in-progress versions of the articles, until they are completed, although they would be willing to let David read a pre-publication assessment version and receive his comments. However, between Alice and Bob there is no such barrier: each co-author wants to see the latest stored version even if the other one is currently editing the next version, and their authoring tool may even let the two of them edit the document together in what-you-see-is-what-I-see fashion.

Figure 1 shows this motivating scenario. The authoring tool that Alice and Bob use to access the articles should work in tandem with the transaction server to enforce a relaxed (but still consistent) CC policy to allow the co-authors to read and write the articles in a collaborative fashion. The transaction server retrieves the articles from the web servers, using their locking mechanism to protect the private versions from being read by outsiders such as David, who presumably employ conventional web browsers rather than the co-authoring editor (which would in any case not accept outsiders as members of the authoring team).

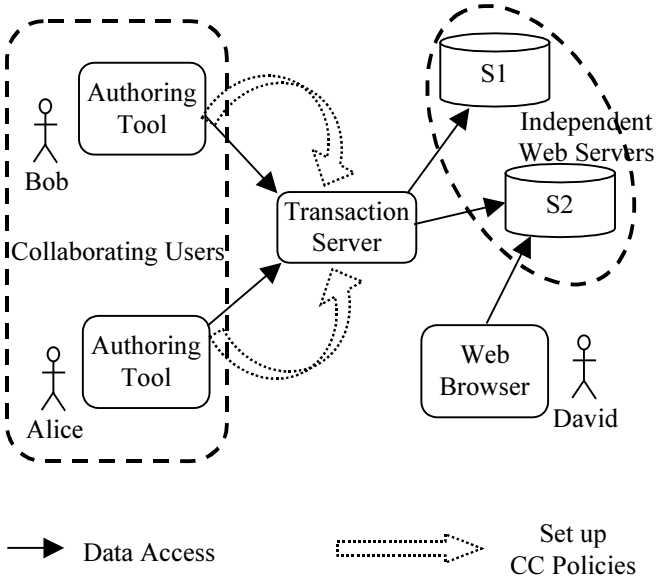

Figure 1. Motivating Scenario

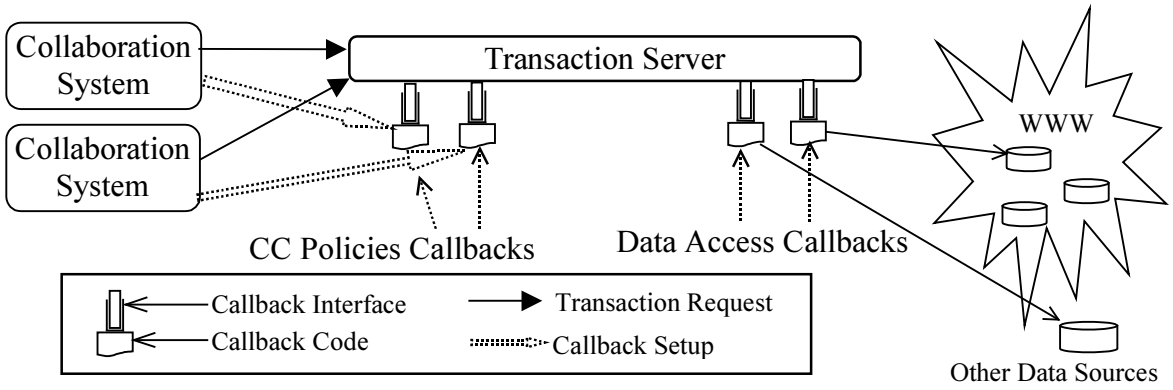

Figure 2. Transaction Server Architecture

Figure 2 shows a generalized architecture depicting how our transaction server might operate in such a scenario. The application used by Alice and Bob makes requests to the transaction server, which enforces the desired CC policy. Without changing the authoring tools, the CC policies are written in various forms of callbacks and sent to the

transaction server from the collaborative systems (in our case Alice and Bob's authoring tools). Among other callbacks, the data access plug-in modules enable the transaction server to access various backend data sources, i.e., the web servers, Hyperbases, or CORBA Servers and the CC policies plug-in modules customize the transaction server so that it can realize the desired ETMs.

The transaction server can be either a full-fledged web server that speaks some HTTP extension with specific URLs representing transaction operations, or an HTTP proxy that sits between client applications and WWW. The choice is an implementation issue. If the applications support HTTP proxies, as most web browsers do, the proxy approach may provide transaction services almost transparently. Our JPernLite implementation can operate either as an HTTP proxy or as a web server.

4.1 External Transaction Manager

Most traditional databases are constructed in such a way that the transaction manager is part of the database management system. When a client asks the database for some data operation, the database (either implicitly or as requested by the user or application) groups the operation with others into a transaction so that the operations are collectively treated as an atomic unit. The application need know nothing about transaction management details, such as how the transactions are aborted or committed, how the locks are obtained, and which locks are conflicting with respect to each other.

When a client needs to access data dispersed across multiple databases, the CC policy is carried out cooperatively by the set of databases – which usually must be configured to know about each other when they are set up. These distributed databases form a federation, using protocols such as the conventional two-phase commit to coordinate with each other.

Instead of applying this approach for web-based hypermedia systems, which would require cooperation between all the web servers accessed during a distributed transaction, we propose the external transaction manager model depicted in Figure 2. That is, let the web servers stay as they are in terms of lacking CC or other transaction functionality, and employ a separate server. This server is thus called an external transaction server.

An external transaction server provides the client applications with transaction services but does not require the web servers to be set up in any application-specific way, e.g., the web servers do not need to know which other web servers to contact when a distributed transaction is performed. This function is moved to the transaction server. When a user starts a task that is intended to be performed as a long-duration transaction, he/she might not know in advance which data objects will be needed. In fact, the task steps often cannot be predicted either. For example, a user might need to perform the step "check agenda" first, to find out what work is most pressing, and then determines

which data will need to be accessed next. The corresponding need for cooperation among the data sources may thus be formed dynamically and cannot be predicted and hardcoded at the time when the servers are configured.

The external transaction server makes it easier to employ heterogeneous information resources. For example, the same application may need to access web pages, CORBA objects, tables in a relational database, and links and anchor information stored in a linkbase such as Chimera 2 [2]. In this case, if we build transaction functionality into the information resources, or in wrappers around them as in Meteor [31], the adaptation effort is likely to be tedious and expensive. And the complexity of the system will be unnecessarily high. In our approach, the transaction server also acts as a CC gateway among the heterogeneous data sources.

4.2 Extensible Concurrency Control Policies

Traditional database systems are targeted for applications that issue large numbers of machine-generated data operations, for example for bank account maintenance, where speed is a high priority. Naturally, traditional databases use atomicity and serializability as the CC correctness criteria. Atomicity guarantees that all the operations in a transaction are done or none of them is done; serializability guarantees that the transactions are executed as if they were executed independently with each other (in some serial order).

Commercial transaction servers such MTS also share the same problem: they can only implement atomic transaction model, with very limited customizability. Although some of them can hook up with back end data sources, they usually require a lot more support, such as recovery and two-phase commit.

