Toward A Model of Self-administering Data

ByungHoon Kang
Division of Computer Science
UC Berkeley
Berkeley, CA 94720
510 642-8468
hoon@cs.berkeley.edu

Robert Wilensky
Division of Computer Science
UC Berkeley
Berkeley, CA 94720
510 642-7034
wilensky@cs.berkeley.edu

ABSTRACT

We describe a model of *self-administering data*. In this model, a declarative description of how a data object should behave is attached to the object, either by a user or by a data input device. A widespread infrastructure of *self-administering data handlers* is presumed to exist; these handlers are responsible for carrying out the specifications attached to the data. Typically, the specifications express how and to whom the data should be transferred, how it should be incorporated when it is received, what rights recipients of the data will have with respect to it, and the kind of relation that should exist between distributed copies of the object. Functions such as distributed version control can be implemented on top of the basic handler functions.

We suggest that this model can provide superior support for common cooperative functions. Because the model is declarative, users need only express their intentions once in creating a self-administering description, and need not be concerned with manually performing subsequent repetitious operations. Because the model is peer-to-peer, users are less dependent on additional, perhaps costly resources, at least when these are not critical.

An initial implementation of the model has been created. We are experimenting with the model both as a tool to aid in digital library functions, and as a possible replacement for some server oriented functions.

Keywords: Self-administering data, data access model, data management, peer to peer, distributed file system, asynchronous collaboration, file sharing, scalable update propagation.

1. Introduction

Central to the digital libraries enterprise are issues of creating, managing and facilitating access to collections of digital objects. At one extreme, emulating traditional libraries, a digital library may serve as finding aid, collection manager, repository, and distributor for a set of digital objects. Alternatively, these services may be disaggregated, with affiliated or independent services providing collection management service, repository service, and so forth. Disaggregation, we suggest elsewhere ([20]), can enable more "democratic" management of resources, with individuals and groups using library-like services to manage their own collections, and to make use of associated services.

JCDL'01, June 24-28, 2001, Roanoke, Virginia, USA.

As digital libraries become more democratic, their interaction with other aspects of the data management process will become more important. For example, a working group may want to house the latest draft of a document in a repository for general access. While they are working on the document, versions need to get passed around or shared, and, as work progresses, the draft in the repository will need to be updated. In addition to incorporation, repositories need to be able to disseminate their data to interested users, who may be able to use related services to comment upon or annotate the document. For example, a user reading the draft may appreciate receiving an updated copy when the draft is changed, or receive notifications of various sorts. Such scenarios represent a need for end-to-end data management, in which collection and repository management form only one piece, and whose boundary with personal and group information management become indistinct.

Here we propose a data management strategy that is largely concerned with personal information management issues as they relate to the larger picture of digital object management. While we are developing this strategy to support digital library tasks, such as incorporation and dissemination of digital objects into conventional digital library structures, the data model we present may have additional implications. In effect, we provide a peer-to-peer digital object management tool. This tool can support interaction with services, but it may also serve as replacement for them. Thus, we propose for consideration a more radical notion of a peer-to-peer version of digital libraries, with no services at all.

2. Scenarios and Design Goals

In this section, we consider a number of scenarios that motivate our design. Mostly, we express our frustration with current tools available for simple processes, and suggest what, to us, seem like more attractive scenarios. Below we present the data processing model we designed to enable these scenarios. Then we describe our initial implementation.

2.1 Co-Authoring across Administering Domains

Example Problem: Suppose a web-page designer is commissioned to create some web pages from a customer. The customer somehow communicates what is desired to the designer, who then creates an initial version of these pages. Perhaps the designer sends these page drafts to the customer by email as attachments, or has the customer download the web-pages from the designer's web site, or uses some other protocol, like ftp, to move copies about. Later, the customer makes some modifications and returns the pages to the designer, and the process iterates.

As a result, both users' email boxes, or file spaces, etc, get filled with email attachments of versions. These versions are often hard to

manage because there is no built-in version management support for email attachments, HTTP, or FTP. Our collaborators could instead try to use some collaboration tool designed for this purpose. Heavy weight document management system like Lotus Notes [10] or even Xerox's Docushare [11] are probably overkill for this purpose; moreover, they may require administrative commitments neither user can make. Web-based file sharing system such as www.desktop.com [17], www.hotoffice.com [18], WebEdit [6], i-drive [12] and BSCW [4], and synchronization services, such as FusionOne [13], provide an interesting alternative. However, such services don't provide control over important aspects of data management, such as back-up, conversion, and merging. Moreover, the users are at the mercy of a potentially overloaded server, perhaps at a precariously financed dot com. Also, adding a third party to the interaction introduces increased vulnerability: Users are not able to perform their sharing operation when the central server is down even though their local machines and services are functioning, and have introduced a new security concern. In addition, they are subject to various, and, we think, avoidable, human errors, such as forgetting to transmit the shared copy to the web repository after every change.

Desired Properties: The above scenario suggests to us the following properties of an ideal system for this task:

- P1: No repeat user involvement in routine data management
- P2: No unnecessary dependence on shared resources, such as shared data repositories or file servers
- P3: No prior administrative set up costs
- P4: Ability to exploit minimal use of central server as only required
- P5: Undo/Redo capability within user's domain
- P6: Secure and safe incorporation of updates at user's domain
- P7: Lightweight enough to be widely deployed

A Proposed Solution: We propose a way of accessing and managing data to achieve the above desiderata. We introduce an infrastructure of *Self-administering Data Handlers*, which are deployed wherever users wish to take advantage of their services. These Self-administering Data Handlers (SD Handlers) administer data according to an attached *Self-administering Data Description*. The Self-administering Data Description (SDD) is metadata describing how, where, and to whom the data are to be copied, updated and otherwise administered. In other words, the SD Handlers are daemon processes that administer data by honoring attached self-administering descriptions.

