ESPRIT Project No. 25 338

Work package C

Information Space

Software Report DC 4, 5.2, 6.3

ID:	InformationSpace DC4, 5.2, 6.3	Date:	17.8.98
Author(s):	Douglas Donaldson, Richard Hayton, APM	Status:	Unreviewed
Reviewer(s):		Distribution:	Project Confidential


FollowMe

Change History

Document Code	Change Description	Author	Date
InformationSpace	InformationSpace DC 1, 2, 3 and 6.1. No changes.	Donaldson, APM	31.3.98
InformationSpace	Update for DC4 and 6.3.	Donaldson, APM	17.8.98
InformationSpace	Update for DC5.2	Hayton, APM	15.10.98

1. Introduction

This document describes the release of the FollowMe WorkPackage C, InformationSpace (IS), of June 1998. This software was released to the FollowMe project together with a release of WorkPackage B, Mobile Object Workbench (MOW), as MOW2.0. The software and this report constitute deliverables DC4 (Object Sharer software, known as White Box Storables), DC5.2 (Software) and DC6.3 (Software Report).

This software description assumes familiarity with:

- The FlexiNet concepts which underpin the MOW and IS.
- The MOW and IS concepts of Cluster, Capsule and Nucleus.
- Deliverables DC1, 2, 3 and 6.1 which describe the requirements, motivation and design rationale, and the key interfaces. It also describes some early implementations of IS components.

1.1 Interfaces

The key external interfaces to the IS are

- **StoreFactory** (called StoreManager in DC6.1) offers newStore and redd methods for creating and destroying Stores. A store is a receptacle for storable objects, typically objects that are to be managed as a collection (for example objects belonging to one user).
- **Store** encapsulates a set of Storables. Storables are stored objects (or groups of objects). Each storable is associated with a directory entry, primarily for management purposes. A storable may be accessed via its directory entry, or more usually, via an interface reference returned upon its creation. The Store interface has only one significant method, ddtodotlelorl . All stores and directories are reachable from this.
- **Directory** offors methods for creating and destroying Storables. Its methods can be grouped into three groups:
  - 'Black box' storable methods (dfllto , ddcll , etc.), for copying objects by value into and out of the Directory. These methods are analogous to file operations.
  - 'White box' storable methods (newStoredl , ddcltoredl , etc.), for creating empty Storable Clusters. A 'white box' storable is *transparently* persistent. That is to say that the creator is passed a reference to (an interface on) an object within the storable, and may the access it as it it were a local java object. This reference may be passed to other clients, or even stored within other storables. The fact that the object is both remote and persistant is make transparent.
  - Directory methods (newlelorl , ddlelorl , ddlent , etc.), for creating and managing a hierarchy of Directories.
- **Storable** is the management interface of a Storable, and includes operations such as dll and ddtrol .

Figure 1 shows the relations between these classes. See the extensive JavaDoc in the code for a full description of these interfaces.

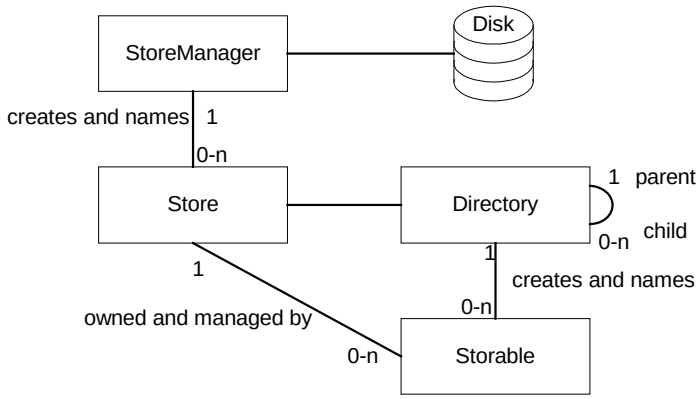

Figure 1. Class Diagram of Stores, Directories and Storable Objects.

