# COMPARING THE XAM API WITH FILE SYSTEM PROGRAMMING

BY

STEPHEN J. TODD

B.S.C.S. , University of New Hampshire, 1987

THESIS

Submitted to the University of New Hampshire

in Partial Fulfillment of

the Requirements for the Degree of

Master of Science

in

Computer Science

December, 2006

This thesis has been examined and approved.

_____
Thesis Director, Dr. Philip Hatcher
Professor of Computer Science

_____
Dr. Robert Russell
Associate Professor of Computer Science

_____
Scott A. Valcourt
Research Project Manager

_____
Date

## DEDICATION

To my grandfather, William Todd

to my father, James Todd

to my mother, Mary Todd

and with great affection to my own wife and family,

Katy, Becky, Matthew, and Boomer Todd.

## ACKNOWLEDGEMENTS

I would like to thank my company, EMC Corporation (and before that, Data General), for funding my Master's Degree. Thanks also to my EMC co-workers: Zoran Cakeljic, Scott Ostapovicz, Tom Teugels, and Mike Kilian.

I would also like to thank my Thesis Advisor, Phil Hatcher, for being present for all twenty years of my quest to finish my Master's Degree!

## TABLE OF CONTENTS

## LIST OF TABLES

## LIST OF FIGURES

ABSTRACT

COMPARING THE XAM API WITH

FILE SYSTEM PROGRAMMING

by

STEPHEN J. TODD

University of New Hampshire, December 2006

XAM is an application programming interface (API) currently being standardized by the Storage Networking Industry Association (SNIA). XAM stands for “eXtensible Access Method”. The XAM API will allow application developers to store content on a new class of storage systems known as “fixed-content” storage systems. This new class of storage system is optimized for the storage and retrieval of unstructured data that changes infrequently.

While it is anticipated that SNIA will eventually ratify a XAM application programmer’s interface (API), it is important that the new API presents an interface that is useful yet offers the richness of the XAM protocol to the developer. An ideal goal for the XAM API would be to become as ubiquitous and familiar to developers as the block file system function calls such as fread(), fwrite(), remove(), etc.

This thesis will describe a XAM simulation framework that allows for the building of XAM applications without the need for actual XAM hardware. It will also compare the XAM API to the familiar block file system API. The XAM simulation can form the basis of a robust framework for continuing XAM research; the results of comparing XAM to file system applications will be fed back into the SNIA community.

# INTRODUCTION

The data storage industry has experienced tremendous growth in innovation in the past twenty years. Starting with the introduction of RAID technology in the late 1980s, storage technologies began to develop a complex and robust feature set, including intelligent caching technologies, snapshots, remote mirroring, fiber channel and storage area networks (SANs), internet SCSI (iSCSI), storage virtualization, and others. Storage has gone from being a peripheral to being a central consideration in any company's data center.

Innovation in the storage industry continues. In particular, storage technologies that deal with the handling of fixed content are on the rise. Fixed content has been defined by the Storage Networking Industry Association (SNIA) as inactive and unchanging content, as opposed to active and changing information (XAM FAQ, 2006). Consider a medical X-RAY image that is being placed on a storage system. Is it likely that this medical image will be subsequently opened, edited, and written back out again? The answer is no. An X-RAY, therefore, can be considered fixed content. Other examples include scanned check images or email. The storage industry has begun building innovative storage systems designed specifically for fixed content.

SNIA has recognized the importance of fixed content storage systems and is attempting to ratify a new application programmer's interface (API) for fixed content called XAM – the eXtensible Access Method. Developers that code to the XAM API

can store their fixed content on storage systems that are designed with fixed content in mind.

The adoption of XAM by application developers is the focus of this research. How does the XAM API look? How can the API be evaluated? How does the API compare with the block file system API? The starting point for the research begins with the evolution of the XAM specification and its introduction into the SNIA community.

# CHAPTER 1

## THE CREATION OF XAM

In 2002, EMC Corporation began shipping a storage system called Centera. Centera is designed specifically for fixed content. The data written to Centera is fixed in nature (not likely to be modified), and the following features are present in the storage system:

- Content Addresses – when content is stored to Centera it is assigned a unique handle based on a hash of the content. This content address provides location independence (the application has no control over or knowledge about the physical placement of the content), and content authentication (the hash value guarantees that the fixed content has not been tampered with or corrupted).
- Associated Metadata – When storing fixed content, Centera allows for additional metadata to be stored along with the content. Metadata can be both application-generated and Centera-specific.
- Retention – When storing fixed content, Centera also allows for an application to specify a length of time for which the content must be retained (i.e. delete is disallowed until the retention expires).

In order to take advantage of these innovative features, a new and proprietary application programmer's interface (API) was created known as the Centera API (EMC Centera, 2006). The Centera API allows an application to store content, receive a content address in return, and associate application-specific metadata with content. The API can also associate retention values with specific content.

As the storage industry recognizes the importance of fixed content storage systems, several vendors (including EMC) have united to create an industry standard API known as XAM (eXtensible Access Method). Beginning in 2004, EMC and IBM began working on a specification. In early 2005, several other vendors joined the effort, including Hewlett-Packard, Hitachi, and Sun. Subsequently a joint proposal was made (XAM Version 1.1) to certain independent software vendors, and finally in late 2005, XAM Version 1.2 was proposed to SNIA under SNIA's intellectual property rules. The Fixed Content Aware Storage Technical Working Group (FCASTWG) was formed with the goal of creating and ratifying a formal XAM interface.

The input from the different vendors has resulted in a description of a system which is similar to Centera. However, there are significant differences. Chief among those differences is the fact that content need not be strictly fixed content. Flexibility has been built into XAM for participating and non-participating content. Participating content is hashed, and the resulting hash value participates in the construction of the object identifier. Non-participating content does not contribute in any way to the object identifier. This feature of XAM allows application developers more flexibility in keeping the same identifier for an object if they so wish.

# CHAPTER 2

# THE XAM API

The XAM API will ultimately be implemented as a set of header files and a shared library that will be linked in by user-space applications. XAM presents several new concepts with which an application developer must become familiar, including SYSTEMS, XSETS, properties and streams (also referred to as “blobs”), and XUIDs (also referred to as XSET names). The material in this section is a short summary of the version 1.2 specification currently being analyzed by SNIA (XAM Specification, 2006).

## SYSTEMS

A SYSTEM in XAM represents a fixed content storage subsystem, and is a repository for fixed content. The XAM API will provide a connection string which will allow XAM applications to connect to one or more SYSTEMs. A SYSTEM has properties and capabilities that can be queried and/or set via the XAM API. An example of a property would be an ASCII name. An example of a capability would be retention, that is, the ability of the SYSTEM to disallow the deletion of certain fixed content items.

An example of a SYSTEM would be the Centera system described in the previous chapter.

It is not necessary to think of a SYSTEM as one piece of hardware. XAM provides for multiple pieces of hardware to work together to form one SYSTEM. In this sense a SYSTEM is a logical construct and can also be defined as a repository for XSETs, which are described below.

## XSETS

An XSET is the main programming construct in XAM. It represents content that is written to a SYSTEM. The XSET can be made up of zero or more streams, which roughly translate into files. (Note that in the 1.2 version of the XAM Specification, streams are referred to as blobs). The XSET is also made up of zero or more properties. Properties are name/value pairs that can be used to describe content (e.g. metadata:data about the data) or can be used to describe policies about the XSET (e.g. FixedRetentionPeriod="1 year", meaning disallow deletes on this XSET for one year). Every XSET has default properties, such as creation time.

XSETs also provide for describing fixed and non-fixed streams or properties within an XSET. These attributes are referred to as Participating or Non-participating in the XAM specification. Participating fields contribute to the naming of the XSET (see XUID). If an XSET is retrieved from a SYSTEM and a

participating field is modified, the XSET will receive a different name (XUID) when it is re-written back to the SYSTEM.

## XUIDS

A XUID is the name of an XSET (the version 1.2 specification exclusively uses “XSET name” and not XUID, which has been adopted post version 1.2). Unlike filenames, a XUID is generated by the SYSTEM, not by the application. Also, a XUID does not present any location information to an application in the same way that an absolute pathname does. Within XAM, all XSETs (and the XUIDs that reference them) live in a flat namespace within the SYSTEM. When an application wishes to access an XSET, the XUID that maps to the XSET must be presented to the SYSTEM.

The creation of XUIDs in XAM has a specific set of rules, including the participating versus non-participating rule mentioned previously. If any property or stream is stored to an XSET as participating, the XUID created must be formed by using the content of the property or stream (e.g. run an MD5 hash over the value). If that property or stream is subsequently modified, the XUID must change. If a new participating property or stream is added to an existing XSET, the XUID must also change.

In version 1.2 of the XAM specification, there is not a formal XAM naming proposal. The XUID will likely consist of a naming version, vendor information,

an XSET specific identifier, and potential vendor-specific extended information (see section 3.3 of the XAM specification).

## XAM FUNCTION CALLS

Before describing the individual XAM routines, the relationships between the SDK, SYSTEMs, XSETs, and streams merit description.

The following is the general hierarchy of XAM function calls:

- A complete sequence of SYSTEM, XSET, or stream calls must begin with a call to SDK_Initialize() and end with a call to SDK_Shutdown();
- A complete sequence of XSET or stream calls must begin with a call to SDK_OpenSystem() and end with a call to SYSTEM_Close();
- A complete sequence of stream calls must begin with a call to either SYSTEM_OpenXSET() or XSET_Create(), and end with a call to XSET_Store(), followed by a call to XSET_Close().

