Information Management – Extensible Access Method (XAM) – Part 1: Architecture

Version 1.0

"Publication of this Working Draft for review and comment has been approved by the FCAS TWG. This draft represents a "best effort" attempt by the FCAS TWG to reach preliminary consensus, and it may be updated, replaced, or made obsolete at any time. This document should not be used as reference material or cited as other than a "work in progress." Suggestions for changes to this document should be directed to the SNIA Technical Council Managing Director at tcmd@snia.org."

WORKING DRAFT

April 2, 2008

Revision History

Version	Date	Originator	Sections	Comments
1.0	4/2/08	M. McMinn	Various	Section 7.2.2, Table 18 (p.45) - Changed the second instance of XSystem.asyncOpenXSet to XSystem.asyncCopyXSet; Section 8.4.1, Table 23 - Added XSystem.isXSetRetained; Foreword - updated doc titles; All - changed status from Internal Use Draft to Working Draft.

Contents

Figures

Tables

Foreword

Parts of this Standard

This standard is subdivided into the following parts:

- Information Management – Extensible Access Method (XAM) – Part 1: Architecture
- Information Management – Extensible Access Method (XAM) – Part 2: C API
- Information Management – Extensible Access Method (XAM) – Part 3: Java API

SNIA Web Site

Current SNIA practice is to make updates and other information available through their web site at http://www.snia.org

SNIA Address

Requests for interpretation, suggestions for improvement and addenda, or defect reports are welcome. They should be sent to the Storage Networking Industry Association, 500 Sansome Street, Suite #504, San Francisco, CA 94111, U.S.A.

Introduction

Purpose and Audience

This document is intended to be used by two broad audiences. The first audience consists of application programmers who wish to use the XAM Application Programmers Interface (API) to create, access, manage, and query reference content through standardized methods that are collectively referred to as XAM (eXtensible Access Method). The second audience consists of those who implement reference content stores that wish to provide access to their stores through the XAM standardized methods.

Organization

The chapter contents of this document are described as follows:

Chapter	Contents
Chapter 1, “Scope”	Defines the subject of the document and the aspects covered.
Chapter 2, “References”	Lists the referenced documents that are indispensable for the application of this document.
Chapter 3, “Terms and Conventions”	Defines the terms and typographical conventions used in the document.
Chapter 4, “Business Overview”	Gives the background of the SNIA XAM and how XAM addresses the industry and market needs.
Chapter 5, “Overview of the XAM Architecture”	Gives an overview of the XAM architecture, including how the XAM Storage System vendors connect their storage systems through the XAM standard APIs.
Chapter 6, “XAM Objects and Common Operations”	Defines the XAM objects as well as the common data structures and operations that can be associated with the XAM object.
Chapter 7, “XAM Library and XSystems”	Specifies the XAM application’s logical view of the XAM Storage System and the software modules that comprise XAM.
Chapter 8, “XSet Operations”	Defines the behavioral and semantic model of an XSet by describing applicable methods on individual XSets and their elements.
Chapter 9, “XSet Management”	Specifies XSet lifecycle management capabilities, including retention, deletion, and hold.
Chapter 10, “Query”	Describes the query language grammar and the two levels of query supported through the XAM API.
Chapter 11, “Security”	Describes three XAM security functions: XAM application authentication, XSystem authorization, and XSet access control.
Annex A, “(normative) XAM Toolkit”	Describes toolkit methods to simplify some common operations within XAM.
Annex B, “(normative) Canonical XSet Interchange Format”	Describes the XSD (XML Schema Definition) for the XML manifest that is used by the XSet canonical format when importing and exporting an XSet.

1 Scope

This part of the XAM standard is a normative specification of the general architecture and semantics of the XAM API. It applies to programmers who are generating XAM applications in any programming language. It also applies to storage system vendors who are creating vendor interface modules (VIMs).

This document uses an object model to describe syntax in examples; these examples are informative only. It is not a normative specification of the syntax of the XAM interfaces in any language binding. The normative specification of the syntax of the C language binding is defined in the XAM C API Specification [XAM-C-API]. The normative specification of the syntax of the Java language binding is defined in the XAM Java API Specification [XAM-JAVA-API].

2 References

2.1 Normative References

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

[CRC] Williams, Ross, “A Painless Guide to CRC Error Detection Algorithms”, Chapter 16, August 1993, http://www.repairfaq.org/filipg/LINK/F_crc_v3.html

[IANA-SASL] “Simple Authentication and Security Layer (SASL) Mechanisms” http://www.iana.org/assignments/sasl-mechanisms

[IEEE754] IEEE Standard 754-1985. IEEE Standard for Binary Floating-Point Arithmetic. IEEE, New York, 1985.

[ISO8601] “Data elements and interchange formats -- Information interchange -- Representation of dates and times”, ISO 8601, http://isotc.iso.org/livelink/livelink/4021199/ISO_8601_2004_E.zip?func=doc.Fetch&nodeid=4021199

[RFC2046] Freed, N, et al., “Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types”, RFC 2046, November 1996.

[RFC2387] Levinson, E. “The MIME Multipart/Related Content-type”, RFC 2387, August 1998.

[RFC3066] Alvestrand, H. “Tags for the Identification of Languages”, RFC 3066, January 2001.

[RFC3454] Hoffman, P, et al., “Preparation of Internationalized Strings ("stringprep")”, RFC 3454, December 2002.

[RFC3491] Hoffman, P, et al., “Nameprep: A Stringprep Profile for Internationalized Domain Names (IDN)”, RFC 3491, March 2003.

[RFC3629] Yergeau, F., “UTF-8, a transformation format of ISO 10646”, RFC 3629, November 2003

[RFC3986] Berners-Lee, T., “Uniform Resource Identifier (URI): Generic Syntax”, RFC 3986, January 2005.

[RFC3987] Duerst, M, et al., “Internationalized Resource Identifiers (IRIs)”, RFC 3987, January 2005.

[RFC4013] Zeilenga, K., “SASLprep: Stringprep Profile for User Names and Passwords”, RFC 4013, February 2005.

[RFC4234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", RFC 4234, October 2005.

[RFC4422] Melnikov, A. Ed, “Simple Authentication and Security Layer (SASL)", RFC 4422, June 2006.

[RFC4616] Zeilenga, K. “The PLAIN Simple Authentication and Security Layer (SASL) Mechanism”, RFC 4616, August 2006.

[UNICODE] The Unicode Consortium, “The Unicode Standard, Version 2.0”, 1996, http://www.unicode.org

[XAM-C-API] “Information Management - Extensible Access Method (XAM) - Part 2: C API”, SNIA draft specification.

[XAM-JAVA-API] “Information Management - Extensible Access Method (XAM) - Part 3: Java API”, SNIA draft specification.

[XOP] “XML-binary Optimized Packaging”, http://www.w3.org/TR/xop10/

2.2 Informative References

[JSR170] David Nuescheler, Content Repository for Java technology API, Java Specification Request (JSR) 170, April 2006. (http://jcp.org/en/jsr/detail?id=170)

[RBAC] “An Introduction to Role Based Access Control,” NIST CSL Bulletin on RBAC (December, 1995) http://csrc.nist.gov/rbac/NIST-ITL-RBAC-bulletin.html or http://csrc.nist.gov/publications/nistbul/csl95-12.txt

[SMI-S] – “Storage Management Initiative Specification,” - ISO/IEC 24775:2007, http://webstore.ansi.org/RecordDetail.aspx?sku=ISO%2fIEC+24775%3a2007

3 Terms and Conventions

3.1 Terms

For the purposes of this document, the following definitions apply.

3.1.1 application field

A XAM field with a name attribute outside the system field namespace of .* (see Term 3.1.20, “system field”).

3.1.2 binding field

An XSet field that affects the value assigned to the XUID when the XUID is created while committing the XSet to persistent storage.

3.1.3 binding modification

A modification that includes adding or deleting binding fields, changing binding field values, changing the binding field type, changing the binding flag (changing it from binding to nonbinding or vice versa), resetting the management properties, or opening a binding XStream in writeonly or appendonly mode.

3.1.4 CRC

Acronym for cyclic redundancy check.

3.1.5 field

A piece of uniquely identifiable data that can be attached to an XSet, an XSystem, or a XAM Library. Two types of fields exist: a property and an XStream.

3.1.6 field attribute

One or more features associated with a field. A field has multiple attributes associated with it, including type (MIME type), binding (TRUE/FALSE), readonly (TRUE/FALSE), and length (length of the value).

3.1.7 method

An abstract interface definition for a specific piece of functionality. When mapped to a specific programming language definition of the method, this may be a function, a procedure, a macro, or an object-oriented method.

3.1.8 MIME

Acronym for Multipurpose Internet Mail Extensions. MIME types are used by XAM to indicate the format of the data contained in an XStream. For additional information, see [RFC2045] and [RFC2046].

3.1.9 nonbinding field

An XSet field that does not affect the value assigned to the XUID when the XUID is created after committing the XSet to persistent storage.

3.1.10 nonbinding modification

A modification that includes all field editing that is not a binding modification, specifically, adding or deleting nonbinding fields, changing nonbinding field values, changing the nonbinding field type, or opening a nonbinding XStream in writeonly or appendonly mode.

3.1.11 OID

Acronym for object identifier.

3.1.12 property

A field whose MIME type attribute is one of the XAM-defined simple types (stypes).

3.1.13 RBAC

Acronym for Role Based Access Control.

3.1.14 reference information

Data that changes infrequently once written. Also commonly referred to as fixed content.

3.1.15 Role Based Access Control

An approach to controlling the system access of authorized users based on the user's role within an organization. For additional discussion, see Chapter 10, “Query”. For more information, see [RBAC].

3.1.16 SASL

Acronym for Simple Authentication and Security Layer.

3.1.17 Simple Authentication and Security Layer

A framework for providing authentication and data security services in connection-oriented protocols and interfaces via replaceable mechanisms. See [RFC4422].

3.1.18 stype

A set of MIME types defined in the XAM specification that are used for field type attributes and are commonly referred to as simple types.

3.1.19 synthetic field

A XAM field that may not be physically represented within the XAM Storage System. Some fields may be derived from other state information within the XAM Storage System. From a XAM application perspective, it does not matter whether a specific XAM field is synthetic or not.

3.1.20 system field

A XAM field with a name attribute in the reserved system field namespace of .* (see Term 3.1.1, “application field”). XAM applications are prevented from creating fields in this portion of the field namespace, though they may be able to change a system field value at the discretion of the XAM Library and VIM implementations.

3.1.21 system properties

A property that is created and managed by the XSystem or the XAM Library.

3.1.22 TLS

Acronym for Transport Layer Security.

3.1.23 VIM

Acronym for Vendor Interface Module.

3.1.24 Vendor Interface Module

A vendor-specific shared library that implements the standard API calls (known as the VIM API) used by the XAM Library to communicate with a specific storage system.

3.1.25 VIM API

The methods that the XAM Library uses to communicate with the VIMs.

3.1.26 XAM

Acronym for eXtensible Access Method.

3.1.27 XAM API

The methods that a XAM application uses to communicate with an XSystem, via the XAM Library.

3.1.28 XAM application

An entity that uses the XAM API to access services provided by an XSystem.

3.1.29 XAM job

A XAM mechanism to submit work to the XSystem. The only XAM job defined is the query job.

3.1.30 XAM Library

A shared library that implements the standardized abstraction layer between the XAM API used by applications and XAM Storage System VIMs (see Section 7.1.1, “Vendor Interface Modules”).

3.1.31 XAM QL

Acronym for the XAM query language.

3.1.32 XAM query

A way to search an XSystem to identify specific content. XAM uses an SQL variant, known as XAM QL, as its query language. Query results consist of a list of XUIDs identifying the XSets that match the query.

3.1.33 XAM session

The communication mechanism and context (e.g., authentication, authorization) used to access a logical container of XSets and the XSystem itself. XSystem instance is a synonym.

3.1.34 XAM Storage System

A storage system that provides XAM-compliant storage services. Typically this system is for data that is not expected to change during its lifetime (e.g., fixed content, reference information, archival data). The contents of a XAM Storage System are exposed to applications via one or more XSystem objects in the XAM API.

3.1.35 XRI

Acronym for XSystem Resource Identifier.

3.1.36 XSet

The primary storage abstraction in XAM. An XSet binds data and metadata into a single entity that is stored and retrieved as a unit. MIME types are used to specify data and metadata formats.

3.1.37 XSet instance

An instantiation of an XSet object. An XSet handle manipulates an XSet instance.

3.1.38 XSet Unique Identifier

A globally unique external reference identifier for a specific XSet. Abbreviated XUID.

3.1.39 XStream

A field that is not a simple type (stype). The XSystem does not type check the value of an XStream. An XStream's MIME type attribute is any defined type except for the XAM-defined stypes.

3.1.40 XStream instance

An instantiation of an XStream object. An XStream handle manipulates an XStream instance.

3.1.41 XSystem

A logical container of XSets that is visible to XAM applications as an abstraction in the XAM API.

3.1.42 XSystem instance

An XSystem (i.e., a logical container of XSets) and the communication mechanism and context (e.g., authentication, authorization) that is used to access the XSystem. XAM session is a synonym. It is also an instantiation of an XSystem object.

3.1.43 XSystem Resource Identifier

An identifier used to specify a target XSystem that can include optional parameters that are useful when connecting to a XAM Storage System. The syntax is similar to an Internationalized Resource Identifier (IRI), with field definitions specific to XAM.

3.1.44 XUID

Acronym for XSet Unique IDentifier.

3.2 Conventions

Typographical conventions used in this document include the following:

Convention	Description
Note:	Contains additional or useful informative text.
CAUTION:	Indicates that you should pay careful attention to the probable action, so that you may avoid system failure or harm.
`Fixed-width text`	Indicates text that you enter at a keyboard or text that is displayed on an output device, such as a screen. This convention is most commonly used for command syntax and examples.
<XAMHandle>	In a method name, indicates the handle to the XAM primary object: XAM Library, XSystem, or XSet.
<op>	In a method name, indicates one of three operations to be performed: get, set, or create.
<stype>	In a method name, indicates one of six stypes: Boolean, Int, Double, XUID, String, or Datetime.
<attribute>	In a method name, indicates one of four attributes: Binding, Length, ReadOnly, or Type.
Italicized text	Indicates a property or field name, i.e., *.xset.xuid*.

Note: The names and syntax of methods described in this document are informative only. The normative descriptions of the actual method names and the syntax of those methods in C and Java can be found in the XAM C and Java API Specifications [XAM-C-API] and [XAM-JAVA-API], respectively.

4 Business Overview

4.1 Background of the SNIA XAM

The amount of reference Information (also known as fixed content) has been growing rapidly each year. At the same time, business demand for timely access to that data, in both the private and public sectors, has been growing. Beyond timely access to this data, businesses need a way to relocate data across diverse hardware platforms, without compromising data integrity.

Current products for storing and managing reference information have significant limits when integrating with other storage products and applications. Vendor-specific data access and data management methods (e.g., for naming, retention, and deletion) are common, requiring application software changes, sometimes extensively, to integrate with each storage product. These integration obstacles also limit the ability to share reference information among applications, and no standards exist for moving reference information across different storage products.

To meet these challenges, the storage industry requires a set of standard interfaces to enable more functional and sophisticated products. These interfaces need to allow multiple vendors to provide different classes of hardware and software products that store, retrieve, and manage reference information reliably and seamlessly.

The Storage Networking Industry Association (SNIA) was formed to ensure that storage networks become efficient, complete, and trusted solutions across the IT community. Comprised of more than 300 members, SNIA is uniquely committed to delivering standards, education, and services that will propel open storage networking solutions to the broader market.

4.2 The XAM Approach

The SNIA XAM (eXtensible Access Method) specification has been developed to address these industry and market needs. XAM will enable products to seamlessly interoperate, allowing customers the flexibility to select among alternate software and hardware vendors when constructing their storage environments. Broad adoption of the standards that are defined in this specification will increase customer satisfaction and accelerate acquisition of new storage technology, expanding the market for reference storage. In addition, a common interface will reduce time to market for new products and solutions from software vendors, hardware vendors, and system integrators.

XAM provides an application programming interface (see [XAM-C-API] and [XAM-JAVA-API]) that allows XAM applications to store data in a fashion that does not depend on the specific storage system. XAM provides the following important functionality to applications and storage systems:

- **Reference information is associated with a globally unique name.** By binding reference information to a unique name, an application can efficiently manage the reference information without concern for the data's actual location. Location independence provides a mechanism for implementing Information Lifecycle Management (ILM) practices within a XAM-based storage system itself.

- **Metadata is raised to the same level of importance as the reference information itself**. By bundling together data and metadata (contextual data about the information being stored), applications can more easily manage and share reference information, which facilitates ILM.

- **Storage systems are accessed via a standard, pluggable architecture**. By standardizing the architecture, customers can add and remove storage products without impacting applications. The XAM architecture is a software framework that allows XAM-enabled applications to interface with XAM-compliant vendor devices in order to store and retrieve reference information in a vendor-independent and location-independent manner.

- **A standard XAM storage provider interface.** XAM Storage System vendors can plug their systems into the XAM API by creating a provider for the Vendor Interface Module API (VIM API). XAM also provides a standardized set of management disciplines and semantics for fixed content, such as retention, expiration, etc.

4.3 Benefits of XAM

Application developers, storage vendors, and storage management professionals all benefit from XAM:

- Application developers benefit because the standard XAM interface no longer requires vendor-specific code for reference information storage. This feature enables application developers to spend less time writing and maintaining vendor-specific code so that they can devote more time to features that add value to their applications. These developers can rely on a standard interface—an interface that is designed and supported by the vendors that are involved in the reference information industry.

- Storage vendors benefit because XAM provides a consistent set of data access and data management capabilities across XAM-compliant storage systems. Therefore, these vendors can plug their hardware into a standard interface that enables application vendors using XAM to transparently use their storage platform. XAM also enables standards-based techniques to move data between diverse storage platforms and information management applications to design solutions based on a common set of features and functions.

- Storage management professionals benefit because they can easily relocate data to accommodate storage system replacement. In addition, they can select best-of-breed application programs and storage systems without being constrained by integration issues.

4.4 Recommendations for Additional Standards

While this specification defines the base interface, additional standards should be developed over time to foster greater levels of functionality and integration. These standards include, but are not limited to:

- Implementation-common metadata to support data classification standards, allowing storage-centric ILM practices that do not depend on the specific application that generated the data.

- Reference information-naming schemas, including standards for embedding structured data into a flat object space. These schemas will facilitate application integration when processing file systems, databases, and other data resources with significant internal structure.

- Interfaces for bulk movement of reference information, such as migrating, backing up, and replicating reference information among XAM-type systems.

- SMI-S [SMI-S] profiles, allowing vendor-specific XAM system implementations to be managed and monitored in a unified manner. These profiles will provide common functions, while allowing the management of the rest of the storage infrastructure.

5 Overview of the XAM Architecture

The XAM architecture is a software framework that allows XAM-enabled applications to interface with XAM-compliant vendor devices. The goal of this architecture is to allow applications to take advantage of the XAM Application Programming Interface (API) to store and retrieve reference information in a vendor-independent and location-independent manner.

A primary requirement of the XAM architecture is the ability to support access to multiple vendors' XAM Storage Systems and multiple versions of the same vendor's XAM Storage System. That is, different versions of the XAM specification must be able to access the same XAM Storage System, or, the same version of the XAM specification must be able to access different versions of a XAM Storage System. This architecture also allows multiple applications to access the same XAM Storage System.

The XAM architecture provides a mechanism for XAM Storage System vendors to create Vendor Interface Modules (VIMs) that act as bridges between the standard XAM APIs and the vendor's storage systems. How the VIMs connect to their respective devices (for example, TCP/IP, SCSI, or a file system) is transparent to the XAM API and the application. The connection is completely encapsulated by the VIM; the applications should be unaware of the VIM's existence and functionality.

5.1 XAM Software Modules

The software modules of the XAM architecture are shown in Figure 1, "XAM Software Modules". XAM standardizes two interfaces between the XAM software modules:

- **XAM API** is to be used by applications that want to communicate through standard interfaces to storage that has been optimized for reference information.
- **VIM API** is to be used by XAM Storage System implementers that want to communicate through standard interfaces to applications that have been optimized for reference information.

The semantics of the XAM API and VIM API are very similar; thus, the XAM Library is intended to be a thin software layer between the application and the VIM. The semantics of the XAM API and VIM API are defined in this specification. The syntax of these APIs, however, are defined in the XAM C API Specification [XAM-C-API] and the XAM Java API Specification [XAM-JAVA-API].

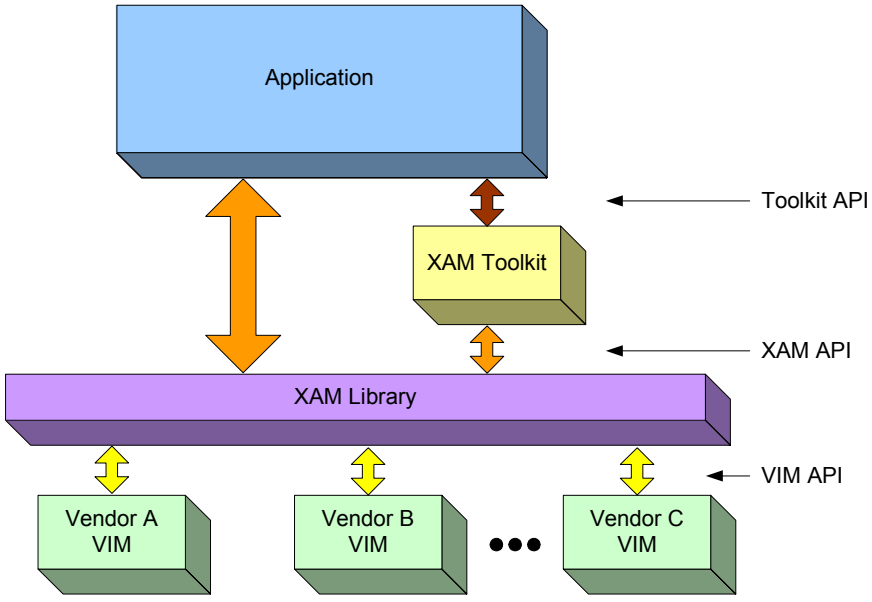

Figure 1 – XAM Software Modules

In Figure 1, the application binds to one of the XAM API language bindings supplied by the XAM Library. XAM standardizes two bindings: a C language binding and a Java language binding. When the application requests access to a specific XSystem, the XAM Library discovers the appropriate VIM to use to dispatch the request to the XSystem. Once a XAM session has been created to connect the application to the XSystem, the XAM Library dispatches additional application requests to the XSystem using the selected VIM. The VIM then communicates with the XAM Storage System (not shown), executes the request, and returns the response to the XAM Library, which in turn sends it to the application. The application can also use convenience interfaces in the XAM Toolkit. See Annex A, “(normative) XAM Toolkit” for more information.

XAM allows the VIM to act as a cache so that it can optimize communication between the VIM and the XAM Storage System. Note that the VIM may be operating in the same context as the application, and thus is potentially subject to malicious attacks. To ensure data security and integrity in the XAM Storage System, the XAM architecture requires all data security and integrity checks to be performed when the application’s data is committed to persistent storage. XAM also strongly recommends that these checks occur at the time the application first modifies the data, so that an application can more directly correlate any issues to the specific operation that caused the issue.

5.2 XAM Object Model

Section 5.1, “XAM Software Modules” defines the three software modules (Toolkit, XAM Library, and VIM) within the XAM architecture. The XAM architecture uses these software modules to create a logical view of the XAM Storage System. This logical view defines a set of objects that are arranged hierarchically, providing a consistent abstraction that is independent of a variety of implementation approaches.

XAM has three primary objects: the XSet, the XSystem, and the XAM Library objects.

- An XSet is the logical unit of data that an application can commit to persistent storage within XAM.
- An XSystem is a logical container of one or more XSets.
- The XAM Library enables an application to discover and communicate with multiple XAM Storage Systems.

The XAM architecture enables a single XAM Storage System to contain one or more XSystems. Additionally, a XAM Storage System may enable an XSystem to span multiple XAM Storage Systems. Regardless of the physical topology underneath the VIM API, the application’s logical model of the XSystem is the same.

When an application tries to access a XAM Storage System, a series of logical XAM object instances are created. When the XAM Library is loaded, a XAM Library object instance is created. When the application connects to an XSystem, an XSystem object instance is created. When the application opens or creates an XSet, an XSet object instance is created. Thus, a strict hierarchical relationship exists between XAM primary objects.

The XAM API also defines two secondary objects, XStreams and XIterators, which can be attached to any of the XAM primary objects. An XStream is a binary data stream, which can potentially be quite large. When attached to an XSet, the XStream’s data is committed to persistent storage with the XSet. An XIterator is a temporary object that cannot be committed to persistent storage. It is used to discover the names of data that are attached to a primary object.

Each XAM object has an associated finite state machine (FSM). The FSM defines normative behavior that is visible to the application. It is not intended to represent all of the states a XAM Storage System vendor might need to implement storage management. The focus of the FSMs is to clearly define the standard behavior that is visible to the application.

5.3 XAM Fields

Data can be attached to any of the XAM primary objects. XAM defines a field as a data-carrying attribute that can be attached to a primary object, of which there are two kinds: properties and XStreams. Properties are used to contain simple kinds of data (strings, integers, etc.), and have a simple set/get style API. XStreams are used to contain larger and potentially more complex data (JPEGs, XML files, or binary data) and are accessed as a stream of data through a read/write style API. Regardless of the object to which the field is attached, the same XAM field-manipulation APIs are used; they are scoped to the appropriate object on which they operate (XAM Library, XSystem, or XSet).

Note that for some language bindings, both properties and XStreams may be treated as objects. However, in the XAM architecture, this is neither a requirement nor a convention. Therefore, the individual language binding must resolve how the properties of the XAM architecture definitions are mapped to the object-oriented view of properties.

Each of the XAM primary objects has a set of properties that provide information about the object from the perspective of the XSystem or XAM Library. This set of properties is referred to as system properties. For the XAM Library and XSystem, system properties are typically related to configuration information. For an XSet, system properties include information such as the time the data was originally stored, the time it was last accessed, and other XSet management properties.

5.4 XAM Persistent Storage: the XSet

An XSet is the addressable unit of storage in the XAM architecture from the application's perspective. For an application to store data in an XSystem, the application must create an empty XSet, populate the XSet fields with its data, and then commit the XSet to persistent storage. If the commit is successful, the application is given a name for the XSet, called a XUID. The application can use the XUID to access the data it stored, exchange the XUID with another application so that it can retrieve the XSet, use it to create application-specific relationships between XUIDs, or use it for other purposes.

When an application creates any XSet field, it must specify whether it wants the field to be binding or nonbinding. A XUID is linked to the binding fields of an XSet. Any change to a binding field, including creating the field, deleting the field, changing the attributes, or changing the value, creates a new XSet with a new XUID, on successful commit of the change. The original XSet shall not be modified in this case. If only nonbinding fields are modified, a new XSet shall not be created on successful commit of the change. This behavior allows applications to store and freely change information that is not associated with the identity of the XSet (the XUID), while also ensuring that the information that is associated with the identity cannot be modified on a specific XSet.

5.5 XSet Management

An XSystem provides both XSet data access and XSet data management. XSet data access methods specify how to create, store, locate, retrieve, update, and delete an XSet contained within an XSystem. These methods are primarily used by data-consuming XAM applications. On the other hand, XSet data management methods specify how an XSystem manages an XSet until it is deleted. These methods are primarily used by data management XAM applications. In most cases, XAM applications will use a combination of both XSet data access and management methods. XAM application developers are strongly encouraged to understand XSet data management, as it differs greatly from data management that is available from familiar data access interfaces, such as file systems.

An XSystem may provide additional capabilities for XSet data management. These capabilities, which may include management of resources, security, migration, virtualization, resiliency, and performance, are outside the scope of XAM.

XSet data management, or XSet management, is further classified into four management disciplines: XSet retention, XSet deletion, XSet storage, and XSet hold.

- **XSet retention.** Retention management uses time criteria to determine the time period(s) during which XSet deletion from the XSystem is prohibited.
- **XSet hold**. Hold management pertains to the capability by which XSystems enforce readonly XSet data access and prohibit XSet deletion.
- **XSet deletion**. Deletion management pertains to the end-of-life or deletion of an XSet in an XSystem.
- **XSet storage.** Storage management pertains to the XSet storage management capabilities, which may be available in an XSystem, but are outside the scope of XAM.

5.6 XAM Security

The security functionality of XAM provides control over which applications can access a XAM Storage System and the types of access and operations that are allowed. A XAM application is authenticated as part of establishing a XAM session with an XSystem; authorization restrictions on the types of allowed accesses and operations are linked to that authentication. Beyond this, specific XSets may be subjected to additional access control restrictions.

XAM Security functionality consists of the following three security disciplines: XAM application authentication, XSystem authorization, and XSet access control.

- **XAM application authentication**. The SASL (Simple Authentication and Security Layer) framework [RFC 4422] allows an application that uses the API to provide authentication to the XSystem when it connects to it. SASL may also establish an authorization identity for authorization and access control purposes.
- **XSystem authorization**. As part of connecting to an XSystem, XAM determines the functional elements of the API that are allowed to be called during the resulting XAM session. If a SASL authorization identity is established, that identity is an input to this determination. If an application tries to call an unauthorized function, that call will return a non-fatal error.
- **XSet access control**. An XSet may have an access control policy applied to it that determines whether an API method is allowed. XSet access permissions can be set to allow only read access or to deny all access to the XSet. A XAM policy name is used to refer to the XSet access control policy.

5.7 XAM Query

The general purpose XAM API is intended to provide a vendor-independent method for storing and retrieving application data. In addition to specific API methods to be used for accessing data, a query-based interface is also included. This interface enables an application to access data using content-based criteria. These criteria are expressed as relationships between XSet properties and, in some cases, content queries of some XSet streams.

The XAM query language is modeled on the database standard query language (SQL), with some important differences:

- All XSets visible within the scope of an XSystem instance are examined for the query.
- XSet property fields contain the relational data to express which XSets are to be returned.

- The return values are always XUID values.
- The XAM architecture specification requires XAM Storage Systems to support relational query of property values; support for content-based query of XStream data is optional.