Consider how the task above might be performed if SD Handlers were available to the collaborating parties. When the web-page designer creates web-pages, she saves them into a directory or folder somewhere on her local disk, as is her standard practice. Her SD Handler detects this action and pops up a UI with a self-administering description for the saved web pages, probably representing her defaults. She examines the default preferences, checks a couple of choices and adds a new destination, in this case, a location specifier provided by the client. Then the SD Handler attaches to the data objects their respective self-administering description.

Suppose the designer specified that these pages should be delivered to client's public web folder whenever she updates one of those. When a page is updated, the SD Handler will automatically sign it with the designer's private key and encrypt the result with the client's public key. The signed and labeled data object is deposited into the network of SD Handler infrastructure.

The client's SD Handler receives and verifies the authenticity of the self-administering description. In this case, it interprets the description as instructing incorporation of the data object into client's web folder. The client's SD Handler logs this event of data incorporation. If the designer's name is not found in the client's *trustee list*, the incorporation is denied. If the recipient's SD Handler is not available, the SD Handler could retry or delegate the retrying of delivery to a pre-negotiated server.

If the designer prefers strong update serialization, the SD Handler might be configured to first contact a pre-negotiated central serialization service, (say, a CVS[8] server or the Oceanstore [2] service) and have her changes merged according to the arrival order at the central serialization server. The merged data are then delivered back to the designer's SD Handler, which forwards the merged data to the destinations specified in the self-administering description.

Such a network of SD Handlers provides a lightweight asynchronous collaboration infrastructure for sharing data in a secure way. Centralized servers may be exploited in this process, but only when there is some particular need that justifies the cost, such as a desire for strong serialization of updates.

2.2 Data-Collection

Example Problem: Let us briefly consider a less desktop-centric scenario. Suppose a botanist takes pictures of plants in a field with her digital camera. She wants to transfer these to multiple remote designations, including her own web page, her research group's database, and her collaborator's disk. To do so, she must go to her office desktop and download the image from the digital camera into some buffer space, and then copy it into her own web page folder. She must then open up a database connection, authenticates herself, and then upload the data into database. She would also pop up an email client, create a new email message and upload the image data as the message's attachment. Then she sends the email to her collaborators, asking them to download the attached image onto their disk space. She repeats these procedures whenever she takes a picture or pictures she wishes to so incorporate.

This scenario provides comparable desiderata to the initial one, except that one would like to deploy our proposal as close to the data as possible. Thus, we must modularize SD Handler functionality so we can implement its services within a device's limited resources. For example, the camera might be enabled with a simple interface for using some pre-downloaded self-administering descriptions. Services that the camera couldn't perform locally could be performed by an affiliated proxy server. The camera need only reach the proxy server for the rest of the tasks to be automated as above.

We envision data collection involving SD Handlers from a wide variety of a simple special purpose devices, include scanners, smart cards, smart mobile phones, PDAs, and lightweight widely distributed sensors. These devices, perhaps together with a helpful proxy, simply deposit their tagged data into the infrastructure, which takes care of all routine data management and transport issues.

3. Self-administering Data Model

As suggested above, we envision a network of SD Handlers, each "close" to a user or device that it serves. To a first approximation,

there would be one SD Handler per networked device, perhaps more. Some would be associated primarily with users, some with data collection in devices, others with services, such as digital object repositories, each supporting basic SD Handler functionality, but perhaps implementing services associated with the particular characteristics of its application. Such a network is illustrated in Figure 1.

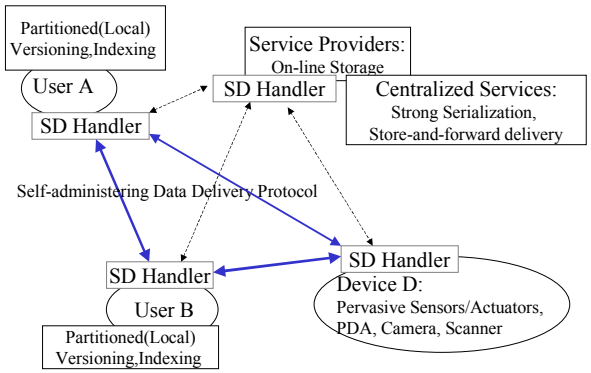

Figure 1: Network of SD Handlers

SD Handlers form a network, within which data is moved in accordance with the SD Handler's discipline. In addition, each SD Handler may provide an interface to a local collection or stream of data. The data may be a user's file system, web space, database, or other collection, administered by some mechanism other than the SD Handler. While these may be administered by a wide variety of mechanisms, the data looks the same once it is with the SD Handler network. We refer to each diverse collection of data as a data *realm*. In effect, the SD Handler bridges a realm into the SD Handler infrastructure.

Figure 2 presents an overview of the architecture of each SD Handler. SD Handlers are required to implement the bottom tier of the architecture, we define its basic functions. These are named bottling, floating, popping, and logging, and are described further below. To exploit capabilities fully, however, it is recommended that SD Handlers also implement an additional tier of functions on top of the basic services. These are called notifying, diff-ing, and versioning. Applications of various sorts may be built on top of these functions. In addition, GUIs and API need to be provided, to communicate with the user, and to form a bridge between the SD Handler and the user's data realms.

3.1 Basic Functionality

Here we present the basic building blocks of SD handling. These are *bottling*, *floating*, *popping*, and *logging*. We then describe how other functions can be built on top of these basic functions.