During the past two decades, researchers in areas such as CAD/CAM, software development environments and distributed authoring have found the traditional database transaction model inappropriate for their systems [21]. One of the problems is that the traditional transaction model prevents partial results from being seen by other transactions, and therefore explicitly disallows the possibility of collaboration. As a result, they proposed a number of Extended Transaction Models [21] that relax the traditional CC criteria.

To make the job of building extended transaction managers easier, Pern introduced the idea of an extensible transaction component that can be integrated with various environment frameworks via a plug-in mechanism [25]. In our recent research work, we adopted some ideas from Pern, especially how to make the transaction server extensible via callbacks. To realize the transaction server architecture shown in Figure 2, we introduced new features such as persistent transactions, an extensible set of information resources, and dynamic loading of the plug-in callback code over the Internet.

5 Transaction Service

We have realized the general transaction server architecture presented in the previous sections. The implementation is called JPernLite, since it is not yet a full implementation of all the ideas. The transaction features provided by JPernLite can be categorized into four areas: data object management, concurrency control, persistent transaction management, and session management. In this section we describe each of the above four areas, their functionality and how they work as part of the transaction server.

Figure 3 shows a sample communication between the transaction server and the client. When functioning as a web server, JPernLite speaks HTTP extended with some JPernLite-specific headers that carry various information. The client requests to lock an object whose URL is `http://www.cs.columbia.edu` in read (R) mode. The values of `PernSessionID` and `Transaction` ID were previously obtained from JPernLite. Upon success, JPernLite replies with the message that tells the client that the object has been successfully locked, and that the lock ID is `DATAOP://TX://Transaction/1/1`.

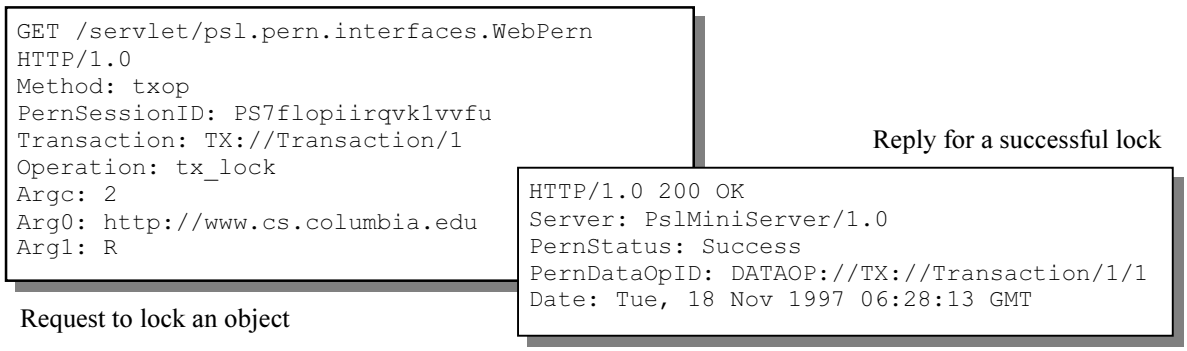

Figure 3. Sample Request and Reply to Lock an Object

5.1 Data Objects

Each datum accessed by a client is represented as an object in JPernLite's internal object base. In the current implementation, we use ObjectStore PSE as the tool to implement the object base. These "stub" objects represent the targets for locking and reference their real-world peers.

The clients may access the data directly from their home data sources. However, in order to obtain the concurrency control guarantees, the client must follow the convention of asking JPernLite for permission first. Or clients may request that JPernLite access the data on their behalf, in which case JPernLite automatically caches the objects for recovery and distributed commitment purposes.

The JPernLite server currently does not remove an object from its object base even after all transactions have unlocked it. This is due to a trade-off between the size of this database and run-time efficiency, since the object base also caches the data content. For example, JPernLite might cache a web page (HTML or binary data). When some

client asks for this web page again, JPernLite employs HTTP conditional GET to fetch the content only if it has been changed, reducing network transmission costs. A garbage collection command is available for administrators to clear the cache.

Whenever a client requests a lock, the JPernLite server first locks the corresponding object in its own database; if that lock was placed successfully, then JPernLite tries to lock the datum in its home data source. For example, if the datum is a web page and its web server supports the WEBDAV locking mechanism, JPernLite tries to obtain the lock from the web server. For web servers that have no notion of locking, the only locks are those kept in the JPernLite server.

5.2 Concurrency Control

JPernLite provides two basic mechanisms for concurrency control: locks and timestamps. Event notification is not built into the JPernLite server, but can be added by extension code, as we discuss in the next section.

Each operation in a transaction has a timestamp that can be used by a serializability implementation such as timestamp ordering. Since JPernLite normally uses conventional two-phase locking to ensure serializability, as do most database transaction managers, the timestamps are left for extension code to exploit - although even the two-phase locking is technically an "extension", i.e., its not hardwired in our implementation.

Locks in JPernLite are more like user-controlled locks as opposed to the implicit locks of traditional databases and conventional transaction servers. The client needs to explicitly request to place a lock on a datum and to remove a previously placed lock. Locking is not performed implicitly as a side effect of data access. The lock modes are configurable by a lock table, in which the administrator specifies available lock modes and their mutual compatibilities. This way, the application administrators can invent their own lock mode, such as notify or dirty read locks, that are very useful for collaborative applications. The lock compatibility table can only specify whether two lock modes (e.g., read vs. write) are allowed on the same datum at the same time or not. More sophisticated lock modes such as intention locks on ancestors or linked entities must be implemented via the plug-in mechanism. (An intention lock is a standard mechanism for preventing deletion or other modification to a composite object containing the regularly locked datum.)

5.3 Persistent Transactions

All transactions in JPernLite are persistent, in the sense that if the JPernLite server crashes and reboots, the clients might not even notice it, and can continue their work as if nothing had happened. This kind of failure support is quite different from the traditional "rollback" recovery, where the client’s work is undone and later has to be redone. The purpose of persistent transactions is to recover the transaction server to be synchronized with the transaction

states in the clients, as opposed to vice versa. This is important for human-oriented systems where operations such as editing and debugging takes a long time to finish, and rolling back the operations, e.g., reverting the file that has been edited, may lead to heavy loss of human work. A client will detect the crash only if it tries to access JPernLite before it is restarted, analogous to the crash and restart of a normal web server. However, once restarted, unsaved changes, ongoing tasks, etc. on the client side can still refer to the transactions that they were working on.

To achieve transaction persistency, all the transactions are stored in the object base as first-class objects as well. That is, JPernLite itself is a client of a traditional database, thus maintaining crash recovery is made easier. The recovery mechanism is also used to recover the "stub" objects in case of server crash. However, remember that the original server(s) is (are) still where the data lives. Therefore, JPernLite's crash recovery does not affect the actual data stored in the web servers but rather its own copy of the data.

5.4 Sessions

A session represents a user agent (either an end-user or an application) that employs the transaction service. The concept of session can be found in many human-oriented application systems. Establishing a session with the transaction server gives the user agent permission to access its services. It is quite similar to a "login" operation.