To submit a XAM query job, an application must first create an XSet, initialize the job command to a query, and store the query string. The application then submits the job. The XSystem processes the query, putting the XUID results into an XStream that is contained in the XSet. The application may store this XSet to make it persistent. On XSystems that support the functionality, an application can also commit a running job to persistent storage, to allow the query to continue, even if the XSystem instance is closed or disconnected.

Committed query job XSets may be accessed by another application, if the application knows the XUID value. These XSets may also be exported to and/or imported from another XAM Storage System, using the same import and export mechanisms that are defined for normal XSets.

5.8 Extending XAM

XAM Storage System vendors can extend XAM in a variety of ways. They can create vendor-specific system fields to advertise or control storage system behavior on the XAM Library, the XSystem or the XSet; they can specify a vendor-specific storage management capability by assigning semantics for XSet storage management; and they can specify additional security roles for XSystem authorization. Vendors can also define new XAM jobs to perform vendor-specific work on the XSystem. A mechanism for vendors to extend the XAM API to support vendor-specific methods is beyond the scope of this specification.

In addition, application vendors may extend XAM by creating their own application-specific XAM fields for XSets. This application-specific metadata is not interpreted by the XSystem implementation, but is preserved as part of the XSet and is available to be queried. The rules for these XAM fields are described in Section 6.3, “XAM Fields”.

6 XAM Objects and Common Operations

The XAM architecture defines a hierarchy of objects to represent information. A XAM object is a data structure that is a package of multiple pieces of data and metadata bundled together for access under a common external name. This chapter provides a normative definition of the XAM objects as well as the common data structures and operations that can be associated with the XAM object.

6.1 XAM Objects

XAM defines three primary XAM objects and two secondary objects, each of which is visible to the XAM application. Primary objects have a strict hierarchical relationship to each other. The three primary objects are the XAM Library object, the XSystem object, and the XSet object. XAM secondary objects can be attached to any primary object. The two types of secondary XAM objects are the XStream object and the XIterator object. Additionally, an asynchronous object is used for asynchronous I/O methods; asynchronous objects are independent of primary and secondary objects. For further information on asynchronous I/O, see Section 8.10, “Asynchronous Operations”.

This specification defines a Finite State Machine (FSM) for each XAM object instance except the XAM Library object. The FSMs specify for each object what operations can occur in each state and what state transitions shall occur as the result of the successful or unsuccessful completion of these operations. The XAM Library does not have an FSM because the XAM Library is expected to be stateless, except for discovery of VIMs and XSystems. Discovery of VIMs and XSystems is an implementation detail of the XAM Library and is thus outside the scope of this specification.

Note that an object is different from an object instance. Specifically for the XSystem object and the XSet object, the object exists independently of a XAM application. An XSet is a logical collection of data and metadata that is identified as a single unit using an XSet Unique Identifier, or XUID. An XSet instance is the XAM application’s current view of the XSet, which may include changes to the XSet that have not been committed to persistent storage. Thus, if two XAM applications are viewing the same XSet and XAM application A modifies data associated with its view of the XSet instance, XAM application B will not see the changes until XAM application A commits the changes. See Section 8.6, “Distributed Access to the Same XSet” for additional information on how XAM supports two applications viewing the same XSet on the same XSystem. Further, after a commit, the XSet instance points to a different XSet if a new XUID is returned as a result of the commit.

Similarly, an instance of an XSystem object is a XAM application’s view of an XSystem. An XSystem is simply a collection of XSets and does not include the communication mechanisms or context (e.g., authentication, authorization) used to communicate with the XSystem. When a XAM application connects to an XSystem object, the XSystem object instantiation includes this additional information. Thus, for example, if a XAM application is not authorized to view all of the XSets in the XSystem, it will see a smaller number of XSets than a different XAM application that is authorized to view all XSets within the XSystem. Note that an instance of the XSystem object is simply called the XSystem instance.

Figure 2, “Contrasting XSystems and XSystem Instances” provides an example set of XSystems, XSystem instances, and XAM applications.

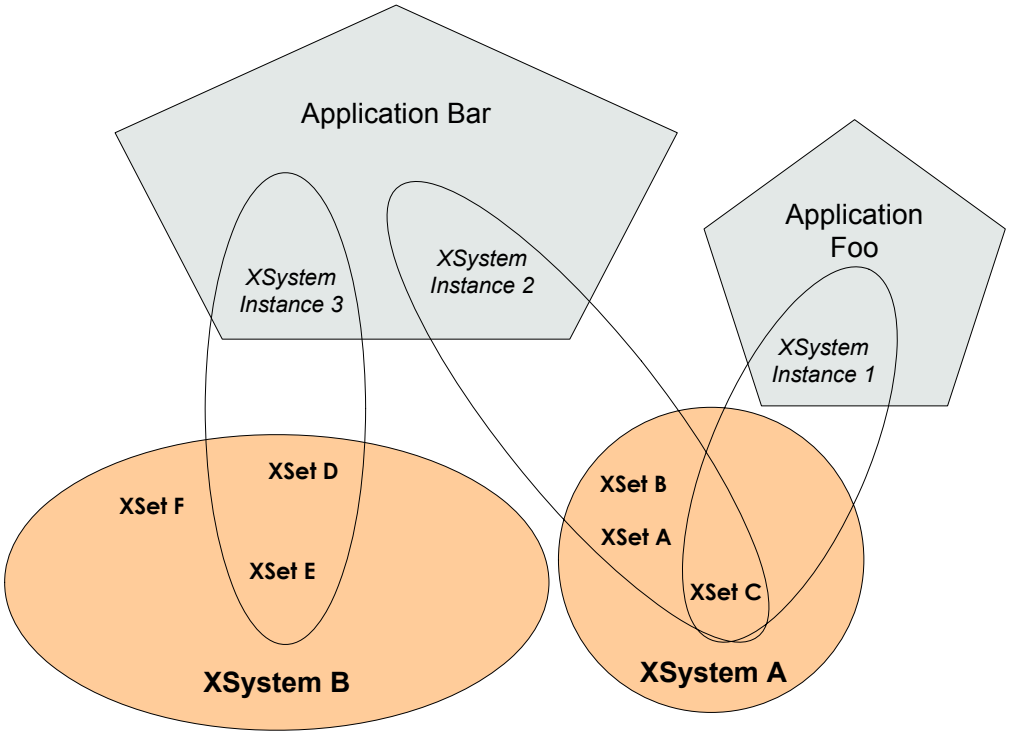

Figure 2 – Contrasting XSystems and XSystem Instances

As shown in this figure, when Application Foo successfully connects to XSystem A, XSystem instance 1 is created. The access rights that were assigned to Application Foo as part of the connection process only allowed it to access XSet C. At the same time, when Application Bar successfully connects to XSystem A, XSystem instance 2 is created. Application Bar has broader access rights and can access XSet A, XSet B, and XSet C. Application Bar also wishes to access XSets on XSystem B. When it successfully connects to XSystem B, a second XSystem instance is created, XSystem instance 3. Application Bar’s access rights are somewhat restricted on XSystem B; it can only access XSet D and XSet E.

The XAM specification allows any of the XAM primary objects to contain fields. Fields contain data that is associated with the particular object (see Section 6.3, “XAM Fields”). A XAM application may change fields on an XSystem instance; however, these changes shall not be persisted in the corresponding XSystem. Thus, if two XAM applications are accessing the same XSystem at the same time (i.e., Application Foo and Application Bar, as shown in Figure 2) and Application Bar modifies a field defined by the XAM specification as an XSystem field, the field shall not change from Application Foo’s perspective, as seen through its XSystem instance 1. A mechanism to persist XSystem instance changes made by a XAM application back to the XSystem is beyond the scope of this specification. Further, a XAM Library or XSystem shall not be required to support addition of fields to the XAM Library instance or XSystem instance, respectively. They shall be required to support the fields as specified in this document (see Chapter 7, “XAM Library and XSystems”.

Note: This specification uses XSystem instance and XAM session interchangeably. XAM session has particular relevance when discussing connecting to the XSystem (see Section 7.2, “XSystem”).

6.1.1 XAM Primary Object Hierarchy

This section defines the normative semantics for creating and releasing XAM primary objects. XAM primary objects shall have a strict hierarchical relationship, as defined in Figure 3, “Primary Object Hierarchy” and Table 1, “Primary Object Hierarchy Transitions”. The methods defined shall be the only mechanisms to create and release primary object instances and associated resources.

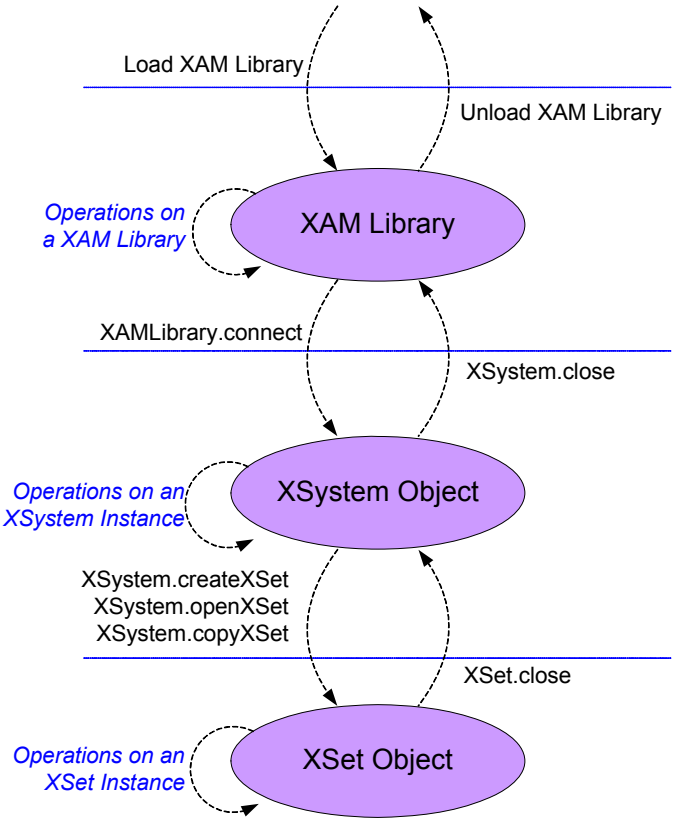

Figure 3 – Primary Object Hierarchy

A XAM Library object is a singleton; that is, a XAM application shall have exactly one instance of the XAM Library. When the XAM Library loads, the XAM Library instance is created. The XSystem instance is created after the XAM application successfully connects to an XSystem. More than one XSystem instance may be attached to the XAM Library. A XAM application can create or open one or more XSet instances within a specific XSystem instance, subject to the access control rights that the XAM application has within the XSystem instance.

Table 1 – Primary Object Hierarchy Transitions

Primary Object Create and Release Methods	Description
Load XAM Library	Success shall instantiate exactly one XAM Library instance. See Section 7.1, “XAM Library” for additional information.
Unload XAM Library	Success shall release the XAM Library instance and all associated resources. See Section 7.1, “XAM Library” for additional information.
XAMLibrary.connect	Success shall instantiate an XSystem instance. See Section 7.2, “XSystem” for additional information.

Table 1 – Primary Object Hierarchy Transitions

Primary Object Create and Release Methods	Description
XSystem.close	Success shall release an XSystem instance and all associated resources. See Section 7.2, “XSystem” for additional information.
XSystem.createXSet XSystem.openXSet XSystem.copyXSet	Success shall instantiate an XSet instance. See Chapter 8, “XSet Operations” for additional information.
XSet.close	Success shall release an XSet instance and all associated resources. See Section 7.2, “XSystem” for additional information.

Note: The mechanism for loading and unloading the XAM Library is platform and language dependent and, therefore, is outside the scope of this specification. The normative definitions of actual method names and syntax can be found in the XAM C API Specification [XAM-C-API] and in the XAM Java API Specification [XAM-JAVA-API].

6.1.2 Primary Object Operations

The normative definitions of the semantics of operations on primary objects are in the following sections:

- For the XAM Library object, see Section 7.1, “XAM Library”.
- For the XSystem object, see Section 7.2, “XSystem”.
- For the XSet object, see Section 8, “XSet Operations”.

6.2 XAM Secondary Objects

The XStream and the XIterator are the only XAM secondary objects. They are secondary objects because they must be attached to a XAM primary object instance. XStreams are a type of field, and XStream instances can be used to store and retrieve data. XIterator instances are used to discover the fields that are attached to a specific primary object.

XStream and XIterator instances shall be created and released as shown in Figure 4, “Secondary Object Hierarchy” and as described in Table 2, “Secondary Object Hierarchy Transitions”.

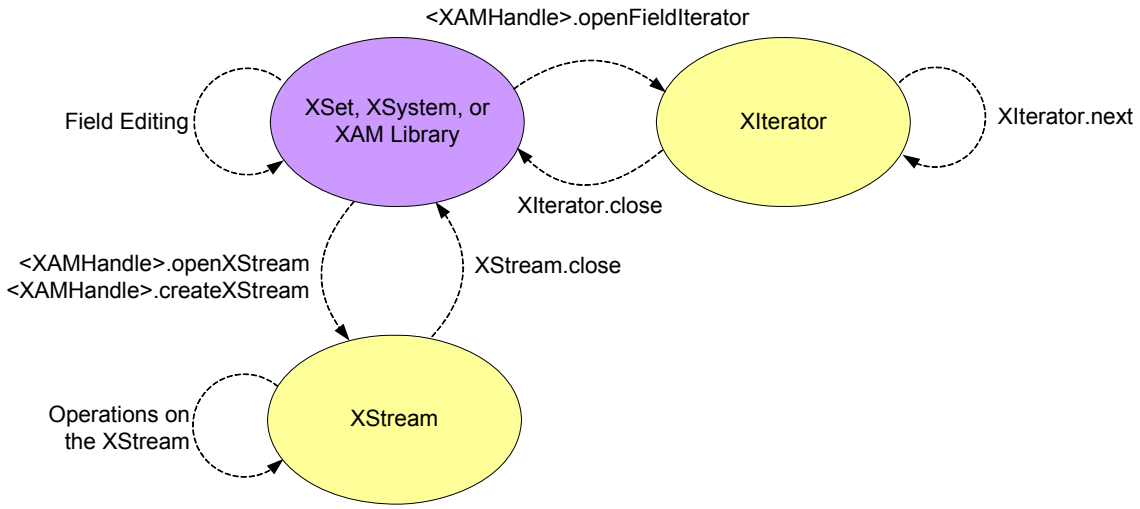

Figure 4 – Secondary Object Hierarchy

In Figure 4 and Table 2, <XAMHandle> is the handle to the XAM primary object. The methods defined shall be the only mechanisms to create and release secondary object instances and associated resources.

Table 2 – Secondary Object Hierarchy Transitions

Secondary Object Create and Release Methods	Description
<XAMHandle>.openXStream	Success shall instantiate an XStream instance.
<XAMHandle>.createXStream	Success shall create an XStream field and instantiate an XStream instance.
XStream.close	Success shall release an XStream instance and all associated resources.
<XAMHandle>.openFieldIterator	Success shall instantiate an XIterator instance.
XIterator.close	Success shall release an XIterator instance and all associated resources.

The XSystem does not interpret the contents of the XStream, except for XSystems that support a Level 2 query, where some XStreams may be processed for later query (see Chapter 10, “Query” for more information on the XAM query facility). Vendor extensions for other types of XStream processing shall be allowed, as long as they are compatible with defined XAM semantics. For a list of methods that interact with XStreams, see Section 6.5, “Operating on Secondary Objects – XStreams and XIterators”.

The XIterator object is used to enumerate the fields contained within a XAM primary object. This object allows the XAM application to specify a prefix for the field name to enable iteration through a subset of the primary object’s fields. For a list of methods that interact with XIterators, see Section 6.5.2, “Operating on XIterators”.

6.3 XAM Fields

Any XAM primary object can have fields contained within it. Each field has a name, attributes that describe how to interact with the object, and a value.

6.3.1 Field Namespace

Each field has a name that is scoped to the XSet. Field names shall be UTF-8 encoded strings (see [RFC3629]) with a maximum length of 512 bytes. Field names shall be case sensitive, so *.xam.org.snia.field* is distinct from *.xam.org.snia.Field*. The field names shall not have embedded NULL characters. To maximize application interoperability, applications are encouraged to use the IDN profile of **stringprep** as defined in [RFC 3491] and [RFC 3454].

XAM divides the field namespace into sub-namespaces, which are statically allocated between the Storage Networking Industry Association (SNIA) standardization, XAM Storage System vendors and XAM application vendors.

To avoid field name conflicts between XAM Storage System vendors and to avoid requiring a central XAM field name registry, XAM takes an approach similar to the Domain Name System (DNS). For XAM Storage System vendor-defined fields, the first portion of the field name shall be an organization's domain name in reverse order, followed by the vendor-defined field name.

Example: *com.example.company.email.fromHeader*
net.example.company.source
org.example.company.standardField

It is strongly recommended that XAM application vendors also use the inverse DNS namespace for application-defined field name definitions.

Table 3, “Static Allocation of Namespace” specifies the static allocation of the namespace, where “*” is used as a wildcard to mean one or more of any valid characters.

Table 3 – Static Allocation of Namespace

Namespace	Description
.xam.*	The XAM Library-owned portion of the namespace. Fields in this namespace shall be defined in this specification and its follow-ons and shall not be extended by XAM Storage System vendors.
.xsystem.*	The XSystem-owned portion of the namespace. Fields in this namespace shall be defined in this specification and its follow-ons and shall not be extended by XAM Storage System vendors.
.xset.*	The XSet-owned portion of the namespace. Fields in this namespace shall be defined in this specification and its follow-ons and shall not be extended by XAM Storage System vendors.
.vnd.<reverseDNS>.*	The XAM System vendor-owned namespace within the XSystem namespace, where <reverseDNS> is the XAM Storage System vendor’s reverse DNS name.
org.snia.*	Reserved for non-XSystem-owned fields. This namespace is owned by the Storage Networking Industry Association (SNIA) and is reserved to enable SNIA standardized fields in the future.
org.snia.xam.*	Reserved for non-XSystem-owned fields and specified in XAM standard documents. This namespace is owned by the SNIA Fixed Content Addressable (FCAS) technical working group.

The ".*" portion of the field namespace is reserved for the XSystem and XAM Library. Fields defined in this portion of the field namespace are known as system fields (see Term 3.1.20, "system field"); fields defined outside this portion of the field namespace are known as application fields (see Term 3.1.1, "application field"). The XSystem shall return a non-fatal error if a XAM application tries to create a new system field on any XAM object instance. The XSystem shall also return a non-fatal error if a XAM application tries to create a field using a name which is an invalid UTF-8 string [RFC3629]. In some situations, a XAM application may be able to change system fields. Values within the ".*" namespace that are not specified above shall be reserved for future use.

6.3.2 Field Attributes

Table 4, "Field Attributes" lists the field attributes and their normative behavior, which shall be present for all XAM fields. Attributes may be set to any valid value except for the binding attribute; the binding attribute shall only be allowed to be set to TRUE for a field contained within an XSet object.

Table 4 – Field Attributes

Attribute Name	Description
Type	The MIME type of the value [RFC 2045, RFC 2046]. The type attribute shall be US-ASCII encoded with a maximum length of 512 bytes.
Binding	A Boolean value indicating if the field is bound to the XUID of the XSet. The behavior associated with the binding attribute is described in more detail in Chapter 8, "XSet Operations".
Readonly	A Boolean value indicating if the field is protected against modification by the standard field operations.
Length	The length of the value, in bytes. The value of length shall be set by the XSystem; it shall not be set by the XAM application.

Type

The type attribute is the MIME type of the value of the data stored in the field. Depending on the MIME type, a field is either a property or an XStream. A property is intended to be used to store a small amount of data that is descriptive of or associated with the XAM object. Conversely, an XStream is intended to store potentially large amounts of data. The XAM API defines get/set style methods to manipulate properties and stream-oriented read/write methods to manipulate XStreams.

The property MIME types defined in this specification are referred to as simple types, or stypes. The XSystem shall type check the stypes and set the actual MIME type based on the specific method that the XAM application uses to create the field. A property is strongly typed to help XAM applications to interoperate between other XAM applications and to enforce consistency checks when using the XAM API.

On the other hand, the XAM application directly sets an XStream's type attribute and shall format the XStream's MIME type as described in [RFC2045]. XSystems shall return a non-fatal error if the type field of an XStream is empty or improperly formatted. The XStream data, or value, is intended to be a MIME document body that matches the MIME type. However, the XSystem does not check the contents of the MIME document body. From the XSystem's perspective, the XStream is simply an unstructured stream of bytes.

Binding

The binding attribute shall only be allowed to be set to TRUE on a field that is within an XSet object. The binding attribute enables the XAM application to specify the set of fields that are used as input when the XSystem generates the XUID for an XSet. If the binding attribute of a field is TRUE when the XSystem generates the XUID, (i.e., when the XSet is committed to persistent storage), any subsequent changes to the field shall cause the XSystem to generate a new XSet with a new XUID, when the changes are successfully committed to persistent storage. Further, if the XAM application changes any field in the set of fields that have the binding attribute set to TRUE (by adding, deleting, or changing), this change shall also cause the XSystem to generate a new XSet (and associated XUID), when the changes are successfully committed to persistent storage.

Readonly

Only the XSystem shall set the readonly attribute, which shall be FALSE, unless otherwise specified. XAM applications shall not be able to change it. If the readonly attribute of a field is TRUE, then the XAM application shall not be allowed to change the field using the standard field operations of set, create, write, and delete. If the XAM application tries to change the field using any of these operations, the XSystem shall return a non-fatal error. If the readonly attribute of an XStream is TRUE, and the application tries to open the XStream in any mode other than readonly, the XSystem shall return a non-fatal error. If the readonly attribute of a field is FALSE, then no error occurs. For more information, see Section 6.4.1, “Operating on Properties”, Section 6.4.3, “Deleting Fields” and Section 6.5.1, “Operating on XStreams”.

The XSystem uses the readonly attribute for fields that only it can change or for preventing changes to XAM job input parameters while the job is running (see Section 8.9, “XAM Jobs and XAM Job Control” for more details). However, when setting the readonly attribute, the XSystem shall not be restricted to these cases. XAM defines specific methods to change some system fields whose readonly attribute is TRUE (see Section 9.2.1, “XSet Retention” for examples).

Length

The XSystem shall derive the value of the length attribute based on XAM application behavior; the XAM application shall not set it. For property fields, the value of the length attribute depends on the stype and shall be as specified as in Table 5, “stypes”. For XStream fields, the XSystem shall derive the value of the length attribute from the number of bytes that are successfully written to it.

6.3.3 Properties

If the field is a property, it shall be one of the stypes and behave as specified in Table 5, “stypes”.

Table 5 – stypes

stype Name	MIME Type string	Description	Value of the Length Attribute
xam_boolean	application/vnd.snia.xam.boolean	Shall be either TRUE or FALSE.	1
xam_int	application/vnd.snia.xam.int	Shall be a signed twos complement 64-bit integer.	8
xam_double	application/vnd.snia.xam.double	Shall be an [IEEE754] double-precision floating point value.	8

Table 5 – stypes

stype Name	MIME Type string	Description	Value of the Length Attribute
xuid	application/vnd.snia.xam.xuid	Shall be a XUID and use the XUID format as defined in Section 6.3.4, “XUID Format”. Note that a XUID field’s value is not required to refer to an XSet currently in existence.	Actual length, in bytes. Between 9 and 80 bytes, inclusive
xam_string	application/vnd.snia.xam.string	Shall be a UTF-8 encoded Unicode string, with a maximum length of 512 bytes.	Actual length, in bytes
xam_datetime	application/vnd.snia.xam.datetime	Shall be a timestamp identifier, as specified by [ISO8601], as profiled below, using UTF-8 encoding.	Actual length, in bytes

Note: Applications should avoid using MIME types that begin with “application/vnd.snia.xam”. This namespace is reserved for future use.

When a field value is stored, the XSystem performs the consistency checks that are defined in Section 6.3.5, “Field Consistency Checks Performed by the XSystem”. If it is not a MIME type, as defined in Table 5, then the XSystem shall treat it as an XStream. The xam_datetime stype format shall be a proper subset of the [ISO8601] specification and is intended to be a compatible subset of [JSR170], Section 6.2.5.1. Specifically:

- Four-digit years shall be used.
- Week dates or ordinal dates shall not be used.
- Midnight shall not be represented with 24:00.
- Time zone designators may be used.
- Duration or interval formats shall not be used.
- All time shall be no finer than millisecond resolution.

6.3.4 XUID Format

The XSystem shall create the XUID, which identifies an XSet. The XUID shall be globally unique and shall conform to the format defined in Figure 5, “XUID Format”. The native format of a XUID is a variable-length byte sequence and shall be a maximum length of 80 bytes. A XAM application should treat XUIDs as opaque byte strings. However, the XUID format is defined such that its integrity can be validated, and XAM Storage System vendors can assign unique XUID values independently.

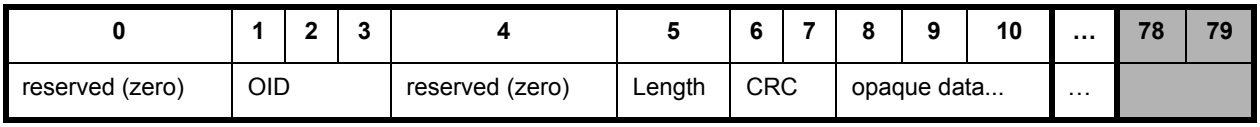

Figure 5 – XUID Format

As shown in Figure 5,

- The reserved bytes shall be set to zero.
- The OID (object identifier) field is the SNMP enterprise number of the XAM Storage System vendor that created the XUID, in network byte order. See [RFC2578] and http://www.iana.org/assignments/enterprise-numbers. 0 is a reserved value.
- The 5th byte is reserved and shall be set to zero.
- The 6th byte contains the full length of the XUID, in bytes.
- The CRC field allows the XUID to be validated for integrity. The CRC field shall be generated by running the algorithm [CRC] across all bytes of the XUID, as defined by the length field, with the CRC field set to zero. The CRC function shall have the following parameters:

  Name : "CRC-16"
  Width : 16
  Poly : 0x8005
  Init : 0x0000
  RefIn : True
  RefOut : True
  XorOut : 0x0000
  Check : 0xBB3D

  This function defines a 16-bit CRC with polynomial 0x8005, reflected input and reflected output. This CRC-16 is specified in [CRC].
- The native format for a XUID is binary. When necessary, XUID textual representation should be base64-encoded, as described in Section 6.8 of [RFC2045], which uses US-ASCII.

6.3.5 Field Consistency Checks Performed by the XSystem

An XSystem shall perform the following consistency checks when a XAM application creates a field. If the check fails, the method shall return with a non-fatal error and leave the field unmodified.

- Field name shall not begin with “.”
- Field name shall be a valid xam_string value (see Table 5, “stypes”).

The XSystem shall perform the following consistency checks when a XAM application creates or modifies a property. If a check fails, the method shall return with a non-fatal error and shall leave the XSet in the same state it was in before the method was called.

- xam_boolean shall be either TRUE or FALSE.
- xam_int shall be 8 bytes.
- xam_double shall be 8 bytes.

- xam_string:
  - Shall be <= XAM_MAX_STRING bytes. See [XAM-C-API] and [XAM-JAVA-API].
  - Shall be a correctly formatted UTF-8 string [RFC3629].
  - Shall not have NULL values embedded within the string.

  For further restrictions on a per language binding, see XAM C API Specification [XAM-C-API] and XAM Java API Specification [XAM-JAVA-API].
- xam_datetime shall satisfy all of the xam_string checks. In addition, xam_datetime:
  - Shall use four-digit years.
  - Shall not use week or ordinal dates.
  - Shall not represent midnight as 24:00.
  - Shall not use a duration or interval format.
  - Shall have a resolution no finer than one millisecond.
- The XUID:
  - Shall have a correct CRC.
  - Shall have a length less than or equal to 80 bytes.
  - Shall not be required to exist on the XSystem.

  The XSystem shall not validate that the reserved fields are set to zero, which enables future use of the reserved fields.
- The type attribute (MIME type):
  - Shall be a US-ASCII string with a length less than or equal to 512 bytes
  - Shall consist of two tokens (type and subtype) separated by a "/" without white space. Each token:
    - Shall contain at least one character.
    - Shall not contain the SPACE character or any control character.
    - Shall not contain any of the following special characters: "(", ")", "<", ">", "@", <,>, ";", ":", "\", <">, "/", "[", "]", "?" and "=". Note that <.> is allowed, and “/” shall only be used to separate the two tokens.

  The XSystem shall not check that the type attribute is a valid MIME type.

When a XAM application modifies the value of an XStream, the XAM application shall not perform any validation on the value being stored and shall allow the type of the XStream to be set to any value.

6.4 Methods that Operate on Fields

This section provides the normative behavior of the methods that can be used to discover what fields are attached to a XAM primary object and to control field attributes and properties. For fields that are XStreams, the methods are addressed in Section 6.5, “Operating on Secondary Objects – XStreams and

XIterators". See the XAM C API Specification [XAM-C-API] and the XAM Java API Specification [XAM-JAVA-API] for additional information on procedural and object oriented bindings of the XAM API.

Note: The XAM API only specifies interfaces between the XAM application and the XAM Library and XSystem; any other interfaces used by the XAM application are beyond the scope of this specification.

6.4.1 Operating on Properties

XAM properties can be operated on using methods that require knowledge of the property's type. In Table 6, "Property Methods", <XAMHandle> is the handle to the XAM primary object and <op> is the operation to be performed, which is one of create, set, or get.

Table 6 – Property Methods

Property Methods	Description
<XAMHandle>.<op>Boolean	Operate on a xam_boolean property
<XAMHandle>.<op>Int	Operate on a xam_int property
<XAMHandle>.<op>Double	Operate on a xam_double property
<XAMHandle>.<op>XUID	Operate on a xuid property
<XAMHandle>.<op>String	Operate on a xam_string property
<XAMHandle>.<op>DateTime	Operate on a xam_datetime property

These operations are semantically defined as:

- create - creates a property field with the specified field name, value, and binding attribute
- set - replaces a property field's value with the specified value without changing any other attributes of the field
- get - returns the field value for the specified property field

Note: The normative definitions of actual method names and syntax can be found in the XAM C API Specification [XAM-C-API] and in the XAM Java API Specification [XAM-JAVA-API].