Prior to this process, a self-administering data description is attached to the data object, akin to creating a packing slip for a shipment. This description contains the shipper's preferences for handling the data, as well as the lists of recipients and/or destinations. The data preference can include high-availability, strong-serialization and default archival support. The destinations can be an on-line storage of collaborator's, a database, a PDA, a smart phone, a speaker/media device and pervasive sensors/actuators. In accordance with this packing slip, at some point, the data object is *bottled*, i.e., prepared for shipping. To make a data bottle, the labeled data object is signed and encrypted. Then the bottle is *floated* across a sea of data. Finally, the bottle is *popped* at its destination(s), and the data extracted. All events are logged, so that support for other services, e.g., version control, and be readily accommodated.

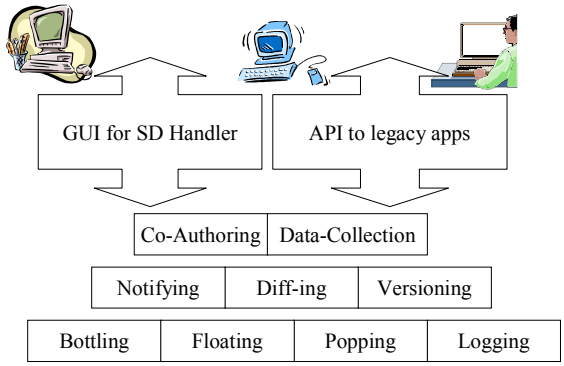

Figure 2: The Tiered SD Handler Architecture

3.1.1 Preparation

Prior to a SD Handler performing any operation on a data object, the object must be bridged into the infrastructure. I.e., a SD Handler has to be made aware of the object, and of the user's specifications for it. This is done by attaching a Self-administering Data Description (SDD) to the data. Since a self-administering data description can have many options and get quite complicated, we assume that most users never work with one directly. Instead, users interact with a UI. We have implemented a standard UI for a SD Handler running on desktop, which we discuss below. We assume that a different UI would be suitable for different devices, and that there would be default description templates for each user and each device, perhaps inherited or cascaded together as a function of the user and device environment.

Once an object has a SDD attached to it, the SD Handler aware of it will begin monitoring the object and attempting to enforce the specification of the SDD. Doing so typically results in sending a copy of the object to one or more recipients.

3.1.2 Bottling

When a SD Handler decides it must send a data object to a recipient, it first prepares a data *bottle*. It does so by signing the self-administering description and its data with its user's private key, for authenticating the sender at the recipient's SD Handler. The result is then encrypted with the destination's public key so that only the real destination can access the description and the data. The sender's credentials are checked against receiver's trustee list to allow appropriate access in incorporating the data at the destinations. Then the bottle is floated, i.e., dropped into the SD Handler network. We describe floating below, but first examine the inverse operation of bottling, *popping*, which occurs when a SD Handler receives a bottle destined for a known user.

3.1.3 Popping

A delivered bottle is inspected for its integrity and the sender is authenticated for appropriate access right. Then the bottle is uncapped with matching encryption keys to be incorporated into the destination realm according to the packing slip. For safe incorporation, every incorporation is logged for undoing or redoing operations.

The trustee-list maintained by SD Handler is used for giving or denying the delivery action from the sender. When SD Handler daemon process receives the bottled data, it authenticates the sender with trustee-list and decrypts the self-administering description to guide the incorporation activities.

Incorporation is based on appending; SD handling never overwrites data, but may shadow it. Since every incorporation is logged, it is always possible to revert the changes back to a specific version.

The bottled data is incorporated through SD Handler into any number of places, and in any number of different manners: onto a user's desktop, PDA, collaborator's domain, online-storage (NFS, Web), database entry, and even subdocument elements, such as anchor points in HTML page. The SD Handler running on a desktop computer maintains the history for the versioned content, and the incorporation activities. If the destination is database, the incorporation could comprise adding new entry; if the destination is a collaborator's online storage, the incorporation may create a newly updated file in a sandboxed location.

The followings are the examples of incorporations at various destinations.

A bottled data delivered

- onto UNIX file system, creates an i-noded file.
- onto a database, creates an updated (appended) database entry.
- onto a repository, creates a new index entry and is moved into repository space.
- onto a speaker device, creates voice data at the speaker
- onto another trusted user's file system, creates an i-noded file in a sandboxed location.
- onto a calendar/address book in a personal information managing application, creates anchor contains new data or new hyperlink pointing to a file in a sandboxed location.
- onto an anchor in a HTML document owned by another trusted user's, creates an anchor contains new data or new hyperlink pointing to a newly updated data in a sandboxed location.
- onto a writable CD, creates a newly added data on the writable CD

3.1.4 Floating

A bottled data object is dropped into the SD Handler network infrastructure. The infrastructure provides the delivery of the bottled data to the destination, as illustrated in Figure 3. SD Handler has its own delivery protocol (SDDP: Self-administering data Delivery Protocol) but SD Handlers can also uses legacy protocols such as HTTP, FTP, and SMTP by tunneling SDDP.

The floating architecture of SD Handler provides store-and-forward service for delivering the data to a recipient who is not available at the time of delivery. It also provides the update serialization service, where the updates from the multiple participants are serially ordered according to the arrival time at the server. The bottled data has to flow into the serialization server and flow out to its destinations.