XSET manipulation is the heart of XAM. XSETs are initially created within the context of a SYSTEM via the XSET_Create() call. Streams are created (Stream_Create()), filled (Stream_Write()), and closed (Stream_Close() in the context of the XSET handle. Name/value pairs are added to the XSET within the context of this handle.

It is critical to note that none of the streams or name/value pairs are officially persisted until the XSET_Store() routine is called and a XUID is created. Equally important is that the XSET_Close() routine should never be called before the XSET_Store() operation because streams and properties will be lost if this happens.

Note that all XAM routines are synchronous, with the exception of the XSET_Query() routine. The reason for this routine's asynchronous behavior is that the XSET_Query() routine may take a long time to run due to the potential for the routine to iterate over millions of XSETs within the XAM SYSTEM.

The full set of XAM function calls are described below.

**SDK_Initialize()**

This function is called by an application in order to initialize the XAM API. The rationale for having this routine includes auto-loading of vendor-specific shared libraries (see architecture section).

**SDK_Get()**

This function is called in order to fetch the current parameters and settings from the SDK such as the version of the SDK, a description of the XAM SYSTEMs that the SDK is currently managing, and the format of the connection string that will connect the application to a specific XAM SYSTEM.

**SDK_Set()**

This function is called in order to set name/value pairs in the SDK in whatever future way an application developer might wish to use.

**SDK_Shutdown()**

This function shuts down the XAM API and causes the SDK to release any resources that it has allocated. After calling SDK_Shutdown(), no other XAM function calls can be made until SDK_Initialize() is called once again.

**SDK_OpenSYSTEM()**

This function is called after SDK_Initialize(). The function accepts a connection string identifying the SYSTEM with which the application wishes to communicate. The connection string format is returned in the SDK_Get() function call. This SYSTEM should be capable of storing and retrieving XSETs.

**SYSTEM_Get()**

This function fetches properties and policies from the SYSTEM, such as whether or not the system supports specific features like retention.

**SYSTEM_Close()**

This function closes an open SYSTEM. No operations can be performed on the SYSTEM unless a subsequent call to SDK_OpenSystem() occurs.

**SYSTEM_XSETExists()**

This function accepts a XUID as an argument and returns whether or not the XSET exists on the SYSTEM.

**SYSTEM_DeleteXSET()**

This function removes an XSET from a SYSTEM.

**SYSTEM_ImportXSET()**

This function imports an existing XSET (which was previously exported from another SYSTEM) and stores the XSET into the given SYSTEM.

**SYSTEM_ExportXSET()**

This function fetches an XSET from a SYSTEM for the purpose of importing it into another SYSTEM.

**SYSTEM_OpenXSET()**

This function accepts a XUID as an argument and opens an XSET for subsequent processing by the application.

**XSET_Create()**

This function is called when the application wishes to create a new XSET that it will ultimately store on a SYSTEM.

**XSET_Get()**

This function allows the application to retrieve XSET name/value pairs, such as default XSET parameters (i.e. creation time) as well as user-defined name/value pairs.

**XSET_Set()**

This function allows the application to set XSET name/value pairs. These name/value pairs can be application-defined (i.e. color="red") or XAM-defined (FixedRetentionPolicy="3 years").

**XSET_Store()**

This function commits the XSET to the SYSTEM. Upon a successful completion of this API, call a XUID will be returned to the application. The application can then use this XUID to refer to this specific XSET.

**XSET_Close()**

This function ends all processing on the currently open XSET. Any subsequent processing on this XSET must be accomplished via an open call.

**XSET_DeleteStream()**

This function removes a stream (blob) from an XSET.

**XSET_Query()**

This function specifies query or search criteria. The SYSTEM, upon receiving the query, returns a list of XUIDs representing XSETs that satisfy the search criteria. The list of XUIDs is returned as a stream within the XSET; aborting the query operation is accomplished by either closing the XSET or telling the stream to abort the query. This call is asynchronous and is evaluated in a subsequent section.

**XSET_OpenStream()**

This function opens a specific stream within an XSET. The stream is identified via a name that is passed as an argument to this routine.

**Stream_Create()**

This function is called in the context of an XSET, and it prepares the XSET to associate a subsequent stream of data (a series of bytes) with a name that the application assigns to the stream.

**Stream_Read()**

This function accepts a buffer as an argument and XAM will place contiguous bytes of content from the particular XSET stream into the buffer.

**Stream_Write()**

This function passes a buffer to XAM in the context of an XSET stream, and XAM temporarily stores the data until a subsequent XSET_Store() operation.

**Stream_StopQuery()**

This function halts a query which was previously submitted via the XSET_Query() routine. The stream contains a list of XUIDs that satisfied the XSET_Query(); but by calling this routine the stream is destroyed.

**Stream_Close()**

This function closes a previously opened stream.

## XSET System PROPERTIES

The XSET_Get() routine allows an application to fetch predefined XSET name/value pairs (called properties). These names and their meanings are described below.

**system.xset.class**

For version 1.2, this field is always set to “Default”. In the future, it is anticipated that standard classes will be defined which require strict metadata formats, and the class will describe the metadata type, such as “email”, or “legal”.

**system.xset.version**

For version 1.2 of the standard this field is always set to “1.0”.

**system.xset.parent**

XSET name (XUID) of a parent from which this XSET was derived (if any). This field is non-NULL if a previously existing XSET is modified and results in a new XUID being generated.

**system.xset.name**

This field represents the name of this XSET (the XUID).

**system.creator.vendor**

This field represents the name of the storage vendor providing the storage for this SYSTEM (e.g. IBM).

**system.creator.version**

This field represents the version of software running on the SYSTEM that initially stored this XSET.

**system.creator.user**

This routine stores the name of the user/application that originally created this XSET.

**system.ctime**

This field represents the time of initial XSET creation.

**system.mtime**

This field represents the last time the XSET was modified (i.e. XSET_Store() was called).

**system.stime**

This field represents the time the XSET was initially stored (or imported) into the current SYSTEM.

**system.etime**

This field represents the time of a retention event. Retention events, as defined in version 1.2, determine whether or not an XSET can be deleted.

## XSET POLICIES

The XSET_Set() routine allows an application to set predefined XSET name/value pairs that map to policies supported by the underlying SYSTEM. These names and their meanings are described below.

**FixedRetentionPeriod**

This XSET parameter is set if the application wishes to retain a given XSET for a particular amount of time. The SYSTEM must prevent deletion if the FixedRetentionPeriod has not expired.

**FixedRetentionClass**

This parameter is similar to FixedRetentionPeriod except a retention class is actually a string that the SYSTEM maps to a numeric retention period. It allows an administrator to modify a retention period without having to modify every individual XSET that holds said value.

**VariableRetentionPeriod**

This parameter is similar to FixedRetentionPeriod except the retention period clock begins ticking from the time when a retention event occurs. Refer to the XAM Specification version 1.2 for a more detailed description or retention events.

**EBRStateEnabled**

When set to TRUE, this parameter means that the XSET is a candidate for Event Based Retention. Retention is only measured after the occurrence of a given event.

**EBRState**

When set to TRUE, this parameter means that an event has occurred and the current time should be recorded in the XSET (as a name/value pair tagged specifically EBRState), and all retention activities should be based on this time.

**RetentionHold**

When set to TRUE, this parameter means that the XSET should be permanently retained.

**Delete**

When set to TRUE, this parameter means that the XSET should be deleted after its retention period has expired.

**Shredding**

When set to TRUE, this parameter means that the XSET should be electronically shredded after it has been deleted. This assumes that any and all copies of the XSET are overwritten on the original medium where the XSET was stored.

# CHAPTER 3

## REQUIREMENTS FOR A XAM SIMULATION

In order to evaluate XAM application code, a XAM simulation environment must first be created. There are currently no XAM systems or software available to help with an evaluation effort. Indeed, there won't be any XAM systems or software available until the XAM specification has at least been ratified.

A full-scale XAM simulation is certainly valuable beyond this thesis. It would allow many developers to try their hand at writing XAM applications without the need for XAM hardware. In addition, if a full-scale XAM simulation was endorsed by SNIA as a functionally correct implementation of XAM, it would allow application developers to write their software and be assured that the code that runs against the XAM simulator would most likely run against an actual XAM system (so long as the XAM system was compliant). Such a XAM simulator should run as part of the XAM architecture which is listed in Figure 1 (XAM Specification, 2006).