JPernLite allows the same user to have multiple sessions, and each session can be disconnected and reconnected in order to continue working on the transactions that were left unfinished in a previous session. This allows mobile user agents, e.g., PDAs, to attach to the service at an arbitrary time and location. Mobile clients and applications with low bandwidth connection can be built in such a way that they only need to remember its own session ID, reconnect to JPernLite whenever they have internet connection, and continue the task or transaction in JPernLite at anywhere.

The notion of session also facilitates collaboration among users because users can now "transfer" sessions among each other back and forth, passing around the on-going transaction information. For example, one user can start a session, start some transactions, and perform some data manipulation, then transfer the session to another user (e.g., by calling the other user and telling him the session ID). Then, the other user can carry on the task that is associated with the session, by probably create more transactions, commit or abort some of the transactions created by the first user, or work on some of the transactions that he has been given.

Basically the notion of session is another level of indirection between the users and the transactions. By not forcing the users to be associated with each transaction, JPernLite allows the application builders to construct more dynamic systems, such as agents moving around the Internet.

6 Extending JPernLite

JPernLite is designed and implemented with the primary goal of extensibility. In this section, we present the methodology and mechanisms an administrator might use in order to design and build application-specific ETMs on top of JPernLite. We conclude this section with several examples that explore various extension areas of JPernLite.

6.1 Designing a JPernLite Extension

Application administrators (or, possibly, end-users) extend JPernLite by writing Java™ plug-in functions that each operates as a "callback" of a specified transaction operation. The idea of using plug-in code to customize a transaction manager is discussed in more detail in [25]. JPernLite separates its functionality into the following five areas:

- **Locks:** JPernLite supports flexible lock modes and lock compatibility via a customizable lock table. One can easily define lock modes and specify the compatibility among them, and by default JPernLite guarantees the integrity of an object by disallowing conflicting locks. However, an administrator can write plug-in code to further specialize the lock semantics. One example is the implementation of two-phase locking: a transaction cannot acquire new locks after it starts to unlock. Such sophisticated semantics that cannot be represented by the locking table must be coded in the plug-ins via the API of JPernLite [46].
- **Timestamp:** Each transaction operation request is associated with a timestamp. JPernLite does not directly use the timestamp, but the plug-in callback code can use it to implement timestamp-based concurrency control policies such as timestamp ordering [11].
- **Transaction Operations**: Higher-level transaction operations such as to begin a new transaction, commit or abort a transaction, are also customizable. By writing plug-in code, one can alter the recovery/rollback policies and commitment procedure. By default, JPernLite supports nested transactions, with a standard two-phase commitment procedure and an atomic recovery policy. One can also add new transaction operations to JPernLite by providing extra public `tx_` methods in the extended `Transaction` classes (refer to section 6.3.1), if the transaction model to be implemented has operations other than begin, abort and commit.
- **Event Notification**: One can also notify clients for events of potential interest. For example, applications might want to be notified of any accesses by relevant users on designated entities or repositories. Putting the event notification capability in the transaction server (as oppose to in the clients) makes information about the ongoing transactions available to the event notification callbacks. The callbacks can use whatever communication protocol that the client recognizes, such as HTTP or RMI, to contact the client.
- **Data Operations**: To support applications utilizing several different kinds of back end data sources, e.g., SQL databases in addition to WWW, JPernLite provides means for introducing new data access protocols. Such data operation plug-ins are also useful for utilizing the concurrency control policies available from the back end data sources, e.g., a web server that implements WEBDAV.

We illustrate the procedure for designing a JPernLite extension in Figure 4. The process is divided into three stages: first determining the transaction operations, then deciding what concurrency control mechanism to use, and last, analyzing what other necessary facilities such as event notification and data operations are needed. In the first stage, one determines the extra transaction operations the specific ETM needs, the differences between the ETM and the standard atomic transaction model in begin, commit and abort procedures, and the transaction structure. If changes are necessary, one then should design the interface for the transaction operations, and write plug-in callback code to alter the built-in begin, commit or abort procedures, by extending (subclass) the `Transaction` class of JPernLite. When JPernLite starts a new transaction, the client can specify (by name) the type of transaction it wants to start, and JPernLite will instantiate the corresponding `Transaction` class.

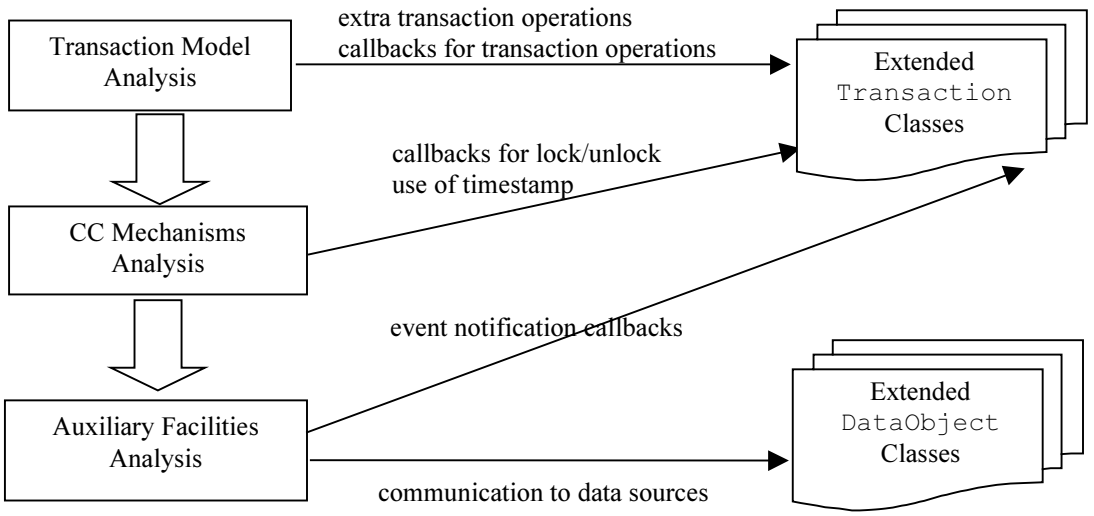

Figure 4. Design Process of JPernLite Extension

During the process of defining a transaction model, one must also consider what kind of concurrency control mechanism it uses, and design the necessary lock or timestamp-based concurrency control algorithms. Then, a new lock table should be defined and used to customize JPernLite via a lock table file. Sometimes the new lock modes may require more sophisticated semantics than mutual (in)compatibility. In this case, it is necessary to write plug-in code to realize such semantics. The plug-in code is written as the callback code of `tx_lock` and `tx_unlock`.

Finally, if the extension involves features such as event notification or utilizes any non-standard web servers or other back end data sources, additional callback plug-ins should be written to realize such features. As we mentioned before, a web server may or may not implement concurrency control features such as web page locking, therefore the transaction server must be prepared to be customized to use such features. The data operation callbacks are done by extending the `DataObject` class in JPernLite.

JPernLite allows different transaction models to co-exist: each transaction model inherits and extends the `Transaction` class, and one can specify what class of transaction to start in the arguments to the `tx_begin` transaction operation.