6.4.2 Determining Field Existence

The method shown in Table 7 is used to confirm or deny whether a field exists within a XAM primary object. <XAMHandle> is the handle to the XAM primary object.

Table 7 – Field Existence Query Method

Field Methods	Description
<XAMHandle>.containsField	Shall return a value of TRUE when the specified field exists within the XAM object and a value of FALSE when the field does not exist within the XAM object

Note: The normative definition of the actual method name and syntax can be found in the XAM C API Specification [XAM-C-API] and in the XAM Java API Specification [XAM-JAVA-API].

6.4.3 Deleting Fields

The method shown in Table 8 is used to delete a field. <XAMHandle> is the handle to the XAM primary object.

Table 8 – Field Deletion Method

Field Methods	Description
<XAMHandle>.deleteField	Delete a field from the XAM object.

Note: The normative definition of the actual method name and syntax can be found in the XAM C API Specification [XAM-C-API] and in the XAM Java API Specification [XAM-JAVA-API].

6.4.4 Operating on Field Attributes

The methods shown in Table 9 are used to set and retrieve field attributes. <XAMHandle> is the handle to the XAM primary object.

Table 9 – Field Attribute Methods

Field Methods	Description
<XAMHandle>.setFieldAsBinding	Set the specified field binding attribute to TRUE.
<XAMHandle>.setFieldAsNonbinding	Set the specified field binding attribute to FALSE.
<XAMHandle>.getFieldType	Retrieve the MIME type value for the specified field.
<XAMHandle>.getFieldLength	Retrieve the field length value for the specified field.
<XAMHandle>.getFieldBinding	Retrieve the binding attribute value for the specified field.
<XAMHandle>.getFieldReadOnly	Retrieve the readonly attribute value for the specified field.

Note: The normative definitions of actual method names and syntax can be found in the XAM C API Specification [XAM-C-API] and in the XAM Java API Specification [XAM-JAVA-API].

6.5 Operating on Secondary Objects – XStreams and XIterators

This section defines the methods that applications use to operate on XStream and XIterator instances. XStreams and XIterators are secondary XAM objects, and as such, shall only be instantiated via factory methods in primary objects. The primary object becomes the parent of the secondary object. Primary objects that have secondary object instances open against them cannot be closed. If a XAM application tries to close such an object, the XSystem shall generate a non-fatal error and the primary object shall not change state.

6.5.1 Operating on XStreams

XAM operations on XStreams (see Table 10, “XStream Methods”) require an XStream instance. The XAM application shall use <XAMHandle>.createXStream or <XAMHandle>.openXStream to create an XStream instance on which to operate. During XStream creation, the XAM application specifies the XStream field

name and attributes and is returned an XStream instance in writeonly mode. If the XAM application opens an existing XStream, it must specify whether it wants to open it in readonly mode, writeonly mode, or appendonly mode. Once the XStream instance is created, the XAM application uses conventional read and write semantics to operate on the XStream. The act of opening an XStream in either writeonly or appendonly mode shall be treated as a change to the XStream, even if no change actually occurred.

- **Open in readonly mode**: Opening the XStream in readonly mode shall initialize the current byte offset to zero. XStream.seek shall be supported to allow reading at arbitrary byte offsets within the XStream value. XStream.seek shall set the current byte offset to the specified byte offset. XStream.seek shall return a non-fatal error if the computed destination byte offset lies outside the boundaries of the XStream, and the current byte offset shall not change. XStream.read shall return the actual number of bytes read, along with the sequential XStream bytes, starting at the current byte offset. When read completes, XStream.read shall increment the current byte offset by the number of bytes actually read. XStream.write shall return a non-fatal error, and the current byte offset shall not change. XStream.tell shall return the current byte offset, and the current byte offset shall not change.

- **Open in writeonly mode**: Opening the XStream in writeonly mode shall initialize the current byte offset to zero, delete the XStream value, and set the XStream length to zero. XStream.write shall return the actual number of bytes written into the XStream value, starting at the current byte offset. When write completes, XStream.write shall increment the XStream length and the current byte offset by the number of bytes actually written. XStream.read and XStream.seek shall return a non-fatal error, and the current byte offset shall not change. XStream.tell shall return the current byte offset, and the current byte offset shall not change.

- **Open in appendonly mode**: The behavior is the same as writeonly mode except for the initialization. Opening the XStream in appendonly mode shall initialize the current byte offset to the XStream length; it shall not change the XStream value or the current XStream length.

Once operations on the XStream are complete, the XAM application is expected to close the XStream, which causes all resources associated with the XStream instance to be freed. Note that any changes made to an XStream are only associated with the instance of the parent object (i.e., XAM Library, XSystem, or XSet). If the XAM application closes the parent object, all changes will be lost. For an XSet, the XAM application can persist the changes to an XStream by using XSet.commit. See Chapter 8, “XSet Operations” for more information on this operation.

Table 10 – XStream Methods

XStream Methods	Description
<XAMHandle>.createXStream	Create an XStream.
<XAMHandle>.openXStream	Open an existing XStream.
XStream.read	Read from the current byte offset position.
XStream.write	Write to the current byte offset position.
XStream.asyncRead	Initiate asynchronous read operation from the current byte offset position.
XStream.asyncWrite	Initiate asynchronous write operation to the current byte offset position.
XStream.seek	Move the current byte offset position.
XStream.tell	Retrieve the current byte offset position.

Table 10 – XStream Methods

XStream Methods	Description
XStream.abandon	Allows the XStream instance to be closed in all cases.
XStream.close	Close the XStream.

Note: The normative definitions of actual method names and syntax can be found in the XAM C API Specification [XAM-C-API] and in the XAM Java API Specification [XAM-JAVA-API].

6.5.2 Operating on XIterators

XAM provides a mechanism called an XIterator to discover the fields attached to a XAM object. This interface was created because XSets, XSystems, and XAM Libraries can all have a variable number of fields that can be assigned almost any name, as specified by the XAM application, XSystem, or XAM Library that creates the field.

Table 11 – XIterator Methods

XIterator Methods	Description
<XAMHandle>.openFieldIterator	An XIterator is used to enumerate the field names of the fields on an XSet, XSystem, or XAM object. It shall set the cursor to the beginning of the field list, which allows applications to specify a field name prefix. If such a prefix is specified, only those fields whose name begins with the prefix shall appear in the iteration.
XIterator.hasNext	Shall return a value of TRUE, when there are more fields remaining in the iteration, and a value of FALSE, when there are no fields remaining in the iteration.
XIterator.next	Shall retrieve the next field name in the iteration.
XIterator.close	Shall release all resources associated with the XIterator.

Note: The normative definitions of actual method names and syntax can be found in the XAM C API Specification [XAM-C-API] and in the XAM Java API Specification [XAM-JAVA-API].

This specification defines a number of system fields using a naming convention that takes advantage of the field name argument for the <XAMHandle>.openFieldIterator method to form a collection of related fields (for an example, see the *.xsystem.auth.SASLmechanism.list.<mechanism>* definition in Section 7.2.3, “XSystem Fields”).

Example: To iterate over the fields composing the *.xsystem.auth.SASLmechanism.list.<mechanism>* collection on a particular XSystem instance, a XAM application would call XSystem.openFieldIterator using a field name prefix of *.xsystem.auth.SASLmechanism.list.* Subsequent calls of XIterator.next would return the field names for all of the SASL authentication mechanisms supported by this XSystem instance. The resulting collection of fields could include:

.xsystem.auth.SASLmechanism.list.ANONYMOUS
.xsystem.auth.SASLmechanism.list.PLAIN
.xsystem.auth.SASLmechanism.list.CRAM-MD5

6.6 FSMs for Secondary Objects – XStreams and XIterators

This section defines the normative FSMs associated with the XStream and XIterator instance. The FSM tables (Table 12, Table 13, and Table 14) define the normative transitions for the respective FSM. The associated figures (Figure 6, Figure 7, and Figure 8) show the respective FSM and associated state transitions. To interpret the transition labels in the tables and figures, first consider each transition to be the format Method(status). The transitions occur as follows:

1. The application initiates the method.
2. The object instance responds with a status (typically via a return code on the called method) and a transition in the state machine.

   Note: Status ‘ok’ is considered a non-fatal (recoverable) error return; status ’fatal’ is considered a fatal (non-recoverable) error return code. For a definition of which method return codes are considered fatal and non-fatal, please see [XAM-C-API] and [XAM-JAVA-API].

6.6.1 XStream FSM

This section defines the FSMs for the general XStream instance. There are two FSMs—one for opening an XStream in readonly mode and one for opening the XStream in writeonly or appendonly mode. The XStream has two additional specialized FSMs, one for importing an XSet and one for exporting an XSet. For those FSMs, see Section 8.8, “XSet Import and Export”.

XSet.abandon or XSystem.abandon applied to any of the XStream’s parent objects shall force an exit from the XStream FSM (reader or writer) from any state within the FSM, and all associated resources shall be returned to the operating environment.

Note: The XStream instance FSM in no way restricts other methods from being called on the parent object, i.e., the XAM Library, XSystem, or XSet, except as noted.

6.6.1.1 XStream Instance FSM - Reader

The XSystem shall enter the XStream instance reader FSM when a XAM application calls <XAMHandle>.openXStream in readonly mode. The FSM and the state transitions are shown in Figure 6, “XStream Instance - Reader FSM”.

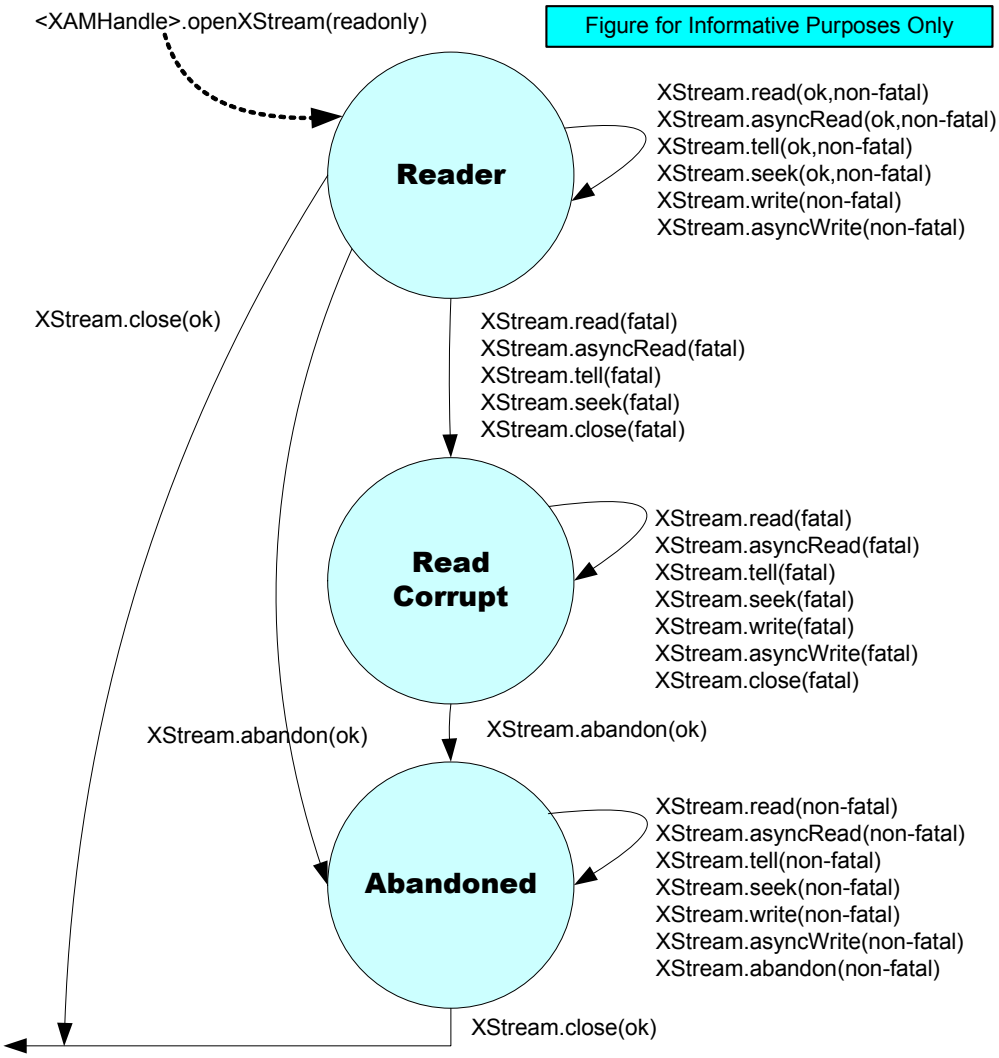

Figure 6 – XStream Instance - Reader FSM

The XStream instance reader FSM shall have the defined inputs, outputs, and transitions as shown in Table 12, “XStream Instance - Reader FSM Transitions”.

Table 12 – XStream Instance - Reader FSM Transitions

State	Transition	To	Comment
NULL	<XAMHandle>.openXStream in readonly mode	Reader	The reader FSM shall be instantiated when the method returns successfully.
Reader	XStream.read(ok) XStream.asyncRead(ok) XStream.tell(ok) XStream.seek(ok)	Reader	The requested operation shall have completed successfully.
Reader	XStream.read(non-fatal) XStream.asyncRead(non-fatal) XStream.tell(non-fatal) XStream.seek(non-fatal)	Reader	The requested operation shall have failed, and the XStream state shall stay the same.
Reader	XStream.close(ok)	NULL	Shall free the XStream instance and release all resources associated with it.
Reader	XStream.close(fatal)	Read Corrupt	If any XAsync instance children have not been closed, XStream.close shall return a fatal error and the reader FSM shall transition to the read corrupt state. Additionally, the XStream may return a fatal error for other reasons.
Reader	XStream.read(fatal) XStream.asyncRead(fatal) XStream.tell(fatal) XStream.seek(fatal)	Read Corrupt	All fatal errors returned from XStream operations in the reader state shall transition the FSM to the read corrupt state.
Reader	XStream.write XStream.asyncWrite	Reader	Shall perform no action and shall always return a non-fatal error.
Reader	XStream.abandon	Abandoned	Shall transition the XStream instance reader FSM to the abandoned state and shall always return success. When it enters the abandoned state, resources associated with the XStream instance and any XAsync child instances may or may not be freed.
Read Corrupt	XStream.read XStream.asyncRead XStream.tell XStream.seek XStream.write XStream.asyncWrite	Read Corrupt	Shall perform no action and shall always return a fatal error.
Read Corrupt	XStream.abandon	Abandoned	Shall transition the XStream instance reader FSM to the abandoned state and shall always return success. When it enters the abandoned state, resources associated with the XStream instance and any XAsync child instances may or may not be freed.

Table 12 – XStream Instance - Reader FSM Transitions

State	Transition	To	Comment
Abandoned	XStream.read XStream.asyncRead XStream.tell XStream.seek XStream.write XStream.asyncWrite XStream.abandon	Abandoned	Shall perform no action and shall always return a non-fatal error.
Abandoned	XStream.close	NULL	Shall always return success, shall free the XStream instance, and shall release all remaining resources associated with the XStream instance and any XAsync child instances.

Note: The normative definitions of actual method names and syntax can be found in the XAM C API Specification [XAM-C-API] and in the XAM Java API Specification [XAM-JAVA-API].

To exit the XStream instance reader FSM, the XAM application shall call XStream.close. This call shall cause the FSM to be uninstantiated and the associated resources to be returned to the operating environment. If XStream.close completed successfully, no error occurred in closing the XStream instance.

If the XAM applications opens the XStream in readonly mode, XStream.close shall always return either success or a fatal error; non-fatal errors are not allowed. If a fatal error is returned, XStream.close causes a transition to the read corrupt state and may cause the parent object of the XStream to eventually transition to the corrupt state (i.e., the XSet, the XSystem, or the XAM Library). This transition could occur, for example, if connectivity was lost to the XAM System. XStream.close might return a fatal error, and the parent object (i.e., the XSet, XSystem, or XAM Library) would also transition to the corrupt state when XIterator.next is called on that object.

In the read corrupt state, all XStream methods shall return a fatal error, except XStream.abandon, which shall return success and transition to the abandoned state. When this transition occurs, the XSystem can either immediately release the resources associated with the XStream instance and any XAsync child instances, or it shall release the resources when it exits from the XStream instance reader FSM.

6.6.1.2 XStream Instance FSM - Writer

The XSystem shall enter the XStream instance writer FSM when a XAM application calls <XAMHandle>.createXStream or when it calls <XAMHandle>.openXStream in writeonly or appendonly mode.

The FSM and the state transitions are shown in Figure 7, “XStream Instance - Writer FSM”.

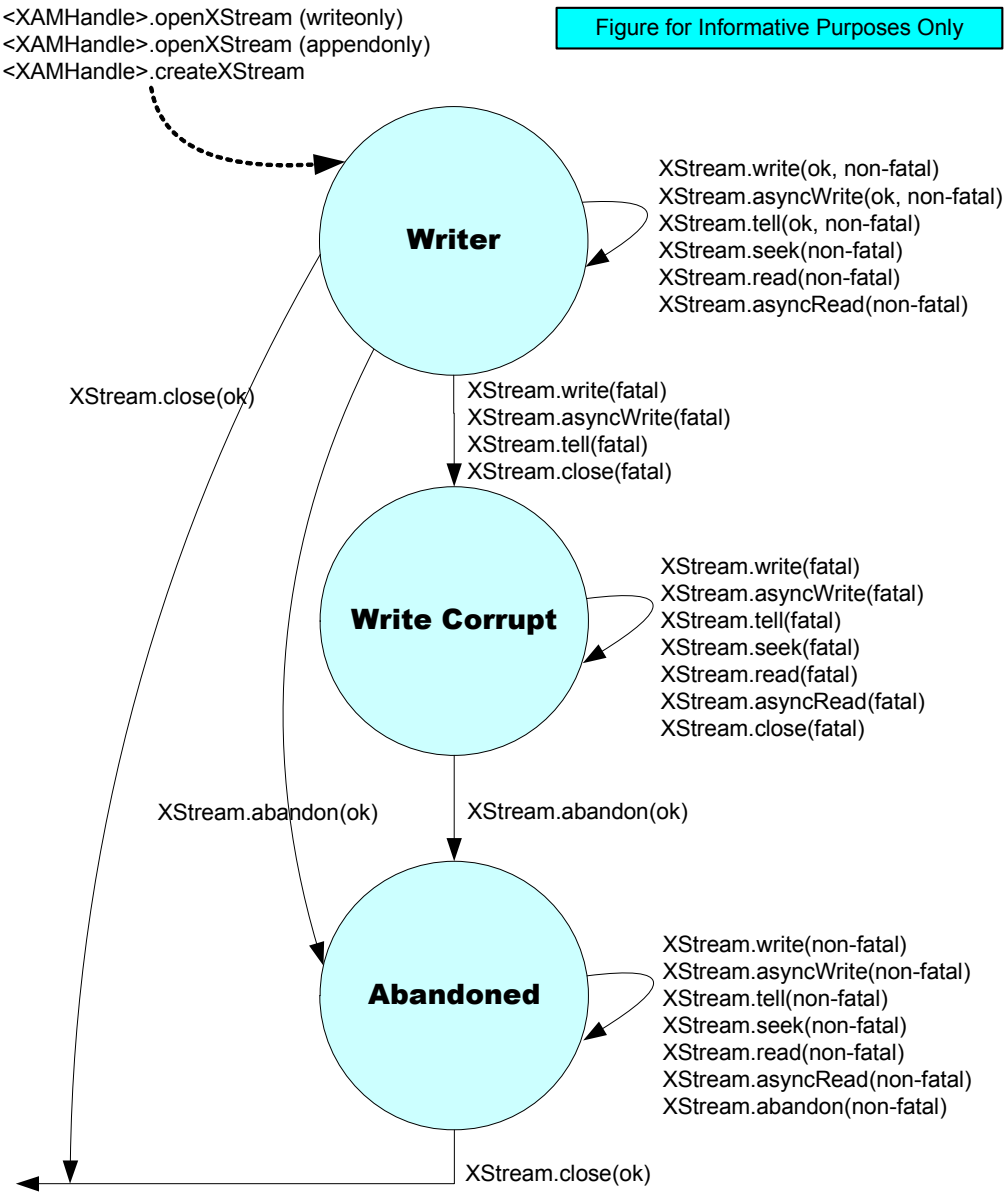

Figure 7 – XStream Instance - Writer FSM

The XStream Writer FSM shall have the defined inputs, outputs, and transitions as shown in Table 13, “XStream Instance - Writer FSM Transitions”.

Table 13 – XStream Instance - Writer FSM Transitions

State	Transition	To	Comment
NULL	<XAMHandle>.openXStream in writeonly or appendonly mode <XAMHandle>.createXStream	Writer	The Writer FSM shall be instantiated when the method returns successfully.
Writer	XStream.write(ok, non-fatal) XStream.asyncWrite(ok, non-fatal) XStream.tell(ok, non-fatal)	Writer	For completion status of either success or non-fatal error, the Writer FSM shall stay in the Writer state.
Writer	XStream.close(ok)	NULL	Shall free the XStream instance and release all resources associated with it.
Writer	XStream.close(fatal)	Write Corrupt	If any XAsync instance children have not been closed, XStream.close shall return a fatal error and the Writer FSM shall transition to the Write Corrupt state.
Writer	XStream.write(fatal) XStream.asyncWrite(fatal) XStream.tell(fatal)	Write Corrupt	A fatal error occurred while accessing the XStream.
Writer	XStream.read XStream.asyncRead XStream.seek	Writer	Shall perform no action and shall always return a non-fatal error.
Writer	XStream.abandon	Abandoned	Shall transition the XStream instance writer FSM to the abandoned state and shall always return success. When it enters the abandoned state, resources associated with the XStream instance and any XAsync child instances may or may not be freed.
Write Corrupt	XStream.write XStream.asyncWrite XStream.tell XStream.read XStream.asyncRead XStream.seek XStream.close	Write Corrupt	Shall perform no action and shall always return a fatal error.
Write Corrupt	XStream.abandon	Abandoned	Shall transition the XStream instance reader FSM to the abandoned state and shall always return success. When it enters the abandoned state, resources associated with the XStream instance and any XAsync child instances may or may not be freed.

Table 13 – XStream Instance - Writer FSM Transitions

State	Transition	To	Comment
Abandoned	XStream.write XStream.asyncWrite XStream.tell XStream.read XStream.asyncRead XStream.seek XStream.abandon	Abandoned	Shall perform no action and shall always return a non-fatal error.
Abandoned	XStream.close	NULL	Shall always return success, shall free the XStream instance, and shall release all remaining resources associated with the XStream instance and any XAsync child instances.

Note: The normative definitions of actual method names and syntax can be found in the XAM C API Specification [XAM-C-API] and in the XAM Java API Specification [XAM-JAVA-API].

To exit the XStream instance writer FSM, the XAM application shall call XStream.close. This call shall cause the FSM to be uninstantiated and the associated resources to be returned to the operating environment. If XStream.close completed successfully, no error occurred in closing the XStream instance.

Note: The XAM specification makes no guarantees as to how much of the data that was written to the XStream has been moved to persistent storage. Only XSet.commit provides guarantees of data having reached persistent storage. The XAM specification provides no persistence guarantees for XStream data that is the child of the XSystem or the XAM Library, as these persistence guarantees are outside the scope of this specification.

If XStream.close returns a fatal error (see [XAM-C-API] and [XAM-JAVA-API] for what constitutes a fatal error), the XSet state for the XStream shall be reset to the value it contained when the last commit occurred, unless the parent object of the XStream also transitions to the corrupt state when XIterator.next is called on the parent object. If the XStream field did not exist at the last commit, the field value shall be set to the empty stream (i.e., the field is not deleted, but all bytes written into the XStream value are lost), unless the parent object of the XStream also transitions to the corrupt state when XIterator.next is called on the parent object. See Section 8.5, “XSet Instance Finite State Machine (FSM)” and Section 7.3.1, “Authentication State Machine” for additional information regarding parent object error states.

Note: Fatal errors, such as connectivity loss to the XAM System, may affect both the XStream FSM and the parent object’s FSM. In this situation, the XStream state from the prior commit operation may be unrecoverable after a fatal error on XStream.close.

In the write corrupt state, all XStream methods shall return a fatal error, except XStream.abandon, which shall return success and transition to the abandoned state. When this transition occurs, the XSystem can either immediately release the resources associated with the XStream instance and any XAsync child instances, or it shall release the resources when it exits from the XStream instance reader FSM.

6.6.2 XIterator FSM

This section defines the FSM for the XIterator instance. XSet.abandon or XSystem.abandon applied to any of the XIterator’s parent objects shall force an exit from the XIterator FSM, and all associated resources shall be returned to the operating environment.

No abandon method for the XIterator exists, and thus no abandoned state. Thus, managing the XIterator object is simpler. Regardless of the completion status of XIterator.close, all resources associated with the XIterator instance are released; thus, no explicit abandon operation is required.

The XIterator instance FSM in no way restricts other methods from being called on the parent object, i.e., the XAM Library, XSystem, or XSet, except as noted.

The XSystem shall enter the XIterator instance FSM when a XAM application calls <XAMHandle>.openFieldIterator. The FSM and the state transitions are shown in Figure 8, “XIterator Instance FSM”.

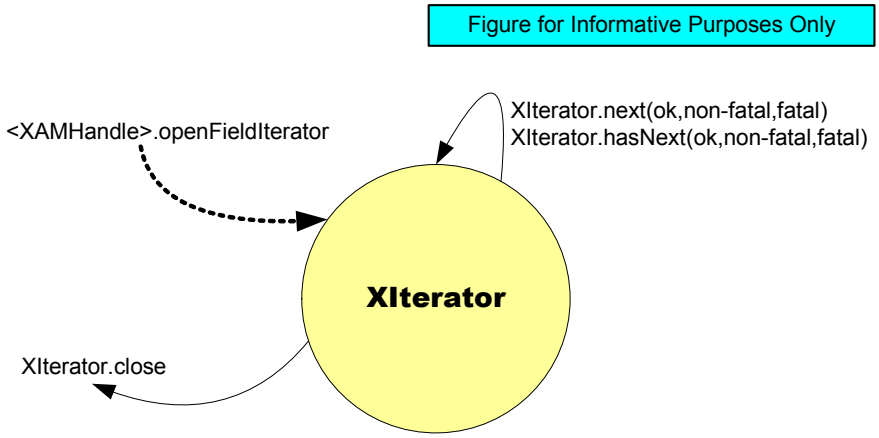

Figure 8 – XIterator Instance FSM

The XIterator instance FSM shall have the defined inputs, outputs, and transitions as described in Table 14, “XIterator Instance FSM Transitions”.

Table 14 – XIterator Instance FSM Transitions

State	Transition	To	Comment
NULL	<XAMHandle>.openFieldIterator	XIterator	The XIterator FSM shall be instantiated when the method returns successfully.
XIterator	XIterator.hasNext(ok)	XIterator	The operation shall have completed successfully and the return value shall be valid. The XIterator FSM shall stay in the XIterator state.
XIterator	XIterator.hasNext(fatal,non-fatal)	XIterator	The operation has encountered an error and the return value may or may not be valid. The XIterator FSM shall stay in the XIterator state.
XIterator	XIterator.next(ok)	XIterator	The operation shall have completed successfully. The return value shall be valid and the state of the iteration shall advance. The XIterator FSM shall stay in the XIterator state.

Table 14 – XIterator Instance FSM Transitions

State	Transition	To	Comment
XIterator	XIterator.next(fatal,non-fatal)	XIterator	The operation has encountered an error and the return value may or may not be valid. The state of the iteration shall not have changed. The XIterator FSM shall stay in the XIterator state.
XIterator	XIterator.close(ok,fatal,non-fatal)	NULL	Shall free the XIterator instance and release all resources associated with it, regardless of whether the operation completed successfully with a fatal error or non-fatal error.

Note: The normative definitions of actual method names and syntax can be found in the XAM C API Specification [XAM-C-API] and in the XAM Java API Specification [XAM-JAVA-API].

To exit the XIterator instance FSM, the XAM application shall call XIterator.close. This call shall cause the FSM to be uninstantiated and the associated resources to be returned to the operating environment. If XIterator.close completed successfully, no error occurred in closing the XIterator instance.

If the XIterator was successfully opened and XIterator.close returns a non-fatal or fatal error, the fields associated with the parent object shall not be modified. However, if a fatal error occurred, XIterator.close may cause the parent object of the XIterator to eventually transition to the corrupt state (i.e., the XSet, the XSystem, or the XAM Library). This transition could occur, for example, if connectivity was lost to the XAM System. XIterator.close might return a fatal error, and the parent object (i.e., the XSet, XSystem, or XAM Library) would also transition to the corrupt state when XIterator.next is called on that object.

Note: No explicit corrupt state exists for the XIterator instance FSM, which simplifies the interaction with the XIterator. If the XSystem chooses to maintain an internal state that causes the XIterator instance to become unusable, it may simply return a fatal error for any calls made on the XIterator instance.

Regardless of the return value of XIterator.next and XIterator.hasNext, the XIterator instance FSM shall stay in the same XIterator state.

7 XAM Library and XSystems

This chapter examines, in more detail, the XAM Library and XSystem logical view of XAM and the software module perspective, taking into account the VIM and XAM Toolkit. Figure 9 shows an overview of the XAM Library components. This figure was first shown in Chapter 5, “Overview of the XAM Architecture”, which introduced the XAM software modules.