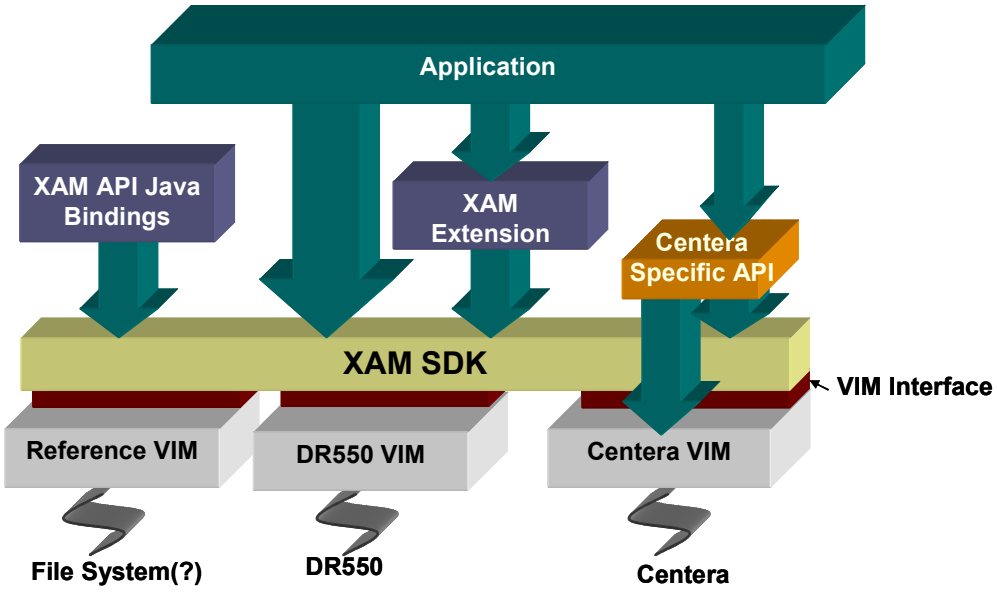


**Figure 1 – XAM Architecture (from XAM Specification)**

The XAM architecture shown in Figure 1 has multiple levels. At the bottom level are two XAM-compliant hardware systems. EMC's Centera product and IBM's DR550 system are shown as examples of systems that could support the XAM API. One level up from the hardware systems are software modules known as VIMs, or Vendor Interface Modules. These modules have a standard interface that converts XAM requests into native requests supported by the underlying hardware systems. For example, a XAM API call that is routed to the Centera VIM will be converted to the Centera protocol and sent over a TCP/IP network to the Centera. The IBM DR550 VIM will behave in a similar manner.

The level above the VIMs is known as the XAM SDK. The SDK is envisioned to be a user-space library that can dynamically load any VIM that is present in the system. The SDK has a set of header files which declare the set of XAM function calls described previously. The implementation of these function calls lives within the SDK library; most of the calls will undoubtedly be mapped to a VIM.

At the very top level of the XAM architecture live the applications that will ultimately use XAM. The goal of this thesis is to write XAM applications that function in the context of this architecture. Note that SNIA is also discussing other pieces of the architecture, such as extensions to XAM, vendor-unique extensions to XAM, and different language bindings such as Java-to-C (XAM Specification, 2006).

Finally, the architecture shows a reference VIM that does not interface to any remote hardware, but rather interfaces to a local store such as a file system. The purpose of the reference VIM is to fully support all XAM API calls such that a XAM application developer could write XAM test code that uses the local store to simulate a SYSTEM that is capable of XSET operations.

This thesis implements a subset of a reference VIM. A reference VIM enables the writing of XAM applications which in turn can be compared to traditional file system applications. It is not necessary to simulate the entire XAM API, but it is more important to simulate those XAM function calls that have to deal with the creation, reading, writing, and deletion of XSETs. Other XAM operations, such as the setting and getting of SYSTEM properties, are less important and need not be implemented for this thesis.

Table 1 lists the entire set of XAM API calls, the section of the XAM specification where the function can be found, and whether or not the routine is supported as part of this thesis. The table also describes the required behavior of the reference VIM for supported routines.

**Table 1 - Supported XAM API Routines**

| XAM Routine | Spec | Supported? | Implementation Strategy |
|---|---|---|---|
| **SDK_Initialize()** | **6.4.2** | **YES** | The reference VIM must read a configuration file describing the SYSTEM name, directory location, and authorization privileges (i.e. user=”steve”). |
| **SDK_Get()** | **6.4.3** | **NO** | |
| **SDK_Set()** | **6.4.4** | **NO** | |
| **SDK_Shutdown()** | **6.4.5** | **YES** | The reference VIM must cleanly shut down. |
| **SDK_OpenSYSTEM()** | **6.4.6** | **YES** | The reference VIM must verify the existence of the SYSTEM, and verify the existence of a proper directory to store/retrieve XSETs. |
| **SYSTEM_Get()** | **6.4.7** | **NO** | |
| **SYSTEM_Close()** | **6.4.8** | **YES** | The reference VIM must close an open SYSTEM. |
| **SYSTEM_XSETExists()** | **6.4.9** | **NO** | |
| **SYSTEM_DeleteXSET()** | **6.4.10** | **YES** | The reference VIM must allow a XAM application to delete an XSET for a given XUID. |
| **SYSTEM_ImportXSET()** | **6.4.11** | **NO** | |
| **SYSTEM_ExportXSET()** | **6.4.12** | **NO** | |
| **SYSTEM_OpenXSET()** | **6.4.13** | **YES** | The reference VIM must accept a XUID and return a handle to the corresponding XSET. |
| **XSET_Create()** | **6.4.14** | **YES** | The reference VIM must create a data structure that tracks a new XSET. |
| **XSET_Get()** | **6.4.15** | **YES** | The reference VIM must support fetching policies and properties from an XSET. |
| **XSET_Set()** | **6.4.16** | **NO** | The reference VIM must support the setting of policies and properties to an XSET . |
| **XSET_Store()** | **6.4.17** | **YES** | The reference VIM must persistently store the XSET (and stream) that was recently created and/or modified, and return a XUID to the application |
| **XSET_Close()** | **6.4.18** | **YES** | The reference VIM must close any resources associated with the XSET. |

**Table 1 Continued**

| | | | |
|---|---|---|---|
| **XSET_DeleteStream()** | **6.4.19** | **NO** | |
| **XSET_Query()** | **6.4.20** | **NO** | |
| **XSET_OpenStream()** | **6.4.21** | **YES** | The reference VIM must return a stream (file) handle for a given XSET and stream name. |
| **Stream_Create()** | **6.4.22** | **YES** | The reference VIM must create a byte stream for the XSET and give it a name. |
| **Stream_Read()** | **6.4.23** | **YES** | The reference VIM must allow a XAM application to read from a specific XSET stream. |
| **Stream_Write()** | **6.4.24** | **YES** | The reference VIM must allow a XAM application to write bytes to a given XSET stream. |
| **Stream_StopQuery()** | **6.4.25** | **NO** | |
| **Stream_Close()** | **6.4.26** | **YES** | The reference VIM must free the byte stream created by either the create or open method. |

Not all details of the Version 1.2 specification are covered in the above table. Additionally, Version 1.2 of the XAM spec is not complete in its treatment of certain topics. Therefore, the prototype reference VIM implemented for this thesis will have certain limitations.

**VIM Auto-Load – Section 5.1**

There is no description of the specific algorithm that XAM will use to dynamically locate and auto-load VIMs. Therefore, this prototype will simply assume that the reference VIM is available and will always load it.

**Storage Pools – Section 4.3.1**

There is no description of exactly how an application XSET gets stored into a XAM storage pool. There is no description on how to limit applications to write/read to and from their own storage pool. Therefore, the reference VIM will present one flat, globally accessible XSET name space for this thesis.

**XAM Naming Model – Section 3.3**

The version 1.2 specification, while describing a general structure of XSET names (XUIDs), does not provide specific details on the structure of XSET names. Therefore, the prototype reference VIM will create random XUIDs based on timestamps and sequence numbers. Algorithms for modifying XUIDs based on participating and non-participating properties and streams are also not fully described. The prototype reference VIM will not support participating streams because they are not necessary for the comparison to the file system API.

**Multiple Application Threads**

The application threading model for XAM is not described. Given that threads are not necessary to compare XAM to a file system API, support for multiple threads will not exist in the prototype VIM.

## Error Handling

Version 1.2 of the XAM specification defers description of specific error values and error-handling behavior (until implementation). Therefore the prototype reference VIM will simply return two error codes (success or failure) and not attempt to categorize specific types of XAM errors.

## Credential String

The specification describes a credential string which allows an application to supply a specific value that authenticates the application and allows for the creation and manipulation of XSETs. The specification does not provide any details on the structure of the credential string. The specification also does not provide any details on applications being granted specific access rights for writing versus reading or creating versus deleting. Therefore, the reference VIM will expect the credential string to contain a user name. If the user name is allowed to access a given SYSTEM, then the application will be allowed to access any and all XSET operations.

## Retention Library – Section 4.2

The XAM specification describes building an extension library to XAM that implements retention. This library is not implemented, but an example of one retention feature (FixedRetentionPeriod) will be supported via the XSET_Set() routine to demonstrate the retention features of XAM.

**Close() Semantics**

The specification is unclear on how to handle close operations (on either the SYSTEM or XSET level) should streams and/or XSETs be left open by the application. On close operations, the prototype reference VIM will not check to see if all other open handles have been properly closed.

# CHAPTER 4

# THE DESIGN OF A REFERENCE VIM

The reference VIM, hereafter noted as the RVIM, can be implemented using a hierarchical set of directories and files as described below.

## The RVIM Config File (RCF)

The first file to consider in the implementation of an RVIM is a configuration file loaded by the RVIM at system startup. The RCF can be located in the current working directory, or an environment variable can be set to instruct the RVIM of the location (and perhaps the name) of the RCF.

In order to implement a prototype RVIM, the RCF should contain the following information:

- The name of a SYSTEM (an ASCII string, i.e. “xam-system-1”);
- The XAM SYSTEM Base Directory (XBD, defined below); and
- The names of the users allowed to access the SYSTEM (as specified in the SDK_OpenSYSTEM() call).

Note that these items are sufficient for the purpose of writing a subset of XAM code. This file is read-only for the RVIM as there is no need to support the setting of

SYSTEM properties as part of this project. In order to more fully implement XAM, the following could be added to the RCF:

- The maximum number of allowable XSETs (in order to simulate a "full" SYSTEM;
- SYSTEM properties (i.e. name/value pairs such as "retention=true");
- Multiple SYSTEM names (and base directories) that allow an application to simulate simultaneous use of multiple SYSTEMs, including importing XSETs from one SYSTEM and exporting to another; and
- The number of XSETs currently in the SYSTEM Base Directory. Dynamically keeping this information for every new XSET would allow the RVIM to simulate XSET capacity restrictions.