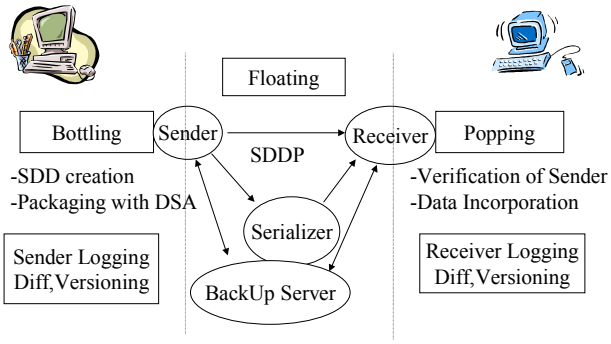

Figure 3: From Sender to Receiver

Finally, the infrastructure of network of SD Handler provides a naming service to map the user's SD Handler's location into its current IP address. Each SD Handler that does not have static IP address, register its current IP address to the name resolution server in the infrastructure. And the SD Sender would cache the latest mapping and use it until the host becomes unreachable, and then it contacts the name resolution server for the current IP address of the participant's SD Handler.

3.1.5 Logging

The data and packing slip and its bottling/popping activities are logged for undoing/redoing and auditing purposes. The logging history can be flushed to a designated archival repository from the local space of the bottling and popping point.

The log can be incremental in that the delta of changes is recorded. Doing so, of course, increases the dependency between logged objects, although it saves the disk space.

3.2 Advanced Functions

There are a set of functions that are useful, but not required, of compliant SD Handlers. We describe these here.

3.2.1 Versioning

Each SD Handler maintains its own version tree at its own realm by enhancing the logging feature. Decentralization is achieved by naming the same resource uniquely along with its version number across different administrative domain. This is done by prefixing the owner-name to a local name of resource, as each SD Handler has its own name space per given owner-name. We use the owner name to uniquely locate its public key. The typical owner name could be email address where the uniqueness is being maintained at its organization or email service provider. In CVS[8] and WebDAV[7] there is one version tree that is maintained at the central server with its single name space. In contrast, in SD Handler, different version trees are maintained at each realm. They share only a naming convention that uniquely addresses identifiable resources across different version trees.

3.2.2 Diff-ing

Given two unique resource names (which includes version numbers), the SD Handler shows the differences of the two data objects. If one or both of data objects are not available, SD Handler looks at the self administering description and contacts the SD Handler which maintains the logs of requested data. Users should be able to diff the changes before and after the transaction so that one

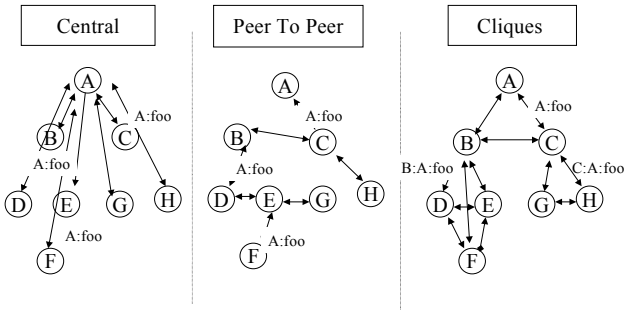

Figure 4: Update Propagation Models

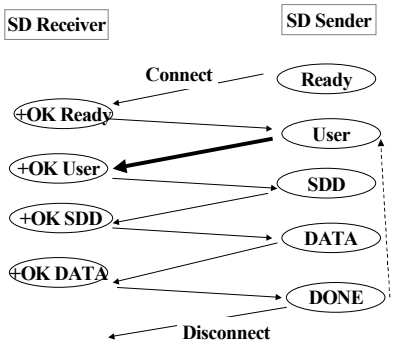

Figure 5: Self-administering Data Delivery Protocol

can verify that the automated updates from trusted source are meaningful.

3.2.3 Notifying

The SD Handler provides a notification of the basic activities to the end user. This can be achieved by simply showing the log entry that has been added most recently. A user's latest activity involving the same data that is under the established self-administering control such as saving after modification, deleting, would be monitored by SD Handler and notified to the user for further actions.

3.3 Scalable Update Propagation Model

We have developed a scalable update propagation model based on cliques. We define a clique as a strongly connected group of users who share the same SDD. As shown in Figure 4, the update for the data, A:foo, is reported back to the central coordinating user/server, A in the diagram, and then propagated back to the rest of the participants in the central model. In a peer-to-peer model, as shown in Bayou's anti-entropy algorithm [3], the updates are reconciled among peers in an arbitrary order whenever they are connected to each other; then the update is propagated to other members. In our model, changes are immediately propagated within cliques; secondary propagation to other cliques is through the junction point, as shown in B, C in the "cliques" diagram. We can imagine that B and C can filter and aggregate the changes made by members in the clique, so that the tiny changes made within B's clique and C's clique are not visible to A. We believe this approach will fit naturally to group interaction.

Both central and the peer-to-peer propagation models use the same name for all the propagation. However, in our model, the naming carries more clique interaction information. For example, given C:A:foo, one can deduce that this object originated from data A:foo and that user C created a new clique by creating a new SDD associated with C:A:foo. The SDD for A:foo would have a different member's policy and serialization preference. For the data, A:foo, the clique composed of A,B,C would specify strong serialization and back-up support, while, the clique composed of B,D,E,F might not need to use such strong serialization and back-up support among them.

3.4 Self-administering Data Delivery Protocol

The Self-administering Data Delivery Protocol is illustrated in Figure 5.

The SD Sender parses the SDD file written in SDDL (Self-administering Data Description Language) and makes a connection to each Sharer's SD Receiver. When it receives the "+OK Ready" message from the SD Receiver, it sends the SD Sender's user id, (generally an email address). The SD Receiver then checks the membership of this user's id in its SD Receiver's trustee list. If it is not on the list, the SD Receiver refuses the connection and the SD Sender sends an email notification to the user of the SD Receiver. If the user id is found on the trustee list then the SD Receiver tries to verify whether the user of the SD Sender is who she claims to be. For user authentication, we use a public-private key encryption system, where the user of the SD Sender's identity are verified using a digital signature. To avoid the man-in-the-middle-attack, all the links between the SD Sender and the SD Receiver can be encrypted.

