# File-Based Network Collaboration System

Toshinari Takahashi, Atsushi Shimbo, and Masao Murota

*Communication and Information Systems Research Labs.*
*TOSHIBA R&D Center*
*1 Komukai-Toshiba-cho, Saiwai-ku, Kawasaki 210 Japan*

{takahasi,shimbo,murota}@isl.rdc.toshiba.co.jp

## Abstract

Computer-Supported Cooperative Work (CSCW) requires coordinated access to shared information over computer networks; such networks have tended to use wires, but wireless networks are now becoming common.

There are a large number of tools aimed at helping users to work cooperatively but these tend to be application specific, leading to proliferation and requiring a large amount of development effort. A more general purpose mechanism would keep the number of tools manageable, and would obviate the need to develop a completely new tool for each problem area.

Data security is also a very important requirement in distributed systems. A solution to the problems of cooperative working must take this security requirement into account.

This paper describes a mechanism aimed at both problems: a general purpose tool for cooperative working that is more secure than existing proposals.

Our approach is novel in that we do not require explicit locking, which can lead to a number of problems, particularly in distributed systems, as we shall explain. Client routines act upon user requests to insert or delete blocks in a file, and request a file-server to modify a shared file according to those requests. The file-server receives encrypted requests asynchronously and merges these requests into the current version of the document without decrypting the requests. Indeed, an interesting feature of our proposal is that the server could not decrypt the content of these requests even if it wanted to. We call this mechanism "privacy enhanced merging".

Our current implementation includes a concurrent editing application that we call "Network BBS"; the server is able to make use of a conventional file-system. This is an experimental tool of our proposed "Collaborative File System".

## 1. Motivation

Shared-data management systems have to be able to cope with recent advances in computer and network technology, such as CSCW over networks, wireless networks, and version-control mechanisms. They also have to be able to take into account security requirements such as data encryption and user authentication. Actually, we often meet such situations when we want to allow a person who does not belong to our domain to edit a specified file we own without giving a user account on our domain. Our approach to these problems is a novel file system architecture which provides an asynchronous editing mechanism and encryption facilities.

Network distributed file systems allow users to share read-access to files, but concurrent modification of those files is more cumbersome. Traditional approaches to this problem include the use of a locking mechanism to provide mutual exclusion, but for various reasons this is often inconvenient.

For example, a user wishes to modify a file and so acquires the lock and starts an editor (in practice the editor is likely to acquire the lock on the user's behalf); while the user holds this lock nobody else may access the file. Alternatively, a user wishes just to read the file and so does not take out an exclusive lock; he then realizes that he needs to change the file, but in the meantime others have modified the file under his feet. Sometimes these problems are merely a nuisance, but sometimes they can lead to the destruction of valuable information, particularly if users are tempted to override the locking mechanism because of its inconvenience. Previous work concerned with such problems has focused on the provision of a comprehensive library of editing primitives [1].

Our system does not require a user to take out an exclusive lock when he or she wishes to modify a file, nor will he or she be frustrated by finding that somebody else holds such a lock. Users take local copies of a file and operate concurrently on those; a file comparison utility such as "diff" [2] determines what changes were made. This allows users to continue to use their favorite editors, such as "vi" or "emacs". The client sends a list of differences, insertions and deletions, as modification requests to the file server. Requests are sent to the server asynchronously and the file server merges these requests into appropriate positions in its stored version of the file. This uses information we call "target versions" which we shall describe later in this paper. A similar approach was used for the CVS system [3].

We realize that our approach does not use a strict consistency mechanism, but instead relies upon a certain amount of common sense on the part of the users. In practice this is sufficient to address many of the requirements of modern distributed systems. It will still be necessary, however, to adapt security mechanisms to the needs of new technologies, in particular to new network technologies.

One commonly used approach to the problems of confidentiality across networks is that of PEM (Privacy Enhanced Mail) [4]; this is particularly convenient to users of wide-area networks. However, PEM is a dedicated application that does not address all of the needs of distributed processing. In particular PEM offers no facilities aimed at cooperative working. Rather than follow the PEM approach we have chosen to follow the example of CFS [5], which delegates responsibility for confidentiality to the file system rather than a mail system. We note that in the simple case where a user does not need to cooperate or communicate with anybody else, this provides confidentiality where the mail paradigm would appear most inappropriate.