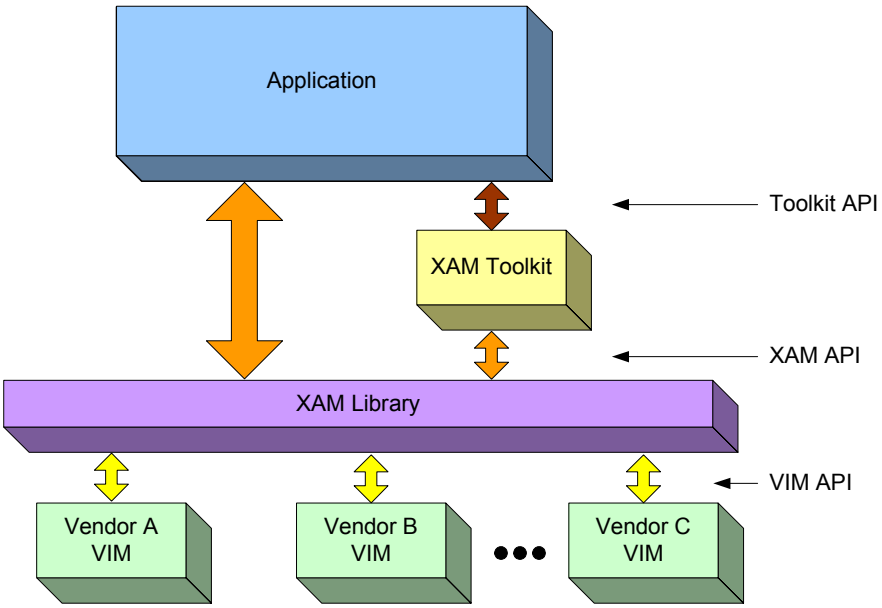

Figure 9 – XAM Library Components

The logical view of XAM was first introduced in Section 5.2, “XAM Object Model” as a set of hierarchical objects. Section 6.1, “XAM Objects” formally defined the object hierarchy and how individual object instances are created and destroyed. This chapter defines the specific attributes of the XAM Library and XSystem objects.

7.1 XAM Library

This section describes the XAM Library, which is a software component that implements the XAM Application Programming Interface (API). The XAM Library allows XAM applications to create and manage XAM sessions and to connect to and manipulate XSystems. The XAM Library also contains fields (properties or XStreams) that describe its capabilities, configuration, and characteristics.

The XAM Library may have different implementations and characteristics, depending on the programming language and operating environment for which it is designed. To illustrate, it may be helpful to think of the XAM Library as a dynamically linked library (DLL) that implements the C language XAM API described in [XAM-C-API]. Using this analogy, the source code of a XAM application that wants to use the XAM API must be configured to call the XAM API methods, and the XAM application’s executable code must be linked with the XAM Library DLL.

It is strongly recommended that the design and implementation of the XAM Library impose no less or more constraints on concurrency beyond what the target operating environment supports. An application that uses XAM Library resources should have no effect on other concurrent applications.

If the operating environment supports it, individual applications may use multithreading to call independent requests to the XAM Library concurrently; hence, all XAM Library method implementations in these operating environments shall be thread safe. Calling a block on a XAM API method causes a thread within the method to block until the XAM Library is able to complete or fail the method.

The XAM Library provides asynchronous (i.e., non-blocking) methods for a subset of the XAM API. By invoking a non-blocking XAM API method, control is returned immediately to the thread that called the method. The XAM application can either poll or use a callback method to determine when the operation has completed (either successfully or with an error). If a callback method is defined when the non-blocking method is called, then the callback interface shall be called when the non-blocking method has completed. If a callback method is not specified, the XAM application uses XAsync.isComplete to poll for the status of the asynchronous method.

7.1.1 Vendor Interface Modules

The XAM Library is a software layer that implements the XAM API methods and abstracts any storage implementation and interface details away from the application. To communicate with the actual implementations of XAM Storage Systems, the XAM Library uses Vendor Interface Modules (VIMs), which are software modules provided by the respective vendors of said systems. The VIM's purpose is to help translate standard XAM API commands, which the application calls using the XAM API, into whatever vendor-specific actions are required to implement them. The XAM Library interfaces with the VIMs using a standard VIM API described in [XAM-C-API] and [XAM-JAVA-API].

The XAM Library is responsible for discovering and managing all VIMs installed and configured in its environment. While XAM Library implementations may differ in specifics, all XAM Libraries should support adding new VIMs and deleting and upgrading existing VIMs. All XAM Libraries should also support the ability to, on demand, dynamically load and initialize VIMs during normal operations.

XAM Library implementations should seek to optimize the consumption of local resources. The XAM Library should manage VIMs and their run-time behavior in such a way that a program error encountered in one VIM should not interfere with unrelated requests in that VIM or other registered VIMs. The XAM Library should also seek to ensure that VIM management operations do not cause unintended resource leaks or system crashes.

The functionality of VIMs should be hidden from the application, which enables the application to concern itself only with the XAM Library and the XAM API.

7.1.2 XAM Toolkits

Besides general applications, optional XAM toolkits may be built on top of the standard XAM API. Such toolkits may be provided by any storage vendor or interested third party. The XAM standard toolkit and list of available toolkit methods can be found in Annex A, "(normative) XAM Toolkit".

7.1.3 Methods on the XAM Library Object

The full specification of XAM API methods can be found in [XAM-C-API] and [XAM-JAVA-API]. Table 15 describes the methods on the XAM Library object.

Table 15 – Methods on the XAM Library Object

Method	Input	Output	Comment
XAMLibrary.connect	XRI	XSystem instance	Establishes a XAM session to a specified XSystem

Note: Disconnecting from an XSystem is listed in Section 7.2.2, "XSystem Methods".

The XRI is used to specify the XSystem to which the application wishes to connect. The XRI may optionally include a vimName which indicates the preferred VIM to use for the connection. See Section 7.2.1, “XSystem Resource Identifier”, for details.

7.1.4 Fields of the XAM Library Object

Table 16, “Fields of the XAM Library Object” defines a set of XAM Library object fields that shall be supported by every XAM Library implementation. Please note that these fields are not necessarily static and persistent, as the XAM Library object may synthesize them at run time.

The concept of binding/nonbinding field attributes doesn’t apply to XAM Library object fields; therefore, all XAM Library object fields shall be nonbinding. Some XAM Library object fields may be readonly and intended to be only inspected by the application. Others may be modifiable by the application to effect a change in the behavior of the XAM Library object. Changes to XAM Library object fields shall not be persisted.

Table 16 – Fields of the XAM Library Object

Field Name	Type	Binding	ReadOnly
.xam.identity	xam_string	FALSE	TRUE
.xam.log.level	xam_int	FALSE	FALSE
.xam.log.verbosity	xam_int	FALSE	FALSE
.xam.log.path	xam_string	FALSE	FALSE
.xam.apiLevel	xam_string	FALSE	TRUE
.xam.vim.list.<name>	xam_string	FALSE	TRUE

.xam.identity

is a xam_string indicating the identity of a particular XAM Library. Applications should treat this field as an opaque string for informative purposes and shall not make functional decisions based on its contents. XAM Library vendors shall use this field to indicate the origin, version, and other identifying qualities of a particular XAM Library.

Example: *.xam.identity* contains “Example Corp XAM Library, version 2.5, build 006”.

.xam.log.level

is a xam_int property indicating the run-time severity of which loggable events are written to the log file. The log level applies to the XAM Library object and all associated VIMs. This field can be modified by the

application, which can use it to increase/decrease the severity of which events are logged. A value of 0 shall mean “no logging”. The permissible values for *.xam.log.level* shall be:

All	5
INFO	4
WARN	3
ERROR	2
FATAL	1
OFF	0

Example: *.xam.log.level* contains 0 (no logging).

.xam.log.verbosity

is a xam_int property indicating the run-time verbosity of a XAM Library object and all associated VIMs. This field can be modified by the application, which can use it to increase/decrease the verbosity of logged events. A value of 0 shall mean “no logging”. The effect of changing this field, as well as the possible values of this field, are specific to the implementation of the XAM Library.

Example: *.xam.log.verbosity* contains 0 (no debugging).

.xam.log.path

is a xam_string property indicating the run-time path to the XAM Library log file. This field can be modified by the application, which will create the log file at the specified location. The effect of changing this field is specific to the implementation of the XAM Library.

Example: Example:.*xam.log.path* “/home/application/xamlog.txt”

.xam.apiLevel

is a xam_string property indicating the XAM API version supported by a XAM Library. The apiLevel shall always be in the format “xx.yy.zz”. XAM Version 1.0 shall be “01.00.00”.

.xam.vim.list.<name>

is a set of xam_string properties indicating a list of VIM names that are generated by the XAM Library based on available VIMs configured or discovered in the current operating environment. Each of these fields is a xam_string containing the vimName. The vimName may be taken verbatim and used to construct an argument (called an XRI) which is passed into XAMLibrary.connect.

Example: *.xam.vim.list.com.example.v0814* contains “com.example.v0814”.

A possible resulting XRI would be “snia-xam://com.example.v0814!xsystemname”

7.2 XSystem

This section describes the concept and characteristics of the XSystem. The XSystem is a logical container of XSets and fields (properties or XStreams), which are pertinent to the XSystem, and describe its capabilities, configuration, and other characteristics.

It is important to realize the difference between the XSystem and the XAM Storage System. The XSystem is a logical concept. It defines, contains, and provides access to the set of XSets which are associated with a unique XAM session. The XAM Storage System is a set of software and hardware components needed to implement and support the behavior of one or more XSystems. As such, the XAM Storage System is a concept relevant to storage providers only, and is not exposed to XAM applications. How an individual XSystem maps to one or more or a subset of XAM Storage Systems is resolved by the implementation of the VIM and the XAM Storage System and may not be visible to the XAM applications at all.

For example, an XSystem may map to a single storage array supplied by a storage vendor, or maybe to a physical or logical partition of this array. It may also map to an aggregation of several arrays, or several partitions residing on the same or different arrays, supplied by the same or different vendors. The implementations of these arrays may include different types of storage hardware and media, e.g., Fibre Channel or SATA disk drives, or optical disks or tape drives. All of these details concern only the XAM Storage System implementor and are abstracted away from the XAM application, which experiences the XAM world through the simplified virtual concept of an XSystem.

Throughout the XAM documentation, the terms XAM Storage System, XSystem, XSystem instance, and XAM session are used extensively. These terms are clarified as follows:

- A XAM Storage System is some combination of storage software and hardware and cannot be directly manipulated by the XAM application. An XSystem is a logical concept, or a virtual container of XSets, which ultimately maps (by some particular storage vendor's implementation) to some XAM Storage System.
- A XAM session is a logical connection between an application and a particular XSystem. This connection presents the application with a specific view of the XSystem, which is also referred to as an XSystem instance. An example would be if the XSystem is configured to grant conditional visibility of its fields or XSets, based on the identity and authentication of the application establishing the XAM session. Thus, two different applications connecting to the same target XSystem (via two separate XAM sessions) may see two different XSystem instances, which, in fact, may be slightly different, as the XSystem view depends on the context of the actual XAM session.

Thus, the terms XAM session and XSystem instance are synonymous and interchangeable, with the former term putting an emphasis on the logical connection (and its lifecycle), and the latter term referring to the object that is visible through this logical connection.

7.2.1 XSystem Resource Identifier

A XAM application connects to an XSystem by calling XAMLibrary.connect in the XAM API and specifying the XSystem's Resource Identifier (XRI) string as its parameter. The XRI shall be an IRI (Internationalized Resource Identifier, [RFC 3987], and its syntax shall be as follows (in ABNF notation). See [RFC 4234] for details on ABNF.

```
xam-xri     = "snia-xam://" [vimname] xsystemname [params]
vimname     = iuserinfo "!"  ; allows same characters as user info
xsystemname = ihost ; xsystemname has same syntax as host
params      = "?" param "=" value [ more-params ]
more-params = 1*( "&" param "=" value )
param       = 1*ifragchar
value       = 1*ifragchar
ifragchar   = ipchar / "/" / "?"; characters allowed in a fragment
```

The fields iuserinfo, ihost, and ipchar shall be as defined in [RFC 3987]. This syntax allows percent-encoding in any element. See [RFC 3986] for more details on percent-encoding. This syntax forbids

private use Unicode characters, even though [RFC 3987] allows their use in the query component of an IRI. This syntax shall be further restricted by the following rules:

- "!" shall be forbidden in the VIM name.
- An xsystemname shall follow the rules for DNS names. A name consists of a sequence of domain labels separated by ".", each domain label starting and ending with an alphanumeric character and possibly also containing "-" characters. The rightmost domain label of a fully qualified domain name in DNS may be followed by a single "." and should be, if it is necessary to distinguish between the complete domain name and some local domain.
- "&" shall be forbidden in param and value.

As indicated, param=value clauses in the XRI may be used to specify session-related parameters. These parameters and their values are not defined within the XAM standard and are expected to be vendor specific. However, the character "." (a dot) shall be reserved as the first character of parameters that may be standardized by the SNIA in the future.

Site administrators are expected to create the xsystemnames when their XSystems are configured, and therefore will be familiar with these names. Likewise, the storage vendors shall create the vimnames for their respective VIMs. Applications should be designed and coded in a way to allow the XRI to be easily configured at run time.

Per the XRI definition above, vimname is an optional component of the XRI. If specified, it shall be a directive to the XAM Library for which VIM to use to reach the specified xsystemname. If not specified, the XAM Library may need to use some type of algorithm to find the right VIM to use. One possible algorithm is for the XAM Library to iterate through its list of installed VIMs, trying to establish a connection through each of them, until it finds the right VIM able to connect to xsystemname. The XAM Library may also choose to persistently cache previously discovered xsystemname-to-VIM mappings. Since the discovery process may be inefficient, applications using the XAM Library may want to minimize the impact by specifying the vimname whenever practical. Prudent application writers may want to be careful not to unconditionally hard code the vimnames, since the vimnames may change as the VIMs are upgraded or replaced.

When the application creates an XSystem instance (using an XRI and the XAMLibrary.connect method), the XAM Library shall load and initialize the VIM. Loading and initializing the VIM shall not require any special methods to be called by the calling application; this is done automatically as a part of the connect. The transfer of information from the XAM Library to the VIM is mediated by the XSystem instance. When constructed, a field shall be created on the XSystem instance. This field shall be named *.xsystem.initializing* with a value of TRUE and with readonly also being TRUE. Then, all fields on the XAM Library shall be created on the new XSystem instance with the same field names, attributes, and values. Finally, *.xsystem.initializing* will be removed. The VIM shall take this information and process it accordingly. Finally, the unauthenticated XSystem instance shall be returned to the application.

7.2.2 XSystem Methods

Table 17 and Table 18 list the synchronous and asynchronous methods for the XSystem.

Table 17 – XSystem Synchronous Methods

Method	Input	Output	Comment
XSystem.authenticate	XStream (writeonly mode)	XStream (readonly mode)	Starts authentication handshake
XSystem.abandon	-	-	Abandons existing XAM session
XSystem.close	-	-	Closes existing XAM session

Table 17 – XSystem Synchronous Methods

Method	Input	Output	Comment
XSystem.createXSet	-	XSet instance	Creates a new XSet
XSystem.openXSet	XUID	XSet instance	Opens specified XSet
XSystem.copyXSet	XUID	XSet instance	Creates a new XSet with some system and all application fields identical in name, attributes, and value to those of the input XSet
XSystem.deleteXSet	XUID	-	Deletes specified XSet
XSystem.isXSetRetained	XUID	xam_boolean	Verifies whether the XSet is subject to XSet retention criteria
XSystem.accessXSet	XUID	xam_boolean	Verifies whether the XSet can be accessed
XSystem.holdXSet	XUID, xam_string	-	Marks specified XSet for hold
XSystem.releaseXSet	XUID, xam_string	-	Releases specified XSet from hold
XSystem.getXSetAccessTime	XUID	xam_datetime	Returns *.xset.time.access* from specified XSet without changing it

Table 18 – XSystem Asynchronous Methods

Method	Input	Output	Comment
XSystem.asyncOpenXSet	XUID, Callback, XOPID	XAsync	Opens specified XSet
XSystem.asyncCopyXSet	XUID, Callback, XOPID	XAsync	Creates a new XSet with some system and all application fields identical in name, attributes, and value to those of the input XSet

7.2.3 XSystem Fields

Table 18, “XSystem Fields” defines a set of XSystem fields that shall be supported by every XSystem implementation. These fields are not necessarily static and persisted within the XSystem; they may be synthesized by the XSystem at run time, within the scope of an established XAM session. The visibility of these fields depends on the specific state within the authentication state machine. See Section 7.3.1, “Authentication State Machine” for more details on the authentication state machine.

The concept of binding/nonbinding field attributes is irrelevant for XSystem fields; therefore, all XSystem fields shall be nonbinding. Some XSystem fields may be readonly and intended to be only inspected by the application. Others may be modifiable by the application to effect a change in the behavior of the XSystem.

Table 19 – XSystem Fields

Field Name	Type	Binding	Readonly
.xsystem.identity	xam_string	FALSE	TRUE
.xsystem.time	xam_datetime	FALSE	TRUE
.xsystem.limits.maxFieldsPerXSet	xam_int	FALSE	TRUE

Table 19 – XSystem Fields

Field Name	Type	Binding	Readonly
.xsystem.limits.maxSizeOfXStream	xam_int	FALSE	TRUE
.xsystem.auth.SASLmechanism.list.<mechanism>	xam_boolean	FALSE	TRUE
.xsystem.auth.SASLmechanism.default	xam_string	FALSE	TRUE
.xsystem.auth.granule.list.<granule>	xam_boolean	FALSE	TRUE
.xsystem.auth.identity.authentication	xam_string	FALSE	TRUE
.xsystem.auth.identity.authorization	xam_string	FALSE	TRUE
.xsystem.auth.expiration	xam_int	FALSE	TRUE
.xsystem.access	xam_boolean	FALSE	TRUE
.xsystem.access.policy.list.<name>	xam_string	FALSE	TRUE
.xsystem.job.commit.supported	xam_boolean	FALSE	TRUE
.xsystem.job.list.<jobType>	xam_boolean	FALSE	TRUE
.xsystem.job.list.xam.job.query	xam_boolean	FALSE	TRUE
.xsystem.job.xam.job.query.continuance.supported	xam_boolean	FALSE	TRUE
.xsystem.job.xam.job.query.level1.supported	xam_boolean	FALSE	TRUE
.xsystem.job.xam.job.query.level2.supported	xam_boolean	FALSE	TRUE
.xsystem.retention.enabled.policy.list.<name>	xam_string	FALSE	TRUE
.xsystem.retention.duration.policy.list.<name>	xam_string	FALSE	TRUE
.xsystem.deletion.autodelete	xam_boolean	FALSE	TRUE
.xsystem.deletion.autodelete.policy.list.<name>	xam_string	FALSE	TRUE
.xsystem.deletion.shred	xam_boolean	FALSE	TRUE
.xsystem.deletion.shred.policy.list.<name>	xam_string	FALSE	TRUE
.xsystem.storage.policy.list.<name>	xam_string	FALSE	TRUE
.xsystem.management.policy.list.<name>	xam_string	FALSE	TRUE
.xsystem.management.policy.default	xam_string	FALSE	TRUE

.xsystem.identity

is a xam_string indicating the identity of an XSystem. Applications should treat this field as an opaque string for informative purposes and shall not make functional decisions based on its contents. XSystem implementers shall use this field to indicate the identity and other identifying qualities of a particular XSystem.

Example: *.xsystem.identity* contains “Example Corp – Lab Test System #16”.

.xsystem.time

is a xam_datetime property indicating the current time on an XSystem. The time shall always be reported as UTC (Coordinated Universal Time), though the precision and granularity depends on the XSystem implementation.

Example: 2008-08-09T05:52:30.188Z

.xsystem.limits.maxFieldsPerXSet

is a xam_int property indicating the maximum number of fields that each XSet that is created on a particular XSystem may contain. This value shall be in the range [2^14, 2^63-1] (i.e, from 16,384 to 1.05567e+49 inclusive), depending on the XSystem implementation. The actual limit may be lower, depending on run-time resource constraints.

.xsystem.limits.maxSizeOfXStream

is a xam_int property indicating the maximum number of bytes each individual XStream created on a particular XSystem may contain. This value shall be in the range [2^36, 2^63-1] (inclusive of boundaries), depending on the XSystem implementation. The actual limit may be lower, depending on the constraints of run-time resources.

.xsystem.auth.SASLmechanism.list.<mechanism>

is a set of xam_boolean properties indicating a list of valid SASL mechanism keywords, as defined and maintained by IANA; see [IANA-SASL]. The value of each property in this list indicates whether that particular mechanism is supported by the XSystem.

Example: *.xsystem.auth.SASLmechanism.list.CRAM-MD5* (= TRUE) indicates that the XSystem supports the CRAM-MD5 SASL mechanism.

.xsystem.auth.SASLmechanism.default

is a xam_string property whose value indicates the default SASL mechanism in use by an XSystem. Applications are encouraged to use the default whenever possible.

Example: *.xsystem.auth.SASLmechanism.default* (= “CRAM-MD5”) indicates that the XSystem’s SASL mechanism defaults to CRAM-MD5.

.xsystem.auth.identity.authentication
.xsystem.auth.identity.authorization

are two xam_string properties whose values indicate the SASL authentication and authorization identities within the scope of the current XAM session. See Section 11.2, “XAM Application Authentication and SASL” for an in-depth discussion of SASL and these identities.

.xsystem.auth.granule.list.<granule>

is a set of xam_boolean properties indicating a list of valid XAM granules. See Section 11.3.1.1, “XSystem Authorization Elements” for a normative list of granules. The value of each property in this list indicates whether that particular granule is granted to the application.

Example: *.xsystem.auth.granule.list.hold* (= TRUE) indicates that the application is granted the granule to call XSystem.holdXSet and XSystem.releaseXSet.

.xsystem.auth.expiration

is a xam_int property indicating the number of seconds remaining before the XSystem will require a re-authentication. This value is an estimated value and should be treated as a hint. The real value may be longer than the estimated value. A value of -1 (negative one) shall mean that the expiration estimate is infinite or that no re-authentication is required.

.xsystem.access

is a xam_boolean property indicating whether an XSystem supports (and has enabled) XSet access control policy. See Section 11.3.2, “XSet Access Control Policy” for an in-depth explanation of XSet access control policy.

.xsystem.access.policy.list.<name>

is a set of xam_string properties indicating a list of XSet policy names that are pertinent to the XSystem’s XSet access control policy capability. See Section 11.3.2, “XSet Access Control Policy” for an in-depth explanation of XSet access control policy.

.xsystem.job.commit.supported

is a xam_boolean property indicating whether the XSystem supports invoking XSet.commit on an XSet, which, at present, identifies a running job.

.xsystem.job.list.<jobType>

is a set of xam_boolean properties indicating a list of valid XAM job types. The xam_boolean value of each individual property in this list indicates whether that particular job type is supported by the XSystem. JobType “query” shall be supported by all XSystem implementations.

Example: *.xsystem.job.list.xam.job.query* (= TRUE) indicates that the XSystem supports query functionality. See Chapter 10, “Query” for an in-depth explanation of XAM query).

.xsystem.job.list.xam.job.query

is a xam_boolean property whose presence and value indicates that the XSystem supports the “query” jobType.

.xsystem.job.xam.job.query.continuance.supported
.xsystem.job.xam.job.query.level1.supported
.xsystem.job.xam.job.query.level2.supported

is a set of xam_boolean properties indicating the capabilities supported by the “query” jobType on a particular XSystem. See Section 10.4, “Level 1 Query: Where Clause Operators” and Section 10.5, “Level 2 Query: Where Clause Content Search Operators” for a detailed explanation of XAM query capabilities.

.xsystem.retention.enabled.policy.list.<name>
.xsystem.retention.duration.policy.list.<name>

are xam_string properties indicating lists of XSet policy names that are pertinent to various aspects of the XSystem’s XSet retention policy management and available on a particular XSystem. See Section 9.3.3, “XSet Management Policy” for an in-depth explanation of XSet retention management policies.

.xsystem.deletion.autodelete

is a xam_ boolean property indicating whether an XSystem supports (and has enabled) the XSystem deletion of an XSet after XSet retention criteria is satisfied and the XSet is not on-hold. See Section 9.2.2, “XSet Deletion” for an in-depth explanation of XSet automated deletion.

.xsystem.deletion.autodelete.policy.list.<name>

is a set of xam_string properties indicating a list of XSet policy names that are pertinent to the XSystem’s autodelete capability. See Section 9.3.3, “XSet Management Policy” for an in-depth explanation of XSet management policies.

.xsystem.deletion.shred

is a xam_ boolean property indicating whether a XAM Storage System supports (and has enabled) the shred after delete capability. See Section 9.2.2, “XSet Deletion” for an in-depth explanation of XSet shredding.

.xsystem.deletion.shred.policy.list.<name>

is a set of xam_string properties indicating a list of XSet policy names that are pertinent to the XSystem’s shred after delete capability. See Section 9.3.3, “XSet Management Policy” for an in-depth explanation of XSet management policies.

.xsystem.storage.policy.list.<name>

is a set of xam_string properties indicating a list of XSet policy names that are pertinent to the XSystem’s storage management capabilities. See Section 9.3.3, “XSet Management Policy” for an in-depth explanation of XSet management policies.

.xsystem.management.policy.list.<name>

is a set of xam_string properties indicating a list of XSet principal management policy names that are pertinent to the XSystem’s XSet management capabilities. See Section 9.3.3, “XSet Management Policy” for an in-depth explanation of XSet management policy.

.xsystem.management.policy.default

is a xam_string property whose value indicates the default principal management policy available on an XSystem. See Section 9.3.5, “XSet Management Policy Default” for an in-depth explanation of XSet management policy).

Example: *.xsystem.management.policy.default* (= “somepolicyname”) indicates that the XSystem’s XSet management policy defaults to “somepolicyname”.

7.3 XAM Session

Note: In this section, the phrases “connecting to an XSystem” and “opening/establishing a XAM session” are synonymous, as are “disconnecting from an XSystem” and “closing/terminating a XAM session”.

To access an XSystem, an application shall first connect to it and establish a XAM session. The XAM session is a logical connection between the application and the XSystem and persists until either the application or the XSystem decides to terminate it. The session is also considered terminated, when either the application or the XSystem terminates abnormally. The XSystem may require an application to specify

its identity after a XAM session is established and may authenticate this identity before access to data within the XSystem is granted.

A single XAM application may establish one or more concurrent XAM sessions to the same XSystem; a single XAM application may establish several XAM sessions to several XSystems simultaneously; and several XAM applications may be connected (maintain established XAM sessions) to a single XSystem concurrently.

Again, it is important to realize that the XAM session is not equivalent to a TCP/IP network connection or any other type of connection to a vendor's storage array. The XAM session is a logical concept in the XAM world. The implementation of the XAM Storage System may use several network connections or any other means to implement and support the behavior of the XAM session.

XSystems may be configured by their administrators (by means outside of the scope of the XAM standard) to require applications to identify themselves and to authenticate their identity before granting them any type of access to data contained in the XSystem. Authentication always happens within the context of a XAM session and is not an optional step. All XSystems will feature some level of authentication, though the strength (and complexity) of the authentication process may differ with different XSystem implementations and their actual configurations in end-user environments.

The implementation of the authentication mechanism in XAM is based on SASL (see Term 3.1.16, "SASL") and may include several challenge/response handshake interactions between the application and the XSystem. The XAM API provides methods for applications to initiate the authentication handshake and to respond to any challenges.

All XSystem implementations shall support the SASL "ANONYMOUS" and "PLAIN" mechanisms, though the XAM Storage System administrator may choose not to use these mechanisms. XAM Storage System vendors are encouraged to implement more sophisticated authentication mechanisms.

7.3.1 Authentication State Machine

XSystems may be configured to require the application to authenticate once on XAM session initiation, or to re-negotiate their authentication periodically, e.g., after some amount of elapsed time or data traffic. Since initial authentication shall be required always, applications shall initiate the authentication handshake and present their credentials immediately after establishing the XAM session (connecting to the XSystem). Failure to authenticate shall cause the XSystem to deny the application any access to data contained within it.

The state machine in Figure 10, “Authentication State Machine” represents the various states of the XAM session. These states reflect the authentication status and the status returns that the XAM application may experience as a result of the method calls. XSystems shall support these methods and status returns.

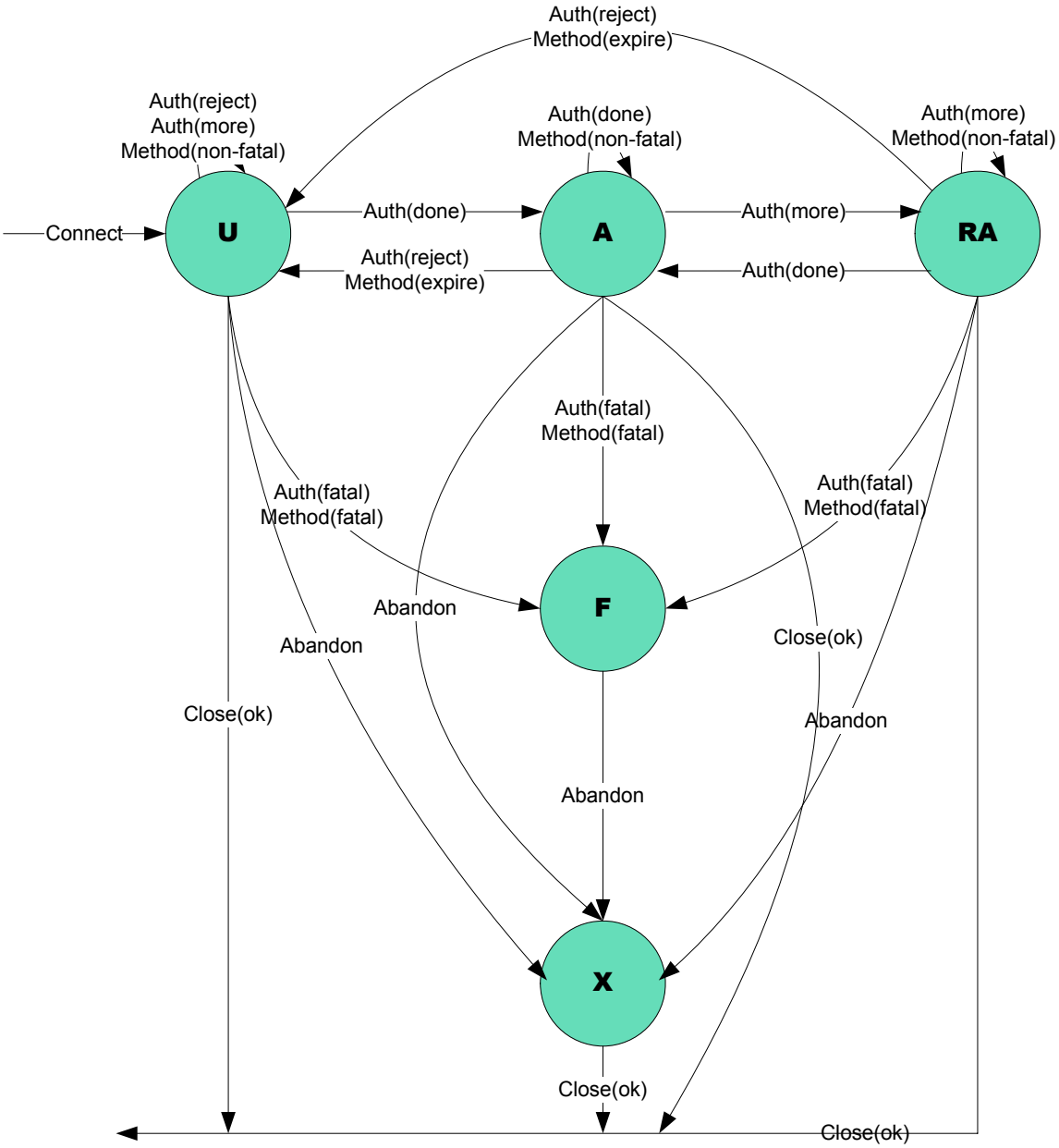

Figure 10 – Authentication State Machine

The states in Figure 10 are labelled U (unauthenticated), A (authenticated), RA (re-authenticating), F (failed) and X (abandoned). When in states A and RA, the application shall be considered authenticated and have full access to the XSystem resources. On the other hand, states U, F, and X are states where the application is in various non-authenticated states, and therefore shall not have access to XSystem resources. For convenience, this section will refer to a XAM session in states (A, RA) as logged in and in states (U, F, X) as logged out. Thus, within the context of an established XAM session, an application may

be logged in or logged out, depending on its current state in the authentication state machine as shown in Figure 10.