After the authentication is finished, the SD Sender sends the SDD to the SD Receiver, followed by the DATA command. The SD Receiver then parses the received SDD and follows its description, which may involve synchronizing the shared document with latest copy given by the DATA command.

If the more files are coming from the SD Sender, it repeats its protocol from the authentication point onwards, until all files have been received; then the connection is closed. The SDDP thus provides a selective delivery acceptance mechanism using a trustee list. Only the owner of the trustee list can add a new user id into its own trustee list.

3.5 Self-administering Data Description (SDD)

The basic responsibility of a SD Handler is to interpret SDDs, i.e., a Self-administering Data Description attached to a data object. SDDs are written in SDDL that is based on XML. Figure 6 provides a typical example.

We stipulate that every SDD has one owner, but may contain multiple users as sharers. Each user in the sharer element can have its own multiple self-administering data server locations. For example, the user "Robert Wilensky" in the following example contains four different SD server locations, the last of the four being his own archival server.

The archival server is an instance of SD server where the SD Handler governs the archival repository for its subscribed users. The owner "B. Hoon Kang" provides its own archival server to be accessible by the sharer. The central server element provides the central server location for SD Handler's "floating" operations (to be

described below), such as store-and-forward data delivery, serializing the updates at a central location, and dynamically mapping the user's SD server name into its current IP address.

```
<SELFDATA
 ownername="B.Hoon Kang"
 UAN =”dlib2001 selfdata paper”
 archivalsupport ="yes"
 secureaccesscontrol="yes"
 consistency="yes"
 availability="high"
 changenotification="always">
<UAN name = “dlib2001 selfdata paper”>
 <ITEM location = “selfdata01.doc”/>
 <ITEM location = “diagram-image01.gif”/>
 <ITEM location = “diagram-image02.gif”/>
</UAN>
<SHARER coherency = "SERIALIZE" >
 <USER name="B. Hoon Kang"
       id="hoon@cs.berkeley.edu"
       initial="B.H.Kang">
  <SELFDATASERVER
  name="alpine.cs.berkeley.edu" port="7070"
  location="Soda Rm 493" />
  <SELFDATASERVER
  name="sb.index.berkeley.edu" port="7070"
  location="145 wilson st" />
  <ARCHIVALSERVER
    name="dlibarchiver.cs.berkeley.edu" port="7070" />
  <ALTERNATEEMAIL
    name="hoon@now.cs.berkeley.edu" />
 </USER>
 <USER
  name="Robert Wilensky"
  id="wilensky@cs.berkeley.edu" initial="RW">
  <SELFDATASERVER name="bonsai.cs.berkeley.edu"
   port="7070" />
  <SELFDATASERVER
   name="mobile-ip.cs.berkeley.edu" port="7070" />
  <SELFDATASERVER
  name="home-ip.eecs.berkeley.edu" port="7070" />
  <ARCHIVALSERVER
  name="myarchiver.eecs.berkeley.edu" port="7070" />
 </USER>
</SHARER>
<CENTRAL>
 <CENTRALSERVER
   name="galaxy.cs.berkeley.edu" port="7070" />
</CENTRAL>
<ARCHIVE>
 <ARCHIVALSERVER
  name="galaxy.cs.berkeley.edu"
  port="7070" />
</ARCHIVE>
</SELFDATA>
```

Figure 6: A Typical Self-administrating Data Description

A UAN (Unique Activity Name) is used to uniquely specify a SDD. I.e., there is one to one mapping between UAN and SDD. The Activity Name is unique within the owner's "realm", or unit of computing administration, as known to the SD infrastructure. In the SD Handler's network, the GUAN (Global UAN), which is composed of owner-name and UAN, is used to uniquely specify the data of interest. The activity is defined in 4.1.2

The UAN tag contains one or more items, each of which maps into a data handle. There are a variety of different types of handles, depending on the nature of the realm. For example, when data item is from a file system structure, then the data handle would be a file or directory name.

4. Implementation: Self-administering Data Handler (SD Handler)

We have built a prototype of the SD Handler in Java. We have modularized the basic tier functionalities and have built the GUI for desktop environment. We have used the Java's built-in security library for public/private key management and its encryption/decryption. In our experimentation, we have focused on co-authoring as our initial target application.

4.1 SDD Viewer/Editor (Bottling)

4.1.1 From Legacy File System

In order to bring data into SD Handler's world, we need to create its SDD first. As shown in the top of Figure 7, the user can browse the file or directory through a File Tree Viewer. If the user places a cursor on a file or directory that has previous SDD, the SDD Viewer/Editor shot will pop up, as shown in the bottom of Figure 7. If this is a new SDD, an empty SDD Viewer/Editor pops up. The user can add or delete sharers and can specify the preferences. (In the current prototype, the preference fields for strong serializing server, store-and-forward server, back-up repository server are associated per SD Handler daemon process, not per SDD instance.) Then, the user can synch out the associated data, using DSA (Digital Signature Algorithm), to every participant: The SD Handler's bottling functionality wraps the content with SDD and encrypt with the sender (owner)'s private key and the recipient's public key. The owner is asked to type in her pass-phrase to authenticate the use of her own private key. The delivery status is recorded.