Our proposed system provides a concurrency mechanism for shared editing which is also suitable for disconnected or intermittent operation[6], and a confidentiality mechanism designed to protect against wiretappers who might eavesdrop on communication between a client and the file server. The system also provides protection against misbehavior of the file server itself. The reason this is useful is that it reduces the utility to an attacker of breaching the security of the server since such a breach will not necessarily lead to discovery of any confidential information. The current implementation provides only rudimentary safeguards against loss or corruption of data, but this is a topic worthy of further study.

Our design had to take into account the protection mechanisms, the most suitable data structures for the encrypted file system, a version-control system, the encryption algorithm, and the merging policy, in order to ensure that these interact in a harmonious way. The system provides shared editing and version control. The techniques chosen have other benefits; for instance they work well even if network bandwidth is limited, and they are resilient in the event that communication is interrupted occasionally. Also, an isolated user with no need of the concurrency mechanism may use the same system to provide version control and confidentiality.

## 2. Basic data structure

Figure 1 shows the structure of a hypothetical shared file. Every shared file within our system has a list of member IDs, a list of encrypted keys (actually the same key, FK as we shall describe, encrypted under several other keys, one for each member), and a list of data blocks. Member IDs are the names of users who are permitted to access the file; such a username is usually the internet e-mail address of the user, although we have considered other mechanisms by which even these names may remain confidential. The contents of each data block are encrypted under a common file key FK, which is not known to the server; users encrypt blocks before sending them to the server and decrypt blocks after receiving them from the server, so the server needs to process only ciphertext. Each user (member) can determine the file key by retrieving their appropriate entry from the list of encrypted keys and decrypting this using their own personal key.

Let us look at the data blocks in more detail. Each data block has associated "traverse information", "data characteristics", an "initialisation vector", and the encrypted contents. Traverse information describes the structure of a particular view of a file; for example the generation and deletion times of a block allow the system to form a view of a version of a file existing at a particular time, which is similar to the mechanism used by the SCCS(Source Code Control System). When a user (member) requests a particular version of a file the system links together the necessary blocks in the appropriate order - the member is responsible for decrypting the individual blocks.

In addition to constraining a version according to the times associated with a block it is also possible to associate other constraints. For example, one user may wish to retain a private version of a shared file (here we mean "private" in the sense that the user does not wish to confuse other users by showing them the file while it is in the process of being updated - we do not mean "private" as a security constraint). The blocks which distinguish a private version of the file from the public version may be tagged in the traverse information, so that they are not returned when another user attempts to read the file.

There is other information associated with each block. For example some blocks may not be enciphered because speed of access is

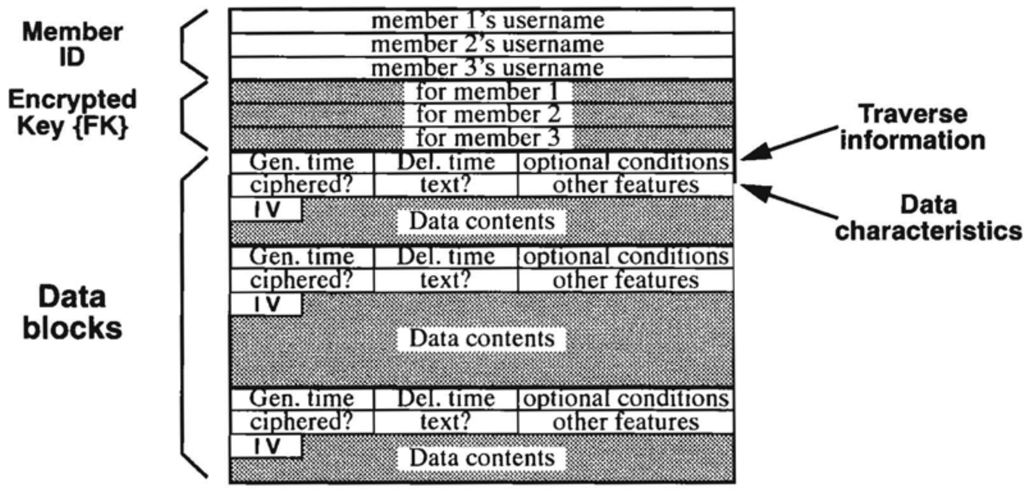


Figure 1. Data structure of shared file.

deemed more important than confidentiality. The data characteristics fields allow such attributes to be held with the corresponding blocks. Each enciphered block has an individual initialisation vector, IV, to protect against certain cryptographic attacks, although we shall not elaborate on this in detail.