## The SYSTEM Base Directory and XSET subdirectories

The XAM SYSTEM base directory (XBD) is used as a working directory when a XAM application connects to a given SYSTEM name. The XBD contains one type of object: directories that represent XSETs. If an XBD is currently empty, that means the SYSTEM currently contains no XSETs. If the XBD contains one or more subdirectories, then these subdirectories store XSETs that are contained by the SYSTEM. The name of the subdirectory that stores XSET information will be the actual XUID for that XSET.

XSET subdirectories (which have the XUID as the name) contain two types of files: (1) an XSET XML file which contains the names of all properties, policies, and

streams that are associated with the XSET, and (2) the actual streams (files) stored as part of the XSET.

The XSET XML file has the following XML structure:

- A properties tag which contains all properties/policies of the XSET, which encloses an individual property containing the name/value pair for a given property or policy within the XSET; and
- A streams tag which contains all streams of the XSET, which encloses an individual stream tag containing the name of the stream that was stored into the XSET.

The XSET subdirectory also contains files that represent the streams stored as part of the XSET. The names of the files are the names given to the stream when stored to the XSET.

By way of example, if a XAM application developer wishes to create an XSET of an X-RAY (with a stream named “x-ray”) for a patient named “Steve”, the RVIM would take the following steps:

- create an XSET subdirectory with a temporary name,
- within that subdirectory, store the X-RAY in a file named “x-ray”,
- store the following XML on an “XSET_Store()”

```
<xml>
    <properties>
        <property name=”creation.time” …
                                o
                                o                    “/>
        <property name=”Patient” value=”Steve”/>
    </properties>
    <streams>
```

```
            <stream name="x-ray"/>
        </streams>
    </xml>
```

- change the name of the subdirectory to match the returned XUID

The last step of naming the subdirectory takes into account that the prototype reference VIM names XUIDs based on timestamps and sequence numbers, which are known ahead of time. Future reference VIMs will have to consider that the XSET directory will need to be created before all properties and streams are known, as well as participating and non-participating parameters, and the initial directory name will likely be temporary and may require a rename once the XUID is known.

Directories that use the XUID as a name need not be created at all for an implementation that uses a database to map XUIDs to directories. For the sake of this project, however, a database will not be used, but XUIDs will intrinsically map to directories, as will be described in the "Algorithms" section below.

## RVIM Algorithms

Given the directory and file structure described above, this section describes the algorithms that are implemented by the RVIM. VIMs plug into a larger XAM SDK framework that is not yet in existence. Therefore, the VIM API for this prototype was coded as a straight one-to-one mapping of the XAM SDK API to a VIM API. What follows is a listing of all VIM APIs and the design of the functions based on the files and directories described above.

**VIM_SDK_Initialize()**

This routine initializes the RVIM. It loads the RVIM Config File (RCF) and parses the XML that contains the names of SYSTEMs, the base directory for each SYSTEM, and the users permitted to access the SYSTEM. These values are then placed in an RCF memory structure for future access. This RCF memory structure contains all of the information for the different SYSTEMs.

**VIM_SDK_Shutdown()**

This routine shuts down the RVIM by simply freeing the memory associated with the RVIM Config File (RCF).

**VIM_SDK_OpenSYSTEM()**

This routine accepts a connection string and security credentials and then returns a handle to a SYSTEM. The connection string for the RVIM should be the name of a SYSTEM found in the RVIM Config File. The security credentials for this project will simply be a string containing a user name. The RVIM scans the RCF in-memory data structure that contains the RCF information, locates the appropriate SYSTEM, and verifies the security credentials. If the credentials match, the RVIM returns a handle pointing to an offset in the RCF in-memory data structure that contains the information for this SYSTEM.

**VIM_SYSTEM_Close()**

This routine closes a SYSTEM for XSET operations. The algorithm accepts a SYSTEM handle and converts it to a previously allocated in-memory SYSTEM data structure, and frees that data structure.

**VIM_SYSTEM_DeleteXSET()**

This routine attempts to delete an XSET. The algorithm examines the in-memory data structure for the SYSTEM and extracts the SYSTEM base directory pathname. Then it appends the XUID name onto this pathname and determines that the XSET directory exists. Upon success, the RVIM prototype must first check the XSET data structure on disk to determine if the “FixedRetentionPeriod” parameter is set. If "FixedRetentionPeriod" is set, and the creation time plus retention period is not greater than the current time, the delete request must be rejected. If it is not set, the RVIM removes the on-disk XSET XML file, removes any streams (files) in the directory, and then removes the XSET directory itself. The XSET no longer exists within the context of that SYSTEM.

**VIM_SYSTEM_OpenXSET()**

This routine accepts a SYSTEM handle and a XUID and returns an XSET handle (if the XSET exists within the SYSTEM). The algorithm maps the SYSTEM handle to the in-memory SYSTEM data structure and extracts the SYSTEM base directory. The XUID is concatenated to the base directory string and verifies that the XSET directory exists. If the directory exists, an attempt is made to open the

standard XSET XML file that should be present in the XSET directory. A parser is then used to populate an in-memory XSET XML file with property and stream information present in the file. The address of the in-memory XSET XML file is converted into an XSET handle and returned to the application.

**VIM_XSET_Create()**

This routine accepts a handle to a SYSTEM and returns a handle to a new XSET. The routine maps the SYSTEM handle to the SYSTEM offset in the RVIM in-memory data structure, and from this information it can locate the SYSTEM base directory where it should store the new XSET. At this point, in a more robust RVIM implementation, the number of XSETs can be incremented and checked against the maximum allowable.

The next step for the RVIM is to create a directory for the new XSET. As mentioned previously, the name of the XSET directory will map to the XUID. The algorithm described here creates a directory name that will ultimately be used as the XUID. The algorithm concatenates the current counter of XSETs in this system with a timestamp such as the one returned from the time() system call. So the fifth XSET written to a SYSTEM at a given time may have a XUID of “005-1152654373”, and the RVIM creates a subdirectory of the same name. Note that the format of this XUID is not necessarily standard, however the XUID is opaque to the application and therefore it serves the purpose for an RVIM.

Once the XSET subdirectory has been created, an in-memory version of the XSET XML file is created, and the number of streams is set to zero. The “system

properties" are given default values and added to the in-memory XSET structure. A handle is then returned to the application which maps to this in-memory version.

**VIM_XSET_Get()**

This routine accepts an XSET handle and a character string for a specific property or policy within the XSET, as well as a pointer to a string that will eventually contain the value of that property or policy, if it exists. The algorithm maps the XSET handle to the in-memory XSET XML file and searches for the specified property. If the property name is found, it copies the value into the character string supplied by the caller.

**VIM_XSET_Set()**

This routine accepts an XSET handle and a name/value pair of character strings. It maps the XSET handle to the in-memory XSET XML file and either changes or inserts the name/value pair in the in-memory XSET XML file.

**VIM_XSET_Store()**

This routine accepts an XSET handle and returns a XUID. As mentioned previously, this version of the RVIM will never modify an existing XUID because participating streams and properties are not supported. The algorithm that this routine will use is to map the XSET handle to the in-memory XSET XML data structure, and then create (or overwrite) the in-memory XML file (which contains all

stream names and property tags) into an actual XML file in the XSET directory. The XUID for the XSET is then returned to the application.

**VIM_XSET_Close()**

This routine accepts an XSET handle and closes the XSET so that no more operations can be performed on it. The algorithm simply frees the in-memory data structure being used to track the XSET XML file. For this prototype, no checks are made as to whether all streams have been closed, because the XAM specification does not specify how to handle unclosed streams.

**VIM_XSET_OpenStream()**

This routine accepts an XSET handle and a stream name and attempts to open an existing stream for the application. The XSET handle is mapped to the XSET XML in-memory data structure to find the XUID and the pathname to the XSET directory. The stream name passed into this routine is then concatenated and an attempt is made to open the file. If the open succeeds, the file handle is converted to a stream handle and returned to the application.

**VIM_Stream_Create()**

Once an application has created an XSET, a next logical step would be to create a stream, a series of bytes that will be given a name and stored as part of the XSET. This routine accepts an XSET handle and a stream name. The XSET handle will point to the in-memory version of the XSET XML file, which contains the

XSET base directory and the XUID name, which allows an absolute pathname to be built. The name of the stream is concatenated to this pathname. A file can then be opened for writing, and if this call succeeds, the stream name is added to the in-memory XSET XML file, and a handle to the file is returned to the application.

**VIM_Stream_Read()**

This routine accepts a stream handle, a buffer, length, and offset, and attempts to read from the open file into the buffer. The stream handle is converted to a file descriptor and the operation is attempted by the RVIM.

**VIM_Stream_Write()**

Once the application has a stream handle, it can issue writes to the stream by passing the stream handle along with a buffer and length. The RVIM converts the stream handle to a file handle and attempts to write the buffer to the file.

**VIM_Stream_Close()**

This routine simply closes the stream handle by converting it to a file descriptor and issuing the close command.

Given the file system layout, configuration files, and RVIM algorithms described above, application code can now be written to exercise and evaluate the XAM API.