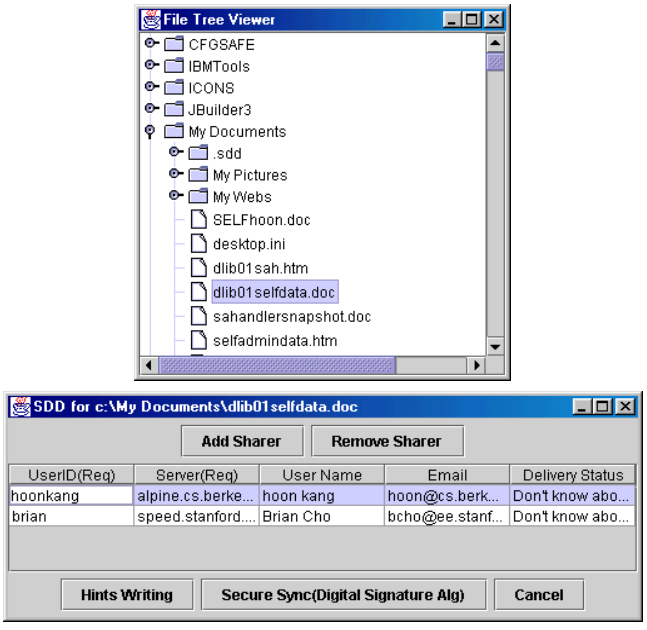

Figure 7: SDD Viewer/Editor from File Tree Browser

4.1.2 *From Activity Browser*

We have found that it is cumbersome to remember to go to the file or directory every time we want to use SD Handler's bottling service. Therefore, we have introduced the *activity* as a mnemonic reminder for the collection of items that share the same SDD. Each item is a representative of data unit such as a file and an image. Thus, the activity is a unit of SDD association. A new activity can be created explicitly from an activity browser and implicitly from file tree browser. A new activity named as the data handle is created implicitly when a SDD is associated with data file from the file tree browser. Later, the user can find the SDD using either the activity browser or the file tree browser. In order to provide the most active activity item in the first page of the activity browser window, the activities are sorted according to their number of accesses and by the priority that it is given by the user.

We also prototyped the activity browser tree to provide hierarchical activity management, however, the complexity of tree management to the user seems to outweigh the benefit of SDD inheritance in a hierarchical activity management. The activity interactions are illustrated in Figure 8.

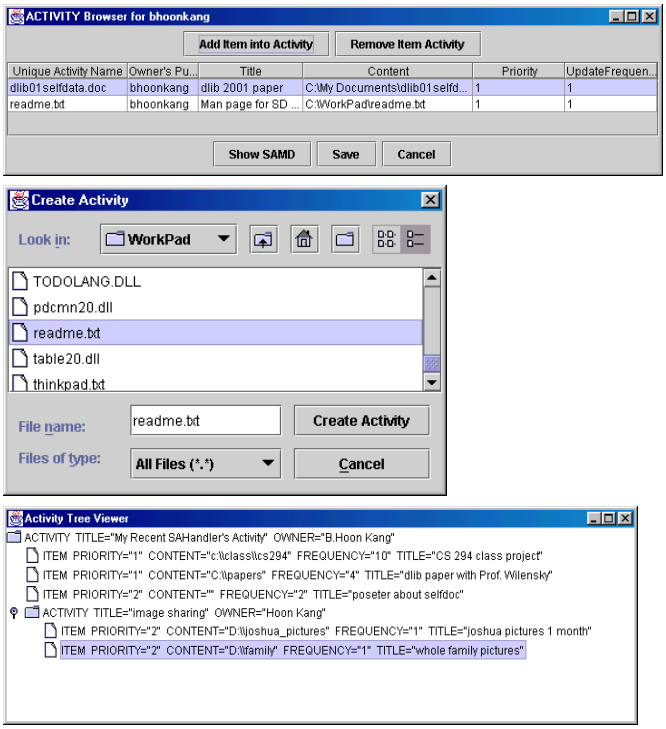

Figure 8: Activity Browsers: flat(top), hierarchical(bottom)

4.2 Public/Private Key-Store Manager

The Java framework provides a keystore architecture and a command line interface, keytool. We provide our own public/private keystore manager to store other collaborator's public key and trust level. (See Figure 9.) The trust level "new" means that the associated public key has never been added to the owner's keystore before and the owner of keystore has to decide whether to accept the key or not. If the owner accepts the public key with new status, the status is changed to "trusted". If the owner denies the public key of a collaborator, the status is set to "untrusted" and further contact from this source is denied at the earliest stage in the SDDP protocol stack.

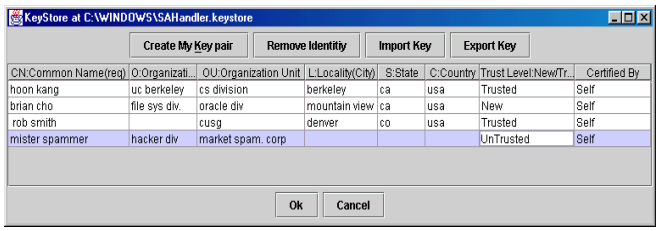

Figure 9: Key Store Manager

4.3 SD Sender/Receiver (Floating)

Our current prototype uses SDDP to deliver the bottled data directly between SD Handlers. The desktop version of the SD Handler has both the SD Sender and the threaded SD Receiver. We designed the SD Handler to minimally use the centralized highly available servers for common P2P infrastructure services such as store-and-forward delivery, naming/tracking the current location (IP address) of SD Handler, and the strong serialization of updates. The infrastructure services are not being shared among SD Handlers across administrative domains. Rather, each SD Handler can subscribe to its own infrastructure services.