We can also tag blocks with data-type information so that an appropriate viewer can be invoked for each of the different types of block within a file. At the moment, the system returns all text blocks so that they may be edited using a conventional text editor. Multimedia applications might tag certain blocks as being audio samples and perhaps others as digitised video. The mechanism is sufficiently flexible to accommodate other types of data that we have yet to imagine. For the moment, the file server is oblivious of these data types, leaving them to be handled by the client application, but we can imagine a server that treats different blocks in different ways.

## 3. File sharing strategy

Each shared file includes a membership list that contains the names of the members who are permitted to access the file; write access is also determined for each user independently of read access.

We describe the system as "asynchronous groupware"; that is to say that each user is happy to edit in isolation without being warned of changes by other users. This assumes a certain degree of prior agreement between users but we feel this to be a realistic expectation for many applications. The system provides consistency in that if two or more different users edit different parts of the same file concurrently, the result is the same as if each user performed their edits in turn.

Every read request from a client includes a "traverse condition" constraints upon which blocks should be returned. In order to decide whether to return a particular block the server tests the condition against the block. For example, a client might request a non-current version of the file - blocks have been inserted and deleted from this version to form the current version. When the client later writes back a changed version this will be tagged with the Original version ID (OID).

Every write request consists of an insertion or deletion and an OID as mentioned above. Insertion is represented as an offset into the file and data to be inserted at that offset. Deletion is represented as two offsets; the start and end points between which deletion should take place. Offsets are relative to the traverse condition implied by the associated OID. In other words the OID describes a file version, and implicitly numbers the bytes of the file. A different version might associate different offsets with the same data. Blocks that did not exist at the time of the OID are not numbered and cannot be referenced, which is the expected behavior. This strategy allows several clients to work on a file concurrently and each will see a consistent view of the file.

When data is inserted into a file this may require that a block be broken into two blocks to accommodate the change. The inserted data becomes a third block, distinct from the other two since they have different traverse information (time stamps in this case).

This is easier to understand given a diagram (figure 2). A file is created at time t0. A client received a file at time t1 and requested insertion at the 3rd byte, and deletion of the 14th and 15th byte. A second client received the file at time t2 and requested deletion of the 11th to 13th bytes. All of these requests are merged into the stored representation of the file. Note that the editor used by the client is oblivious of this behavior - the user may continue to make use of his or her favorite text editor, such as Emacs.

Client behavior is as follows. A client requests a particular version of a file, which is constructed by the server and returned to the client. All textual data blocks are merged into a contiguous sequence of text that can be processed by a conventional editor. Non-text