# CHAPTER 5

## COMPARING XAM TO THE FILE SYSTEM API

Now that a framework has been put in place to enable the writing of XAM applications, XAM applications can be written. These XAM applications can be placed side by side with similar applications written using a file system API. This exercise will point out areas of XAM that need improvement in order to move toward a goal of the XAM API becoming ubiquitous.

This thesis will use the POSIX file system API as a basis for comparison. In particular, comparisons will be made against file system routines described in the Open Group Base Specification, which is freely available for reading on the www.opengroup.org website (Open Group).

There are five general points of comparison when XAM code is written and compared to file system code. These comparison points are (1) creation of an XSET, (2) reading data from an XSET, (3) reading metadata from an existing XSET, (4) deleting an XSET, and (5) the error handling infrastructure. Note that these comparisons are made against functionality that is *already supported* by the POSIX file system API. Areas where XAM provides functionality *not supported* by the POSIX file system API will be covered in the next chapter.

## Coding Criteria

The XAM and file system software that will be written will adhere to the same coding styles so as to facilitate comparison. In particular, (1) programs will only include comments that delineate major sections of code (e.g. “open”, “write” “close”), (2) error values will not be inspected by the application (this results in a reduced amount of code to compare), (3) comparable programs will accept similar input when applicable, and (4) programs will be written in C. Note that ignoring error values *completely* is not a good idea, so a code sample will be included that ***does*** evaluate XAM error handling.

## Creation of an XSET

This application creates an XSET using XAM, and creates a file using the file system API. The application code simply writes the alphabet into a file. In both cases the file name is called alphabet. Figure 2 shows the two sets of code side by side.

**XAM**

```
#include "stdio.h"
#include "xamapi.h"

main(int argc, char *argv[])
{
// declare variables

STATUS_TYPE  status;
SDKHandle    sdkhandle;
SYSTEMHandle systemhandle;
XSETHandle   xset;
STREAMHandle stream;
char xuid_array[XAM_STRING_LENGTH];

char WRITE_STRING[] =
          "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
int bytes_written;

if (argc != 3)
{
    printf("\nUsage: write system username\n");
    exit(-1);
}

// open

status = SDK_Initialize(&sdkhandle);
status = SDK_OpenSYSTEM(sdkhandle,
                        argv[1],
                        argv[2],
                        &systemhandle);
status = XSET_Create(systemhandle, &xset);
status = Stream_Create(xset, "alphabet", 0,
                       &stream);

// write

status = Stream_Write(stream,
                      sizeof(WRITE_STRING),
                      WRITE_STRING,
                      &bytes_written);
status = XSET_Store(xset, &xuid_array[0]);

// close

status = Stream_Close(stream);
status = XSET_Close(xset);
SYSTEM_Close(systemhandle);
status = SDK_Shutdown(sdkhandle);

printf("%s\n", xuid_array);
}
```

**FILE SYSTEM**

```
#include <stdio.h>
#include <stdlib.h>

main(int argc, char *argv[])
{
// declare variables

FILE *fp;
char WRITE_STRING[] =
      "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
int bytes_written;

if (argc != 1)
{
    printf("\nUsage: fs_write\n");
    exit(-1);
}









// open

fp = fopen("alphabet","w");









// write

bytes_written = fwrite(WRITE_STRING,
                       sizeof(WRITE_STRING),
                       1,
                       fp);


// close

fclose(fp);

printf("\n");
}
```

**Figure 2 - Application Write**

What are the items that stand out from the side by side comparison?

**Number of Handles**

The XAM application code requires four handles for this simple exercise (SDK, SYSTEM, XSET, and stream). The file system code requires only one (FILE).

**Number of Routines**

The XAM application code requires ten routines to create an XSET, while the file system code requires three.

**Program and Function Arguments**

The main XAM application code requires two arguments: the name of the XAM SYSTEM and the security credentials string. The main file system code requires no arguments. The XAM code requires no flags when opening an XSET, while the file system code requires permissions such as "w".

**Return Value**

The XAM application code returns a XUID. The file system code returns nothing.

**General Comments on Creating an XSET**

A subtle difference that is not apparent in XSET creation is that the XAM code specifies location via a SYSTEM, while the file system code specifies location via its

current working directory. For file systems the environment of a process always contains the root directory location "/" and the current working directory. XAM does not have this paradigm, and it asks the application to specify a physical storage subsystem as a target.

**Reading an XSET**

Figure 3 shows code that reads the alphabetical string.

**XAM**

```
#include "stdio.h"
#include "xamapi.h"



main(int argc, char *argv[])
{
// declare variables

STATUS_TYPE  status;
SDKHandle    sdkhandle;
SYSTEMHandle systemhandle;
XSETHandle   xset;
STREAMHandle stream;

char READ_STRING[XAM_STRING_LENGTH];
int bytes_read;

if (argc != 4)
{
    printf("\nUsage: read system user xuid\n");
    exit(-1);
}

// open

status = SDK_Initialize(&sdkhandle);
status = SDK_OpenSYSTEM(sdkhandle,
                        argv[1],
                        argv[2],
                        &systemhandle);
status = SYSTEM_OpenXSET(systemhandle,
                         argv[3],
                         &xset);
status = XSET_OpenStream(xset,
                        "alphabet",
                         &stream);

// read

status = Stream_Read(stream,
                     0,
                     sizeof(READ_STRING),
                     READ_STRING,
                     &bytes_read);

// close

status = Stream_Close(stream);
status = XSET_Close(xset);
SYSTEM_Close(systemhandle);
status = SDK_Shutdown(sdkhandle);

printf("%s\n", READ_STRING);
}
```

**FILE SYSTEM**

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_BUF 512

main(int argc, char *argv[])
{
// declare variables

FILE *fp;
char READ_STRING[MAX_BUF];
int bytes_read;

if (argc != 1)
{
    printf("\nUsage: fs_read\n");
    exit(-1);
}




// open

fp = fopen("alphabet","r");









// read

bytes_read = fread(READ_STRING,
                   sizeof(READ_STRING),
                   1,
                   fp);


// close

fclose(fp);

printf("%s\n", READ_STRING);
}
```

**Figure 3 - Application Read**

**Number of Handles**

The XAM code again uses many more handles (found on lines 11-14). The file system code only uses one (found on line 10).

**Number of Routines**

The XAM code in this case uses nine function calls as opposed to three for the file system. Starting on line 27 both applications execute open() logic, but it takes the XAM application four routines to execute an open(), whereas the file system code accomplishes an open() using one line of code.

**Program and Function Arguments**

The XAM application in this use case requires three parameters: the XAM SYSTEM, the credentials, and the XUID of the XSET that contains the alphabetical string. The file system code still requires no program arguments. Again, the file system code requires a flag on its open call such as "r".

**General Comments on Reading an XSET**

Again, the XAM application must specify the context of the read command by specifying a SYSTEM on the command line. Similarly, the file system application assumes the context is its current working directory.

**Properties of an XSET**

Figure 4 shows code that dumps XSET statistics. Note that the equivalent in the file system API is the “stat” routine, which takes a file name and returns a structure which contains attributes about a file.

**XAM**

```
#include "stdio.h"
#include "xamapi.h"




main(int argc, char *argv[])
{
// declare variables

STATUS_TYPE  status;
SDKHandle    sdkhandle;
SYSTEMHandle systemhandle;
XSETHandle   xset;
STREAMHandle stream;
xset_prop_list_t props;

int i;

if (argc != 4)
{
    printf("\nUsage: dump sys user xuid\n");
    exit(-1);
}

// open

status = SDK_Initialize(&sdkhandle);
status = SDK_OpenSYSTEM(sdkhandle,
                        argv[1],
                        argv[2],
                        &systemhandle);
status = SYSTEM_OpenXSET(systemhandle,
                         argv[3],
                         &xset);

// fetch attributes

status = XSET_GetProperties(xset, &props);

// print attributes

for (i=0;i < props.n_properties;i++)
{
  printf("%s: %s\n", props.name[i],
                     props.value[i]);
}









// close

status = XSET_Close(xset);
SYSTEM_Close(systemhandle);
status = SDK_Shutdown(sdkhandle);
}
```

**FILE SYSTEM**

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

main(int argc, char *argv[])
{
// declare variables

struct stat buf;
int err;

if (argc != 2)
{
    printf("\nUsage: fs_dump filename\n");
    exit(-1);
}









// fetch attributes

err = stat(argv[1],&buf);

// print attributes

printf("st_dev: %d\n", buf.st_dev);
printf("st_ino: %d\n", buf.st_ino);
printf("st_mode: %d\n", buf.st_mode);
printf("st_nlink: %d\n", buf.st_nlink);
printf("st_uid: %d\n", buf.st_uid);
printf("st_gid: %d\n", buf.st_gid);
printf("st_rdev: %d\n", buf.st_rdev);
printf("st_size: %d\n", buf.st_size);
printf("st_blksize: %d\n", buf.st_blksize);
printf("st_blocks: %d\n", buf.st_blocks);
printf("st_atime: %d\n", buf.st_atime);
printf("st_mtime: %d\n", buf.st_mtime);
printf("st_ctime: %d\n", buf.st_ctime);
}
```

**Figure 4 – Dump Statistics**

**Number of Handles**

The XAM code again uses many handles. The file system routine uses no handles.

**Number of Routines**

The XAM code in this case uses seven function calls as opposed to one for the file system. Note that the primary reason for the amount of function calls in XAM is that the application must drill down to the XSET level before it can acquire XSET statistics, while the file system application simply calls stat() and passes the pathname of the file it is interested in.