6.2 Testing Plug-in Code

Testing the plug-in callback code to ensure the correctness of the transaction model is also an important issue. It is desirable to be able to test the plug-in code alone, without the supporting JPernLite framework. However, since the plug-in code might access JPernLite's global databases, that would require a test module which simulates them. Building such simulation databases could be time consuming. One would have to construct the exact database interfaces, support all the features they export and create the objects (ongoing transactions, data objects being accessed, etc.) in the databases to match the test scenario.

We separate the testing process into two parts: testing the transaction callbacks and testing data access callbacks. The transaction callback testing focuses on the extended `Transaction` classes, and data access testing focuses on the extended `DataObject` classes. This way, when administrators test the transaction callbacks, unnecessary network traffic to access the data objects can be avoided.

Testing the data access callbacks is rather simple. A simple driver can be written to call the `Read`, `Write`, `ExternalOp` and `UndoExternalOp` methods and check the effects of these methods by looking at the actual data stored in the data sources, e.g., the web servers.

To test the transaction callbacks, one could integrate them into JPernLite with no extended data access modules. Instead, a `DUMB_DataObject` class could be used to handle all the data object accesses. `DUMB_DataObject` normally returns success on data read and write. To test the cases where the transaction model deals with data access failures, a random factor can be introduced. To verify the correct execution of transactions for our example ETMs, we produced a log of all the transaction requests and their results and manually went over the list. Specific case-based tests can also be done in this fashion, by setting up the scenario one wants to test first.

6.3 Plug-in Interface and API

In order to realize ETMs on top of JPernLite, one needs to subclass (or extend, in Java™ terminology) two Java™ classes: `Transaction` and `DataObject`. The extended `Transaction` class holds the plug-in code for locks, timestamps and transaction operations; the extended `DataObject` class holds the plug-in code for back end data access. Event notification code may be written in either or both classes.

6.3.1 *The* `Transaction` *Class*

The basic `Transaction` class (Figure 5) defines transaction methods such as begin/commit/abort a transaction and lock/unlock an object. All the transaction methods take three parameters: `"global"` contains the handlers to databases such as the transaction base and recovery base; `"ses"` describes the session entry from which this transaction method is issued; `"args"` passes client parameters to the transaction method. Some examples of client parameters include the URL of the object to be locked, the lock mode of the lock that the client requires, or the name of the transaction class to start. The client can pass as many parameters to the transaction method as it wishes, but the default transaction methods as defined in the base `Transaction` class only use the first few as indicated in Figure 5. However, since the client parameters are passed to the callbacks, they serve as an important bridge between the client and the plug-in code.

```
public class Transaction {
  public TxOpResult tx_begin(PernGlobal g, SessionEntry s, String arg[]);
  public TxOpResult tx_abort(PernGlobal g, SessionEntry s, String arg[]);
  public TxOpResult tx_commit(PernGlobal g, SessionEntry s, String arg[]);
  public TxOpResult tx_lock(PernGlobal g, SessionEntry s, String arg[]);
  public TxOpResult tx_unlock(PernGlobal g, SessionEntry s, String arg[]);
  public TxOpResult tx_dataOp(PernGlobal g, SessionEntry s, String arg[]);
}
public CBresult tx_unlock_after(PernGlobal g, SessionEntry s, String arg[], TxOpResult r);
```

Figure 5. The `Transaction` Class

Application administrators subclass the `Transaction` class and write callbacks to alter the execution of the methods. There are three kinds of callbacks, categorized by when they are called:

- `"before"` callback: called before the method is executed
- `"after"` callback: called after the method is successfully executed
- `"error"` callback: called when the method fails. An error code and a line of text explaining why the operation was failed.is passed to the callback via the `op_res` parameter so that the failure reason is both machine recognizable and human readable.

All the methods listed in the interface can have callbacks. In addition, an implicit method `Rollback`, which is called when a transaction is aborted, can also have callbacks. The name of each callback method is their category name (`"before"`, `"after"`, `"error"`) underscore-prefixed by the transaction method name (`"tx_begin"`, `"tx_lock"`, `"Rollback"` etc.). The interface for callback methods is also shown in Figure 5. For the `"error"` and `"after"` callbacks, the `"op_res"` parameter contains the result of the transaction method.

The administrator can choose to write one or more of the possible callbacks for any of these operations. The callbacks can ask JPernLite (via the "CBresult" return value) to ignore errors, stop the current operation and start another one instead, or return certain information to the client. The callbacks have access to a wide range of objects in JPernLite through the API of the databases in "global". They can also make various outside connections on their own, such as notifying the client of certain events.

The administrator can also choose to write additional transaction operations for the specific transaction model. For example, an operation to check if a lock is available or has expired can be done in this fashion. Such additional transaction operations can simply be added to the specific Transaction class as public methods, e.g., a tx_check_lock method that has the same interface as the pre-defined transaction operations.

6.3.2 The DataObject Class

The DataObject class models the data object that resides on the WWW or other potential data source, such as an SQL server, a legacy database that speaks HTTP, a CORBA object and so forth. By default, JPernLite does not know how to communicate with these data sources to access and manipulate the objects. Whenever an object is accessed, JPernLite searches in the subclass classes of the DataObject class and looks for the one that is designed to handle such data objects.

All the data objects must have a string name, although the choice of the name is totally up to the applications. For example, if an object is a CORBA object, the name could be corba://myorb/objectName. JPernLite would look for a class that is willing to accommodate such an object by calling the Processes method. The Java™ interface for DataObject is shown in Figure 6. There are two methods for read and write, which are called by JPernLite whenever a client accesses the object.

```
public interface DataObject {
  public boolean Processes (String name);
  public TxOpResult Read(PernGlobal g, SessionEntry s, Transaction tx, String args[]);
  public TxOpResult Write(PernGlobal g, SessionEntry s, Transaction tx,
                          String new_content, String args[]);
  public TxOpResult ExternalOP(PernGlobal g, SessionEntry s, Transaction tx, String args[]);
  public TxOpResult UndoExternalOP(PernGlobal g, SessionEntry s, Transaction tx,
                                   DataOperation dop);
}
```

Figure 6. The DataObject Interface

There are also two generic methods, ExternalOp and UndoExternalOp, to perform other operations specific to that type of data object, and remove the effects of such data operations when JPernLite rolls back a transaction. For example, JPernLite calls the ExternalOP of a data object to perform lock operations at the data source. If a

specific type of data supports locking (e.g., a web page on a web server that implements WEBDAV), the `ExternalOp` method can be implemented to support it. These methods can be called by the transaction classes that use specific types of back end data objects. For example, a CORBA object can be accessed by writing a `CORBA_DataObject` class that is essentially the stub for the interfaces. Methods other than read and write are naturally mapped to `ExternalOp` and the recovery of such methods are written in the `UndoExternalOp`.

6.3.3 Global Databases API

When a callback is invoked, it is given the `global` parameter that contains a collection of databases of JPernLite:

- *Transaction base:* the database that stores all the on-going transactions;
- *Recovery base:* the database that stores all the stub objects of the data objects currently being used;
- *Session base*: the database that stores all active sessions;
- *Lock table:* this is not exactly a database, but rather an object that stores the lock table used by JPernLite.

The API of these databases (or data structures) is the interface between the callback code and JPernLite. It is fully specified in [46].