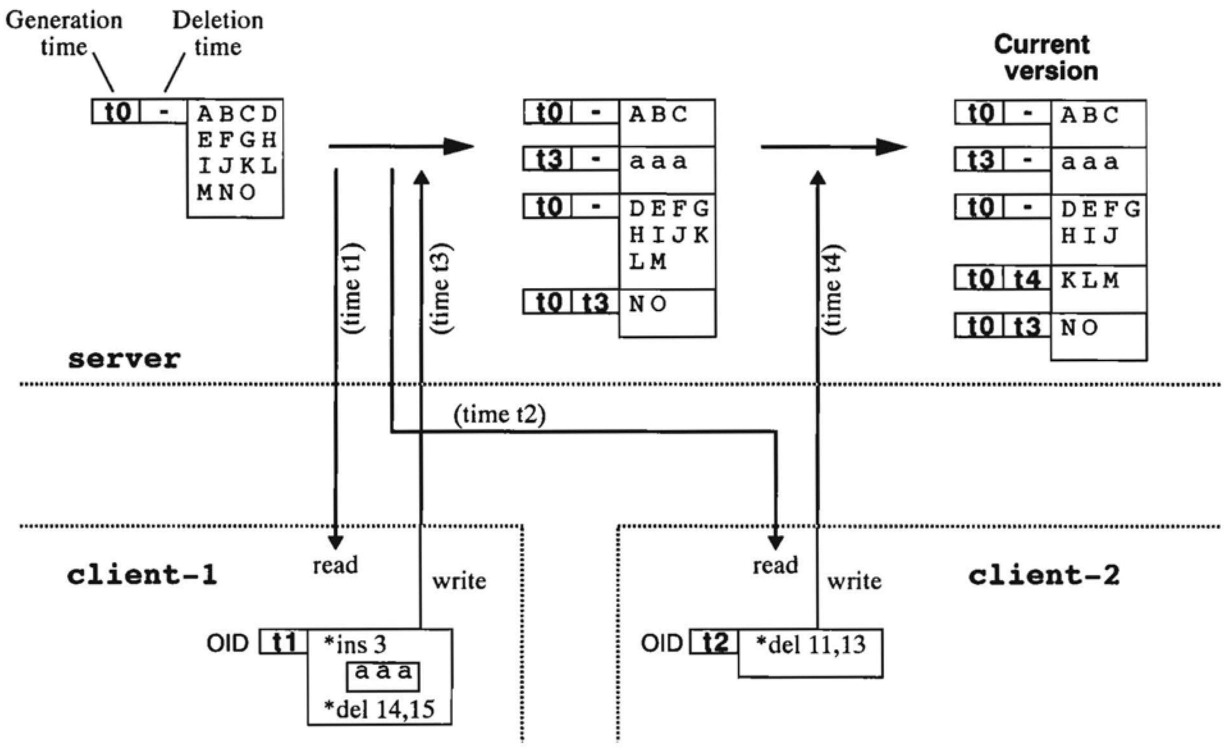


**Figure 2. Example of concurrent editing and file transition.**

blocks may be routed to different editors according to their type. Possibly a client might display certain data types, but allow the user to edit only a subset of what he or she sees; alternatively the editor might ignore all unrecognized blocks, treating them as though they didn't exist. This is not strictly true, since we probably do not want these blocks deleted when the file is written back to the server, but, as far as the user is concerned, it may as well be true.

After the user has finished editing the file (he or she performs whatever incantation is necessary to write the file to disc) the client determines a set of insertions and deletions that, given the original file, would result in the new version. For this purpose,, we currently use a novel file-comparison utility based on the "diff"[2] algorithm. This utility operates at the level of characters but an alternative difference tool operates at the level of words, which can be more understandable to the user. Non-textual data is not currently processed. In practice, many strategies are possible provided they result in the expected version of the file.

It is important that this process does not introduce semantic inconsistency. For example, if two users simultaneously insert data at the same place in a file. Such inconsistency may render a file unintelligible. The server can diagnose such a problem but is not responsible for resolving it. A client that retains the editing history of the file should be able to recover in the event of a conflict.

In the current implementation, as a default arbitration, if two users simultaneously insert data at the same place in a file, both are reflected in the file in last-insert-first-appear order; and if two users simultaneously delete the same part in a file, they are counted as a single deletion at the time of the first deletion.