**Program and Function Arguments**

The XAM application again requires three parameters: the XAM SYSTEM, the credentials, and the XUID of the XSET. The file system code requires the name of the file. Note also that the file system code requires a struct that has pre-defined fields. The XAM code requires an array containing a list of name/value pairs in the form of strings.

**Return Value**

The XAM protocol returns an array of name/value pairs, which are then printed out by the application. The file system protocol returns a struct stat and the application must print out each individual field of the structure. XAM gives the application developer flexibility in viewing new XSET properties without having to

rewrite any code.

## Deleting an XSET

Figure 5 shows the code that deletes an XSET.

**XAM**

```
#include "stdio.h"
#include "xamapi.h"




main(int argc, char *argv[])
{
// declare variables

STATUS_TYPE  status;
SDKHandle    sdkhandle;
SYSTEMHandle systemhandle;
XSETHandle   xset;
STREAMHandle stream;

if (argc != 4)
{
    printf("\nUsage: delete sys user xuid\n");
    exit(-1);
}

// initialize

status = SDK_Initialize(&sdkhandle);

status = SDK_OpenSYSTEM(sdkhandle,
                        argv[1],
                        argv[2],
                        &systemhandle);

// perform delete

status = SYSTEM_DeleteXSET(systemhandle,
                           argv[3]);

// close

SYSTEM_Close(systemhandle);

status = SDK_Shutdown(sdkhandle);
}
```

**FILE SYSTEM**

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

main(int argc, char *argv[])
{
// declare variables

struct stat buf;
int err;

if (argc != 2)
{
    printf("\nUsage: fs_delete filename\n");
    exit(-1);
}









// perform delete

err = remove(argv[1]);
}
```

**Figure 5 – Application Delete**

## Number of Handles

The XAM code for deleting uses many handles. The file system routine uses no handles.

## Number of Routines

The XAM code in this case uses five function calls as opposed to one for the file system.

**Program and Function Arguments**

The XAM application again requires three parameters: the XAM SYSTEM, the credentials, and the XUID of the XSET. The file system code requires the name of the file.

**Return Value**

Neither the XAM nor the file system library returns any value other than the status of the delete command.

**Evaluating XAM error handling**

Figure 6 shows two XAM applications. The first application (copied from Figure 2) does not check status values, while the second application does check error values.

**XAM NO ERROR HANDLING**

```
#include "stdio.h"
#include "xamapi.h"

main(int argc, char *argv[])
{
// declare variables

STATUS_TYPE  status;
SDKHandle    sdkhandle;
SYSTEMHandle systemhandle;
XSETHandle   xset;
STREAMHandle stream;
char xuid_array[XAM_STRING_LENGTH];

char WRITE_STRING[] =
          "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
int bytes_written;

if (argc != 3)
{
    printf("\nUsage: write system username\n");
    exit(-1);
}

// open

status = SDK_Initialize(&sdkhandle);
status = SDK_OpenSYSTEM(sdkhandle,
                        argv[1],
                        argv[2],
                        &systemhandle);
status = XSET_Create(systemhandle, &xset);
status = Stream_Create(xset, "alphabet", 0,
                       &stream);

// write

status = Stream_Write(stream,
                      sizeof(WRITE_STRING),
                      WRITE_STRING,
                      &bytes_written);
status = XSET_Store(xset, &xuid_array[0]);

// close

status = Stream_Close(stream);
status = XSET_Close(xset);
SYSTEM_Close(systemhandle);
status = SDK_Shutdown(sdkhandle);

printf("%s\n", xuid_array);
}
```

**XAM WITH ERROR HANDLING**

```
#include "stdio.h"
#include "xamapi.h"

main(int argc, char *argv[])
{
// declare variables

STATUS_TYPE  status;
SDKHandle    sdkhandle;
SYSTEMHandle systemhandle;
XSETHandle   xset;
STREAMHandle stream;
char xuid_array[XAM_STRING_LENGTH];

char WRITE_STRING[] =
          "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
int bytes_written;

if (argc != 3)
{
    printf("\nUsage: write system username\n");
    exit(-1);
}

// open

status = SDK_Initialize(&sdkhandle);

if (status != STATUS_TYPE_NO_ERROR)
{
  printf("SDK_Initialize()");
  exit(-1);
}

status = SDK_OpenSYSTEM(sdkhandle,
                        argv[1],
                        argv[2],
                        &systemhandle);

if (status != STATUS_TYPE_NO_ERROR)
{
  printf("SDK_OpenSystem()");
  close_handles(&sdkhandle,NULL,NULL,NULL);
  exit(-1);
}

status = XSET_Create(systemhandle, &xset);

if (status != STATUS_TYPE_NO_ERROR)
{
  printf("XSET_Create()");
  close_handles(&sdkhandle,&systemhandle,
                NULL, NULL);
  exit(-1);
}

status = Stream_Create(xset, "alphabet", 0,
                       &stream);

if (status != STATUS_TYPE_NO_ERROR)
{
  printf("Stream_Create()");
  close_handles(&sdkhandle,&systemhandle,
                &xset, NULL);
  exit(-1);
}

// write

status = Stream_Write(stream,
                      sizeof(WRITE_STRING),
                      WRITE_STRING,
                      &bytes_written);

if (status != STATUS_TYPE_NO_ERROR)
{
  printf("Stream_Write()");
  close_handles(&sdkhandle,&systemhandle,
                &xset, &stream);
  exit(-1);
}

// close

status = Stream_Close(stream);