6.4 Example JPernLite Extensions

6.4.1 Two-Phase Locking

The first example we show here is the callback code to implement Two-Phase Locking in JPernLite. Two-phase locking is the standard algorithm used to guarantee serializability among conflicting transactions in traditional database systems. Here we implement a common variant of it: a transaction is divided into an "growing" phase where a transaction acquires locks and a "shrinking" phase where a transaction releases all locks.

```
public CBresult tx_lock_before(PernGlobal g, SessionEntry s, String arg[], TxOpResult res) {
  if ((Boolean)GetAtt("shrinking")) {
    TxOpResult r;
    r=new TxOpResult(TxOpResult.ERROR, "PernError", "Cannot lock after started unlock");
    return new CBresult(r); // let tx_lock return "r" immediately
  } else return new CBresult(); // let tx_lock continue normally
}

public CBresult tx_unlock_after(PernGlobal g, SessionEntry s, String arg[], TxOpResult res) {
  SetAtt("shrinking", new Boolean(true));
  return new CBresult();    // let tx_unlock continue normally
}
```

Figure 7. Two Phase Locking

Figure 7 shows our implementation of two-phase locking using two callbacks: `tx_unlock_after` and `tx_lock_before`. `tx_unlock_after` marks the transaction to be in the "shrinking" phase upon a successful unlock operation, and `tx_lock_before` prevents a transaction in the "shrinking" phase from obtaining any new locks.

6.4.2 Sagas

Sagas [22] are based on the concept of a *compensating* transaction, where each transaction may be compensated to semantically undo its effects. Note that the unit of compensation is not atomic data access such as incrementing an integer by one, but instead is the unit of a transaction. A Saga is a long-lived transaction-like unit that consists of a set of sub-transactions (T1, T2, ... Tn), each one associated with a compensating transaction (C1, C2, ... Cn). The execution of a Saga is that either the sequence T1, T2, ... Tn or the sequence T1, T2, ... Tk, Ck, ... C2, C1 (where 1<=k <=n) is executed. The degree of concurrency is increased by the fact that each sub-transaction (T1, T2, ... Tn) can be committed right after it finishes, and therefore resources can be released as early as possible and reused by other transactions.

We realize Sagas by adding two callbacks, `tx_begin_after` and `tx_abort_before`, which are shown in Figure 8. In `tx_begin_after`, we obtain the type of the sub-transaction from the client parameter, and store it in the transaction base by setting an attribute of the current transaction. Later, when we are about to abort a Saga, we execute the compensating transactions in reverse order.

```
public class Sagas_Transaction extends Transaction {
  public CBresult tx_begin_before(PernGlobal g, SessionEntry s, String arg[], TxOpResult r) {
    if (GetAtt("TransactionType")!=null) { // this is a leaf node transaction
      TxOpResult res = new TxOpResult(TxOpResult.ERROR, "PernError",
                          "Cannot start sub transaction on leaf node transactions");
      return new CBresult(res);
    }
    return new CBresult();
  }

  public CBresult tx_begin_after(PernGlobal g, SessionEntry s, String arg[], TxOpResult r) {
    if (arg.length > 1) {                       // leaf node transaction
      String tx_id = r.TxInfo("PernTxID");  // ID of the new transaction
      Transaction t = g.transactionBase.FindTX(tx_id);
      t.SetAtt("TransactionType", arg[1]);      // for use of compensation
    }
    return new CBresult();
  }

  public CBresult tx_abort_before(PernGlobal g, SessionEntry s, String arg[], TxOpResult r) {
    Transaction t;
    String tx_type = (String)GetAtt("TransactionType");
```

```
    if (tx_type == null) {           // this is a top level Sagas transaction
      // compensate the sub transactions in reverse order, no compensation failure
      for (int i = children.size()-1; i>=0; i--) {
        t = g.transactionBase.FindTX((String)children.elementAt(i));
        tx_type = (String)t.GetAtt("TransactionType");
        if (tx_type == null) {       // this is another Sagas transaction
          // MethodCaller will call the callbacks if they are defined.
          MethodCaller.CallTXmethod(g, s, t, "tx_abort", arg);
        }else{                       // this is a leaf node in the transaction tree
          switch (t.Status()) {
          case ALIVE:                // hasn't commited yet
            MethodCaller.CallTXmethod(g, s, t, "tx_abort", arg);
            break;
          case COMMITTED:
            // ask itself to run the compensate transaction
            t.compensate(g, s, arg);
            break;
          case ABORTED:              // already aborted, do nothing
            break;
          }
        }
      }
      // after compensation, simply let tx_abort to return success
      return new CBresult(new TxOpResult());
    }
    return new CBresult();  // otherwise, continue normal abort
  }
}
```

Figure 8. Extended `Transaction` Class for Sagas

6.4.3 Access Notification

This example shows how the lock table is useful in extending JPernLite. The example is to realize an access notification feature where a client can be notified whenever another user accesses a certain data object. One would set up a lock table as shown in Figure 9, where a new lock mode E that is compatible with all other lock modes is introduced. The lock table is loaded in to JPernLite via a lock compatibility file. Therefore, the lock table is global to JPernLite. All the transaction classes share the same set of lock modes and their mutual compatibility. This design choice is made so that the transaction model designers have to explicitly deal with the lock table conflicts: different ETMs may have different compatibility between the same set of lock modes. If we hide the difference by providing each transaction model a separate lock table, the chances to have an unknown lock table conflict will be greatly increased.

There are several places where the plug-in callback code can be written to notify all users who hold an E lock on the target datum. One possibility is in the `tx_lock_after` callback, which is called when the target datum is

successfully locked. The second possibility is in the `tx_lock_before` callback, which is called when a user attempts to obtain a lock. One can also write a `tx_dataOp_before` or `tx_dataOp_after` that notifies the client right before or after the actual data operation. The administrator should make the choice based on the exact semantics that the access notification is intended to achieve. Figure 10 shows an example callback for `tx_lock_after`. The communication between the callback and the client is hidden in the method `notify_client()`, which is application-specific.

	R	W	E
R	Y	N	Y
W	N	N	Y
E	Y	Y	Y

Figure 9. Lock Table for Access Notification

```
public class AN_Transaction extends Transaction {

  public CBresult tx_lock_after(PernGlobal g, SessionEntry ses, String arg[], TxOpResult r) {
     RecoveryItem  data = g.recoveryBase.ensureItem(s, this, arg[0]);
     Vector        locks = data.locks();

     for (int i=0; i<locks.size(); i++){
       DataOperation lock = (DataOperation)locks.elementAt(i);
       if (((String)lock.GetAtt("lock_mode")).equals("E")) {
         Transaction t = lock.TX();
         SessionEntry s = g.sessionBase.FindSession(t.Session());
         // notify the user of session s, data is locked in mode arg[1] by someone
         notify_client(s, data, arg[1]);
       }
     }
     return new CBresult();
  }
}
```