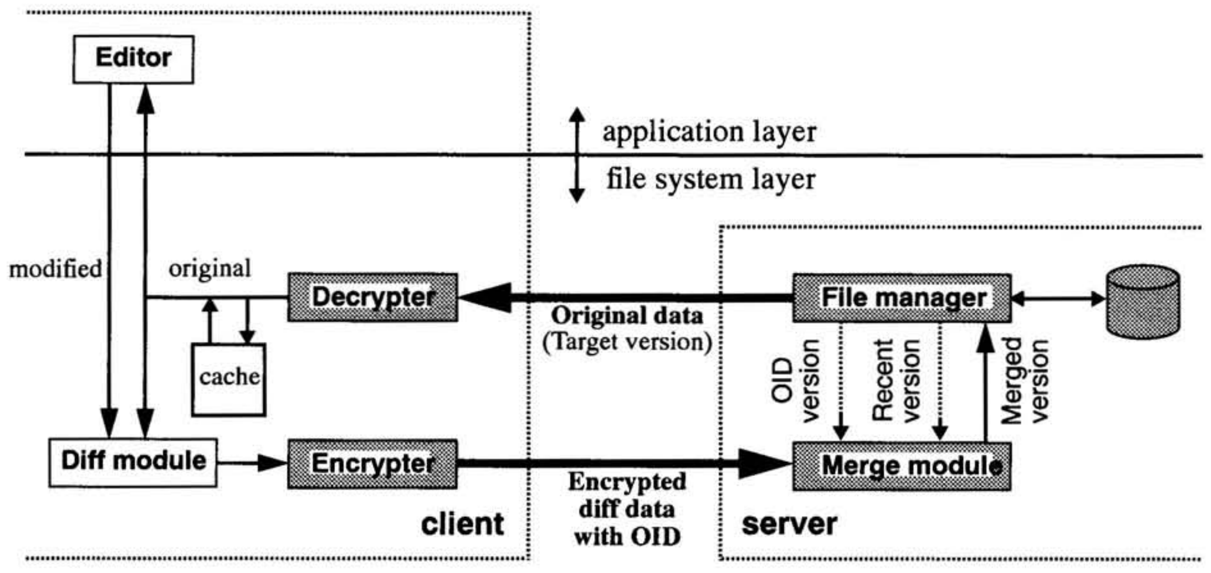


**Figure 3. Flow of the encrypted/decrypted data.**

## 4. Privacy Enhancement Mechanism

What we have described so far essentially duplicates the facilities in SCCS as regards a version control mechanism. We have hinted at cryptographic processing but have not yet explained how this works. This section describes the security architecture which involves user authentication, symmetric-key encryption, and a merging mechanism to allow multiple versions. Our architecture is interesting in that although the server maintains version information it can do so while retaining the encrypted form of the files. We call our mechanism "privacy enhanced merging".

### 4.1. System Data Flow

Figure 3 shows the system architecture, incorporating this privacy enhanced merging mechanism. The client encrypts and decrypts the file contents so that these contents are encrypted both while stored on the server and while in transit between the client and server. The shaded modules in figure 3 handle encrypted data. When the client has a cache data including an old file and its version ID, it is also possible to read only the differential part between the version of the cache which it has and the newest version the server contains.

In our system, the clients are oblivious of the merging mechanism - this is performed entirely by the server. Delays caused by mutual exclusion are limited to those delays required by the server, rather than an arbitrarily long delay at a client, as can happen in systems that employ locks to enforce exclusion. The server delay is essentially the time required to perform the merge operation. This scheme works well even if the network connections are slow, or intermittent - perhaps the user has a portable machine that is not always attached to the network.

The reader may like to refer back to figure 1 to see that while the contents of a file are hidden (encrypted) the file's structure is not. The contents are encrypted block by block while the traverse information and data characteristics are not. This allows the server to operate upon the file without requiring decryption of the components.