To interpret the transition labels in the authentication state machine, first consider each transition to be the format Method(status). For clarity, Method(status) is used to denote a XAM API method call, excluding the authentication, close, and abandon methods, which are particularly relevant to the state machine and are labelled explicitly. The transitions occur as follows:

1. The application initiates the method.
2. The XSystem responds with a status (typically via a return code on the called API method) and a transition in the state machine.

   Note: In the table below, (non-fatal) is considered a recoverable error return, and (fatal) is considered a non-recoverable error return. See [XAM-C-API] and [XAM-JAVA-API] for what constitutes a fatal or non-fatal error.

Table 19, “Authentication State Machine” describes the state machine transitions in a more formal way.

Table 20 – Authentication State Machine

State	Transition	To	Comments
NULL	XAM session not established		
	Connect	U	XAMLibrary.connect succeeds.
U	**Unauthenticated**: XAM session established but application not logged in.		
U	Close(ok)	NULL	XSystem.close succeeds. Application terminates XAM session gracefully.
U	Abandon	X	XSystem.abandon succeeds. Application abandons XAM session ungracefully.
U	Auth(fatal) Method(fatal)	F	XSystem.authenticate returns a fatal error, or any XAM API method returns a fatal error.
U	Auth(more) Auth(reject) Method(non-fatal)	U	XSystem.authenticate returns ‘more’ (requiring that the handshake be continued), ‘reject’ (rejecting the authentication attempt without prejudice), or any XAM API method returns a non-fatal error.
U	Auth(done)	A	XSystem.authenticate successfully completes the authentication handshake.
A	**Authenticated**: XAM session established, application logged in.		
A	Close	NULL	XSystem.close succeeds. Application terminates XAM session gracefully.
A	Abandon	X	XSystem.abandon succeeds. Application abandons XAM session ungracefully.
A	Auth(fatal) Method(fatal)	F	XSystem.authenticate or any XAM API method returns a fatal error.
A	Auth(reject) Method(expire)	U	XSystem.authenticate returns ‘reject’ (rejecting the re-authentication attempt without prejudice), or any XAM API method returns a non-fatal expiration error.
A	Auth(done) Method(non-fatal)	A	XSystem.authenticate successfully re-authenticates or any XAM API method returns a non-fatal error.

Table 20 – Authentication State Machine

State	Transition	To	Comments
A	Auth(more)	RA	XSystem.authenticate returns ‘more’ (requiring that the authentication handshake be continued).
RA	**Re-authenticating**: XAM session established, application logged-in, proactive re-authentication in progress.		
RA	Close	NULL	XSystem.close succeeds. Application terminates XAM session gracefully.
RA	Abandon	X	XSystem.abandon succeeds. Application abandons XAM session ungracefully.
RA	Auth(fatal) Method(fatal)	F	XSystem.authenticate or any XAM API method returns a fatal error.
RA	Auth(more) Method(non-fatal)	RA	XSystem.authenticate returns ‘more’ (requiring that the handshake be continued), or any XAM API method returns a non-fatal error.
RA	Auth(reject) Method(expire)	U	XSystem.authenticate returns ‘reject’ (rejecting the re-authentication attempt without prejudice), or any XAM API method returns a non-fatal expiration error.
RA	Auth(done)	A	XSystem.authenticate successfully completes the re-authentication handshake.
F	**Failed**: XAM session established but in a failed state; application may call only XSystem.abandon.		
F	Abandon	X	XSystem.abandon succeeds.
X	**Abandoned**: XAM session established but in abandoned state; application may only call XSystem.close.		
X	Close(ok)	NULL	XSystem.close succeeds.

The major concepts of the state machine include initial authentication and re-authentication, of which there are two types: reactive and proactive.

7.3.2 Initial Authentication

The XAM application enters the state machine by connecting to the XSystem, and the XAM session is initially placed in state U (unauthenticated). In state U, the XAM application shall have access to XSystem fields *.xsystem.auth.SASLmechanism.list.**. Visibility of other XSystem fields in state U depends on the XSystem implementation. In states A and RA, all XSystem fields defined in Table 18, “XSystem Fields” shall be visible. Visibility of XSystem fields in states X and F depends on the XSystem implementation.

The XAM application initiates the authentication handshake by calling XSystem.authenticate. Depending on the SASL mechanism configured on the XSystem and selected by the application, the application may need to respond to a series of challenges by iteratively calling XSystem.authenticate. When authentication succeeds, the XAM session transitions to the A (authenticated) state. When authentication fails, the XAM session either stays in state U (unauthenticated - authentication rejected) or transitions terminally to state F (failed - authentication rejected with prejudice or XSystem experienced a fatal error).

Once in state A, the XAM application can access the contents stored within the XSystem, subject to the capabilities that the site security administrator grants to the application’s identity and role.

7.3.3 Re-Authentication

Some security environments may require that storage users periodically re-authenticate with the local security authority, even if they already hold an established and fully authenticated XAM session. In such environments, XSystems configured to require periodic re-authentication may trigger the authentication handshake at any point throughout the lifetime of the XAM session, by signalling to the application that a re-authentication is required. This signalling is typically done via an appropriate non-fatal error return code, which may be returned by any XAM API method. Authentication expiration errors are deliberately non-fatal; if the application successfully re-authenticates after having been signalled to do so, the XAM session shall be restored to its pre-expiration state. XAM applications should anticipate this scenario and be coded in a way to react to re-authentication requests gracefully.

Alternatively, applications may also proactively re-authenticate their existing XAM session by initiating the authentication handshake at any point during the lifetime of the XAM session. The XSystem maintains a property field (*.xsystem.auth.expiration*), which contains the number of seconds left until a re-authentication is required. Applications may take advantage of this field to proactively re-authenticate and avoid inconveniences, which would otherwise be caused by the XSystem triggering a re-authentication via a XAM API method non-fatal error return code.

Under either re-authentication scheme, the application is expected to re-authenticate using credentials which are valid at the time of the re-authentication operation. If the XSystem security environment has changed during the current XAM session (such as may occur if a user's password is changed during a session) the up-to-date credentials must be presented to the XSystem, rather than the original authentication information. A failure to re-authenticate a running XAM session may cause the application to be logged out and all further access to data contained in the XSystem to be denied.

7.3.3.1 Reactive Re-Authentication

While in state A, the authentication may expire, causing the next XAM API method called to fail with an appropriate non-fatal 'expired' error code and the XAM session to revert to state U.

In this case (application logged out by XSystem), the full run-time context of the XAM session (e.g., open XSets, XStreams, etc.) shall be preserved by the XSystem, as long as the XAM session remains connected. The application may return to normal operations after it successfully re-authenticates with the XSystem (i.e., successfully completes an authentication handshake, as during initial authentication). Alternatively, the application may decide to abandon this XAM session and let the XAM Library and XSystem clean up (i.e., discard) any resources held.

7.3.3.2 Proactive Re-Authentication

While in state A (authenticated) the application may initiate a re-authentication handshake proactively (by calling XSystem.authenticate), thereby entering state RA (re-authenticating).

While in state RA, the application is negotiating a new authentication handshake (similarly to the initial authentication), but still has full access to data within the XSystem, until either the re-authentication fails and the XAM session reverts back to state U, or the XSystem rejects the re-authentication with prejudice and the XAM session advances into the terminal F (failed) state.

A third possibility is for the application to be logged out from its current XAM session while the re-authentication handshake is still in progress, in which case the XAM session reverts back to state U (unauthenticated), and all further access to data contained in the XSystem shall be denied.

7.3.3.3 Closing/Abandoning XAM sessions

To prevent accidental data loss when closing a XAM session, XSystem.close, which disconnects the application from the XSystem, shall always fail with a non-fatal error, if any resources within the XSystem (e.g., XSets) are held open by the application. Under normal circumstances, this behavior encourages the

application to maintain sound design practices and to always close any open XSet before disconnecting from the XSystem.

However, under abnormal circumstances (e.g., the application logged out in a XAM session by an XSystem), the application may find itself unable to return to normal operations (state A), and thus unable to close open XSets, but also unable to disconnect from the XSystem, as it still holds open XSets from before it was logged out. In such cases, the XAM application should call <XAMHandle>.abandon on the XSystem instance, which allows the application to abandon any open XAM session resources, and then disconnect from the XAM session (by closing the XSystem instance).

Having discussed the semantics of closing a XAM session and the related <XAMHandle>.abandon method, it is clear that the application may disconnect from an XSystem either in a graceful way (e.g., closing all open XSets/resources, then disconnecting from the XAM session), or in a forceful way (e.g., abandoning all open XSets/resources, then disconnecting from the XAM session).

CAUTION: The <XAMHandle>.abandon call is not recommended for normal use. It may cause data loss within the context of the affected XAM session and should be used with extreme caution and only when absolutely necessary!

8 XSet Operations

This chapter defines the behavioral model of an XSet and describes applicable methods on individual XSets and their elements. It also defines the virtual semantic model of an XSet.

Note: The physical layout of an XSet within an XSystem is an implementation detail of XAM Storage Systems, VIMs, and XSystems and is beyond the scope of this specification.

8.1 XSet Behavior

An XSet is the XAM object that is used to persist reference information. As such, it has fields (properties and XStreams), some of which the XSystem creates and some of which the application creates. An XSet can have multiple fields, where each field has several attributes. The XSystem or application may designate which fields are binding. These fields define the binding content of the XSet, and if any of these binding fields are added, modified, or deleted, the XSystem creates a new XSet and leaves the original XSet the same. For full details of fields and attributes, see Chapter 6, “XAM Objects and Common Operations”.

Each XSet has a globally unique identifier called a XUID (XAM Unique Identifier). The XUID is tied to the set of binding fields in the XSet, and if any of these fields are added, modified, or deleted, then the XSystem creates a new XSet and generates a new XUID. The mechanism used by the XSystem to create the XUID depends on the implementation; therefore, it is not defined in this specification. For details regarding the rules for determining when a new XUID (and thus a new XSet) must be created, see Section 8.3, “The XUID – Naming an XSet”.

To work with an XSet, a XAM application creates an XSet instance by calling XSystem.createXSet. To persist the information in the XSet instance and create the XSet, the application calls XSet.commit. XSet.commit persists any changes made since either the last XSet.commit was called or the XSet was created (if this is the first commit of the XSet). If any binding fields have changed (added, modified, or deleted) in the XSet instance since the previous commit, the XSystem creates a new XSet and generates a new XUID.

Note: If two XSets have the same XUID, they will have identical binding fields and values when the requirements of this specification are followed. However, if two XSets have identical binding fields and values, they may (or may not) have the same XUID. For example, if two different vendor implementations for XUID generation use different algorithms, even though the binding field inputs to the algorithm are the same, the resultant XUID on commit will not necessarily be the same.

In addition to creating a new XSet instance from scratch, an existing XSet may be opened or copied by calling XSystem.openXSet or XSystem.copyXSet, respectively. These methods use the XUID to access the XSet and create an XSet instance with the fields of the XSet. Opening the XSet preserves the XUID of the XSet, while copying does not. The semantics for an opened or copied XSet instance are the same as for an XSet instance that was created. Changes to this XSet instance are not persisted until XSet.commit is called and completes successfully.

The XSystem may export the contents of an XSet by using the export process described in Section 8.8, “XSet Import and Export”. Exporting an XSet produces a canonical XSet, the format of which is described in Section 8.8.4. The exported XSet can then be imported into another XSystem. Newly created XSet instances form the substrate for importing XSets into an XSystem. The process for importing a canonical XSet into a new created XSet instance is described in Section 8.8, “XSet Import and Export”. On a successful import, the XSet instance is in essentially the same state as if it were newly opened; on commit, the original XUID will be returned. Changes can be made to the XSet instance before committing, but changing binding fields on the XSet instance will result in a new XUID being assigned. If the XSet instance is not committed after a successful import, the XSet will not be created in the importing XSystem.

Note: Import only works with a newly created XSet instance. If an XSet instance is changed, an import attempt results in a non-fatal error. Likewise, import does not work on an XSet instance that is created by opening or copying an existing XSet.

When a XAM application no longer needs an XSet instance, it should call XSet.close, which closes the XSet and releases any resources held. If the XSet instance has changes that have not been saved by invoking XSet.commit, these changes will be lost. If a XAM application tries to close an XSystem instance that contains the XSet without closing the XSet first, the XSystem shall return a non-fatal error.

8.2 XSet Fields

8.2.1 Number of Fields on an XSet

An XSet can contain a large number of fields. The maximum number of fields that an XSet can contain is finite, however, and is a feature of the XSystem that contains the XSet. To promote portability, all XSystems must support a minimum number of fields in an XSet. For complete semantics, see Section 7.2.3, “XSystem Fields”.

The methods that create fields on an XSet shall generate a non-fatal error when used to create a field that would cause the number of fields on the XSet to exceed the maximum number of fields, as described in Section 7.2.3. Importing an XSet that has more fields than supported on the XSystem shall likewise fail with a non-fatal error when the import stream is closed.

8.2.2 Length of a Field on an XSet

XStream field values can contain a large number of bytes. The maximum number of bytes within a single XStream is limited, and the limit is an aspect of the XSystem that contains the XSet (similar to the number of fields on an XSet). For portability, all XSystems must support fields with a minimum length. For complete semantics, refer to Section 7.2.3.

When a request is made that would result in a field length that is longer than allowed by the XSystem (i.e., when writing bytes to an XStream), XStream.write shall fail and generate a non-fatal error. Importing an XSet that has XStream fields with lengths that are longer than supported on the XSystem shall likewise fail with a non-fatal error when the import XStream is closed.

8.2.3 Normative XSet Fields

This section defines XSet fields that are referred to as XSet system fields. The field, if present, shall have the name, type, binding value, and readonly value as specified in Table 21, “XSet System Fields”. See Section 8.5, “XSet Instance Finite State Machine (FSM)” for additional details of when these fields are modified and Section 8.5.4, “Summary of XSet System Fields in each XSet Instance State” for when these fields are present. The values in *xset.xuid* and *.xset.dirty* indicate the current FSM state for the XSet instance.

Table 21 – XSet System Fields

Field Name	Type	Binding	Readonly	Description
.xset.time.creation	xam_datetime	Yes	Yes	Time that the XSet was created
.xset.time.xuid	xam_datetime	Yes	Yes	Time that the XUID was assigned to the XSet
.xset.time.commit	xam_datetime	No	Yes	Most recent time that the XSet was modified
.xset.time.access	xam_datetime	No	Yes	Most recent time that the XSet was opened or committed

Table 21 – XSet System Fields

Field Name	Type	Binding	Readonly	Description
.xset.time.residency	xam_datetime	No	Yes	Time that the XSet was first stored in this XSystem
.xset.xuid	xam_xuid	No	Yes	XUID of the XSet
.xset.dirty	xam_boolean	No	Yes	Presence vs. absence of uncommitted changes

.xset.time.creation **[binding, readonly]**

indicates the creation time of the XSet and shall contain the time that XSystem.createXSet was called to instantiate the XSet instance. This field shall be created in an XSet instance and the value initialized to the current time by the XSystem when XSystem.createXSet is called. This field shall be copied from the input XSet when XSystem.copyXSet is called. The value of this field shall not be later than the value of *.xset.time.xuid*.

.xset.time.xuid **[binding, readonly]**

indicates the time at which the XUID was assigned to the XSet and shall contain the time that the XSet was first committed to an XSystem. This field shall be created and the value initialized to the current time by the XSystem when XSet.commit is called and the current XSet instance state is either in the dirty no XUID state or the clean no XUID state. See Section 8.5, “XSet Instance Finite State Machine (FSM)” for definitions of these states and further discussion. The value of this field shall not be earlier than the value of *.xset.time.creation*.

This field should be removed from an XSet instance when a binding modification is made.

.xset.time.commit **[nonbinding, readonly]**

indicates the modification time of the XSet and shall contain the time that the XSet (i.e., with this XUID) was most recently committed to the XSystem. This field shall be created and the value initialized to the current time by the XSystem on first commit. The XSystem shall update the field’s value on each subsequent commit that makes a modification to the XSet with two exceptions:

- Any change to *.xset.time.access* shall not be treated as a change to the XSet and shall not cause *.xset.time.commit* to be updated.
- Opening an XStream for write shall be considered to be a change to the XSet, even if the contents of the XStream are not subsequently changed. Opening an XStream for write shall cause *.xset.time.commit* to be updated.

The value of this field shall not be later than the value of *.xset.time.access*.

.xset.time.access **[nonbinding, readonly]**

indicates the last access time of the XSet and shall contain the time that the XSet was most recently opened (XSystem.open) or committed (XSet.commit), whichever is more recent, except as noted below. This field shall be created and the value initialized by the XSystem at the time of first commit. The XSystem shall update the value on each subsequent XSet open or commit. Due to concurrency issues, if a concurrent access occurs by another XSet instance in parallel with this XSet instance, this field’s value may be the value when this XSet instance was created (i.e., as a result of XSystem.openXSet), rather than the current value that has subsequently been updated in the XSet due to actions performed via a different XSet instance. The value of this field shall not be earlier than the value of *.xset.time.commit*.

***.xset.time.residency* [nonbinding, readonly]**

indicates the arrival (residency) time of the XSet and shall contain the time that the XSet was first stored to the present XSystem. The XSystem shall create this field and initialize its value at the time of first commit, when the XSet instance is in a no XUID state (either a dirty no XUID state or a clean no XUID state). When an import operation occurs and no binding fields are modified, the XSystem shall update the field's value when the XSet is committed to an XSystem. The value of this field should not be earlier than the value of *.xset.time.xuid*, but because of clock skew among XSystems, the value of this field may be earlier than the value of *.xset.time.xuid*.

***.xset.xuid* [nonbinding, readonly]**

indicates, if present, the XUID value for the XSet. This field shall be present if: 1) the XSet has either been previously committed to the XSystem, or 2) the current XSet instance data is the result of a successfully completed import operation (see Section 8.8.2, "XSet Import Process"). The field shall not be present if: 1) the XSet instance has never been committed to the XSystem, 2) a binding field has been modified on the XSet, or 3) the XSet was opened in copy mode.

***.xset.dirty* [nonbinding, readonly]**

indicates, if present, whether the XSet instance contains any uncommitted changes (i.e., this field shall be present if the XSet contains uncommitted changes). This field is only present in XSet instances, because committed XSets cannot contain uncommitted changes. The XSet instance that is the result of a successfully completed import operation (see Section 8.8.2, "XSet Import Process") shall be treated as containing uncommitted changes until XSet.commit is called. The field shall not be present if the XSet instance contains no uncommitted changes. An XSet opened in readonly mode cannot contain uncommitted changes; hence, this field shall not be present in a readonly XSet instance.

8.2.4 Copying an XSet - Field Behavior

XSystem.copyXset shall remove *.xset.time.xuid, .xset.time.commit, .xset.time.access* and *.xset.time.residency* from the XSet instance that is created. XSystem.copyXset shall have no effect on *.xset.time.access* in the XSet that the instance is copied from.

Any binding modification to an XSet instance shall remove *.xset.time.xuid* and *.xset.time.residency* from the XSet instance that is modified. No change shall be made to *.xset.time.creation* by either XSystem.copyXSet or a binding modification.

8.3 The XUID – Naming an XSet

An XSet's name is its globally unique identifier, called a XUID (XAM Unique Identifier). All XSets have only one XUID. If an application creates, modifies, or deletes a binding field in an XSet, the XSystem shall create a new XSet and generate a new XUID on successful commit. Thus, the XUID is bound to the XSet by the binding fields of the XSet.

Newly created XSet instances shall have no XUID associated with them (i.e., have no *.xset.xuid* property) until they are committed to an XSystem or have been the target of a successful import. When an XSet instance without an associated XUID is committed to an XSystem, the XSystem creates an XSet and assigns the XSet a XUID. The application cannot choose this XUID. After the commit, the XSet instance will have the XUID associated with it. Likewise, newly opened XSet instances will always be associated with the XUID that was bound to the XSet that was opened.

When an XSet instance that is associated with a XUID is modified, it may break the association between the XSet instance and the XUID, depending on which fields were created, modified, or deleted and

whether they were binding fields. Thus, a static XUID guarantees the integrity of the binding fields that make up an XSet. Table 22, “XSet Naming Behavior on Commit” describes these transitions.

Table 22 – XSet Naming Behavior on Commit

XSet Instance Change	Shall create new XSet with new XUID on successful commit
Add a binding field	Yes
Change binding field	Yes
Delete binding field	Yes
Add a nonbinding field	No
Change nonbinding field	No
Delete nonbinding field	No
Change Nonbinding Field to Binding	Yes
Change Binding Field to Nonbinding	Yes

When an XSet instance that is associated with a XUID is committed to an XSystem, it shall not create a new XSet. However, when an XSet instance that has no XUID is committed, it shall create a new XSet with a new XUID. The previous XSet (if any) shall remain in the XSystem, unchanged and addressable with the previous XUID. XSet.commit shall not remove the previous XSet (if one exists).

The exception to this rule is import. When an XSet instance is imported, it will be associated with a XUID. On commit, it may result in an XSet being created within the XSystem it is being imported into, if that XSet does not already exist on that XSystem.

8.4 XSet Methods

8.4.1 XSystem Operations on XSets

Table 23 lists the XSystem methods that operate on XSets. To use these methods, the application must be connected to the XSystem and have a valid XSystem instance. Complete details of these methods are found in the [XAM-C-API] and [XAM-JAVA-API].

Table 23 – XSystem Methods that Operate on XSets

Synchronous Methods	Asynchronous Method	Description
XSystem.accessXSet	-	Determines whether an XSet can be accessed, i.e., if it exists in the XSystem instance and the caller is authorized to access it
XSystem.createXSet	-	Creates a new, empty XSet instance
XSystem.openXSet	XSystem.asyncOpenXSet	Opens an existing XSet on the XSystem instance, creating an XSet instance that preserves the XUID of the XSet

Table 23 – XSystem Methods that Operate on XSets

Synchronous Methods	Asynchronous Method	Description
XSystem.copyXSet	-	Creates a copy of an existing XSet on the XSystem instance, creating an XSet instance that has no XUID; the XSystem shall not assign the same XUID associated with the existing XSet on commit
XSystem.deleteXSet	-	Deletes an existing XSet
XSystem.holdXSet	-	Places a hold on an XSet so it cannot be modified or deleted
XSystem.releaseXSet	-	Releases a hold on an XSet
XSystem.getXSetAccessTime	-	Retrieves *.xset.time.access* from the XSet without changing its value
XSystem.isXSetRetained	-	Verifies whether the XSet is subject to XSet retention criteria

8.4.2 XSet Operations on XSets

Table 24 lists the XSet methods that save, abandon, import, and export XSet instances. To use these methods, the application must have a valid XSet instance. Complete details of these methods are found in the [XAM-C-API] and [XAM-JAVA-API].

Table 24 – XSet Methods that Operate on XSets

Synchronous Methods	Asynchronous Method	Description
XSet.commit	XSet.asyncCommit	Stores any changes to the XSet since it was created, opened, copied or last committed, whichever is later. The XSet remains open.
XSet.close	-	Closes an XSet and releases all resources associated with the XSet. Close will fail if there are XStreams or XIterators open against the XSet.
XSet.abandon	-	Allows the XSet to be closed without regard for open XStreams or XIterators
XSet.openExportXStream	-	Opens a special export stream for the XSet
XSet.openImportXStream	-	Opens a special import stream for the XSet

8.5 XSet Instance Finite State Machine (FSM)

An XSet instance is different than an actual XSet. The XSet instance is effectively a handle that contains a pointer to an XSet. In addition to pointing to an XSet, it encapsulates a variety of temporary states. These states include the application's uncommitted operations on the XSet and additional information, like cached copies of some, all, or part of the fields associated with the XSet. This section defines the normative Finite State Machines (FSMs) associated with the XSet instance.

The tables in this section define the normative transitions for the FSM; the associated figures further illustrate the descriptions in the tables.

Note: The XSet instance FSM does not restrict the application from calling other methods on the parent object (e.g., the XAM Library or XSystem), except where noted.

8.5.1 Defining the FSM Hierarchy

The FSM for the XSet instance is a two-level hierarchical FSM. The top level of the hierarchy is the Master XSet instance FSM or the Master XSet FSM. The Master XSet FSM describes the states that have the same behaviors, regardless of how the XSet is opened. The second level of the hierarchy is a breakdown of the open XSet state of the Master XSet FSM. Three open XSet FSMs exist for the open XSet state, and the open XSet FSM that is applied is based on the mode in which the XSet was opened. The three FSMs include:

- Readonly open XSet FSM: XSets opened in readonly mode
- Restricted open XSet FSM: newly created, restricted XSets, or those opened or copied in restricted mode
- Unrestricted open XSet FSM: newly created, unrestricted XSets, or those opened or copied in unrestricted mode

The Master XSet FSM and an open XSet FSM are instantiated (and uninstantiated) as a pair. Transitions in the Master XSet FSM that appear to enter the open XSet state actually enter a state within the appropriate open XSet FSM, as described above. If a transition leads from an open XSet FSM state to a state on the Master XSet FSM, the currently instantiated open XSet FSM shall stay instantiated. Only when the Master XSet FSM is exited will the FSM pair be uninstantiated.

8.5.2 Master XSet FSM

The Master XSet FSM shall have the defined inputs, outputs, and transitions as described in the following sections:

- Section 8.5.2.1, “Entering the State Machine”
- Section 8.5.2.2, “Entering The Abandoned State”
- Section 8.5.2.3, “Entering the Corrupt State”
- Section 8.5.2.4, “Performing Generic Operations on an Open XSet”
- Section 8.5.2.5, “Exporting an XSet”
- Section 8.5.2.6, “Importing an XSet”
- Section 8.5.2.7, “Exiting the Master XSet FSM”

The FSM and the state transitions are also shown in Figure 11, “Master XSet FSM”. If a method has both a synchronous and an asynchronous version, only the synchronous version is shown in this figure.

Fatal errors and XSet.abandon arcs to the Master XSet FSM from the specific Open XSet FSMs are not shown in Figure 11 to aid in readability. For asynchronous XAM methods, the start of the arc indicates when the application calls the asynchronous operation and corresponds with the instantiation of an XAsync instance that is in the pending state. The end of the arc indicates when the XAsync instance transitions to the completed state.

Given the potential size of the state represented by the XSet instance, to optimize access patterns, the XSet instance points to a new XSet after an XSet.commit, if XSet.commit created a new XSet (i.e., a new XUID). Thus, the Master XSet instance FSM can dynamically refer to a new XSet over time.