Figure 10. Callback for Access Notification

6.4.4 Extending Data Objects

In this example, we extended the `HTTP_DataObject` class for web servers that support GET and PUT. The `HTTP_DataObject` simply maps the read and write requests from the transactions to GET and PUT HTTP requests, respectively. If a web server implements WEBDAV, locking of web pages can also be done this way. Due to the length restriction of this paper, Figure 11 only shows part of the implementation. For more details, please refer to [46].

```
public class HTTP_DataObject {
  private String url_s;
  public HTTP_DataObject(String url_p) {    url_s = url_p;  }
  public boolean Processes (String url) {  return (url.indexOf("http://") == 0); }
  public TxOpResult Read(PernGlobal g, SessionEntry ses, Transaction tx, String arg[]) {
     URL url;

     try{
        url = new URL(url_s);
     }catch (MalformedURLException e){
        return new TxOpResult(TxOpResult.ERROR, "PernError", "Malformed URL");
     }

     HTTPConnection hcon=setHCon(url, sendModifiedSince, args);
     // ... set HTTP connection and GET the content.  Omitted.
  }
}
```

Figure 11. Fragment of `HTTP_DataObject`

7 Discussion

7.1 Application Integration

JPernLite can be integrated with application systems in several ways: It can be run as an independent web server or proxy server on the Internet, receiving explicit or implicit transaction requests from the clients through HTTP. It can also be used as a Java™ servlet, a standard way of integrating services into web servers. Finally, it can also be treated as a Java™ package, that is, tightly integrated into a host server program via its Java™ API. We illustrate the possibilities in this section.

7.1.1 JPernLite as a Web Server

To treat JPernLite as an independent web server that manages distributed transactions, one needs to develop a client that speaks the HTTP dialect of JPernLite. The current version of JPernLite searches for parameters such as type of transaction operation, transaction ID, or session ID in both the header part of the HTTP message and the variables of a POST request. Information returned from the JPernLite server is also put in both the headers of the reply message and a regular, human readable HTML message. The two interfaces are suitable for different clients: custom-made applications that have their own HTTP code can easily use the headers to manipulate the transaction requests and replies. More human-oriented applications that want to use existing browsers can use HTML forms to code transaction requests, and the human can easily read the transaction results.

The current HTTP interface is supported by Java™'s reflection package (`java.lang.reflect`). That is, when such a request is received, it determines the transaction classes and method names through the utilities in `java.lang.reflect`. The requirement for a reflective programming language is simply a limitation imposed by this particular implementation of the transaction server, because JPernLite uses Java™ class extension to define new transaction types, and callbacks are different methods of the transaction classes. An implementation can also choose to define transaction types via a scripting language, and use different parameters to distinguish different callbacks.

7.1.2 JPernLite as a Proxy Server

Although not yet implemented, JPernLite can be modified to run as an HTTP proxy server. In this mode, it catches all the HTTP requests from the web clients, and distinguishes normal data request and transaction requests. A transaction request is one whose URL is the reserved JPernLite URL. When such a request is caught, JPernLite performs the transaction request, and returns information such as session ID and transaction ID as cookies. Cookies is an HTTP header that informs the browsers to remember and send whenever they are accessing a certain URL. Most commercial web browsers, especially both Netscape and MSIE supports cookies. Therefore, when the client makes GET, POST or PUT requests in the future, the JPernLite proxy server can associate the data request with the right transaction context, and apply READ or WRITE locks according to the request type.

Although the semantics of GET and PUT is pretty clear, the semantics of a POST request is inherently ambiguous by its nature in terms of whether it is a read or a write operation. Different web servers use it for different purposes, it could be a data entry form, or it could simply be a query form. The proxy server interface package currently treats all POST requests as write operations. If an administrator wants to be specific, URL-specific data object classes can be written that change the write locks on POST requests to read locks.

7.1.3 JPernLite as a Java™ Component

For a somewhat tighter integration, one can use JPernLite as a Java™ servlet module or simply a Java™ package. Most web servers support servlets as a standard way of adding extensions. The Java™ API of JPernLite provides entry points to the service methods, such as start up JPernLite, execute transaction operations, and setting lock tables. In this case, the host system, either a web server or a legacy system that wants an internal transaction server requests the transaction operations directly through the Java™ API without any network traffic involved. Currently JPernlite does not provide interfaces to receive and configure the callback modules through the Internet, so the web server administrator would need to install the callbacks by hand (that is, by putting the zipped Java™ classes into the right directory). A potential problem with this integration approach is that the host system must be willing to accept the

persistence mechanism that JPernLite uses, i.e., ObjectStore PSE. That means integrating JPernLite also forces the host system to include ObjectStore in its CLASSPATH.

7.1.4 Our Experience

We have carried out several integration experiments. First, we implemented a Java™ applet client of JPernLite that can be run by any web browser supporting JDK 1.1. Human users can start transactions through the applet on the fly while they access one or more web pages. Accesses from multiple users are coordinated via JPernLite and multi-access transactions can be created, committed, and aborted. During this integration experiment, we also implemented a client-side Java™ API for JPernLite. The client side JPernLite API accesses the transaction server through the HTTP interface, and exposes the services to the client application (in our case, the applet) via the Java™ API. Legacy systems that are written in languages other than Java™ would probably need to run JPernLite as a separate web server and write their own client side API.

Another example is that we integrated JPernLite into a workflow environment server [29], which directly uses JPernLite as a Java™ package. The workflow manager in the server utilizes JPernLite via the Java™ API to perform transaction operations. Compared with using JPernLite as an independent web server, this approach helps the workflow manager to synchronize the on-going tasks with the on-going transactions when recovering from server crashes. In such a situation, the workflow manager would be able to ask JPernLite to start certain tasks (via callbacks) when the transaction base recovers from a server crash. If JPernLite was running independently as a web server, such operations might suffer from network disconnection or jam: that is, the recovery process would have to involve further recovery protect against network disasters.

7.2 Back End Data Source Integration

JPernLite differs from some other transaction servers in an important way: it does not impose strict requirements on the back end data source capabilities. In this section we discuss some interesting problems we encountered when the back end data sources do not offer any transactional support. If JPernLite were to make requirements such as directly supporting recovery, two-phase commit, and a certain type of authentication, these problems would not exist in the first place. These problems are handled by the data access modules (the `DataObject` classes) in general. For example, the authentication scheme of the back end data should be encapsulated in the data object classes. They should remember the appropriate header fields (the ones that represent user name and password, for example) each time an access is granted, and use them when the data source needs to be rolled back.

7.2.1 The "Back Door" Problem

Users can always access data from web servers or other information resources without notifying the transaction server. This creates what we call the "back door" problem. If the data source supports locking on its data (i.e., web pages, for web servers), JPernLite could then actually lock them on the server. Then users who access the data directly could not interfere with a transaction accessing the same data in a conflicting mode. Another possible solution is to use access control, such as HTTP authentication in the case of WWW. If the data is protected under HTTP authentication and only JPernLite knows the password, users will not be able to access the data from outside of JPernLite. However, if none of the above mechanisms is available, e.g., many existing web servers that serve read-only web pages do not provide any locking capabilities, in the general case clients could not be informed if someone accesses the web pages directly without notifying the transaction server.