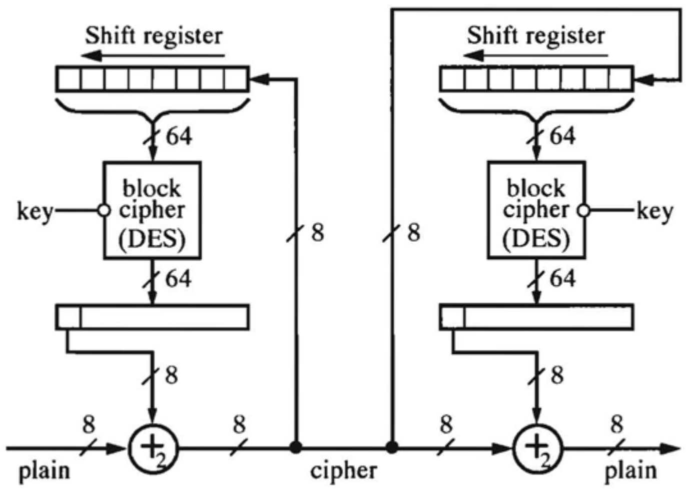


**Figure 4. 8-bit cipher feedback mode.**

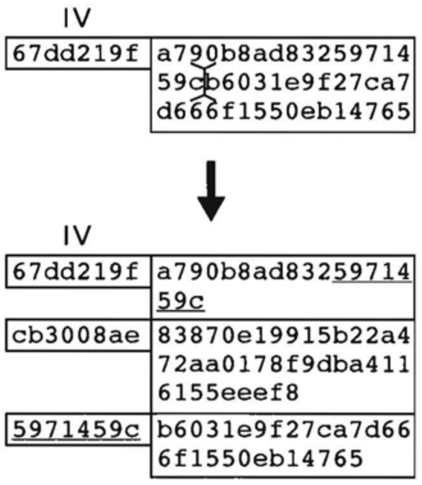


**Figure 5. Data block insertion.**

## 4.2. Encryption Algorithm

We should observe that since the server may have to split a block into two, the encryption algorithm chosen must be such that, if the ciphertext is split, then the client may recover the two halves of the plaintext. An alternative would be to ask a client to decrypt and split the block, but we decided that this approach would be unsuitable.

We have found that self-synchronous stream ciphers fulfill this requirement. In our implementation we have used an 8-bit Cipher FeedBack (CFB) mode of DES (The American Data Encryption Standard). The key, FK, we mentioned earlier is a DES key. This key is encrypted under the public keys of each of the authorized users of a file, using the RSA (Rivest, Shamir, and Adleman) public-key cryptosystem.

Figure 4 illustrates how the 8-bit CFB mode operates. The ciphertext character stream consists of adding, modulo-2, the plaintext character to the left octet character derived from the output of a block cipher(the DES algorithm in our system). And at the receiving end, the plaintext character can be restored adding the same data into the ciphertext character, when the same DES key and the same IV for each shift registers are available. In the proposed system, 8-bit is chosen as the character length on this mechanism for the convenience of dealing with byte-order insertion/deletion.

As described in the third section, data blocks are generally divided according to merge operations (insertion/deletion). Figure 5 illustrates an example of data block insertion. When a data block is split in two, it is necessary to generate a new IV for the second block. This is done by copying the last eight bytes from the first block; if there are less than eight bytes in the first block, then bytes from the first IV must be used. There is no need to change or perform re-encryption for the data contents itself. This move is understandable, considering the fact the input of each block cipher comes from the 64-bit shift registers which contain the most recent 64 bits transmitted as ciphertext.

When a user creates a shared file the client application chooses FK and IV at random. The user specifies which users may access the file, and an encrypted version of FK is generated for each user using their appropriate public keys. The user also specifies the "administrator" or administrators for the file - the users who are authorized to change various security- related properties of the file, such as the list of members. Administrators play the role traditionally associated with the "owner" of a file.

### 4.3. Message Authentication

The system that we have described requires an authentication mechanism: The file server is responsible for authenticating users; an integrity check allows tampering to be detected. An aim of our design is to minimize the overhead of these mechanisms. We employ digital signatures, a message integrity check, and a secure key distribution scheme to enforce security as follows.

1. When a user wishes to access a shared file, the client application computes a digital signature for the shared file ID, the user ID, and the time as the client believes it, using the user's private key. The client sends the initial command (open) with the signature. The RSA public-key cryptosystem is used to compute this signature.
2. After verifying a signature, the server generates a symmetric authentication key, AK, at random. This is encrypted with the user's public key and returned as a response.
3. The client decrypts this response in order to learn AK. Subsequent messages between the client and server are signed with a Message Authentication Code using AK as the signing key. On each communication, the sequence number is incorporated against a forged message as a parameter to the MAC function. The MAC is computed using the MD5 Message Digest function.
4. When the server receives the terminate command (close), the AK is eliminated.

In the above protocol, the client does not strictly authenticate the validity of the server assuming that the server does not have its own public-key. Even if it was a forged server, the fact would be detected by the client at a following read request time, because the forged server could not send meaningful data contents in the encrypted state. Anyway, this problem might be alleviated if the server has its own public-key, of course, using well-known authentication mechanisms.