If they don't use common P2P infrastructure services, the SD Receiver has to be available to receive a data delivery from the sending SD Sender. However, we have come up with a simple solution where the SD Handler of the initial creator of the SDD is used as an infrastructure service point for the duration of the interactions unless one of the participant's SD Handlers specified in SDD provides the infrastructure services for their interactions. For example, if the recipient's SD Receiver does not have store-and-forward delivery service, it can contact the creator of its SDD for the latest logged copy that it might have missed. As a same token, each participant can register its current IP location to the creator of its SDD.

4.4 Receiver Log Viewer (Popping/Logging)

When self-administered data is sent to a receiving SD Handler, the receiver log viewer (See Figure 10.) records the receiving activity, noting its UAN (Unique Activity Name), author (sender), date, and author's note. If the sender is a new/trusted contact, the receiver first creates a versioned copy and then records the data handle to it, then the incorporation status is set to "Pending". The owner can accept or deny the incorporation of this versioned copy into his own realm. The incorporation status shows whether the current logged activity has been accepted or denied. If the incorporation is successful, the status will be set to "Accepted", if not it will be set to "Error". If the user distrusts the sender, then the sender's public key is registered into the owner's keystore as "Untrusted". If the user denies the incorporation entry, then only the associated update is denied and the incorporation status is set to "Denied".

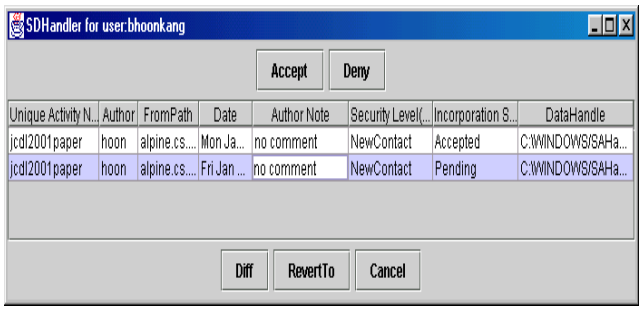

Figure 10: Receiver Log Viewer

Table 1: Comparison of SD Handler with Legacy Applications

Desired properties	Email (SMTP)	CVS/ Web-DAV	FTP	ICQ/ AIM	Groove	SD Handler
P1	No	No	No	No	No	Yes
P2	Yes	No	Yes	Yes	Yes	Yes
P3	Yes	No	No	Yes	Yes	Yes
P4	Yes	No	Yes	Yes	Yes	Yes
P5	No	Yes	No	No	No	Yes
P6	No	No	No	No	No	Yes
P7	Yes	Yes	Yes	Yes	Yes	Yes
P to P	Yes	No	No	Yes	Yes	Yes
Goal	Message exchange	Central version	File transfer	Instant message	P2P services	Self-admin

5. Comparisons

5.1 Comparison to "Legacy" Applications

We compared SD Handler with Email, CVS, and FTP, according to the seven desiderata listed in 2.1. A summary of the comparisons is given in Table 1.

P1. No repeat user involvement in routine data management

All of the applications except SD Handler require repeated user involvement in copying, moving, and sending the data. SD Handler requires SDD creation once for repeated usages.

P2. No unnecessary dependence on shared resources, such as shared data repositories or file servers

Email does not require shared resources for collaboration; CVS require a shared server location.

P3. No prior administrative set up costs

Both CVS and FTP provide the password-controlled access to the data that is being shared among collaborators. Either group account or individual account needs to be set up by an administrator, and need to be distributed to each collaborator to access the data prior to the collaboration. In Email or ICQ [14] or AIM [16], however, the password is not required to send or receive the message and its attached data. The access is purely controlled by the user's discretion whether to accept or refuse the attachment. An orthogonal end-to-end security method, for example, PGP[9] email, could be added. Both the SD Handler and Groove[15] provide public/private key based access control to the data without requiring prior administrative account set up. The user's discretion is guided by the key issuer's certificate or web-of-certificates.

P4. Ability to exploit minimal use of central server as only required

ICQ provides this property, so as to be scalable when their central server is contacted for name resolution of recipient's current IP address and store-and-forward data delivery to the unavailable recipient. By this measure, web-based file sharing systems over-utilize their central server in terms of the network bandwidth, processing power and disk space.

P5. Undo/Redo capability within user's domain

Only CVS support this.

P6. Secure and safe incorporation of updates at user's domain

Email could use DSA (Digital Signature Algorithm) for end-to-end security but the incorporation of email attachment is not sandboxed. ICQ and FTP do not provide safe-guarded incorporation either. CVS's undo capability could provide a safe incorporation since one can go back to the previous change in the case of an incorporation error.

P7. Lightweight enough to be widely deployed

All the applications above are considered to be lightweight since they do not require a heavyweight server infrastructure.

5.2 Declarative vs. Session-Based Data Management

FTP, NFS, HTTP, and derivative applications (e.g. WebDAV[7]) require a session with a resource controlling a data object in order to create, update, move, delete, or otherwise manage that object. Moreover, during this session, the data are managed by procedural commands. Network file systems, e.g., AFS and NFS, basically provide file semantics in sharing data, so, once again, intentions are expressed procedurally. Ficus [5] , Bayou (peer-to-peer optimistic file replication [16]), and Rumor (user-level replication system [19]) use file sharing semantics, and hence are fundamentally procedural as well. Also the overwriting semantics of file systems does not provide the knowledge about who made which changes. Hence one would have to use versioning software like CVS[8] in an explicit way, requiring the user's involvement in setting up check-in, check-out and copying.