There is another kind of the "Back Door" problem, concerning data sources that generate data on the fly, e.g. by using CGI-bin or a URL corresponds to a query interface to a database. In this case, the locks applied by JPernLite is merely on that URL, and does not necessarily mean that the actual data or page is locked. In particular, if two query URLs produce the same document, locking one of them in JPernLite would no prevent someone else to obtain a conflicting lock on the other URL. A solution to this problem is to write the data access modules so that when such a lock is applied, the data access modules contact the database behind the web server, and actually lock the objects or tables inside the database directly.

7.2.2 Supporting Distributed Commit

JPernLite uses a two-phase commit protocol when data from multiple data sources are accessed by a single transaction. Two-phase commit is the standard means for committing data updates over distributed data sources, where a preparation phase (pre-commit) ensures that all the sites agree prior to the final global commitment phase. However, many data sources do not support two-phase commit. For example, servers that allow LOCK and PUT according to the WEBDAV specification may not also have pre-commit and abort (in case pre-commit fails) operations, which are essential to the two-phase commit protocol.

If the data source supports some form of distributed commit protocol, one can use the callback mechanism to extend JPernLite to exploit it. If the data source does not support any form of distributed commit protocol, what JPernLite can do is assume it will eventually agree with the commit and then retry repeatedly to write the current version of the data. This should work, eventually, for web servers that support only the basic HTTP 1.1 protocol (which includes the PUT method) - but the "back door" is extended here to include any accesses during the period of time prior to finalizing global commit.

7.3 Scaling Up

As we described the transaction server in this paper, all data access requests go through it for concurrency control, and thus it becomes a bottleneck or center of failure in its client application systems. This is an inherent problem of any centralized client-server system.

The problem comes mainly from two kinds of delays: network traffic and computation in the transaction server. We simulated the computation cost in situations where 10, 100 and 1000 users are using the server simultaneously. Each user randomly starts transactions to perform tasks such as read, write, lock and unlock. The data they access has a small set of "hot-spots" to simulate the reality of the web: there are a few web sites or web pages that are visited far more frequently than others. On a Pentium™ 200MHz machine with 64M RAM, the time spent by the server to service an individual request, in average, are 115, 151 and 450 milliseconds, respectively. The increase in processing time comes from larger data structures and transaction base, thus makes it harder to search and resolve concurrency control conflicts.

Considering the domain of potential applications, we think the processing time (0.1 to 0.5 second) is reasonable for a community of as many as 1000 users. In the anticipated domains, such as co-authoring, software engineering, CAD/CAM, healthcare, emergency response, etc., human activities take much longer than the CC processing time - and in any case it is unlikely that more than 1000 users would participate in the same team and thus share a single transaction server.

To tackle the network traffic problem as well as to scale up to teams of teams, which may rise in size beyond 1000 users, we are working on the idea of *alliances* among distributed transaction servers, where two or more servers make agreements between themselves to cooperate on enforcing certain subsets of their CC policies, analogous to our lab's previous work on alliances among workflow engines [8]. Multiple instances of the transaction server could share transaction state information with each other, resolve concurrency control conflicts, and allow users to collaborate across transaction servers.

7.4 Concurrency Control without Locking

Throughout this paper, we refer to locking as if it were the main CC mechanism. But this is for expository purposes only: JPernLite is not hardwired in any way to locking. Each operation in the history of a transaction is associated with a timestamp, therefore various timestamp-based CC algorithms can also be implemented.

Although lock-based concurrency control mechanisms are the most popular ones, there do exist other alternatives. For instance, optimistic concurrency control policies verify consistency of data at the commit time of a transaction. Implementing such policies would involve sophisticated use of JPernLite's plug-in extension capabilities.

8 Contributions and Future Work

8.1 Summary

We have presented a middleware approach for distributed concurrency control on WWW and introduced an independent extensible transaction server component. The transaction server fills the gap between the CC features hypermedia systems need and the CC features current web servers provide. A Java™ realization, JPernLite, has been integrated with a workflow system and can also run as a stand-alone server servicing requests from our Java™ applet client. The architecture makes it possible for JPernLite to support distributed transactions on data stored in heterogeneous information resources, although to date we have been using WWW as the primary example of such a data source. Figure 12 lists the transaction service features we summarized in section 2 as requirements for web-based collaborative systems.

* Event Notification	+ Fine-grained Notification	+ User controlled Locking	** Fine-grained Locking
+ Shared Locking	+ Persistency	+ Synergistic Cooperation	+ Extensibility
+ Heterogeneity	+ Supporting Large Scale	+ Support Distributed TXs	** Application Semantics

+: Yes, JPernLite supports this feature

−: No, JPernLite does not support this feature

*: JPernLite provides the opportunity to write event notification callbacks in transaction server, but does not support any specific communication protocols.

**: Yes, through the use of callbacks.

Figure 12. Summary of Features

8.2 Future Work

The only way to extend JPernLite at present is by writing plug-in code in Java™, which is activated by the JPernLite server (or servlet or package, its all the same code with multiple interfaces) when the corresponding transaction operations are performed. We are also defining a scripting language based on the original Pern's Cord notation to describe CC policies at a higher level of abstraction.

To support large-scale collaborative systems, it is desirable for the transaction servers to communicate with each other. Each individual transaction server would service only a subset of the applications and enforce the CC policies needed by those applications. When two or more applications from different transaction servers share data, these transaction servers would interact with each other to coordinate data access according to negotiated CC policies.

9 Acknowledgements

Special thanks to George Heineman, Steve Dossick, and Wenyu Jiang, the authors owe a lot to their excellent work in the Programming Systems Lab and numerous meetings and discussions about this research. The authors also want to thank Laurent Daynës from Sun Microsystems Laboratories who reviewed the paper and gave a lot of valuable comments.

This research work is sponsored in part by the Defense Advanced Research Projects Agency, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0022, and in part by an IBM University Partnership Program Award. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, the Air Force, the U.S. Government or IBM.

10 References

1 K. M. Anderson, "Integrating Open Hypermedia Systems with the World Wide Web," *Proc. The 8th ACM Conference on Hypertext*, Southampton, UK, 1997, pp. 157-166.

2 K. M. Anderson, R. N. Taylor, and E. J. Whitehead, Jr., "Chimera: Hypertext for Heterogeneous Software Environments", *Proc. European Conference on Hypermedia Technology*, Edinburgh, Scotland, Sept. 1994.

3 M. P. Atkinson, L. Daynës, M. Jordan and S. Spence, "An Orthogonally Persistent Java™", *SIGMOD Record,* Vol. 25, p. 68-75, No. 4, December 1996.

4 A. Bapat, J. Wasch, K. Aberer and J. M. Haake, "HyperStorM: An Extensible Object-Oriented Hypermedia Engine", *Proc. The 7th ACM Conference on Hypertext*, Washington, DC, USA, 1996.

5 N. S. Barghouti and G. E. Kaiser, "Concurrency Control in Advanced Database Applications", *ACM Computing Surveys*, Vol. 23, No. 3, Sept. 1991, pp. 269-317.

6 N. S. Barghouti, *Concurrency Control in Rule-Based Software Development Environments,* Ph.D. Thesis, Computer Science Department, Columbia University, Feb. 1992.