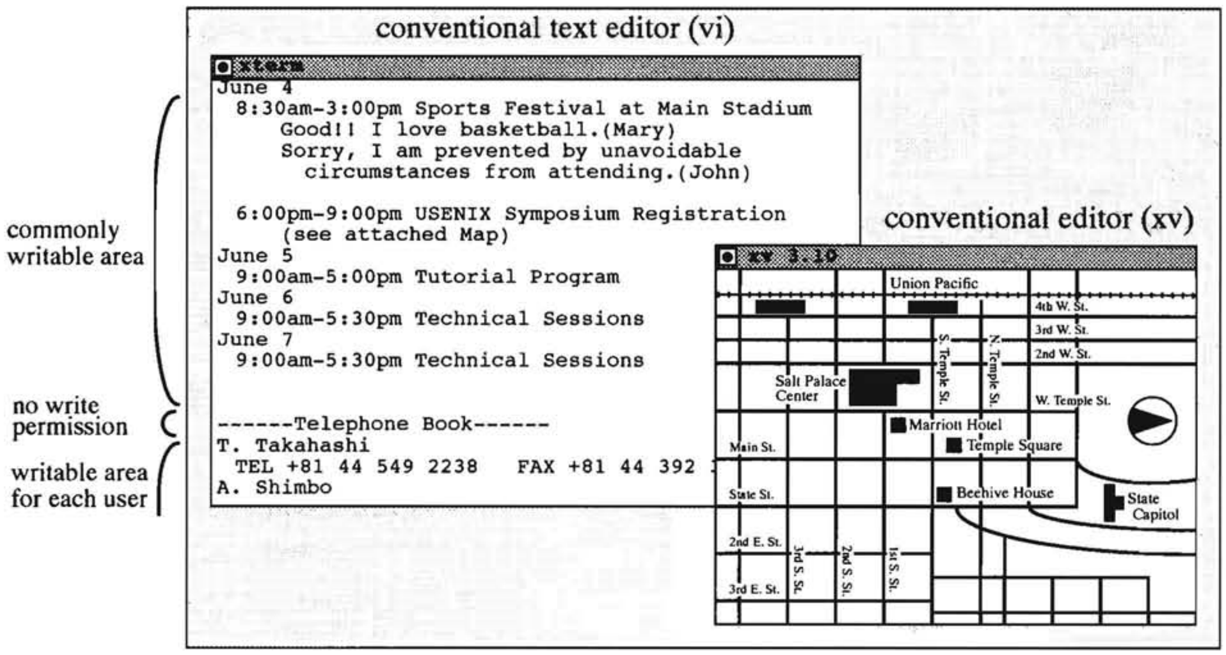
In the point of view of performance measurement, the initial connection (check-in) is the most dominant procedure in the above protocol; each RSA decipherment takes only 1.2 second under the Sun SPARC Station 2 (SS2) [7].

### 4.4. Discussion About the Server's Misbehavior

We should perhaps mention that there are several ways in which the server may misbehave. The server can easily destroy the contents of any shared file, but this is not surprising. The server cannot easily determine the plaintext of the contents of the files that it holds. Assuming that there is recognizable redundancy within the file it is difficult for the server to corrupt the file without detection. In particular it is difficult for the server to forge new blocks. However there are certain types of data corruption that the client should be aware of if it wants to detect this; for example re-ordering or deletion of blocks can result in a valid file. We assume that clients will take whatever measures they deem necessary to counter such misbehavior.

The server can still derive a certain amount of information about the contents of the file. For example, it can detect which blocks most frequently change, and measure the size of blocks. We deem this to be an acceptably small amount of leakage in our environment.