if (status != STATUS_TYPE_NO_ERROR)
{
  printf("Stream_Close()\n");
  close_handles(&sdkhandle,&systemhandle,
                &xset, NULL);
  exit(-1);
```

**Figure 6 – XAM Error Handling**

## General Comments On Error Handling in XAM

The XAM error handling application in Figure 6 is roughly three times longer than the same code without error handling. Lines 97-156 of the coding sample were eliminated for the purpose of fitting the coding sample on one page. A simple application that creates XSETs has ten routines that return error values. Each of these routine return a value that must be processed by the application; this explains the increase in lines of code.

The number of handles required by XAM causes the application to come up with a strategy for closing open handles in the face of errors. The example above displays this behavior by calling a close_handles() routine. The calling of this routine is a burden for the application programmer, but it is required because the XAM version 1.2 specification is silent regarding the exit behavior of the XAM SDK.

Finally, the lack of error code descriptions listed in the XAM specification do not currently allow XAM applications to perform any specific error handling procedures. There is no way to tell if an error occurred at the VIM layer, at the XAM SYSTEM layer, because of bad application logic, or for some other reason.

## Coding Analysis and Recommendations

The coding samples listed above now allow for a more in-depth analysis of XAM and how it compares to traditional file system programming. Given the ubiquity of the file system API, XAM is clearly going to fall short in a comparison of this nature. Recommendations will be made based on this comparison (subsequent sections will compare XAM in a more favorable light).

**Addressing the Number of Handles**

The number of handles required to program XAM is clearly a disadvantage. For most operations a XAM application has to keep track of (1) a handle to the XAM SDK, (2) a handle to the XAM system, (3) a handle to an XSET, and (4) a handle or handles to underlying files within the XSET. Having this many handles adds lines of code and complexity to XAM applications.

XAM could cut the number of handles in half by making the SDK handle and SYSTEM handle implicit by using common techniques. For example, why not have the XAM library itself keep track of the SDK handle during the SDK_Initialize() call? Similarly, the SDK_Initialize() routine can map the application process to a given default XAM system. This mapping could occur, for example, by use of an environment variable or configuration file to map an application to a default XAM System. By using techniques such as this, the application is no longer required to keep track of a handle to call SDK_Initialize(), and the need to call both SDK_OpenSystem() and SYSTEM_Close() disappears. Similarly, XSET operations no longer need to occur within the context of a SYSTEM handle, which further reduces the complexity of XAM application code.

One could argue that such an approach limits the application to the use of one and only one SYSTEM. It is true that certain applications (those that wish to move XSETs between SYSTEMs using XSET export and import routines) need to distinguish specific SYSTEMs. For example, a customer might wish to migrate *from* one XAM SYSTEM because a lease has expired; another customer might wish to

migrate *to* another XAM SYSTEM because it is a higher performing system. Other applications might wish to request the specific policies and properties of a SYSTEM. These can also use the existing API. However, many applications will simply want to manipulate XSETs without regard for the underlying hardware (similar to how many applications use file systems without regard for the hardware underneath). For these applications, eliminating SDK and SYSTEM handles would be beneficial.

**Addressing the Number of Routines**

For most XAM applications there can be at least twice as many function calls as an equivalent file system application. This is due to the fact that at the lowest level XAM has streams and file systems have files, but XAM then layers XSET routines on top of streams.

XAM could reduce the number of routines by providing macros that provide implicit XSET operations underneath the covers when the XSET details are not important to the application. For example, the reading of an XSET stream requires five operations: (1) the opening of an XSET, (2) the opening of a stream within an XSET, (3) the read of the stream, (4) the close of the stream, and (5) the closing of the XSET. If the goal of the application is to simply read from an XSET stream, then steps (1) and (5) can be eliminated. A version of the XSET_OpenStream() routine could accept a XUID and a stream name and allow direct access to a stream without directly addressing the XSET. One might argue that this approach does not allow for advanced XSET techniques (viewing XSET properties), but certainly more macros can be written that provide more advanced features.

## Addressing the Program Arguments

Each XAM application displayed above accepts the XAM SYSTEM name and credential string as arguments.

As mentioned previously, application knowledge of specific XAM SYSTEMs may not be desirable. Indeed file system applications are typically unaware of underlying hardware configuration and/or location. Therefore application knowledge about XAM SYSTEMs can be made unnecessary through the use of techniques such as environment variables.

The credentials string at first seems to be a similar disadvantage. The file system API does not have an equivalent mechanism to authenticate user access, but instead uses the user/group/other technique. The credential string may seem a burden to the enablement of XAM application programming by individual users who simply want to manipulate XSETs. However, there have been complaints in the file system community that the user/group/other concept is too simplistic for file sharing and ownership. This provides the XAM community an opportunity to implement a robust security model.

## Addressing the Return Values

XAM presents a new paradigm to applications wishing to store data; the application does not have control over naming. With file systems the application creates the name of the file. Upon writing and closing the file the file name is unchanged. XAM streams work the same way; the application gets to assign names

to the streams. However, when the XSET is stored, or committed, a XUID is returned. What to do with this XUID presents some unique challenges to XAM applications.

One option for a XAM application is to store the XUID in a database along with certain keywords that assist in the eventual lookup of this XUID. This approach seems wasteful because having the application keep this information is a duplication of information already stored within the XSET. It also places a burden on the application to create and maintain a database. How then can a XAM application open an XSET for a given XUID if it doesn't keep track of all XUIDs?

The XAM answer to this question is to use XSET_Query (), but version 1.2 of the XAM specification is lacking in details about the specific implementation of query. The stated plan is for an application to provide certain metadata tags/keywords to the system and for XSET_Query() to provide an XSET in return. This XSET will then contain a list of any and all XUIDs that satisfy the search criteria specified by the metadata tags/keywords.

This technique for discovering a XUID seems overly complicated when compared to opening an existing file using file system techniques. A XUID is similar to an i-node value in that it means nothing to an application but it means everything to the file system. The level of a XUID in this sense is too-low and not convenient to an application developer. A file system application, however, provides metadata tags/keywords in the form of an absolute pathname, and it gets back one (and only one) file handle. Is there a way for XAM to add something similar?

One suggestion would be for XSETs to be given location tags that are similar to absolute pathnames that lead to files. This location tag can be a system attribute of an XSET. A XAM application would call XSET_SetProperty() to set the attribute and subsequently call a (proposed) new routine such as OpenXSETUsingTag(absolute_pathname). This allows a XAM application to open up an XSET using techniques familiar to a file system programmer.

## Addressing File System Routines

At the lowest level, XAM and file systems both manipulate streams or files. XAM uses an open stream command, file systems use fopen(). XAM uses stream read, write, and close routines, file systems use fread(), fwrite(), and fclose(). So in many ways, XAM and file systems are similar in this regard.

File systems, however, have a much larger toolkit when it comes to manipulating files. For example, the fopen() routine has the ability to open files in a variety of different modes (read, write, append, etc). XAM has no such equivalent. File systems also have a long list of utility routines that can be used on files. Examples include fgetpos(), fseek(), fsetpos(), ftell(), fgetc(), fgets(), fflush(), fstat(), and fsync(), among others. The version 1.2 XAM specification is silent on this functionality. If these routines are valuable to file system applications, their implementation should be considered as part of XAM as well.

The fstat() command likewise contains a large amount of information about files, and should be considered for XAM streams as well.

XSETs do not need to support routines such as fgetpos(), but XAM should consider how these routines apply to low-level streams.

## Addressing File System Stat()

Evaluating the list of options returned by the file system stat() command is illuminating when compared to XAM.

The **st_dev** field, which identifies the device containing the file, does not have an equivalent in XAM. There may be a certain class of XAM applications (e.g. focused on migration XSETs from a given system to another) that would care to know what XAM device currently holds a given XSET.

The **st_ino** field provides a unique identifier for this file within the context of a given device. XAM does provide this information in the form of the XUID (the system.xset.name property).

The **st_mode** field is used for a variety of purposes in the file system, including the file type and access permission information. The file type information (is the file a directory, regular file, character device, etc.) is generally not applicable to XSETs. The access permission information, however, is a different story. File system applications can get/set permissions such as read, write, delete, and execute. These permissions limit accessibility to different users and groups.

The XAM version 1.2 specification does not cover this area. Can applications have read permission on XSETs but not write permission? If so, how does an application modify the permissions of a given XSET? The specification briefly

touches on this topic with its discussion of storage pools, but not to the level of sharing given XSETs among different applications.

The **st_nlink** field describes the number of hard links to this file. The XAM spec need not address links because XAM applications are divorced from location information about how XSETs are actually stored on a XAM system.

The **st_uid** and **st_gid** fields are group and user permission information, and related to the **st_mode** description above. XAM does not distinguish individual users and groups.

The **st_rdev** field is related to special devices and not applicable to XAM.

The **st_size** field describes the length of the file. There is no equivalent in XAM. It would be a very useful field for XAM to supply, because currently there is no way to access the size of an XSET.

The **st_blocksize** and **st_blocks** fields relate to the block size of the underlying storage device and the number of blocks that the file actually occupies. XAM had no such equivalent.

The **st_atime** field reveals the last access of this file (i.e. the last read request). XAM has no equivalent. Note that this may be a good thing for XAM. Implementing atime is complicated for file system developers because it causes additional disk accesses to update the atime field.

The **st_mtime** field reveals the last modification to this file. XAM's equivalent is the system.mtime property of an XSET. However, system.mtime does not have the granularity to differentiate between modification of XSET streams and modification of XSET properties (see st_ctime).

The **st_ctime** field reveals the last time the *attributes* of a file were changed. XAM has no equivalent.

## Addressing File System Error Handling

XAM avoids the convoluted file system errno methodology by always returning a status code that contains the equivalent of errno. Many POSIX routines also follow this approach. The XAM 1.2 specification has currently deferred a discussion of error codes. Generating robust, sensible error codes is a critical step for subsequent versions of the XAM specification.

Minimizing the number of routines by creating macros is also a sensible way to cut down on the amount of application error handling logic currently caused by the large number of XAM routines.

The number of handles currently required by XAM will be problematic for XAM error handling. Consider an application that receives a XAM stream error. If the application wishes to exit, or if it wishes to cease XAM processing, it must potentially close a stream handle, an XSET handle, a SYSTEM handle, and the SDK handle. This amount of layered handle closing adds complexity to the code. The XAM specification should specify SDK exit functionality that ensures that handles are automatically closed by the SDK upon application exit.

## Query Discussion

The XSET_Query() routine has not been implemented in this version of the RVIM, but it needs evaluation. The routine is extremely important because it allows

an application to discover XSETs that contain certain searchable keywords. The current XAM query mechanism is confusing because queries are started via an XSET function call, but a query is aborted using a stream function call.

The XAM specification itself refers to the query API as "convoluted" (XAM Specification, 2006). It attempts to use XSETs to store query results, but implementing an abort request via an XSET stream is too confusing. A separate API which submits a QUERY and aborts a QUERY via a SYSTEM function call makes more sense.

# CHAPTER 6

## XAM BENEFITS VERSUS THE FILE SYSTEM API

Certainly it has been shown that the XAM API, in its initial form, needs improvement in several areas when compared to file systems. There are several things that the file system API can do that the XAM API cannot.

The XAM API, however, can do some things that the file system API cannot. In order to evaluate the benefits of XAM, it is necessary to discuss what type of application would rather use XAM than the file system API.

For the purpose of illustration this paper will examine the use case of *email archiving*, and in particular the U.S. Security and Exchange Commission's rule 17a-4. One of the reasons for the growth of fixed-content storage systems has been for new laws in place which mandate how companies manage their emails. Rule 17a-4 presents several rules which make XAM an attractive option (Cincinnati College of Law, 2003).

Information on the Securities and Exchange Act of 1934 can be found at the U.S. Security and Exchanged Commission's website at www.sec.gov. With the introduction of this act the United States government began to strictly regulate activities in financial markets. As these activities began to take a more electronic form, the SEC decided to place requirements on "preservation" of any email relating to financial transactions. SEC rule 17a-4 spells out these preservation requirements

in detail. A complete summary of rule 17a-4 can be found at the University of Cincinnati College of Law website at http://www.law.uc.edu/CCL/34ActRls/rule17a-4p5203.html.

Archiving and preservation of email is an attractive use case for a fixed-content storage system. Therefore email archiving should be a candidate application for the XAM API. Rule 17a-4 contains at least three important requirements that are suited for XAM: (1) retention, (2) non-rewritable, and (3) the accuracy of the storage media recording process. For each of these cases the XAM API will be discussed and compared to a traditional file system.

**Retention of Content**

The text of rule 17a-4, in section (b-4), state that "every such broker and dealer shall preserve for a period of not less than three years ….. Originals of all communications ….relating to his business as such". An application that wishes to be compliant with this requirement needs to be able to specify a three-year retention period as part of the archiving of the email. Nowhere in the file system API is there a method to accomplish this. However, the XAM API allows the setting of a retention value before the XSET is committed. Consider the code sample listed below in Figure 7 (which is a modified version of the "XAM Write" example listed in Figure 2).

**XAM**