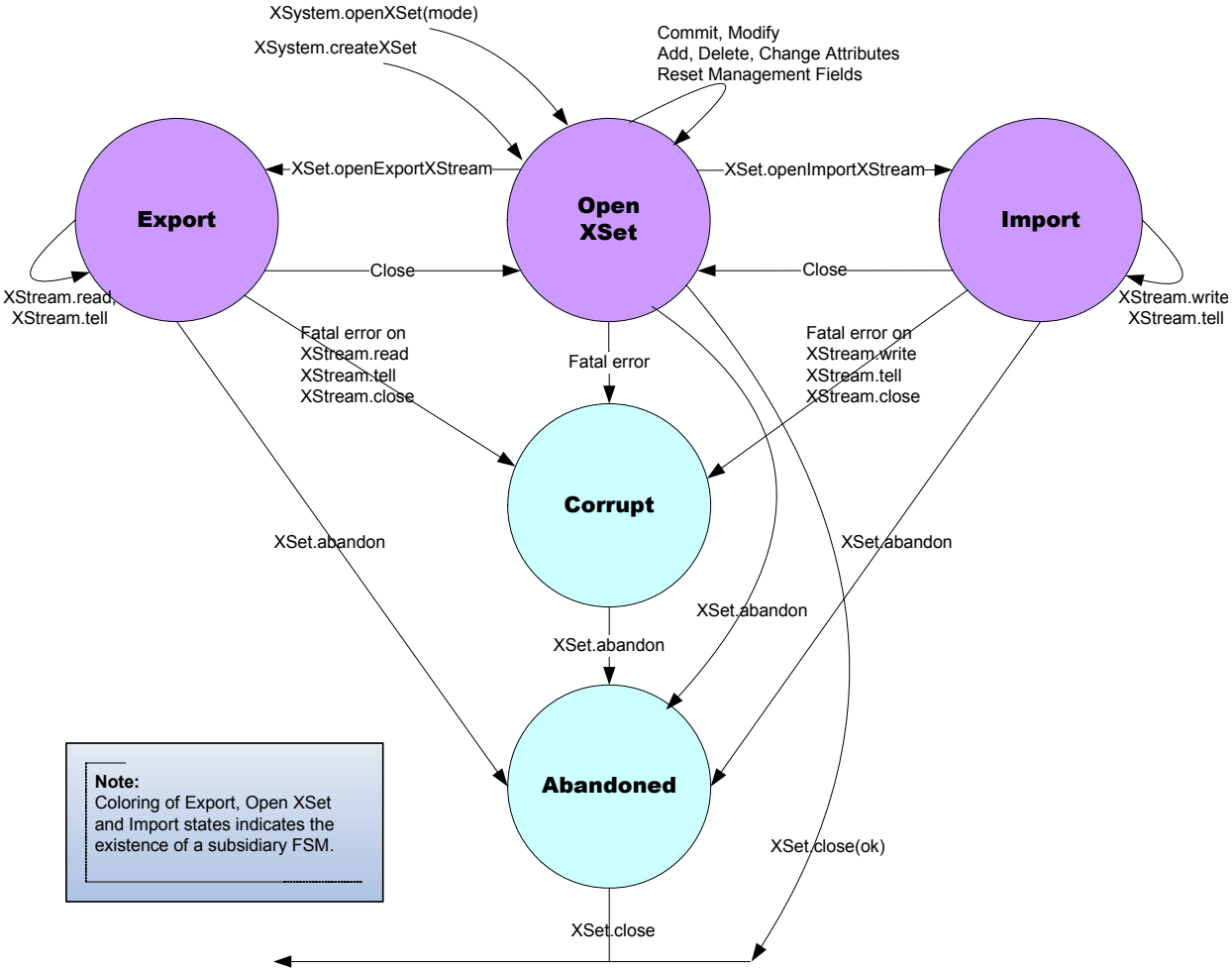

Figure 11 – Master XSet FSM

To interpret the transition labels in the Master XSet finite state machine, first consider each transition to be in the format of Method(status). The transitions occur as follows:

1. The application initiates the method.
2. The XSystem responds with a status (typically via a return code on the called API method) and a transition in the state machine.

Status ‘non-fatal’ is considered a recoverable error return code; status ‘fatal’ is considered a non-recoverable (fatal) error return code. For a definition of which method return codes are fatal and non-fatal, please see [XAM-C-API] and [XAM-JAVA-API].

Table 25, “Entrance to the Master XSet FSM” explicitly shows which methods have both versions by showing the asynchronous version as optional (e.g., [async]).

Table 25 – Entrance to the Master XSet FSM

Start State	Transition	Final State	Comment
Outside of FSM (NULL)	XSystem.[async]OpenXSet	Open XSet	The Master XSet FSM shall be instantiated when the method returns successfully. An open XSet FSM shall be instantiated as dictated by the mode.
	XSystem.[async]CopyXSet	Open XSet	The Master XSet FSM shall be instantiated when the method returns successfully. An open XSet FSM shall be instantiated as dictated by the mode.
	XSystem.createXSet	Open XSet	The Master XSet FSM shall be instantiated when the method returns successfully. An open XSet FSM shall be instantiated as dictated by the mode.

8.5.2.1 Entering the State Machine

The XSystem shall enter the Master XSet FSM when a XAM application calls XSystem.openXSet, XSystem.copyXSet, or XSystem.createXSet. The specific open XSet FSM depends on which open mode was used:

- If readonly mode was used, the readonly open XSet FSM shall also be instantiated.
- If restricted mode was used, the restricted open XSet FSM shall also be instantiated.
- If unrestricted mode was used, the unrestricted open XSet FSM shall also be instantiated.

When a XAM application calls XSystem.openXSet, the XSystem shall check if the XSet has a valid hold placed on it. If the XSet is on hold, and if the application opened the XSet in restricted or unrestricted mode, then the open shall fail. If the open mode was readonly, then the open shall succeed.

The normative transitions into the Master XSet FSM are defined in Table 25. Once the Master XSet FSM is instantiated and enters the open XSet state, a XAM application can perform operations on the XSet. Or, it can call XSet.openExportXStream or XSet.openImportXStream. See Section 8.5.3, “Open XSet FSMs” for further information.

8.5.2.2 Entering The Abandoned State

Regardless of what state the XSet instance is in, if a XAM application calls XSet.abandon, then the XSet instance FSM shall transition to the abandoned state. When an XSet instance is in the abandoned state, all methods (except XSet.close) shall return a fatal error. Note that resources associated with open XStreams or Xiterators will not be released unless the close method of those instances is called.

The normative transitions of the abandoned state in the Master XSet FSM are defined in Table 26.

Table 26 – Abandoned State of the Master XSet FSM

Start State	Transition	Final State	Comment
Abandoned	XSet.setFieldAsBinding XSet.setFieldAsNonbinding XSet.getField<attribute> XSet.containsField XSet.deleteField XSet.<op><stype> XSet.resetManagementFields XSet.openImportXStream XSet.openExportXStream XSet.openFieldIterator	Abandoned	All methods shall return a fatal error.
Abandoned	XSet.close	NULL	Shall free the XSet instance and release all resources associated with it, regardless of completion status.

8.5.2.3 Entering the Corrupt State

A XAM application cannot close an XSet instance when it is in the corrupt state (i.e., XSet.close shall not succeed). When an XSet instance is in the corrupt state, all methods (except XSet.abandon) shall return a fatal error. To close an XSet instance that is in the corrupt state, a XAM application must first abandon the XSet instance (XSet.abandon), transition to the abandoned state, and then close the XSet instance (XSet.close).

No specific API calls exist that allow a XAM application to intentionally enter the corrupt state. The XSystem decides when an XSet instance enters a corrupt state. The decision to place an XSet instance in a corrupt state is an implementation detail of the XSystem and beyond the scope of this specification. In the following example, multiple XSystem instances accessing the same XSystem cause the XSet instance to enter a corrupt state.

Example: In some XAM Storage Systems, multiple XSystem instances may access the same XSystem, including the same XSet. In this situation, multiple XSet instances may be operating on the same XSet. In general, if the XSet is modified by a third party when the XAM application has an open XSet instance to the same XSet, it generates a non-fatal distributed access error. See Section 8.6, “Distributed Access to the Same XSet” for additional information.

The normative transitions of the corrupt state in the Master XSet FSM are defined in Table 27.

Table 27 – Corrupt State of the Master XSet FSM

Start State	Transition	Final State	Comment
Corrupt	XSet.setFieldAsBinding XSet.setFieldAsNonbinding XSet.getField<attribute> XSet.containsField XSet.deleteField XSet.<op><stype> XSet.resetManagementFields XSet.openImportXStream XSet.openExportXStream XSet.close XSet.openFieldIterator	Corrupt	All methods shall return a fatal error.
Corrupt	XSet.abandon	Abandoned	Shall cause the XSet instance to enter the abandoned state.

8.5.2.4 Performing Generic Operations on an Open XSet

The normative transitions describing operational effects on open XSets in the Master XSet FSM are defined in Table 28.

Table 28 – Generic Operation Effects on Open XSets in the Master XSet FSM

Start State	Transition	Final State	Comment
Open XSet	XSet.setFieldAsBinding(ok,non-fatal) XSet.setFieldAsNonbinding(ok,non-fatal) XSet.getField<attribute>(ok,non-fatal) XSet.containsField(ok,non-fatal) XSet.deleteField(ok,non-fatal) XSet.<op><stype>(ok,non-fatal) XSet.resetManagementFields(ok,non-fatal) XSet.[async]Commit(ok,non-fatal) XSet.openFieldIterator(ok,non-fatal)	Open XSet	For completion status of either success or non-fatal error, the Master FSM shall stay in the open XSet state. See Section 8.5.3, “Open XSet FSMs” for additional information. The specific state within the open XSet FSM depends on the mode that was used when creating the XSet instance.
	XSet.openExportXStream(non-fatal) XSet.openImportXStream(non-fatal)	Open XSet	Stay in the open XSet state
	XSet.openExportXStream(ok)	Export	On completion, shall transition to the export state to begin exporting the XSet.
	XSet.openImportXStream(ok)	Import	On completion, shall transition to the import state to begin importing an XSet using the canonical XSet format.

Table 28 – Generic Operation Effects on Open XSets in the Master XSet FSM

Start State	Transition	Final State	Comment
Open XSet	XSet.setFieldAsBinding(fatal) XSet.setFieldAsNonbinding(fatal) XSet.<attribute>(fatal) XSet.containsField(fatal) XSet.deleteField(fatal) XSet.<op><stype>(fatal) XSet.resetManagementFields(fatal) XSet.abandon(fatal) XSet.close(fatal)	Corrupt	A fatal error occurred while accessing the XSet. All further accesses to the XSet instance shall return an error.
	XSet.abandon(ok, non-fatal)	Abandoned	Shall cause the XSet instance to enter the abandoned state.
	XSet.close(ok)	NULL	Shall free the XSet instance and release all resources associated with it.
	XSet.close(non-fatal)	Open XSet	A non-fatal error occurred, possibly due to open XStreams.

8.5.2.5 Exporting an XSet

Exporting an XSet can only be done on an unmodified XSet. Once the FSM enters the export state, the only methods that can be used are XStream.read, XStream.tell, XSet.abandon, and XStream.close. All methods performed on an XSet in the export state shall return a non-fatal error and keep the Master XSet FSM in the export state. See Section 8.8, “XSet Import and Export” for a more detailed description of the export process. The export process is begun by calling XSet.openExportXStream followed by XStream.read (potentially multiple times) to retrieve the canonical description of the XSet. Once the entire XSet is fully exported, XStream.read returns a unique value to indicate that all bytes in the canonical description have been read. If XStream.read or XStream.tell return a non-fatal error, then the method may be retried, and the Master XSet FSM remains in the export state.

The XAM application would then normally call XStream.close once the export completes. If the entire XSet was exported, then XStream.close will complete successfully. See the [XAM-C-API] and [XAM-JAVA-API] for additional information. At any time, the export XStream may be closed. If XStream.close is successful, no changes to the XSet shall occur. No changes shall occur because an export effectively creates a transient (temporary) XStream that does not change the XSet and is gone when the export completes. If XStream.close completes with a fatal error, the Master XSet FSM shall transition to the corrupt state.

The normative transitions of the export state in the Master XSet FSM are defined in Table 29.

Table 29 – Export State of the Master XSet FSM

Start State	Transition	Final State	Comment
Export	XSet.setFieldAsBinding XSet.setFieldAsNonbinding XSet.getField<attribute> XSet.containsField XSet.deleteField XSet.<op><stype> XSet.resetManagementFields XSet.openImportXStream XSet.openExportXStream XSet.close XSet.openFieldIterator XStream.seek XStream.[async]Write	Export	All methods shall return a non-fatal error.
	XStream.[async]Read(ok, non-fatal) XStream.tell(ok, non-fatal) XStream.[async]Close(non-fatal)	Export	See Section 8.8.3, “Import and Export XStream Instance FSMs” for additional information.
	XStream.[async]Close(ok)	OpenXSet	See Section 8.8.1, “XSet Export Process” for additional information. The specific state within the open XSet FSM depends on the mode that was used when creating the XSet instance.
	XStream. [async]Close(fatal) XStream.[async]Read(fatal) XStream.tell(fatal) XSet.abandon(fatal)	Corrupt	See Section 8.8.3, “Import and Export XStream Instance FSMs” for additional information.
	XSet.abandon(ok, non-fatal)	Abandoned	Shall cause the XSet instance to enter the abandoned state.

Note: XSet.close is not a valid transition from the export state, because there is an open XStream.

8.5.2.6 Importing an XSet

Importing an XSet can only be done on a newly created XSet. Once the FSM enters the import state, the only XSet or XStream methods that can be used are XStream.write, XStream.tell, XSet.abandon, and XStream.close. All methods performed on an XSet in the import state shall return a non-fatal error and keep the Master XSet FSM in the import state. See Section 8.8, “XSet Import and Export” for a more detailed description of the import process.

The XAM application starts the import process by calling XSet.openImportXStream followed by XStream.write (potentially multiple times), to input the canonical format byte stream for the XSet. Once the entire XSet is fully imported, the XAM application would normally call XStream.close. The XSystem shall parse, validate, and populate all fields for the XSet instance before XStream.close completes. The XSystem may perform partial validation on each XStream.write call. If any fatal errors are encountered, the XSystem may return the fatal error on completion of XStream.write and shall return the fatal error on completion of the close. If non-fatal errors were encountered on XStream.write, the operation may be retried. If non-fatal errors were encountered on XStream.close, the FSM shall transition to the open XSet state and populate the fields in the XSet instance.

If the entire XSet was imported and the XSystem encounters no fatal or non-fatal errors, then XStream.close will complete successfully. If a fatal error was returned, then the XSet transitions to the corrupt state. See the [XAM-C-API] and [XAM-JAVA-API] for additional information. On successful completion of the import, the Master XSet FSM transitions to the open XSet state. See Section 8.5.3, “Open XSet FSMs” for information on the specific XSet open state that is entered.

The normative transitions of the import state in the Master XSet FSM are defined in Table 30.

Table 30 – Import State of the Master XSet FSM

Start State	Transition	Final State	Comment
Import	XSet.setFieldAsBinding XSet.setFieldAsNonbinding XSet.getField<attribute> XSet.containsField XSet.deleteField XSet.<op><stype> XSet.resetManagementFields XSet.openImportXStream XSet.openExportXStream XSet.openFieldIterator XStream.seek XStream.[async]Read	Import	All methods shall return a non-fatal error.
	XStream.[async]Write(ok, non-fatal) XStream.tell(ok, non-fatal) XStream.[async]Close(non-fatal)	Import	See the Section 8.8.3, “Import and Export XStream Instance FSMs” for additional information.
	XStream. [async]Close(ok)	OpenXSet	The specific state within the open XSet FSM depends on the mode that was used when creating the XSet instance. See Section 8.8.3, “Import and Export XStream Instance FSMs” for additional information.
	XStream.[async]Close(fatal) XStream.[async]Write(fatal) XStream.tell(fatal) XSet.abandon(fatal)	Corrupt	See Section 8.8.3, “Import and Export XStream Instance FSMs” for additional information.
	XSet.abandon(ok, non-fatal)	Abandoned	Shall cause the XSet instance to enter the abandoned state.

Note: XSet.close is not a valid transition from the import state, because there is an open XStream.

8.5.2.7 Exiting the Master XSet FSM

To exit the Master XSet FSM, the XAM application shall call XSet.close. A successful close shall cause the FSM to be uninstantiated and the associated resources to be returned to the operating environment. The FSM shall not be uninstantiated, however, if the application calls XSystem.abandon on the parent XSystem instance.

Unless the XSet instance is in an abandoned state, an XSet.close call shall fail if there are any open XStreams (including import XStreams, export XStreams, readonly XStreams, and writeonly XStreams). Note that the import and export states, by definition, have open XStreams. An XSet instance in the abandoned state can always be closed.

8.5.3 Open XSet FSMs

8.5.3.1 Common States

The open XSet FSMs share the same common states, which are based on the following binary attributes:

- An XSet instance is either dirty or clean
  - dirty: an XSet instance with fields that have been changed since the time of open or last commit. This change includes loss of *.xset.xuid* (e.g., XSystem.openXSet in copy mode) if the changes were binding. The act of opening an XStream for write or append shall cause the XSet that contains the opened XStream to become dirty.
  - clean: defined as not dirty
- An XSet instance either has a XUID or has no XUID
  - XUID: the XSet instance has a XUID bound to it. This designation means that *xset.xuid* is present and the value is the XSet's XUID.
  - no XUID: the XSet instance does not have a XUID and does not have *.xset.xuid*. This happens because it either never had a XUID assigned to it, the XSet instance had a binding modification, and therefore will have a new XUID assigned to it on commit, or the XSet instance was opened in copy mode.

These two binary attributes thus combine to form the four valid states in an open XSet FSM: clean XUID, clean no XUID, dirty XUID, or dirty no XUID.

The state within the open XSet FSM can be derived by examining the following XSet instance fields: *.xset.xuid* and *.xset.dirty*.

- *.xset.xuid* is present in XUID states and not present in no XUID states.
- *.xset.dirty* is present in states labeled dirty and not present in states that are labeled clean. These fields are synthetic properties that are not present on the XSet when it is committed to persistent storage. Because these fields are synthetic, XAM query cannot be used on either of these fields.
- Export includes *.xset.xuid* but not *.xset.dirty,* because export is always initiated from a clean XUID state.

8.5.3.2 The Individual Open XSet FSMs

Note that the Master XSet FSM transitions to and from the import and export state are explicitly shown on the open XSet FSMs. While not shown in the figures for readability purposes, as stated previously, it is possible to exit the FSMs on a fatal error, a call to XSet.abandon, and a call to XSystem.abandon. These transitions are shown in the tables enumerating all possible transitions.

8.5.3.2.1 Readonly Open XSet FSM

When a XAM application opens an XSet in readonly mode and tries to make binding or nonbinding modifications to the XSet instance, the XSystem must return a non-fatal error. Likewise, if a XAM application tries to call XSet.commit on that XSet instance, the XSystem must also return a non-fatal error.

Figure 12 illustrates the readonly open XSet FSM.

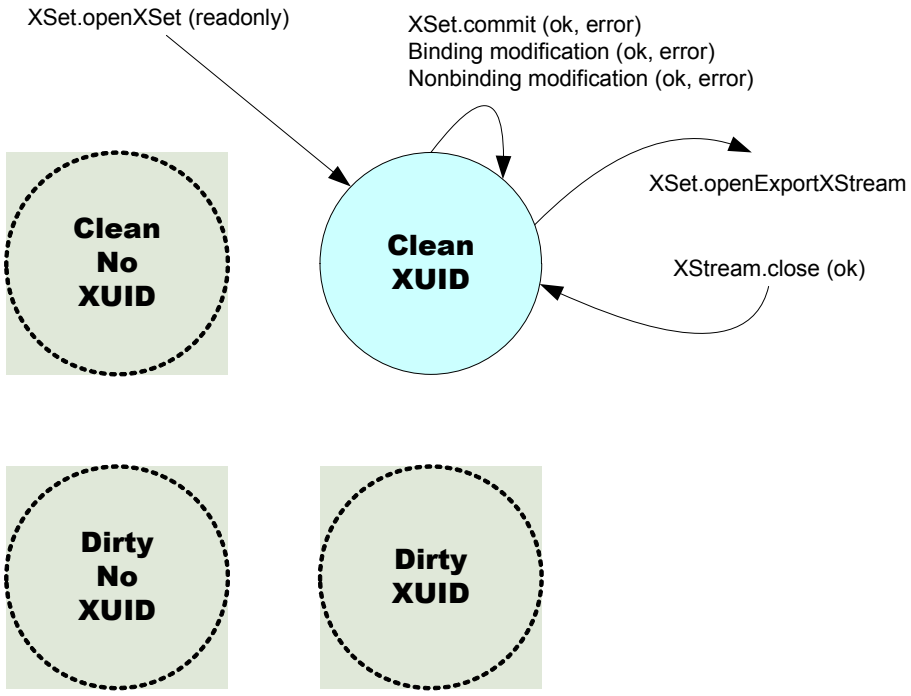

Figure 12 – Readonly Open XSet FSM

8.5.3.2.1.1 Entering the State Machine

To enter the readonly XSet FSM, an XSet must be opened in readonly mode. This open will create an XSet instance that is in the clean XUID state. Even when an XSet has a valid hold on it, it may still be opened in readonly mode. The readonly XSet FSM does not allow binding or nonbinding modifications on the XSet instance, nor does it allow XSet.commit to succeed.

The normative transitions into the readonly open XSet FSM are defined in Table 31.

Table 31 – Entrance to the Readonly Open XSet FSM

Start State	Transition	Final State	Comment
Outside of FSM (NULL)	XSystem.[async]OpenXSet	Clean XUID	The readonly open XSet FSM shall be instantiated when the method returns successfully.

8.5.3.2.1.2 Operations on an Open XSet Instance in the Clean XUID State

The normative transitions describing operational effects on open XSet instances are defined in Table 32, “Operations on an Open XSet Instance in the Clean XUID State”. Note that methods that create/change

fields but that are not specifically mentioned (e.g., job control methods) are governed by the appropriate binding or nonbinding modification entries.

Table 32 – Operations on an Open XSet Instance in the Clean XUID State

Start State	Transition	Final State	Comment
Clean XUID	XSet.[async]Commit Binding modification Nonbinding modification XSet.openImportXStream XSet.openExportXStream(non-fatal)	Clean XUID	Shall return a non-fatal error.
	XSet.getField<attribute>(ok, non-fatal) XSet.get<stype>(ok, non-fatal) XSet.containsField(ok, non-fatal)	Clean XUID	Perform the operation.
	XSet.openXStream(ok)	Clean XUID	The appropriate XStream FSM shall be instantiated, depending on the mode, when the method returns successfully.
	XSet.openExportXStream(ok)	Export	The export XStream FSM shall be instantiated when the method returns successfully.
	XSet.abandon	Abandoned	Shall cause the XSet instance to enter the abandoned state.
	XSet.getField<attribute>(fatal) XSet.get<stype>(fatal) XSet.containsField(fatal) XSet.openXStream(fatal) XSet.openExportXStream(fatal) XSet.[async]Commit(fatal) XSet.close(fatal)	Corrupt	A fatal error occurred. The XSet instance is corrupt.
	XSet.close(non-fatal)	Clean XUID	A non-fatal error occurred, possibly due to open XStreams.

8.5.3.2.1.3 Returning to Readonly FSM after Exporting an XSet

An export of an XSet can only be done on an unmodified XSet instance in the clean XUID state. Export does not change the XSet instance; thus, it returns the XSet instance to the clean XUID state after completion. To complete an export and return to the clean XUID state, the application should call XStream.close on the export XStream instance. If XStream.close is completed with a fatal error, the XSet instance shall transition to the corrupt state in the Master XSet FSM.

The normative transitions returning to the clean XUID state after export are defined in Table 33.

Table 33 – Returning to the Readonly FSM after Export

Start State	Transition	Final State	Comment
Export (not shown)	XStream.close(ok)	Clean XUID	The export completed successfully.
	XStream.close(non-fatal)	Clean XUID	The export completed unsuccessfully.
	XStream.close(fatal)	Corrupt	The export completed unsuccessfully.

8.5.3.2.1.4 Returning to Readonly FSM after Importing an XSet

Importing an XSet can only be done on a newly created XSet instance in the clean no XUID state. Because it is impossible to transition to the clean no XUID state with the readonly XSet FSM, it is impossible to perform an import in this mode.

8.5.3.2.2 Restricted Open XSet FSM

An XSet instance in restricted mode will allow only nonbinding modifications, once the XSet is in a XUID state (clean XUID or dirty XUID). Note that creating an XSet instance in restricted mode (createXSet) will always start in a “No XUID” state; thus, all edits are permitted until the XSet is committed. Trying to make a binding modification on an XSet instance that has been opened in restricted mode will result in a non-fatal error, because the XSet instance starts in the clean XUID state. Because binding modifications are not allowed once the XSet has a XUID, new XUIDs shall never be assigned on successful commit once the XUID is assigned.

Figure 13 illustrates the restricted open XSet FSM.

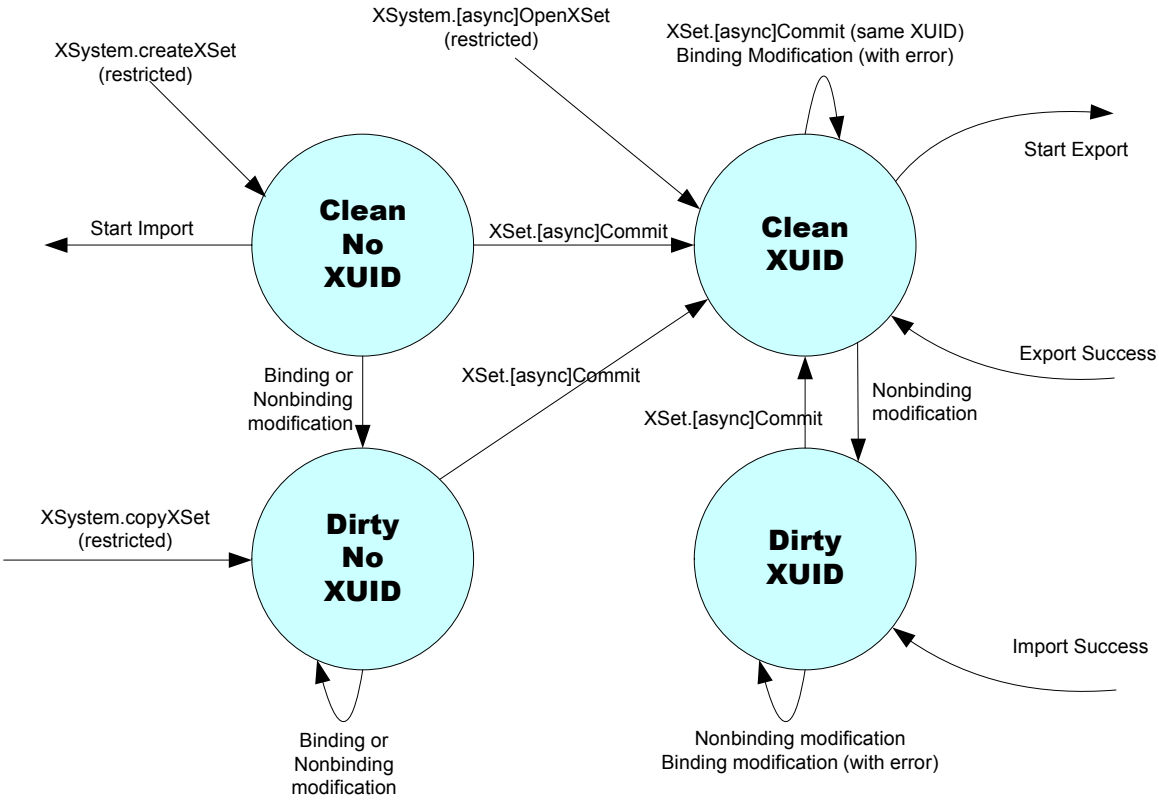

Figure 13 – Restricted Open XSet FSM

8.5.3.2.2.1 Entering the State Machine

To enter the restricted XSet FSM, an XSet must be created, opened, or copied in restricted mode. XSystem.createXSet will create an XSet instance that is in the clean no XUID state; XSystem.openXSet will create an XSet instance that is in the clean XUID state; and XSystem.copyXSet will create an XSet instance that is in the dirty no XUID state. If an XSet has a valid hold on it, trying to open the XSet in restricted mode shall result in a non-fatal error; it shall not result in an error to copy an XSet on hold, however.

The normative transitions into the restricted open XSet FSM are defined in Table 34.

Table 34 – Entrance to the Restricted Open XSet FSM

Start State	Transition	Final State	Comment
Outside of FSM	XSystem.[async]CreateXSet	Clean No XUID	The restricted open XSet FSM shall be instantiated when the method returns successfully.
	XSystem.[async]OpenXSet	Clean XUID	The restricted open XSet FSM shall be instantiated when the method returns successfully.
	XSystem.[async]CopyXSet	Dirty No XUID	The restricted open XSet FSM shall be instantiated when the method returns successfully.

8.5.3.2.2.2 Operations on an Open XSet Instance in the Clean XUID State

The normative transitions describing operational effects on open XSet instances are defined in Table 35, “Operations on an Open XSet Instance in the Clean XUID State”. Note that methods that create/change fields but that are not specifically mentioned (e.g., job control methods) are governed by the appropriate binding or nonbinding modification entries.

Table 35 – Operations on an Open XSet Instance in the Clean XUID State

Start State	Transition	Final State	Comment
Clean XUID	Binding modification XSet.openImportXStream	Clean XUID	Shall return a non-fatal error.
	XSet.[async]Commit(ok)	Clean XUID	Perform the operation. Shall update *.xset.time.access* and return the XUID.
	XSet.[async]Commit(fatal)	Corrupt	Shall return a fatal error.
	Nonbinding modification	Dirty XUID	Perform the operation.
	XSet.getField<attribute>(ok) XSet.get<stype>(ok) XSet.containsField(ok)	Clean XUID	Perform the operation.
	XSet.getField<attribute>(non-fatal) XSet.get<stype>(non-fatal) XSet.containsField(non-fatal)	Clean XUID	A non-fatal error occurred; the XSet instance shall not have changed.
	XSet.openExportXStream(ok)	Export	The export XStream FSM shall be instantiated when the method returns successfully.
	XSet.abandon	Abandoned	Shall cause the XSet instance to enter the abandoned state.

Table 35 – Operations on an Open XSet Instance in the Clean XUID State

Start State	Transition	Final State	Comment
Clean XUID	XSet.getField<attribute>(fatal) XSet.get<stype>(fatal) XSet.containsField(fatal) XSet.openXStream(fatal) XSet.openExportXStream(fatal) XSet.close(fatal)	Corrupt	A fatal error occurred. The XSet instance is corrupt.
	XSet.close(non-fatal)	Clean XUID	A non-fatal error occurred, possibly due to open XStreams.

8.5.3.2.2.3 Operations on an Open XSet Instance in the Dirty XUID state

The normative transitions describing operational effects on open XSet instances are defined in Table 36, “Operations on an Open XSet Instance in the Dirty XUID State”. Note that methods that create/change fields but that are not specifically mentioned (e.g., job control methods) are governed by the appropriate binding or nonbinding modification entries.