2. Design

The design of the IS conforms to the Nucleus, Cluster and Capsule model of FlexiNet and the MOW.

There are three levels of grouping. Each StoreFactory manages a number of Stores. Each Store manages a number of Storables, and each Storable contains a number of separate objects. Working in reverse, the objects that constitute one Storable must be kept separate from those in other Storables, both for security reasons, and in order to identify a boundary for persistence. Storables therefore correspond to the FlexiNet (and ODP) notion of a cluster. Stores are consequently factories for Storables, i.e. Capsules in FlexiNet/ODP terms. The StoreFactory, as a Factory for Stores is an ODP Nucleus. In FlexiNet terms, this is just another capsule.

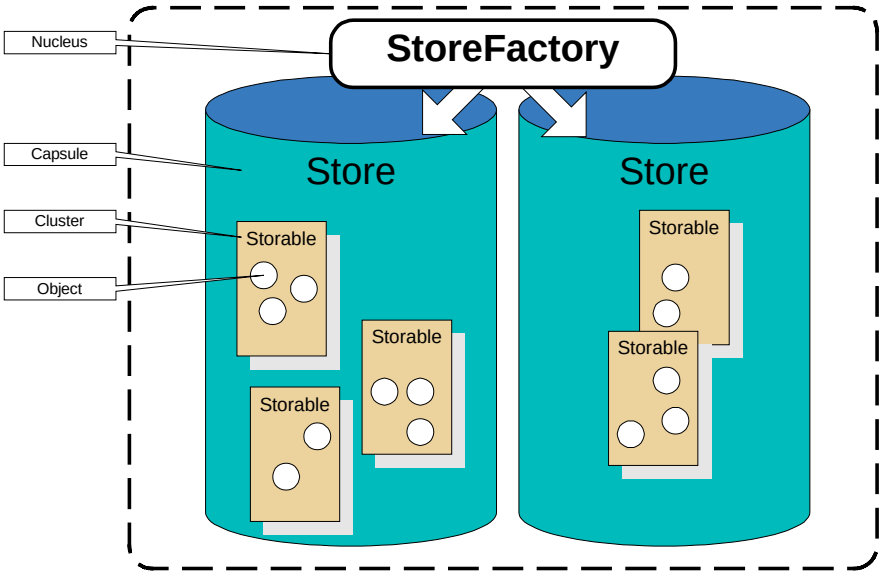

Figure 2 Stores and Storables

Each cluster is encapsulated so that it may be treated as a unit. In practical terms, this means that no objects are shared between clusters, and the Cluster abstraction therefore defines a boundary that may be used when serialising a Storable to disk.

At a computational level, clusters and capsules communicate with each other, and with remote clients via location transparent communication. In Engineering terms, this is achieved by each Cluster and Capsule having a separate 'ClusterComms' communications system managing its personal set of imported and exported references. It would be inefficient for each ClusterComms to be implemented entirely separately, so to allow sharing, all the ClusterComms within a capsule are multiplexed over a CapsuleComms, which manages low level communication, such as access to the network, and multiplexing. In an insecure implementation, different capsules may also share the same CapsuleComms, however in a secure implementation, where each store is 'owned' by a different client, it is more robust, and straightforward for

each capsule to have a separate CapsuleComms. Capsules are therefore entirely separate from each other (and may even reside in different processes or on different physical machines). In addition to the ClusterComms and CapsuleComms, the other related Engineering component is the ClusterManager. This is a management object with each Cluster, which provides two management interfaces, a public interface for initialising a Cluster and a private interface for use by the Capsule. In the case of a Storable, these interfaces are Storable and StorableManager respectively. Figure 3 gives a more detailed view of the internal relationship between a store and its Storables.