```
 1  #include "stdio.h"
 2  #include "xamapi.h"
 3
 4  main(int argc, char *argv[])
 5  {
 6  STATUS_TYPE  status;
 7  SDKHandle    sdkhandle;
 8  SYSTEMHandle systemhandle;
 9  XSETHandle   xset;
10  STREAMHandle stream;
11
12  char WRITE_STRING[] =
13        "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
14
15  char xuid_array[XAM_STRING_LENGTH];
16
17  int bytes_written;
18
19  if (argc != 3)
20  {
21      printf("\nUsage: write system username\n");
22      exit(-1);
23  }
24
25  status = SDK_Initialize(&sdkhandle);
26
27  status = SDK_OpenSYSTEM(sdkhandle,
28                          argv[1],
29                          argv[2],
30                          &systemhandle);
31
32  status = XSET_Create(systemhandle, &xset);
33
34  status = Stream_Create(xset, "alphabet", 0, &stream);
35
36  status = Stream_Write(stream,
37                        sizeof(WRITE_STRING),
38                        WRITE_STRING,
39                        &bytes_written);
40
41  status = Stream_Close(stream);
42
43  status = XSET_SetPolicy(xset,
44                          "FixedRetentionPeriod",
45                           365 * 3);
46
47  status = XSET_Store(xset, &xuid_array[0]);
48
49  status = XSET_Close(xset);
50
51  SYSTEM_Close(systemhandle);
52
53  status = SDK_Shutdown(sdkhandle);
54
55  printf("%s\n", xuid_array);
56  }
```

**Figure 7 - XSET Retention**

Lines 43-45 of Figure 7 display the XSET_SetPolicy() routine, which is the key routine for an application wishing to set a retention policy for content. In this example we are using a policy set in days (365 * 3 = 3 years); the XAM API will eventually formally specify how to specify retention lengths. As a result of receiving a XUID during the XSET_Store() command, the application and the XAM storage system now have a contract stating that this XSET cannot be erased for a total of three years. The contents of this XSET, for the example listed above, would be one or more emails containing information relating to a broker's business.

An equivalent function call to retain files for a given period of time does not exist in the file system API.

**Non-Rewriteable Content**

Another rule mandated by the SEC is that archived email cannot be overwritten, meaning that it cannot be opened and edited, and it cannot be overwritten by another email. Certainly both XAM and file systems have the ability to open content and modify it, but XAM has a feature that prevents edits and overwrites. This capability of XAM was previously described as participating or non-participating fields. When archiving an email as part of an XSET, for example, the application can specify everything in the email to be participating. This forces the XUID to be constructed based on the content of the email. An example of the XAM API to set a field to participating is shown in Figure 8, which is a modification of the XAM code listed in Figure 2.

**XAM**

```
 1  #include "stdio.h"
 2  #include "xamapi.h"
 3
 4  main(int argc, char *argv[])
 5  {
 6      STATUS_TYPE  status;
 7      SDKHandle    sdkhandle;
 8      SYSTEMHandle systemhandle;
 9      XSETHandle   xset;
10      STREAMHandle stream;
11
12      char WRITE_STRING[] =
13            "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
14
15      char xuid_array[XAM_STRING_LENGTH];
16
17      int bytes_written;
18
19      if (argc != 3)
20      {
21          printf("\nUsage: xam_write system username\n");
22          exit(-1);
23      }
24
25      status = SDK_Initialize(&sdkhandle);
26
27      status = SDK_OpenSYSTEM(sdkhandle,
28                              argv[1],
29                              argv[2],
30                              &systemhandle);
31
32      status = XSET_Create(systemhandle, &xset);
33
34      status = Stream_Create(xset, "alphabet", 0, &stream);
35
36      status = Stream_Write(stream,
37                            sizeof(WRITE_STRING),
38                            WRITE_STRING,
39                            &bytes_written);
40
41      status = Stream_Close(stream);
42
43      status = XSET_SetParticipating(xset,
44                                     "alphabet",
45                                     "true");
46
47      status = XSET_Store(xset, &xuid_array[0]);
48
49      status = XSET_Close(xset);
50
51      SYSTEM_Close(systemhandle);
52
53      status = SDK_Shutdown(sdkhandle);
54
55      printf("%s\n", xuid_array);
56  }
```

**Figure 8 - XSET Participating**

Line 43 contains the SetParticipating() call. When the XSET is stored (XSET_Store()) a XUID is returned. If the application wishes to fetch this email, it uses the XUID. The application can then open the email and overwrite the contents. When the XSET is committed to the storage system via the XSET_Store() routine, however, a *new* XUID is returned, and the original remains intact. The XAM system *must* return a new XUID because a participating field within the XSET has been changed. In this way XAM is able to support the 17a-4 requirement for being non-rewritable.

File systems have no such implicit functionality. The ability to overwrite a file is dependent on permissions that are beyond the control of an email archiving application. If the email archiving can write a file to a file system, what prevents it from overwriting the same file? As long as permissions can be compromised, the file can be compromised, and protection against file system overwrites cannot be guaranteed.

**The Accuracy of Content**

Another important rule found in 17a-4 is the accuracy of the storage media that is recording an email. Has the email become corrupt? Another important question relates to overwrites: how can an application guarantee that an email has not been overwritten? The SEC is looking for proof that the data that was originally written is indeed the content that was retrieved.

XAM handles this situation nicely via the same property of participating versus non-participating. If an email is stored using all participating fields in the

XSET, then the XAM system is using a specific algorithm to calculate the XUID (e.g. an MD5 hash algorithm). When an XSET is retrieved based on a certain XUID, these algorithms can be re-run (by the client-side VIM for example) to validate that the bytes in the XSET correctly map to the XUID.

This authenticity checking is indeed an algorithm already used by EMC's Centera product. Network transfers from the Centera storage system are verified at the application side via MD5 checking within the Centera SDK. XAM VIMs are well-positioned, therefore, to use the same type of technique. Note that third party entities could also become involved in checking for this authentication (even the application itself could do it).

File systems cannot offer the same guarantee. It is true that file systems can be mounted upon storage infrastructures that have strong data integrity guarantees. The application, however, cannot call any file system routine to guarantee that this is so. Applications can certainly perform their own checksums and naming techniques, but this may be more of a burden than an application is willing to take on.

**Other Use Cases**

There are other legal use cases that place requirements on a company's fixed content. The Health Insurance Portability and Accountability Act (HIPAA) of 1996, listed at http://www.legalarchiver.org/hipaa.htm, places legal requirements on fixed content in the medical community. Section 1173 of HIPAA contains a SAFEGUARDS section which states "Each person ….. who maintains or transmits health information" must "ensure the integrity and confidentiality of the information".

XAM is clearly on the right track in this regard, as it has the ability to verify integrity (via the participating flag) and it is on track to maintain confidentiality (with the credential string).

There are other non-legal use cases for XAM as well. There are many scientific projects throughout the world that must process fixed content, and they need to verify the integrity of the fixed content so that the results are accurate. Consider the SETI@home project, described at http://setiathome.berkeley.edu. This project uses computers around the globe to analyze radio telescope data (fixed content).

Finally, proposals have been made which use fixed content storage repositories as the center of an individual's collective fixed content life, including music, video, health records, bank records, etc. XAM is well suited for this type of fixed content, which requires guaranteeing the integrity of data as well as the security of who can view that data. One such proposal is the "My World Information Brokerage Built on CAS" concept listed at http://www.cascommunity.org.

# CHAPTER 7

# CONCLUDING REMARKS

This paper has described a new technology in the storage industry: fixed-content storage systems. XAM is a new software API that will allow application access to fixed-content records. The XAM API has been explained and a XAM simulation environment has been proposed and a prototype has been implemented (the reference VIM). This reference VIM was then used to develop XAM applications, and these applications were then compared to equivalent file system applications. The comparison to file systems yielded the following set of recommendations for the industry leaders developing the XAM specification:

- Reduce the number of handles required to program XAM.
- Eliminate the need for addressing XAM systems directly.
- Introduce macros to reduce the number of XAM functions required to accomplish certain tasks
- Consider enhancing XAM stream functions to support traditional file system functionality, such as fstat().
- Fully describe XAM security techniques (storage pools) and XSET sharing (i.e. user and group permissions that are “better” than listed in POSIX file system API).

- Fully describe a robust set of XAM error codes
- Describe SDK exit behavior, especially the closing of any open handles.
- Design an alternative query API.

## LIST OF REFERENCES

Cincinnati College of Law. "Rule 17-4 – Records to be Preserved by Certain Exchange Members, Brokers, and Dealers". Securities Lawyer's Deskbook. May 2, 2003 //www.law.uc.edu/CCL/34ActRls/rule17a-4p5203.html.

EMC Centera Software Developer's Kit. "Centera_SDK_3[1].0_API_Ref_Guide.pdf". Centera Developer's Portal. May 30, 2005. lighthouse.emc.com/portal/modules.php?name=ContentEditor&mode=show_content&target=45

Health Insurance Portability and Accountability Act (1996) www.legalarchiver.org/hipaa.htm

Open Group Base Specification Issue 6. (Version 3, 2004) www.opengroup.org/onlinepubs/009695399.

SETI@home Project (October 2006) http://setiathome.berkeley.edu

United State Securites and Exchange Committee. "Securites and Exchange Act of 1934". March 25, 2005. www.sec.gov.

Van Riel, Jan. "My World Information Brokerage built on CAS", November 2005. http://www.cascommunity.org/portal/read.php?ca=e7aebe9e1df16bd452cedf9dc887b8c0

XAM Team: "XAM Genesis", Storage Networking Industry Association Data Management Forum (September 20, 2006) www.snia-dmf.org/xam/genesis.shtml

XAM Team: "XAM Frequently Asked Questions", Storage Networking Industry Association Data Management Forum (September 20, 2006) http://www.snia-dmf.org/xam/faq.shtml

XAM Team. "XAM Specification Version 1.2". Storage Networking Industry Association Data Management Forum (September 19, 2005) www.snia-dmf.org/xam.