Table 36 – Operations on an Open XSet Instance in the Dirty XUID State

Start State	Transition	Final State	Comment
Dirty XUID	XSet.[async]Commit	Clean XUID	Perform the operation. Persist the nonbinding modifications in the XSet instance to the XSet and return the existing XUID.
	XSet.[async]Commit(fatal)	Corrupt	Shall return a fatal error.
	Binding modification XSet.openImportXStream	Dirty XUID	Shall return a non-fatal error.
	Nonbinding modification	Dirty XUID	Perform the operation.
	XSet.getField<attribute>(ok) XSet.get<stype>(ok) XSet.containsField(ok)	Dirty XUID	Perform the operation.
	XSet.getField<attribute>(non-fatal) XSet.get<stype>(non-fatal) XSet.containsField(non-fatal)	Dirty XUID	A non-fatal error occurred; the XSet instance shall not have changed.
	XSet.openImportXStream	Dirty XUID	Shall return a non-fatal error.
	XSet.openExportXStream	Dirty XUID	Shall return a non-fatal error.
	XSet.abandon	Abandoned	Shall cause the XSet instance to enter the abandoned state.
	XSet.getField<attribute>(fatal) XSet.get<stype>(fatal) XSet.containsField(fatal) XSet.openXStream(fatal) XSet.close(fatal)	Corrupt	A fatal error occurred. The XSet instance is corrupt.
	XSet.close(non-fatal)	Dirty XUID	A non-fatal error occurred, possibly due to open XStreams.

8.5.3.2.2.4 Operations on an Open XSet Instance in the Clean No XUID State

The normative transitions describing operational effects on open XSet instances are defined in Table 37.

Table 37 – Operations on an Open XSet Instance in the Clean No XUID State

Start State	Transition	Final State	Comment
Clean No XUID	XSet.openExportXStream	Clean No XUID	Shall return a non-fatal error.
	XSet.[async]Commit	Clean XUID	Perform the operation. Persist the changes in the XSet instance to the XSet and assign a new XUID.
	XSet.[async]Commit(fatal)	Corrupt	Shall return a fatal error.
	Nonbinding modification	Dirty No XUID	Perform the operation.
	Binding modification	DIrty No XUID	Perform the operation.
	XSet.getField<attribute>(ok) XSet.get<stype>(ok) XSet.containsField(ok)	Clean No XUID	Perform the operation.
	XSet.getField<attribute>(non-fatal) XSet.get<stype>(non-fatal) XSet.containsField(non-fatal)	Clean No XUID	A non-fatal error occurred; the XSet instance shall not have changed.
	XSet.openImportXStream(ok)	Import	The import XStream FSM shall be instantiated when the method returns successfully.
	XSet.abandon	Abandoned	Shall cause the XSet instance to enter the abandoned state.
	XSet.getField<attribute>(fatal) XSet.get<stype>(fatal) XSet.containsField(fatal) XSet.openXStream(fatal) XSet.openImportXStream(fatal) XSet.close(fatal)	Corrupt	A fatal error occurred. The XSet instance is corrupt.
	XSet.close(non-fatal)	Clean No XUID	A non-fatal error occurred, possibly due to open XStreams.

8.5.3.2.2.5 Operations on an Open XSet Instance in the Dirty No XUID State

The normative transitions describing operational effects on open XSet instances are defined in Table 44.

Table 38 – Operations on an Open XSet Instance in the Dirty No XUID State

Start State	Transition	Final State	Comment
Dirty No XUID	XSet.openExportXStream XSet.openImportXStream	Dirty No XUID	Shall return a non-fatal error.
	XSet.[async]Commit	Clean XUID	Perform the operation. Persist the changes in the XSet instance to the XSet and assign a new XUID.
	XSet.[async]Commit(fatal)	Corrupt	Shall return a fatal error.
	Nonbinding modification	Dirty No XUID	Perform the operation.
	Binding modification	Dirty No XUID	Perform the operation.
	XSet.getField<attribute>(ok) XSet.get<stype>(ok) XSet.containsField(ok)	Dirty No XUID	Perform the operation.
	XSet.getField<attribute>(non-fatal) XSet.get<stype>(non-fatal) XSet.containsField(non-fatal)	Dirty No XUID	A non-fatal error occurred; the XSet instance shall not have changed.
	XSet.abandon	Abandoned	Shall cause the XSet instance to enter the abandoned state.
	XSet.getField<attribute>(fatal) XSet.get<stype>(fatal) XSet.containsField(fatal) XSet.openXStream(fatal) XSet.close(fatal)	Corrupt	A fatal error occurred. The XSet instance is corrupt.
	XSet.close(non-fatal)	Dirty No XUID	A non-fatal error occurred, possibly due to open XStreams.

8.5.3.2.2.6 Returning to Restricted FSM after Exporting an XSet

An application can only export an XSet on an unmodified XSet instance in the clean XUID state. Export does not change the XSet instance; thus, it returns the XSet instance to the clean XUID state after completion. To complete an export and return to the clean XUID state, the application should call XStream.close on the export XStream instance. If XStream.close is completed with a fatal error, the XSet instance shall transition to the corrupt state in the Master XSet FSM.

The normative transitions returning to the clean XUID state after export are defined in Table 39.

Table 39 – Returning to the Restricted FSM after Export

Start State	Transition	Final State	Comment
Export (not shown)	XStream.close(ok)	Clean XUID	The export completed successfully.
	XStream.close(non-fatal)	Clean XUID	The export completed unsuccessfully.
	XStream.close(fatal)	Corrupt	The export completed unsuccessfully.

8.5.3.2.2.7 Returning to Restricted FSM after Importing an XSet

An application can only import an XSet on a newly created XSet instance in the clean no XUID state. Because it is impossible to transition to the clean no XUID state with the restricted XSet FSM, it is impossible to perform an import in this mode.

8.5.3.2.3 Unrestricted Open XSet FSM

When an application opens an XSet in unrestricted mode, it can make both binding and nonbinding modifications on the XSet instance. When it makes a binding modification to the XSet instance, the XSystem assigns a new XUID on commit. If the XSet instance is newly created and has never been committed, then the XSystem shall also assign a new XUID on commit. Finally, if the application opens an XSet in copy mode and then calls XSet.commit for the first time on the XSet instance, the XSystem assigns a new XUID. In all other cases, if only nonbinding modifications are made to an XSet instance, then the XSystem, on commit, shall return the same XUID that is currently associated with the XSet.

- The import process allows an application to change binding fields after the import XStream is closed. However, because of the binding modification, the XSystem shall remove the XUID that was associated with the XSet during the successful import process.
- XSet.resetManagementFields changes a binding property (*.xset.retention.base.starttime*); thus, it is a binding modification.

Figure 14 illustrates the unrestricted open XSet FSM.

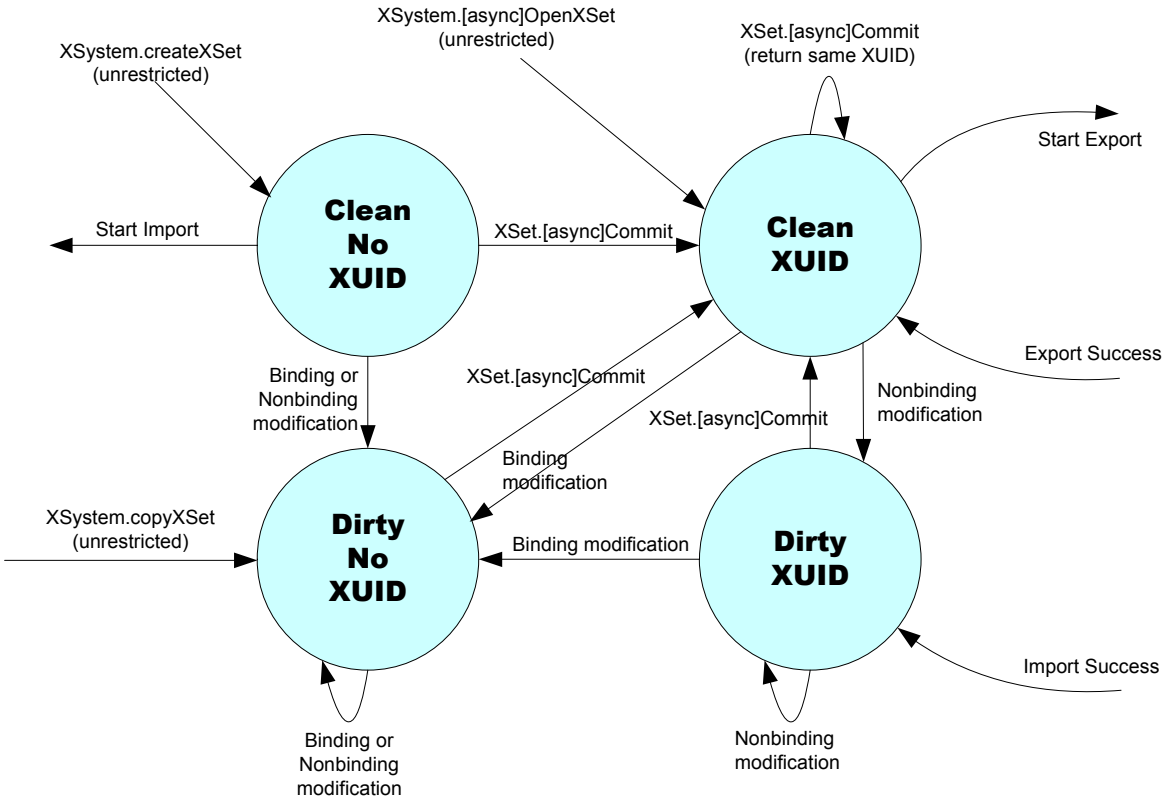

Figure 14 – Unrestricted Open XSet FSM

8.5.3.2.3.1 Entering the State Machine

The unrestricted XSet FSM can be entered in one of three ways:

- An XSet is opened in unrestricted mode, which puts the XSet instance in the clean XUID state.
- An XSet is copied in unrestricted mode, which puts the XSet instance in the dirty no XUID state.
- An XSet instance is newly created, which puts the XSet instance in the clean no XUID state.

If an application opens an XSet instance in unrestricted mode and the XSet has a valid hold on it, then XSystem.openXSet will fail. If an application copies an XSet instance in unrestricted mode and the XSet has a valid hold on it, then the hold will be removed from the newly copied XSet instance. When XSet.commit is called on the new XSet instance, the XSystem assigns a new XUID, as per standard restricted commit semantics.

The normative transitions into the unrestricted open XSet FSM are defined in Table 40.

Table 40 – Entrance to the Unrestricted Open XSet FSM

Start State	Transition	Final State	Comment
Outside of FSM (NULL)	XSystem.[async]OpenXSet in unrestricted mode	Clean XUID	The unrestricted open XSet FSM shall be instantiated when the method returns successfully.
	XSystem.[async]CopyXSet in unrestricted mode	Dirty No XUID	The unrestricted open XSet FSM shall be instantiated when the method returns successfully.
	XSystem.createXSet	Clean No XUID	The unrestricted open XSet FSM shall be instantiated when the method returns successfully.

8.5.3.2.3.2 Operations on an Open XSet Instance in the Clean XUID state

The normative transitions describing operational effects on open XSet instances are defined in Table 41.

Table 41 – Operations on an Open XSet Instance in the Clean XUID State

Start State	Transition	Final State	Comment
Clean XUID	XSet.ImportXStream	Clean XUID	Shall return a non-fatal error.
	XSet.[async]Commit(ok)	Clean XUID	Perform the operation. Shall update *.xset.time.access* and return the XUID.
	XSet.[async]Commit(fatal)	Corrupt	Shall return a fatal error.
	Nonbinding modification	Dirty XUID	Perform the operation.
	Binding modification	DIrty No XUID	Perform the operation.
	XSet.getField<attribute>(ok) XSet.get<stype>(ok) XSet.containsField(ok)	Clean XUID	Perform the operation.

Table 41 – Operations on an Open XSet Instance in the Clean XUID State

Start State	Transition	Final State	Comment
Clean XUID	XSet.getField<attribute>(non-fatal) XSet.get<stype>(non-fatal) XSet.containsField(non-fatal)	Clean XUID	A non-fatal error occurred; the XSet instance shall not have changed.
	XSet.openExportXStream(ok)	Export	The export XStream FSM shall be instantiated when the method returns successfully.
	XSet.abandon	Abandoned	Shall cause the XSet instance to enter the abandoned state.
	XSet.getField<attribute>(fatal) XSet.get<stype>(fatal) XSet.containsField(fatal) XSet.openXStream(fatal) XSet.openExportXStream(fatal) XSet.close(fatal)	Corrupt	A fatal error occurred. The XSet instance is corrupt.
	XSet.close(non-fatal)	Clean XUID	A non-fatal error occurred, possibly due to open XStreams.

8.5.3.2.3.3 Operations on an Open XSet Instance in the Dirty XUID State

The normative transitions describing operational effects on open XSet instances are defined in Table 42.

Table 42 – Operations on an Open XSet Instance in the Dirty XUID State

Start State	Transition	Final State	Comment
Dirty XUID	XSet.openImportXStream XSet.openExportXStream	Dirty XUID	Shall return an error.
	XSet.[async]Commit	Clean XUID	Perform the operation. Persist the changes in the XSet instance to the XSet and return the existing XUID.
	XSet.[async]Commit(fatal)	Corrupt	Shall return a fatal error.
	Nonbinding modification	Dirty XUID	Perform the operation.
	Binding modification	Dirty No XUID	Perform the operation.
	XSet.getField<attribute>(ok) XSet.get<stype>(ok) XSet.containsField(ok)	Dirty XUID	Perform the operation.
	XSet.getField<attribute>(non-fatal) XSet.get<stype>(non-fatal) XSet.containsField(non-fatal)	Dirty XUID	A non-fatal error occurred; the XSet instance shall not have changed.
	XSet.abandon	Abandoned	Shall cause the XSet instance to enter the abandoned state.
	XSet.getField<attribute>(fatal) XSet.get<stype>(fatal) XSet.containsField(fatal) XSet.openXStream(fatal) XSet.close(fatal)	Corrupt	A fatal error occurred. The XSet instance is corrupt.
	XSet.close(non-fatal)	Dirty XUID	A non-fatal error occurred, possibly due to open XStreams.

8.5.3.2.3.4 Operations on an Open XSet Instance in the Clean No XUID State

The normative transitions describing operational effects on open XSet instances are defined in Table 43.

Table 43 – Operations on an Open XSet Instance in the Clean No XUID State

Start State	Transition	Final State	Comment
Clean No XUID	XSet.openExportXStream	Clean No XUID	Shall return a non-fatal error.
	XSet.[async]Commit	Clean XUID	Perform the operation. Persist the changes in the XSet instance to the XSet and assign a new XUID.
	XSet.[async]Commit(fatal)	Corrupt	Shall return a fatal error.
	Nonbinding modification	Dirty No XUID	Perform the operation.

Table 43 – Operations on an Open XSet Instance in the Clean No XUID State

Start State	Transition	Final State	Comment
Clean No XUID	Binding modification	DIrty No XUID	Perform the operation.
	XSet.getField<attribute>(ok) XSet.get<stype>(ok) XSet.containsField(ok)	Clean No XUID	Perform the operation.
	XSet.getField<attribute>(non-fatal) XSet.get<stype>(non-fatal) XSet.containsField(non-fatal)	Clean No XUID	A non-fatal error occurred; the XSet instance shall not have changed.
	XSet.openImportXStream(ok)	Import	The export XStream FSM shall be instantiated when the method returns successfully.
	XSet.abandon	Abandoned	Shall cause the XSet instance to enter the abandoned state.
	XSet.getField<attribute>(fatal) XSet.get<stype>(fatal) XSet.containsField(fatal) XSet.openXStream(fatal) XSet.openImportXStream(fatal) XSet.close(fatal)	Corrupt	A fatal error occurred. The XSet instance is corrupt.
	XSet.close(non-fatal)	Clean No XUID	A non-fatal error occurred, possibly due to open XStreams.

8.5.3.2.3.5 Operations on an Open XSet Instance in the Dirty No XUID State

The normative transitions describing operational effects on open XSet instances are defined in Table 44.

Table 44 – Operations on an Open XSet Instance in the Dirty No XUID State

Start State	Transition	Final State	Comment
Dirty No XUID	XSet.openExportXStream XSet.openImportXStream	Dirty No XUID	Shall return a non-fatal error.
	XSet.[async]Commit	Clean XUID	Perform the operation. Persist the changes in the XSet instance to the XSet and assign a new XUID.
	XSet.[async]Commit(fatal)	Corrupt	Shall return a fatal error.
	Nonbinding modification	Dirty No XUID	Perform the operation.
	Binding modification	Dirty No XUID	Perform the operation.
	XSet.getField<attribute>(ok) XSet.get<stype>(ok) XSet.containsField(ok)	Dirty No XUID	Perform the operation.
	XSet.getField<attribute>(non-fatal) XSet.get<stype>(non-fatal) XSet.containsField(non-fatal)	Dirty No XUID	A non-fatal error occurred; the XSet instance shall not have changed.

Table 44 – Operations on an Open XSet Instance in the Dirty No XUID State

Start State	Transition	Final State	Comment
Dirty No XUID	XSet.abandon	Abandoned	Shall cause the XSet instance to enter the abandoned state.
	XSet.getField<attribute>(fatal) XSet.get<stype>(fatal) XSet.containsField(fatal) XSet.openXStream(fatal) XSet.close(fatal)	Corrupt	A fatal error occurred. The XSet instance is corrupt.
	XSet.close(non-fatal)	Dirty No XUID	A non-fatal error occurred, possibly due to open XStreams.

8.5.3.2.3.6 Returning to Unrestricted FSM after Exporting an XSet

An application can only export an XSet on an unmodified XSet instance in the clean XUID state. Export does not change the XSet instance; thus, it returns the XSet instance to the clean XUID state after completion. To complete an export and return to the clean XUID state, the application should call XStream.close. If XStream.close is completed with a fatal error, the XSet instance shall transition to the corrupt state in the Master XSet FSM.

The normative transitions returning to the clean XUID state after export are defined in Table 45.

Table 45 – Returning to the Unrestricted FSM after Export

Start State	Transition	Final State	Comment
Export (not shown)	XStream.close(ok)	Clean XUID	The export completed successfully.
	XStream.close(non-fatal)	Clean XUID	The export completed unsuccessfully.
	XStream.close(fatal)	Corrupt	The export completed unsuccessfully.

8.5.3.2.3.7 Returning to Unrestricted FSM after Importing an XSet

An application can only import an XSet on a newly created XSet instance in the clean no XUID state. After a successful import, the XSet instance will be in a dirty XUID state. The import model allows an application to change binding fields before committing the XSet, but standard semantics apply (e.g., if it makes a binding modification, the XSystem removes the imported XUID, which removes the XUID field, and assigns a new XUID on commit).

The normative transitions returning to the dirty XUID state after import are defined in Table 46.

Table 46 – Returning to the Unrestricted FSM after Import

Start State	Transition	Final State	Comment
Import (not shown)	XStream.close(ok)	Dirty XUID	The import completed successfully.
	XStream.close(non-fatal)	Dirty XUID	The export completed unsuccessfully.
	XStream.close(fatal)	Corrupt	The export completed unsuccessfully.

8.5.4 Summary of XSet System Fields in each XSet Instance State

Table 47 shows the XSet system fields that shall be present by XSet Instance State. In this table, Yes means that the field is present, No means that the field is not present, Maybe means that the field may be present, and N/A means that all methods that can access the fields return a non-fatal error; thus, whether a field is present does not apply. See Section 8.1, “XSet Behavior” for additional information.

Table 47 – XSet System Field Presence by XSet Instance State

XSet Field	Clean No XUID	Dirty No XUID	Clean XUID	Dirty XUID	Import	Export	Corrupt	Abandon
.xset.time.creation	Yes	Yes	Yes	Yes	N/A	N/A	N/A	N/A
.xset.time.xuid	No	No	Yes	Yes	N/A	N/A	N/A	N/A
.xset.time.commit	No	Maybe	Yes	Yes	N/A	N/A	N/A	N/A
.xset.time.access	No	No	Yes	Yes	N/A	N/A	N/A	N/A
.xset.time.residency	No	No	Yes	Yes	N/A	N/A	N/A	N/A
.xset.dirty	No	Yes	No	Yes	N/A	N/A	N/A	N/A
.xset.xuid	No	No	Yes	Yes	N/A	N/A	N/A	N/A

8.6 Distributed Access to the Same XSet

Distributed access refers to the parallel (concurrent) reading, writing, and deleting of common data (see Figure 15, “Abstract XSet Distributed Access Model”). Concurrency issues can arise when common data is accessed and edited in a distributed fashion. Because distributed edits of common data structures can conflict, it is important to define the semantics describing how these conflicts are resolved when the data is saved.

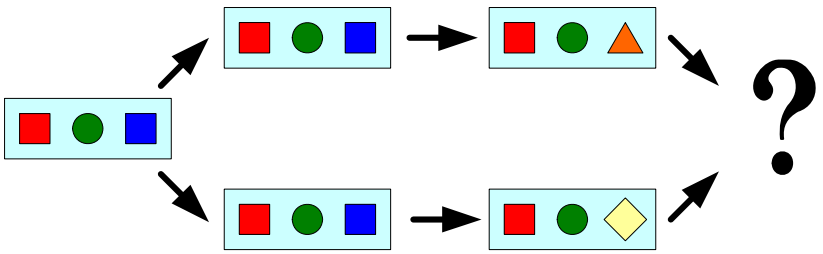

Figure 15 – Abstract XSet Distributed Access Model

In XAM, these concurrency issues can manifest when two XAM applications (or two processes/threads within the same XAM application) open an XSet (by opening its XUID) at the same time. This open causes an XSet instance to be created. In Figure 16, Application A and Application B are opening the same XSet and creating two separate XSet instances. This figure shows a simple conflict case. When each

application modifies, creates, or deletes one or more nonbinding fields within its XSet instance and tries to commit the change to the XSystem, a conflict occurs because the changes are to the same fields.

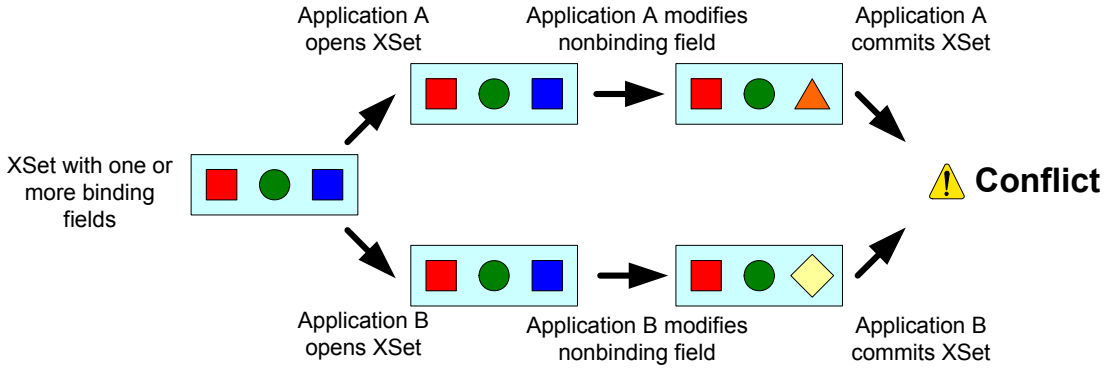

Figure 16 – XSet Distributed Access Example

All changes to the XSet before the commit by Application A are not visible to Application B. Concurrency issues occur when one application performs a commit operation while only changing, adding, or deleting nonbinding fields, because this does not change the XUID. If binding fields are changed, added, or deleted, a new XSet is created, per the Unrestricted Open XSet FSM. Thus, if both applications are editing the same XSet and have modified a binding field, no conflict occurs.

For this reason, a concurrency conflict occurs when an XSystem instance detects that an XSet instance's access to an XSet (or XSet access) has been changed since the XSet instance was created.

An XSet access is defined, for the purposes of concurrency, as follows:

- reading, changing, creating, or deleting a field associated with the XSet
- commiting or deleting an XSet
- placing or releasing a hold on an XSet

An XSet is defined to have changed when an operation causes any of the fields in the XSet, except *.xset.time.access*, to change. Note that changing field values or attributes in the XSet instance does not change values in the XSet. The only methods that change the XSet are XSet.commit, XSystem.deleteXSet, XSystem.holdXSet, and XSystem.releaseXSet. While XSystem.openXSet changes *.xset.time.access*, for the purposes of concurrency, this time change is not considered a change to the XSet. An XSystem is allowed to return an access time for the current XSet instance, rather than the most up-to-date access time of the XSet.

After an XSet instance is created, it shall be able to detect that an XSet has been changed in the XSystem.

8.6.1 Design Goals and Derived Semantics

The concurrency model in XAM is based on the following design goals:

- XAM 1.0 does not support explicit locks. If a XAM application wishes to ensure exclusive access to an XSet, the mechanism to enforce this is outside the scope of the XAM 1.0 specification.
- XAM VIMs may implement transparent use of scratch memory to store, retrieve, or cache intermediate values or partial values of an XSet before a commit occurs. If the actual XSet is modified, the XAM VIM is not required to update any cached data associated with the XSet. As

mentioned previously, the XAM VIM is required to be able to detect the conflict if it accesses the XSet in the back-end storage (i.e., the XSystem).

- When a concurrency conflict exists, the first commit precedes other commits on the same XSet. This precedence rule avoids potential data loss scenarios.

- Simple, clear conflict-error semantics are required for the XAM application and XSystem implementers. Instead of trying to merge partial conflicts within an XSet, XAM 1.0 places the entire XSet instance that detected the conflict into the corrupt state and prevents further operations on the XSet instance.

Thus, XAM 1.0 supports multiple concurrent XAM application readers with a single XAM application writer to the same XSet. However, if multiple application writers try to change the same XSet, XAM 1.0 detects that the conflict occurred but leaves it to the XAM application to recover. Additionally, even the single writer is not transparent to the readers; if the XSet is modified by the writer, XSet readers may be able to continue to read the old data for some amount of time. However, when the XSystem instance detects the conflict, the XSet instance will be placed in the corrupt state, and the XAM application reader will have to abandon and then close the XSet instance. After the XAM application re-opens the XSet, it should assume that any prior data it has read may have changed, unless it has a mechanism for detecting changes within the XSet.

More formally, the first XSet instance commit shall define what the XSet will contain. If two or more XSet instances are opened against an existing XSet, then the first one to commit any changes to the XSet shall have its changes stored in the XSet. The second (and subsequent) XSet instance, on detection of the modified XSet, shall enter the XSet instance corrupt state, causing the access that detected the conflict to return a non-fatal concurrency error and all further accesses to fail with a non-fatal concurrency error. Note that it is acceptable for VIMs that use a cache to succeed while the XSet access is within the cache. However, even for caching VIMs, once a concurrency failure is detected, all subsequent accesses shall return a failure.

Note: The XAM Library and XSystem instance cannot be committed to persistent storage. Any field edits associated with either a XAM Library instance or an XSystem instance are not persistent; they only affect the local instance. Thus, XAM Library and XSystem instances cannot suffer from these concurrency issues.

8.6.2 Use Cases

Distributed access can be divided into two groups:

- Single XSet instance concurrency, which describes concurrent access of an XSet instance by multiple threads within a single process.

- Multiple XSet instance concurrency, which describes concurrent access of a single XSet through the use of multiple XSet instances. Multiple XSet instance concurrency issues can arise in many ways; they can occur within a single process or task on a single host, within multiple processes or tasks on a single host, or within multiple processes or tasks across multiple hosts.

Note: The divergence caused by performing different edits (change and commit) on nonbinding fields on the same XSet (i.e., XSets that have the same XUID) that is stored on multiple physical XAM Storage Systems is not within the definition of a concurrency conflict. This conflict does not become an issue until the XSets converge. Convergence occurs when both XSets are accessed by the same application, or when one XSet is migrated to a XAM Storage System that contains a divergent XSet. Convergence issues that are caused by a XAM application accessing two diverged XSets on two different XSystems is beyond the scope of this specification. XAM migration is defined as the combination of an XSet export followed by an XSet import. XAM enables a XAM application performing the migration (thus causing convergence) to have the tools

to evaluate the data that has diverged and make an appropriate application-specific choice between the diverged data. See Section 8.8, “XSet Import and Export” for further information.

8.6.2.1 XSet Conflict Resolution, Example 1

Process A on the host opens an XSet instance, while process B on the same host opens an XSet instance to the same XSet (both have unrestricted access to the XSet). Process A modifies a nonbinding field on the XSet instance and commits the XSet. Process A can continue to change the XSet instance because the XSet is in sync with the XSet instance. Any edits made to the XSet instance by process B will cause XSet.commit to return a non-fatal error. Based on the state of the VIM cache (if any), tries to read or edit XSet fields in process B may also return a fatal or non-fatal error. Once a fatal error is returned, the XSet instance is transitioned to the corrupt state, and all future operations on the XSet, except for XSet.abandon and then XSet.close, return a fatal error.

8.6.2.2 XSet Conflict Resolution, Example 2

Process A opens an XSet instance in readonly mode, while process B opens the same XSet with unrestricted access. Process A is not allowed to change the fields on the XSet, thus, it cannot cause concurrency problems for process B. Note that process A is still vulnerable to process B causing changes to the XSet, which would result in process A’s XSet instance transitioning to the corrupt state when process A tries to read XSet fields.

8.6.2.3 XSet Conflict Resolution, Example 3

A single process containing two threads opens an XSet instance in each thread (both are unrestricted, and each XSet instance was opened using the same XUID). The XSet contains a field that has an existing value. Thread A modifies the field, while thread B reads the field. Thread order is undefined. XAM does not define if thread B will read the original or the modified value from the field. This type of thread synchronization is the responsibility of the application.

8.7 XSet Policy

An XSet policy is an agreement, between XAM applications and an XSystem, for a set of XAM-specified behaviors or rules that apply to the XSet. An XSystem advertises the list of XSet policy agreements, for which the XSystem is willing and obligated to abide by, to XAM applications via a list of XSystem fields or an XSystem policy list. A XAM application can only select an XSet policy agreement from the XSystem policy list. The XSet policy agreement is finalized once the XAM application commits the XSet instance containing the selected XSet policy.

The XAM specification of an XSet policy is an XSet system property with XSet policy methods for creating, updating, and removing an XSet policy property. An XSet policy property value is referred to as the policy name. The XAM specification of an XSystem policy list is a set of XSystem properties, where each element of the set is an XSystem property with a policy name value that is identical to the policy name that is appended at the end of the XSystem property name. Unless otherwise specified, the XSystem policy list may be the empty set, i.e., no XSystem policy list properties exist. The attributes of the XSet policy and XSystem policy list properties are specified in Table 48.