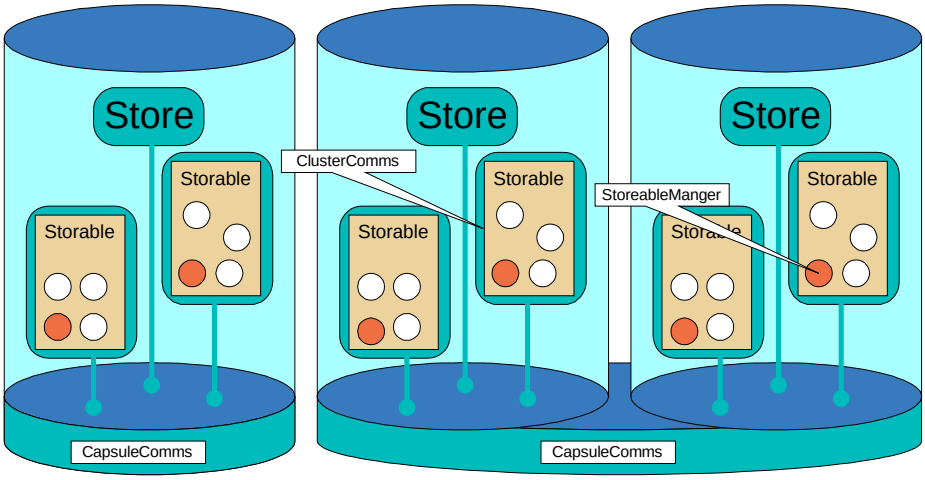

Figure 3. Implementation of the Store and Storables.

2.1 Underlying Storage

The underlying storage is supplied by implementations of DataDirectory. The DataDirectory interface abstracts the provision of storage for byte arrays. DataDirectory and the implementation above a file system, FSDirectory, are described in DC6.1.

3. User Authentication

3.1 Introduction

If the information space is used to store Storables containing private or personal information, then access to it must be controlled. Without suitable mechanisms, a third party might misrepresent themselves as a particular user in order to gain access to private information, or may eavesdrop on the communication between a Store and a client and thus gain illicit access.

The original specification of the Information Space given in the Technical Annex was of a *personal* information space, used to store personal information. Although the design of the Information Space has widened in scope, to include information that is not user-specific, this remains the primary area of concern for IS security.

3.2 Approach

As has been hinted in the previous section, the Store concept is a useful grouping abstraction for Storables that are to be managed as a unit. We envisage that each user will be allocated one or more Stores, and that user authentication is therefore best tackled on a per-store basis. This has a number of advantages

- The model is straightforward for both users and administrators
- The overhead of creating individual Storables is kept small, as authentication is tackled at a courser granularity
- Storables for different users can be managed by the same process
- It fits the expected use of the IS by other workpackages – only a user's Personal Assistant will access Storables within their information space

3.3 SSL

The mechanism used for authentication is based on the SSL protocol (secure socket layer). As each Store has a separate communications stack, each may have a different access policy controlling the credentials a client needs in order to connect. For *personal* information, we this policy will allow only client who have been granted a certificate relating to a particular user to connect. In engineering terms, the 'Crimson' FlexiNet binder provides an SSL-based communications stack, and this is used in both the client and the server to effect an authenticated communication channel between client and server. This is illustrated in Figure 4. Once the storable has been configured to use a particular access policy, and the client has been given access to an appropriate Authentication Certificate, the authentication (and optionally encryption) process is entirely transparent to both client and storable.