7 BEA Corp., BEA TUXEDO, http://www.beasys.com/products/tuxedo/tuxwp_pm/tuxwp_pm1.htm, Nov. 1996.

8 I. Ben-Shaul and G. E. Kaiser, *A Paradigm for Decentralized Process Modeling*, Kluwer Academic Publishers, Boston MA, 1995.

9 R. Bentley, W. Appelt, U. Busbach, E. Hinrichs, D. Kerr, S. Sikkel, J. Trevor and G. Woetzel, "Basic Support for Cooperative Work on the World Wide Web," *International Journal of Human-Computer Studies* Vol. 46, No. 6, June 1997.

10 T. Berners-Lee and R. Cailliau, "WorldWideWeb: Proposal for a HyperText Project", CERN European Laboratory for Particle Physics, Nov. 1990, http://www.w3.org/hypertext/WWW/Proposal.html.

11 P. A. Bernstein and N. Goodman, "Timestamp-based Algorithms for Concurrency Control in Distributed Database Systems," *Proc. 6th Conference on Very Large Databases,* Montreal, Canada, Oct. 1980.

12 P. A. Bernstein, V. Hadzilacos and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

13 A. Biliris, S. Dar, N. Gehani, H.V. Jagadish and K. Ramamritham, "ASSET: A System for Supporting Extended Transactions," *Proc. ACM SIGMOD International Conference on Management of Data*, Minneapolis MN, May 1994, pp. 44-54, Special issue of *SIGMOD Record*, Vol. 23 No. 2, June 1994.

14 B. Campbell and J. M. Goodman, "HAM: A General Purpose Hypermedia Abstract Machine*," Proc. The ACM Conference on Hypertext*, 1987.

15 P. Chrysanthis and K. Ramaritham, "ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior," *Proc. ACM SIGMOD International Conference on Management of Data,* New York, NY, 1990.

16 D. Cooper, "Transaction Internet Protocol (TIP) Demonstration," http://oss.tandem.com:2490/tip/.

17 CyberTeams Corp. "WebSite Director", http://www.cyberteams.com/products/wsd

18 L. Daynës, M. P. Atkinson and P. Valduriez, "Customizable Concurrency Control for Persistent Java™", *Advanced Transaction Models and Architectures*, S. Jajodia and L. Kerschberg, (editors), Kluwer Academic Publishers, 1997.

19 M. Eaddy, S. Rong and J. Shapiro, "WebDAV Project," Dec. 1997. http://www.cs.columbia.edu/~eaddy/webdav.html

20 J. Eliot and B. Moss, "Nested Transactions: An Approach to Reliable Distributed Computing," *Information Systems*, MIT Press, Cambridge MA, 1985, Michael Lesk (editor).

21 A. K. Elmagarmid, (editor), *Database Transaction Models for Advanced Applications*, Morgan Kaufmann, 1992.

22 H. Garcia-Molina and K. Salem, "Sagas," *Proc. the ACM Conference on Management of Data*, May 1987. Pp. 249-259.

23 S. P. Hadjiefthymiades and D. I. Martakos, "A generic framework for the deployment of structured databases on the World Wide Web," *Proc. The 5th International World Wide Web Conference*, Paris, France, May 1996.

24 G. T. Heineman and G. E. Kaiser, " Integrating a Transaction Manager Component with Process WEAVER", Technical Report, Computer Science Dept., Columbia Univ., CUCS-012-94, May 1994.

25 G. T. Heineman and G. E. Kaiser, "An Architecture for Integrating Concurrency Control into Environment Frameworks," *Proc. The 17th International Conference on Software Engineering*, 1995.

26 G. T. Heineman and G. E. Kaiser, "The CORD Approach to Extensible Concurrency Control*," Proc. The 13th International Conference on Data Engineering,* Apr. 1997.

27 IBM, "TXSeries4.2", http://www.software.ibm.com/ts/txseries, 1998.

28 G. E. Kaiser, S. E. Dossick, W. Jiang, J. J. Yang and S. X. Ye, "WWW-based Collaboration Environments with Distributed Tool Services," *Journal of World Wide Web*, Vol. 1, 1998, pp. 3-25, Baltzer Science Publishers.

29 G. E. Kaiser and S. E. Dossick, "Xanth: An Architecture for Effective Utilization of Distributed Heterogeneous Information Resources," Technical Report, Computer Science Dept., Columbia Univ., CUCS-003-98, Mar. 1998.

30 Microsoft Corp., "Microsoft Transaction Server", http://www.microsoft.com/com/mts.asp, Aug. 1998.

31 J. Miller, D. Palaniswami, A. Sheth, K. Kochut and H. Singh, "WebWork: METEOR's Web-based Workflow Management System," *Journal of Intelligent Information Systems*, Vol. 10, No. 2, March 1998. (in press)

32 Netcraft Corp., "The Netcraft Web Server Survey," 1998. http://www.netcraft.com/survey/.

33 Object Management Group, "Object Transaction Service", http://www.omg.org/corba/sectrans#trans, Nov. 1997.

34 Object Management Group, "The Common Object Request Broker", http://www.omg.org/corba/ July 1998.

35 The Open Group: *Distributed TP: The XATMI Specification*, Prentice Hall, ISBN 1-85912-130-6, November 1995.

36 C. Pu, G. Kaiser and N. Hutchinson, "Split-transactions for Open-ended Activities," *Proc. the 14th VLDB Conference*, 1988.

37 Sun Microsystems, "Enterprise JavaBeans™", http://java.sun.com/products/ejb/, July 1998.

38 Sun Microsystems, "Java™ Transaction API (JTA)", http://java.sun.com/products/jta/, Sep. 1998.

39 Sun Microsystems, "Java™ Transaction Service (JTS)", http://java.sun.com/products/jts/, Sep. 1998.

40 TIP Working Group, J. Lyon, K. Evans and J. Klein, "Transaction Internet Protocol (TIP)," Apr. 1998. ftp://ftp.ietf.org/internet-drafts/draft-lyon-itp-nodes-07.txt.

41 WEBDAV Working Group, Y. Goland, E. J. Whitehead, A. Faizi, S. R. Carter, D. Jensen, "Extensions for Distributed Authoring and Versioning on the WWW – WEBDAV," 1997. ftp://ftp.ietf.org/internet-drafts/draft-ietf-webdav-protocol-08.txt.

42 U. K. Wiil, "Concurrency Control in Collaborative Hypermedia Systems," *Proc. The 5th ACM Conference on Hypertext*, Seattle, WA, Nov. 1993, pp. 14-24.

43 U. K. Wiil and J. J. Leggett, "Workspaces: The HyperDisco Approach to Internet Distribution," *Proc. The 8th ACM Conference on Hypertext*, Southampton, UK, Apr. 1997, pp. 13-23.

44 U. K. Wiil and J. J. Leggett, "Hyperform: A Hypermedia System Development Environment," *ACM Transactions on Information Systems*, Jan. 1997.

45 World Wide Web Consortium, "Jigsaw Overview", http://www.w3.org/jigsaw, Sep. 1998.

46 J. J. Yang, "Application Programming Interface of JPernLite", Tech. Report CUCS-010-98, Computer Science Dept., Columbia Univ., New York, N.Y., 1998.