We shall now explain the benefits of our chosen policy.



```
June 4
  8:30am-3:00pm Sports Festival at Main Stadium
      Good!! I love basketball.(Mary)
      Sorry, I am prevented by unavoidable
        circumstances from attending.(John)

  6:00pm-9:00pm USENIX Symposium Registration
        (see attached Map)
June 5
  9:00am-5:00pm Tutorial Program
June 6
  9:00am-5:30pm Technical Sessions
June 7
  9:00am-5:30pm Technical Sessions


------Telephone Book------
T. Takahashi
  TEL +81 44 549 2238    FAX +81 44 392
A. Shimbo
```

**Figure 6. Initial view of NBBS client program.**

## 5. Network Bulletin Board System

We have explained the core parts of the mechanism proposed in this paper. We believe this system, which we call "Network BBS" (NBBS), to be a novel approach to secure file-system design. NBBS has a number of advantages over a conventional BBS on a PC network. NBBS aims to provide support for "groupware", allowing concurrent editing of files while providing certain guarantees about consistency. NBBS is currently implemented as a client-server application that uses a conventional UNIX filestore as the underlying storage mechanism. We plan to provide the file-system facilities that users have come to expect, similar to those already provided by NFS or AFS, however, these systems do not provide as flexible a concurrency mechanism nor the security facilities of our design.

There is a method to expand conventional file systems known as "stackable file system" [8], which we plan to take a similar approach to, although our plan is not yet concrete. Conventional applications continue to use conventional system calls(open/close/read/write) and these are detected by the novel file system libraries. The "open" system call behaves in making local cache copying from the server, the "read" or "write" system call behaves in reading from or writing to the local cache, and the "close" system call behaves in detecting modified parts and sending them to the server. From the point of view of constructing file systems, the NBBS client includes both a program on the application layer and that on the file system layer (Please consider figure 3 again).

Currently, to boot an NBBS client, a user might type

```
% bbs usenix.org:security95/schedule
```

The name "usenix.org", to the left of the colon, is the name of the NBBS server the client should connect to. The name "security95/schedule", to the right of the colon, specifies the name of a file known to that server. Filenames may be specified as either a relative or absolute path. The user is free to choose more convenient aliases for these values.

Finally, let us consider the system in actual use. This has been described in part in section three. Please consider figure 6.

The client joins together all of the textual blocks and presents these to a text editor. Each non-text item is sent to an alternative editor. We currently use the "xv" viewer since this can handle most of the sorts of file that we use, apart from continuous media (audio or video). NBBS has been in use within our office environment and has proven a useful tool to support communication within the office. We want to note that users can type "vi" as an aliased name of "bbs"; NBBS might be used as a secure concurrent version-controlled text editor.

## 6. Conclusion

This paper has introduced the idea of "privacy enhanced merging". We suggest that future file systems will have to provide facilities such as this. This project has suggested a number of topics for future study. In particular we are examining conflict recovery strategies and intend to carry out a performance evaluation. We intend to build a "Collaborative File System" which should be upward compatible with current file systems but will also provide a privacy enhanced mechanism and support asynchronous editing.

## Availability

We are planning to distribute the NBBS source code via FTP around of this summer. Details will be announced in appropriate newsgroups.

## Acknowledgments

We wish to thank Dr. Mark Lomas from The University of Cambridge for the comments about the security mechanism.

## References

[1] Michael Knister & Atul Prakash: Issues in the Design of a Toolkit for Supporting Multiple Group Editors, USENIX Computing systems, Vol. 6, No. 2, Spring 1993, pp. 135-166.

[2] Webb Miller & Eugene W. Myers: A File Comparison Program, Software Practice and Experience, Vol. 15, No. 11, pp. 1025-1040 (1985).

[3] Brian Berliner: CVS II: Parallelizing Software Development, USENIX Conference, Washington D.C., Winter 1990.

[4] J. Linn, et al.: Privacy Enhancement for Internet Electronic Mail, Network Working Group Request for Comments No. 1421-1424 (1993).

[5] Matt Blaze: A Cryptographic File System for Unix, ACM Conference on Communications and Computing Security, Fairfax, VA, November 3-5 (1993).

[6] T. Takahashi: Editing Piled-Style-Data by a Flat Editor, Proceedings of the 32nd Programming Symposium, IPS Japan, pp. 63-68 (1991) (in Japanese).

[7] M. Murota, et al.: Implementation and Fundamental Evaluation of Privacy Enhanced File Sharing System, Proceeding ot the 50th meeting, IPS Japan, vol.1, pp. 241-242 (1995) (in Japanese)

[8] Jeff Cook and Stephen D. Crocker: Truffles - Secure File Sharing With Minimal System Administrator Intervention, Trusted Information Systems, Glenwood, MD (1993).