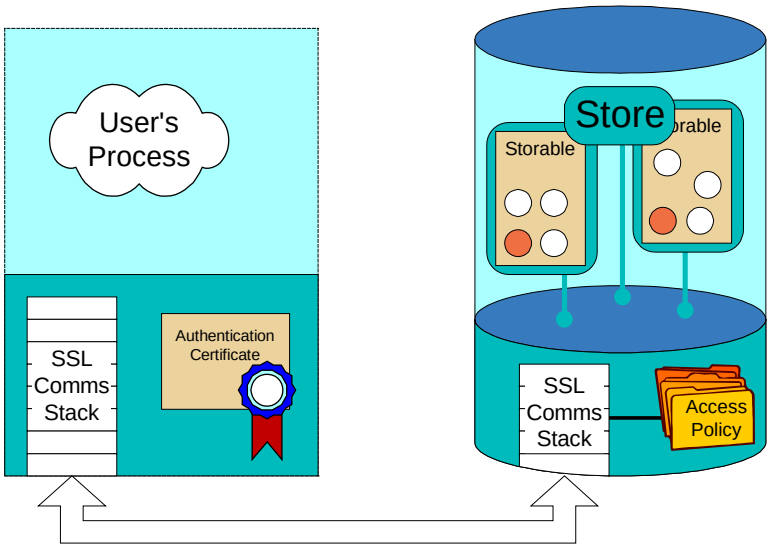

Figure 4 Authenticated Communication

3.4 User Administration

User administration is outside the scope of the Information Space. However, a stand-alone program is provided to act as a certificate authority and generate User authentication certificates. These must be securely transferred to client processes, as it is the clients' possession of these certificates that it actually authenticated. In order to allow more complex security policies, such as pre-storable access control, each certificates contains an additional ‘user id’. The user id related to a particular call may be examined from within the ClusterComms, which may then make block or allow the access based on its own policy.

3.5 Three Tier Systems

It is likely that a single process may wish to act with the authority of a number of different users. For example, users may login to a service that then acts on their behalf to access stores and other FollowMe components. This ‘middle-tier’ service will need to act as a number of different users, and keep certificates (and information) related to these users separate. Within FlexiNet, clusters provide the ideal abstraction for achieving this, as each may be given its own communications infrastructure, and SSL certificates. In addition, the encapsulation provided by the cluster will ensure that an authenticated communications channel to a service for use by user A is not accidentally passed to the code managing user B.

4. Components

4.1 StoreFactoryImpl

A StoreFactoryImpl keeps information about the StoreImpls it has created. The information needed to recreate the Store after a crash is held in a MetaStoreFactory object, saved as a black box storable in the StoreFactory's Directory. This Directory is independent from the Stores' Directory hierarchies. The MetaStoreFactory object is saved under the name *storename*.msf. The information needed for efficient lookup of Stores created is held in a Hashtable of StoreFactoryEntries, hashed by Store name.

The principle subtlety in the implementation of StoreFactoryImpl is the need to create new Stores in their own Capsule, preventing sharing of any Java objects between the StoreFactory Capsule and the Store Capsules.

The implementation of newStore checks for the existence of a *storename* Directory and *storename*.msf object. If they exist, the Store is recreated based on the previously saved parameters. (This is used for restart after failure.) If the store really doesn't exist already, then an appropriate root directory is created and [illegible] is called to initialize a new capsule for the store. [illegible] 'wraps' the proto-capsule to isolate it from the rest of the system, and to turn it into a true capsule, thereby preventing the accidental sharing of objects between capsules. The wrap operation returns an interface to the capsule of type StoreManager. Once wrapped, [illegible] may be called to initialise the capsule.

Note that a similar wrapping process in undergone to wrap the StoreFactoryImpl itself, however this is slightly simplified as the StoreFactoryImpl is the only capsule (at that time) so does not need to be encapsulated from anything else.

4.2 StoreImpl

StoreImpl is the Store capsule implementation. It performs several roles, which are separately defined in the interfaces it implements.

- As a CapsuleManager, its behaviour is inherited from CapsuleManagerImp.
- As a StoreManager it allows StoreFactory to initialise and destroy it.
- As a Store, external Clients can gain access to its Directory hierarchy.
- As a PartNameHandler, it restores Storables that have not yet been loaded from disc.
- As a PartNameHandler, it restores Directories that have not yet been loaded from disc.
- As an XStore, it offers back door services to its Directories.