In contrast, the SD Handler model provides a declarative way of managing data across administrative domains in a wide area scale. The SD Handler model also enables the user to specify that the data needs to be versioned at the different administrative domains. We believe this model can simplify data management, achieving our goals of minimizing the user's participation in routine tasks.

5.3 Scripted Email Attachment

A SD Handler can perform incorporation of received data into the recipient's internal data storage. A similar effect can be achieved by running a script (VBScript, UNIX shell script) with an email attachment. However, as is well-known, doing so is dangerous since the script can run any arbitrary command. However, the SD Handler's incorporation operation is sandboxed within SD Handler's address boundary where the access is limited only through the sanitized SD Handler's incorporation functionality.

5.4 P2P (Peer to Peer) Systems

ICQ [14], AIM [16] provide a peer to peer instant messaging with infrastructure services such as identity (user account) management, store-and-forward delivery and dynamic mapping of user's current IP address. We have found that these infrastructure services are common to most P2P systems. For example, Groove [15] provides collaborative P2P software tools with just these infrastructure services. The "shared space" in Groove provides an interactive collaborative environment where various applications (tools) can be built upon such as instant messaging, file sharing, free form drawing, and real-time conversation. However, unlike SD Handler, the management of data still requires repeated user interactions. The delivered files (attachments) have to be manually downloaded and saved. Versioning and logging are not provided since the

incorporation of data is not automated but depends on manual end-user commands. Moreover, Groove and ICQ/AIM assume each peer to be an end user; in SD Handler the peer could be a personal repository server, a back up server, and a device in addition to other desktop users. Finally, Groove is focused on building a collaboratively shared space (or workspace) in a P2P way. SD Handler is focused on providing new semantics and controls for managing data with minimal user interactions. The co-authoring application is an example of using self-administering data model in a collaborative scenario.

6. Discussion

We have not yet had enough experience with our implementation to draw any forceful conclusions. However, we keep discovering new applications for self-administering data as we progress. For example, with an appropriate SDD, mirrored copies of documents can easily be made, in effect implementing a peer-to-peer RAID, or eliminating the need for tape backups. Doing so makes sense, as the loaded cost of tape backup is probably about one order of magnitude greater than the cost of low-end disk on a PC.

Self-administering data seem especially useful for data incorporation. For example, as in our Personal Library service [20], users may want to place a document in a scanner, and have it incorporated in their repository and added to a collection they maintain. Providing a SD Handler at the scanner provides a simple means to accomplish this goal.

Self-administering data can be used to update data managed by particular remote programs. Consider contact information inside a remote program. One can have one's contact information in an object whose SDD instructs that it be sent to one's contact list upon modification, and that the remote "contact update" program be executed. The remote user would need to specify what such a program is. The result, though, is that everyone's contact information for an individual can in effect be edited by the individual to whom it refers.

Self-administering data can provide an alternative to traditional digital library functions. Documents can be made available to a community of users by copying, rather than via repositories. Again, doing so may make sense, depending upon the cost of servers versus the cost of low-end machines, and the kind of availability one desires.

On the other hand, one-to-many communication is costly in this model, as we currently encrypt each communication on a per user basis. One possibility is to provide an option for users to trade off security for communication efficiency, in the case of one-to-many transactions.

7. Acknowledgements

This research was supported by the Digital Libraries Initiative, under grant NSF CA98-17353.

8. References

[1] Randy Katz, et al. The Endeavour Expedition: Charting the Fluid Information Utility, http://endeavour.cs.berkeley.edu/proposal/

[2] John Kubiatowicz, et al, OceanStore: An Architecture for Global-Scale Persistent Storage, Proceeedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000), 2000. http://oceanstore.cs.berkeley.edu/

[3] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP-16), Saint Malo, France, 1997, http://www.parc.xerox.com/csl/projects/bayou/.

[4] R. Bentley, W. Appelt, U. Busbach, E. Hinrichs, D. Kerr, S. Sikkel, J. Trevor, G. Woetzel, Basic Support for Cooperative Work on the World Wide Web. International Journal of Human-Computer Studies, 46(6), 1997. http://bscw.gmd.de/index.html

[5] T.W. Page, Jr et al, Perspectives on Optimistically Replicated, Peer-to-Peer Filing, Software Practice and Experience, v.28, n.2, February, 1998, http://ficus-www.cs.ucla.edu/travler/ficus_summary.html

[6] Ken Pier, Eric A. Bier, Ken Fishkin, Maureen Stone WebEdit: Shared Editing in a Web Browser. WWW4 Poster Proceedings, 1995. http://www.parc.xerox.com/istl/groups/gir/doc/webedit/webedext.htm.

[7] Jim Whitehead, Collaborative Authoring on the Web: Introducing WebDAV, Bulletin of the American Society for Information Science, Vol. 25, No.1,1998, http://www.webdav.org/papers/

[8] CVS (Concurrent Versions System), http://www.cvshome.org/

[9] PGP (Pretty Good Privacy), http://www.pgpi.org/

[10] Lotus Notes, http://www.lotus.com/

[11] Xerox Docushare, http://www.xerox.com/

[12] I-drive, http://www.idrive.com/

[13] FusionOne, http://www.fusionone.com/

[14] ICQ, http://www.icq.com/

[15] Groove Networks, http://www.groove.net

[16] AIM, http://www.aim.com/

[17] Desktop, http://www.desktop.com/

[18] Hotoffice, http://www.hotoffice.com/

[19] Peter Reiher, Michael Gunter, Gerald Popek, Rumor: A User-Level File Replication Middleware Service, http://fmg-www.cs.ucla.edu/

[20] Robert Wilensky, Personal Libraries: Collection Management as a Tool for Lightweight Personal and Group Document Management (forthcoming).