Similarly to a StoreFactoryImpl, it keeps information about the Storables and Directories within it. This is held in a MetaStore object, saved as a black box storable in the Store's meta Directory. This is a Directory just for this purpose,

which is independent from the Store's Directory hierarchy. The MetaStoreFactory object is saved under the name *storename*.mst. (The StoreFactoryImpl actually initialises the StoreImpl with a meta Directory that shares the same file system directory as itself, so the StoreFactory's *storename*.msf meta information and the Store's *storename*.mst meta information appear in the same file system directory.)

The MetaStore object holds two mappings and their inverses, one from absolute Directory name to Directory interface Id, and one from absolute Storable names to Cluster Address. (See the interface definition of Directory for a definition of absolute names in a Directory hierarchy). These mappings allow Storables and Directories to be recreated at their old addresses after a crash, whether they are looked up by name in a Directory, or referenced by a location and persistent transparent object reference.

The MetaStore object is saved whenever it changes, that is whenever a white box Storable or Directory is created or destroyed.

Similarly to a StoreFactoryImpl, the principle subtlety in the implementation of Store is the need to create new Storables in their own Cluster. The StoreImpl asks its Capsule communications to 'wrap' a Storable with its own Cluster communications, getting back an encapsulated StorableManager interface. This interface is used for most communications with the Storable. However, the StoreImpl needs to use the unwrapped StorableManager interface to bypass encapsulation when a Storable is being destroyed. Similarly, the StorableManager uses an unwrapped XDirectory interface to its Directory for efficient access to the storage.

4.3 DirectoryImpl

DirectoryImpl relies on a DataDirectory object for creating sub Directories and storing Storables. It also keeps a Hashtable of its black box Storable, white box Storable and sub Directory members, indexed by name.

It implements `writeTo` of black box storables by serialising the object onto a buffer, then using the DataDirectory to store the buffer's byte array. Note that to obtain a deserialiser, the Directory needs to be given a reference to its Cluster communications. This means that the Store should not be initialised to construct its root Directory until after it has been wrapped with Capsule and Cluster communications. This is the case, as described in section 4.1, StoreFactoryImpl, above.

It delegates implementation of `newStored` , `restoreStored` and `readStored` to its Store, through the Store's XStore interface. StoreImpl implements `newStored` by creating a new StorableManagerImpl, which is a manager for an empty Storable Cluster (there is initially no application object inside it).

After creating, restoring or removing a sub Directory, Directory informs the Store, again through the Store's XStore interface. This allows the StoreImpl to keep its MetaStore information up to date.

DirectoryImpl prepends a byte to each Storable in its DataDirectory, signifying whether the Storable was copied in or a white box object. This means that the Directory does not need to keep an extra meta information object in its DataDirectory to signify this. The Directory can efficiently read the first byte of each Storable after a crash and work out which are black box and which are white box objects.

Locking of Directory members is fairly complicated. Locking is needed in case two clients try and change a Directory simultaneously, or try to access a white box Storable simultaneously. The locking needs to be at a fine granularity so that independent members of a Directory can be accessed simultaneously.

A Directory creates a Lock for each of its members. The Lock for a white box Storable is shared with the Storable's StorableManager. This allows the StorableManager to lock the Storable from when method activity starts until after the changed Storable has been stored. The Storable cannot be deleted in this interval. The Lock for a sub Directory is shared with that Directory. This allows a parent to lock its child, remove all its members and then remove it.

When a Store is destroyed, or a Directory is recursively removed, the lock acquisition recurses down the Directory hierarchy. The lock for a deleted object can be killed (moved to a dead state). Any waiting activity then fails to acquire the lock, leading to an exception propagating back to the client.

4.4 StorableManagerImpl

StorableManagerImpl performs several roles, which are separately defined in the interfaces it implements.

- As a ClusterManager, its behaviour is inherited from ClusterManagerImp but changed to implement persistence transparency.
- As a StorableManager it allows Stores to initialise and destroy it.
- As a Storable (an extension of Cluster), external Clients can create Objects and copy them to new Directories.

The persistence transparency is implemented in e , called by the Cluster communications after a call to the Storable has finished. i waits for thread activity in the Cluster's ThreadGroup to finish, then stores the ClusterState. The ClusterState is obtained from t e a t ntent , and consists of the application object itself (possibly null), the table of exported interfaces, and the distinguished interface on the application object returned after the application object was initialised. This is precisely the information needed to restore the Storable after a crash. The StoreImpl keeps the Storable's cluster address in its MetaStore object, so the ClusterState table of exported interfaces only needs to map the interfaces to their Ids. The work of restoring all the interfaces at their old identifiers is done by t e a t ntent .

The ClusterState is serialised into a buffer using a ClusterByValue serializer obtained from the Cluster communications. This serialises the entire Cluster, not using the default FlexiNet semantics (objects serialised by value, interfaces serialised by reference).

St t differs from ClusterManagerImp's t t because it locks the Storable against other calls until after e has stored it. This allows the e to wait for thread activity to finish, meaning that the Storable is in a stable state to be stored.

The buffer used for the ClusterState is segmented, with a one-byte segment for the DirectoryImpl's flag, and the rest for the Storable. The StorableManagerImpl passes the buffer to the Directory using its XDirectory interface, and the Directory fills in the flag byte and saves the buffer's byte array to its DataDirectory.

5. Using the IS

Example programs for using the IS are supplied in the MOW and IS release within sub directories of Flex-iNet/TestCode/ISpace. The ReadMe.txt files in those directories describe the mechanics of compiling and running the examples. Two of the examples are described here. The examples assume a StoreFactory is running and has exported its Name for clients. In the absence of a Class Repository allowing the StoreFactory to dynamically load the Storables' classes, the classes of Storables must be on the StoreFactory JVM's Classpath.

5.1 Black

The Black Client

- Binds a variable called sman to a StoreFactory. This is done using a stringfied form of the StoreFactory's Name, obtained from the command line or from System Properties. It would be equally valid to use a trader such as TrivTrader.
- Creates a Store using store = sman.newStore("store").
- Gets the RootDirectory of the Store using root = store.getRootDirectory().
- Copies an object by value (a String with the value "data") into a Directory entry called "data" using root.copyInto("data", "data").
- Looks up the object using result = (String) lookupCopy("data").
- Verifies that the result is "data".
- Removes the Store using sman.remove("store").

The Black CopyClient creates two Stores and copies a black box Storable into the first Store, then copies the data from the first Store to the second using root.copy("data", root2, "data2"). It then modifies the copy in root2 to verify that the two copies are independent.

5.2 White

The White Client

- Binds sman, as for Black.
- Creates a Store using store = sman.newStore("store").
- Gets the RootDirectory of the Store using root = store.getRootDirectory().

- Creates an empty Storable in a Directory entry called “myAccount” using storable = root.newStorable("myAccount").
- Creates a white box Storable object of class AccountImpl using tagged = storable.createObject(AccountImpl.class, new Object[0]). The second parameter indicates that the AccountImpl’s init method expects no arguments.
- Retrieves the Account interface from the tagged using myAccount = (Account) tagged.iface().
- Performs credit, debit and balance invocations on the Account. The results transparently persist in the Store.
- Exits without cleaning up, to allow Client2 to bind to the Account independently.

The White Client2 demonstrates that the Account object persists even after the Store has been interrupted and restarted. Client2:

- Binds sman, as for White Client.
- Looks up the Store using store = sman.lookup("store").
- Gets the RootDirectory of the Store using root = store.getRootDirectory().
- Looks up the white box Storable using tagged = root.lookupStorable("myAccount").
- Retrieves the Account interface from the tagged using myAccount = (Account) tagged.iface().
- Performs credit, debit and balance invocations on the Account and verifies that the results are consistent with the expected persistent state. The results transparently persist in the Store.
- Exits.

The White CopyClient creates two Stores and a white box Storable, and copies the Storable from the first Store to the second using tagged2 = storable.copy(root2, "myAccount2"). It then modifies the Storables to verify that the two are independent.