Table 48 – XSet and XSystem Policy List Properties

Property Name	stype	Binding	Readonly
.xset.<XAM standard name>.policy	xam_string	XAM specified	XAM specified
.xsystem.<XAM standard name>.policy.list.<name>	xam_string	FALSE	TRUE

An XSet policy shall only be created or updated with a policy name found in the XSystem policy list, where the *<XAM standard name>* portion of the XSet policy matches the property names of the XSystem policy list. Unless otherwise specified, the *<XAM standard name>* portion of the XSet policy property name shall match the *<XAM standard name>* portion of the XSystem policy list property name. A XAM application may use the XIterator mechanism (see Chapter 6, "XAM Objects and Common Operations") on the XSystem to discover the valid policy names that can be used to create or update an XSet policy. Alternatively, a XAM application may simply create or update an XSet policy using the appropriate XSet policy method. If a XAM application specifies an invalid policy name (i.e., a policy name that is not included in the XSystem policy list), then the XSet policy method shall generate a non-fatal error.

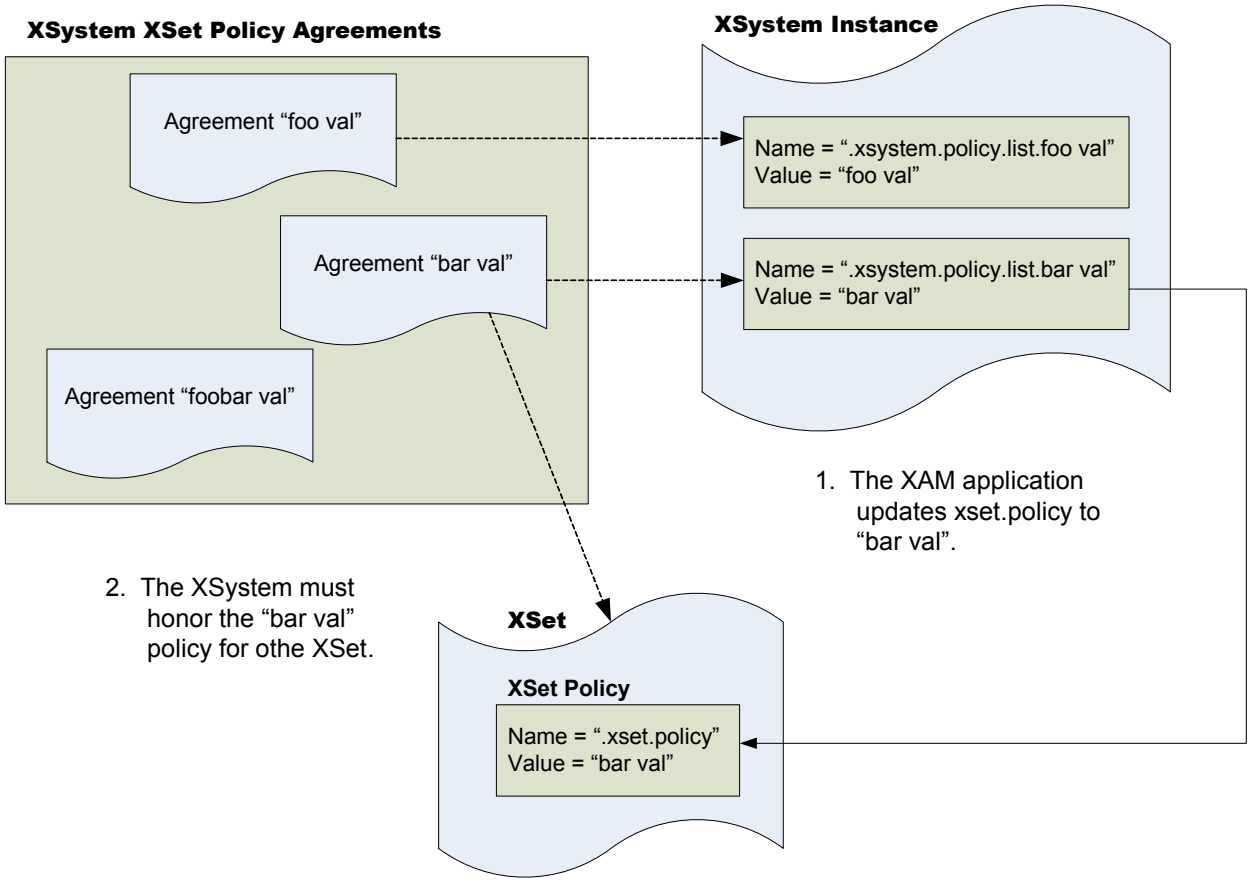

Figure 17 – Policy Relationships Between the XSet and XSystem

An XSystem may change the behavior and rules of an XSet policy agreement or create or remove an XSet policy agreement at any time. The XSystem shall advertise the XSet policy agreement alterations on any XSystem instance created after the changes are made. Conversely, the XSystem shall not advertise the XSet policy agreement alterations on any XSystem instance created before the changes are made.

The XSet policy XAM-specified behaviors and rules, complete specification of property attributes, and associated XSet policy methods are defined in Chapter 9, "XSet Management" and Chapter 11, "Security" as XSet policy pertains to the following XAM management and security XSet disciplines:

- Retention
- Deletion

- Storage
- Access control

Note: Due to the abstracted nature of an XSet policy, XAM does not include the semantics or the data definitions employed by the XAM Storage System to formulate the XSystem policy list. If a XAM Storage System administrator wants to migrate an XSet from one XSystem to another, the XAM Storage System administrator should define policy names on each XSystem, such that migration from one XSystem to another preserves the policies associated with an XSet. The XAM application responsible for the import can detect differences between the exported XSet's getActuals and the imported XSet's getActuals; however, it is beyond the scope of this specification to reconcile the conflict.

8.8 XSet Import and Export

One of the key factors in achieving long-term data persistence is the ability to move XSets between different XSystems. This ability is achieved by using the XSet import and export mechanism. This specification defines the behavior of the methods used to perform the export of an XSet from an XSystem, the methods used to perform the import of an XSet into an XSystem, and the canonical XSet data format that these methods use.

The definition of the canonical XSet format can be found in Annex B, "(normative) Canonical XSet Interchange Format". This definition describes the XAM representation of an XSet when it is outside of an XSystem. The internal representation of an XSet within an XSystem is outside of the scope of this specification. XSystem implementers are free to optimize the internal representation of an XSet.

XSet.openImportXStream opens an import XStream to the XSet instance. The application imports an XSet by using XStream.write to write the bytes of an XSet that is in canonical format into the import XStream. The import is complete when the application closes the import XStream.

XSet.openExportXStream opens an export XStream to the XSet instance. The XSet instance must be in a clean state to begin an export. The application exports the XSet from the XStream by using XStream.read to read the canonical representation of the XSet from the export XStream. The export is complete, when a read of the export XStream generates an End Of Stream indicator, and the application closes the export XStream.

8.8.1 XSet Export Process

The XSet export process shall create an XStream representation of the contents of an XSet. The contents of this export XStream shall be in the canonical XSet format. This XStream is created on a clean (unmodified) XSet instance. The XSystem shall generate a non-fatal error when an application tries to open an export XStream against an XSet instance that is not in a clean state. The existing XSet will be unmodified by creating the export XStream. Note that XStream.seek shall return a non-fatal error when used on an export XStream.

To ensure data integrity and interoperability, the following export rules shall be enforced by the XSystem. An XSystem:

- Shall export all application fields
- Shall export all XAM standard system fields
- Shall export all vendor system fields not created by the exporting XSystem
- May export vendor system fields understood by the exporting XSystem

Exporting an XSet

Exporting an XSet shall be a multi-step process using XSet.openExportXStream, XStream.read, and XStream.close. The following steps shall be required:

1. Open an existing XSet instance using XSystem.openXSet.
2. Call XSet.openExportXStream to begin the export. An XStream handle will be returned. Opening an export XStream shall put the XSet instance into the export state. When an XSet instance is in this state, it may not be modified in any way (i.e., all methods that create, change, or delete shall return a non-fatal error).
3. Read the canonical representation of the XSet from the XStream using XStream.read. XStream.read should be called over and over until it generates an End of Stream indicator.
4. Close the export XStream using XStream.close. This close shall return the XSet to the state it was in before the export operation.

On successful export, the standard system fields (see Section 8.2.3, “Normative XSet Fields”) shall not be modified.

8.8.2 XSet Import Process

The XSet.openImportXStream method is called on a newly created XSet instance. Trying to import into anything but a newly created XSet instance shall result in a non-fatal error. XSet.openImportXStream opens an import XStream on the XSet instance. When an XSet instance has an open import XStream, it may not be edited (i.e., all methods to create, change, or delete shall return a non-fatal error). The application imports an XSet by using XStream.write to write the bytes of an XSet that is in canonical format into the import XStream; this process results in that XSet instance being fully populated and in a dirty XUID state. XStream.seek shall return a non-fatal error when used on an import XStream. The import is complete when the application closes the import XStream.

If the import is successful (i.e., the canonical representation was complete and it was successfully transferred to the XStream), the data will be validated when the import XStream is closed. If it is invalid, the XSet will enter a corrupt state. On success, the XSet instance will have the XUID (as defined in the canonical representation) assigned to it; if the binding fields change in the XSet after the import, the XUID will be removed and a new XUID shall be assigned on commit of the XSet instance. If the XSet already exists in the XSystem, the application may open the existing XSet and compare the two and resolve any conflicts, if needed. This pattern allows the application to do a more sophisticated import, where it can choose to commit (or not commit) the XSet without any consequences.

To ensure data integrity and interoperability, the following import rules shall be enforced by the XSystem:

- The imported XSet shall retain all fields found in the canonical XSet data, although the XAM Storage System is allowed to delete that XAM Storage System’s vendor fields.
- The importing XSystem shall change those nonbinding XAM standard system fields that are required to change (e.g., *.xset.time.commit* may change; *.xset.time.residency* shall change).
- The importing XSystem may change nonbinding vendor system fields understood by the importing XSystem.
- All fields and field contents exist on the XSet instance, but the XSet instance has not yet been committed to the XSystem. This requirement allows the importing application to decide whether to commit, change and commit, or discard the XSet instance.
- If an XSet with the same XUID exists in the importing XAM Storage System, the existing XSet shall be overwritten with the imported XSet on commit of the imported XSet. An XSystem may generate a fatal or non-fatal error on commit, if a field update would result in a violation of specialized field

rules (e.g., cause a retention field to decrease), or if a binding policy is part of the XSet and cannot be enforced on the importing XSet.

Importing an XSet

Importing an XSet is a multi-step process using XSet.openImportXStream, XStream.write, and XStream.close. The following steps shall be required:

1. Create a new XSet using XSystem.createXSet.
2. Call XSet.openImportXStream to allow the import. An import XStream handle is returned. Opening an import XStream shall put the XSet instance into the import state. When an XSet instance is in this state, it may not be modified in any way (i.e., all methods that create, change, or delete shall return a non-fatal error).
3. Write the canonical representation of the XSet to the XStream using XStream.write.
4. Close the XStream using XStream.close. If XStream.close is successful, the XSystem shall validate the imported contents and set up the XSet state accordingly. More details on validation can be found below.

When the stream closes, the XSystem shall validate the canonical XSet data. If the canonical XSet data is invalid, then the XSet instance shall enter a corrupted state. The XSystem implementor must determine if a canonical XSet is valid. Note that an XSet instance in a corrupted state cannot be recovered, and all operations, except abandon, shall fail.

If the data is valid, then the XSet instance shall enter the dirty state (meaning it has been modified), and *.xset.xuid* shall be populated with the XUID from the imported data. The fields on the XSet shall match the original XSet. At this point, the policy fields must be evaluated. Sufficient information shall be encoded in the canonical format to allow the actual values of the policy elements to be determined. If a policy provides a value for any GetActual call for a standard field in the XSet, then that policy and value shall be part of the exported XSet. Policies should be mapped to the policies of the importing XSystem (before XStream.close returns control to the application) as follows:

- Nonbinding policies (name or value): These fields can be modified by the XSystem to match what is supportable by the XSystem; however, this change shall not violate policy behavior (e.g., reduce the retention period).
- Binding named policy fields:
  - The XSystem has a named policy that matches the description, as specified in the canonical format. The reference is correct and shall not cause the import to fail.
  - The XSystem has a named policy that does not match the description, as specified in the canonical format. The import shall fail.
  - The XSystem has no policy by that name. The XSystem can create a policy by that name with the value described in the canonical format, or the import shall fail.
- Binding value policy fields: If the XSystem cannot support the policy, then the XSystem shall cause the import to fail. Otherwise, the import shall succeed.

The XSet instance (if not corrupted) will be valid and have valid policies at this point. The XSet instance will not have been committed; it may then be committed or discarded as any normal XSet instance would be. Unless there is a change to binding fields in the XSet, the XUID in *.xset.xuid* will be returned on commit of the XSet instance.

If the XSet already exists in the XSystem, the application can open the existing XSet and compare the two so that any conflicts can be resolved, if needed. Comparing the two allows the application to do a more sophisticated import, where it can choose to commit (or not commit) the XSet without any consequences.

On successful import, the standard system fields shall be modified as shown in Table 49.

Table 49 – XSet System Field Modification on Import

Field Name	Modified	New Value
.xset.time.creation	No	N/A
.xset.time.xuid	No	N/A
.xset.time.commit	No	N/A
.xset.time.access	Yes	Time XSet was imported
.xset.time.residency	Yes	Time XSet was imported
.xset.xuid	Yes	XUID from imported XSet
.xset.dirty	Yes	Set to TRUE

When the XSet instance that contains the imported XSet is committed, *.xset.time.access* and *.xset.time.residency* shall be set to the current time.

8.8.3 Import and Export XStream Instance FSMs

See Section 8.5, “XSet Instance Finite State Machine (FSM)” for additional information regarding the import and export states of the Master XSet FSM. Policies include the following:

- XStream.close in import corrupt state shall always return a fatal error.
- XStream.close(fatal) on import causes the entire XSet to transition to the corrupt state.
- In import corrupt state, all methods shall return a fatal error except XStream.abandon.
- XStream.close removes the XStream from the XSet.

See Figure 18, “Export XStream Instance FSM” and Figure 19, “Import XStream Instance FSM” for an illustration of these export and import functions.

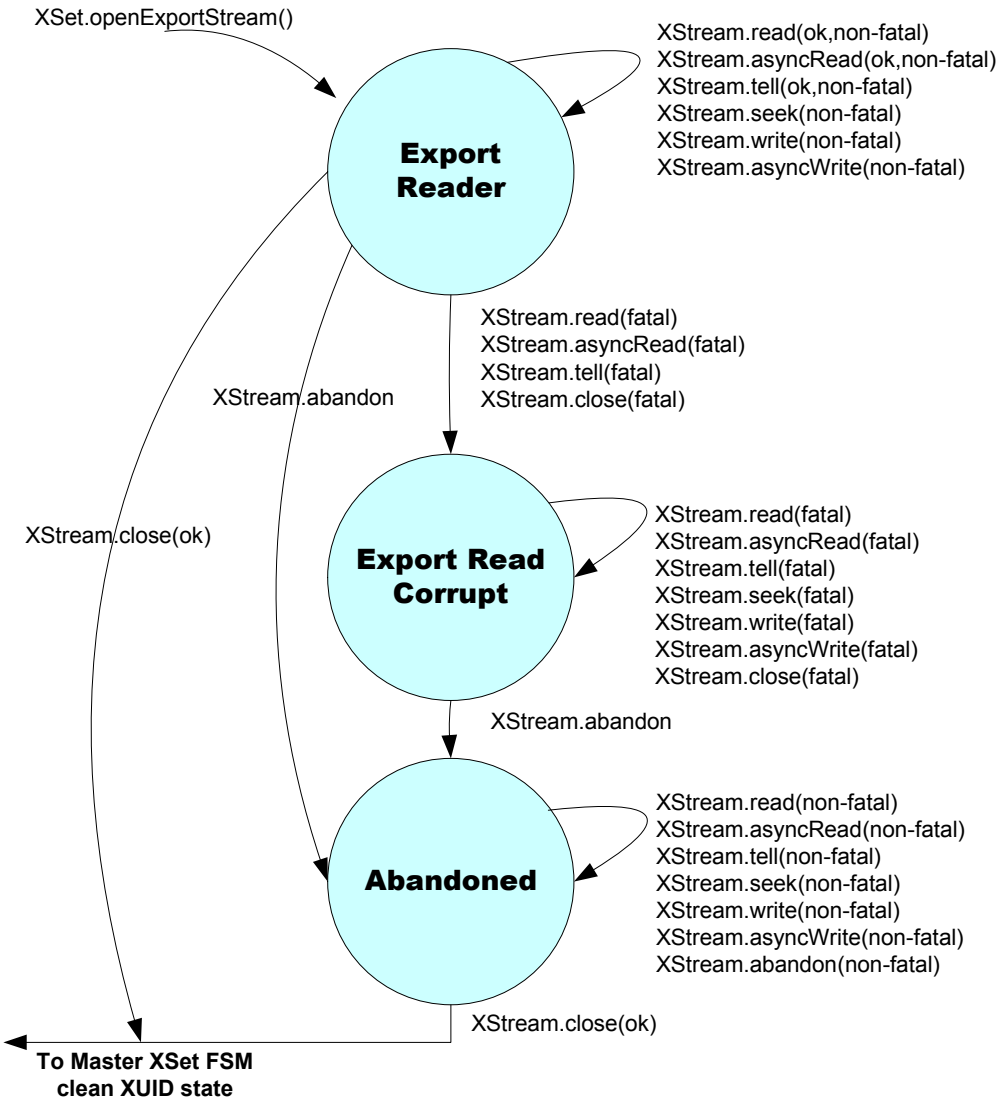

Figure 18 – Export XStream Instance FSM

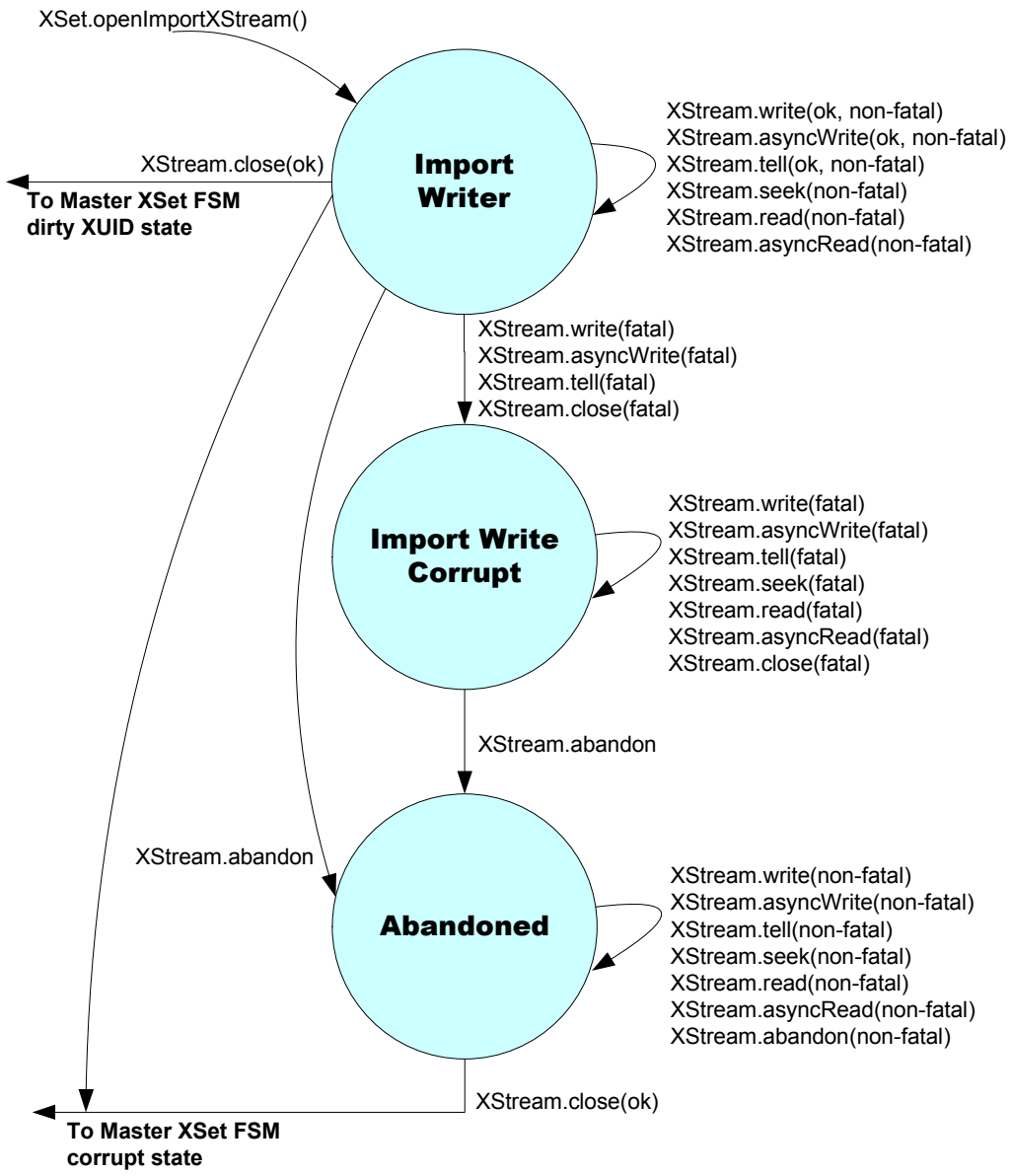

Figure 19 – Import XStream Instance FSM

8.8.4 XSet Canonical Format

The main goals for the canonical format are interoperability and performance. Interoperability is required so that XSets can be moved between different XSystems without a loss of data. Good performance is required to enable XSystems to efficiently export or import large numbers of XSets in a reasonable amount of time. Secondary goals are the use of existing standards and parsers, where possible, and the ability to do offline inspection of exported XSets. The format should also support the ability to describe multiple XSets within the same data.

With these design goals in mind, the canonical XSet format shall be packaged in two main parts: an XML document describing the policies, properties and streams of the XSet, or XSet “manifest”, and the binary representation of the streams. Since properties are compact, they shall be fully defined in the XML document. Since XStreams can be rather large, only the attributes of the stream shall be included in the XML document; the actual contents shall be outside the XML document in a separate part of the package.

The format of the package shall adhere to the XML-binary Optimized Packaging (XOP) format [XOP]. With this format, the binary data is attached as MIME attachments following the XML document. The MIME attachments shall conform to the MIME Multipart/Related Content-type specification as defined in [RFC 2387]. A table of contents attachment shall be added as the first attachment to list the offsets of each of the XStream's binary data. This attachment allows for parallel access to the XStreams if desired. The format of the XML document shall allow multiple XSets to be included in one canonical XSet package, although the current version of the specification supports only import and export of a single XSet at a time. If the XSet does not contain any XStreams, then no MIME attachments shall follow the XML document. As per the XOP format, the order of the MIME parts shall not be considered significant, except for the purpose of determining the root MIME part.

Using XML enables the use of standard XML parsers. However, XML is not designed to handle binary data efficiently; it requires Base64 encoding, thus expanding the size of the data by 33%. By using XOP, the applications can use existing XML tools to parse and understand the XSet structure, while not requiring them (and the XSystem) to Base64 encode/decode the XStreams. The format does not include a digital signature, compression, encryption, or cryptographic XSet integrity.

This specification uses an XML Schema Definition (XSD) to define the format. The use of schema allows for better validation of the contents if applications so choose, while not impacting the performance of applications which do not. The schema is found in Annex B, "(normative) Canonical XSet Interchange Format".

8.8.4.1 XSet Manifest XML Format

The XML format is organized into two parts: policies and XSets. The policies section lists the XSystem policies that are in effect. The XSets section lists the properties and XStreams of each XSet in the manifest. The format of the XML manifest allows for multiple XSets to be included in a single package, although this version of the specification only specifies the inclusion of a single XSet in a manifest.

The root element for the XSet XML format shall be called xsets. This element shall be followed by the element policies, which shall contain the XSystem policies that are referenced by the XSet. Each policy shall be expressed with a policy element, which in turn shall contain one or more property elements. The policy element shall contain name, type, readonly, binding, and length attributes followed by a value element containing the value of the property. These attributes and the value shall correspond to the XSystem policy. The property elements contained in the policy element shall correspond to the policy subtypes. Each property element shall have name, type, readonly, binding, and length attributes followed by a value element containing the value of the property. The attributes for the property shall contain the values of the attributes of the corresponding XSystem property.

Following the policies element shall be one or more xset elements. The xset element shall contain the properties and XStreams for a single XSet. The XSet element shall contain two sub-elements: properties and xstreams. The properties element shall contain the entries for all the properties exported for this XSet. Each property shall be defined by a property element containing the name, type, readonly, binding, and length attributes of the property. A value child element shall contain the value of the property. The attributes for the property element shall contain the values of the attributes of the corresponding XSet properties.

The next element shall be the xstreams element, which defines the XStream information. Each XStream shall be represented by an xstream element, containing the name, type, readonly, binding, and length attributes of the property. One child element, xop:include, is mandated by XOP. It contains the content-id (in an href element), prefaced by cid: to the MIME part containing the XStream contents. The xop:Include element shall also contain an xmlns:xop attribute containing the URI of the XOP specification. The attributes for the XStream property shall contain the values of the attributes of the corresponding XStream.

8.8.4.2 The Canonical Representation Build

The first component of the canonical representation is the root part of the XOP package. As per the XOP specification, it must identify a media type of “application/xop+xml”. The “start” parameter identifies the Content-ID of the “root” MIME object. In the XAM case, this parameter may be set to any value that adheres to XOP by the exporting XSystem, but this value shall be the same as the Content-Id of the XSet manifest part of the XOP package, as described below. This element should be processed first on import. The “start-info” parameter provides additional information on the “root” object, which is the XAM manifest XML. In the XAM case, it shall be used to indicate the format of the object, XML, using the MIME type “text/xml”. The use of XML is required by XOP. A “Content-Description” entry is used to describe the contents of the package. This entry can be used to provide additional information about the contents.

The following is a sample root part of the XOP package:

```
Content-Type: Multipart/Related;boundary=MIME_boundary;
   type="application/xop+xml";
start="<canonicalxset.xml@snia.org>";
start-info="text/xml"
Content-Description=”some text”
```

The second component is the root MIME part containing the XSet manifest. As per XOP, the Content-Type parameter shall be “application/xop+xml”, and the type parameter shall be the same as the start-info parameter above. The contents of the Content-ID parameter shall match the start parameter from the root part of the XOP package as described above. The XSet manifest is contained in this MIME part.

The following is an example of this part of the XOP package:

```
Content-Type: application/xop+xml;
    charset=UTF-8;
    type="text/xml"
Content-Transfer-Encoding: 8bit
Content-ID: <canonicalxset.xml@snia.org>
```

The next MIME part is the XSet “table of contents”, which contains the MIME content-type of “text/text”, content-transfer-encoding of “text”, and a content-id of “<TOC>”. The contents of this part are a list of text lines that contain the offsets of the XStreams. The format of each line is:

```
Offset of [XSet XUID]:[content-id of MIME part with contents]:[offset in bytes]
```

Where,

- [XSet XUID] is the text representation of the XUID
- [content-id of MIME part with contents] contains the content-id of the relevant MIME part
- [offset in bytes] is the offset in bytes from the beginning of the package to the start of the binary data in the package

The remaining MIME parts are the XStreams’ contents, where each XStream shall be contained in a separate MIME part. The Content-Type parameter shall contain the MIME type for the XStream data. The Content-Transfer-Encoding shall represent the encoding used (e.g., binary). The Content-ID shall match the href used in the XML manifest. The use of binary encoding is strongly recommended to reduce the package size. If binary is not possible, the contents should be Base64 encoded.

The choice of which encoding to use depends on the ability of the XSystem to set the MIME boundary string. The MIME boundary string shall be any set of characters not used in any of the MIME part contents. If the exporting system cannot meet this requirement, it shall encode the contents using Base64 encoding, so that it can set the MIME boundary string to a string not occurring in the contents.

8.8.4.3 XSet Export Example

The example in Table 50 shows how an XSet would appear in the canonical format. For this example, we have an XSystem policy and several XSet fields.

Table 50 – Example XSystem Policy Property

Name	Type	Binding	Readonly	Length	Value
org.snia.policy.test	application/vnd.snia.xam.int	FALSE	TRUE	8	1

Table 51 lists the XSet properties and XStreams used for this example.

Table 51 – Example XSet for Export

Name	Type	Binding	Readonly	Length	Value
org.snia.property.test	application/vnd.snia.xam.boolean	FALSE	TRUE	1	1
org.snia.xuid.property	application/vnd.snia.xam.xuid	TRUE	TRUE	8	XUID1234
org.snia.stream.property	image/jpeg	FALSE	TRUE	20000	

Note: "XUID1234" is not a proper value for a XUID. It is listed for pedagogical purposes only and should be interpreted as the base64 encoding of a valid XUID.

The XSet would have the following XML representation:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--Sample XAM XSet -->
<xsets xsi:schemaLocation="http://www.snia.org/2007/xam/export
XAMCanonicalXSetDefinition.xsd" xmlns="http://www.snia.org/2007/xam/export"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xop="http://www.w3.org/2004/
08/xop/include">
    <version>1.0.0</version>
    <policies>
        <policy name=".xsystem.management.policy.list.default" type="application/
vnd.snia.xam.string"
        readOnly="true" binding="false" length="7">
            <string>testing</string>
            <property binding="false" type="application/vnd.snia.xam.int" length="8"
readOnly="true" name="org.snia.policy.test">
                <integer>1</integer>
            </property>
        </policy>
    </policies>
    <xset>
        <properties>
            <property binding="false" type="application/vnd.snia.xam.boolean" length="1"
readOnly="true" name="org.snia.property.test">
                <boolean>true</boolean>
            </property>
            <property binding="true" type="application/vnd.snia.xam.xuid" length="8"
readOnly="true" name="org.snia.xuid.property">
                <string>XUID1234</string>
```

```
            </property>
        </properties>
        <xstreams>
            <xstream binding=”false” type=”mime/jpeg” length=”20000” readOnly=”true”
name=”org.snia.stream.property”>
                <xop:Include href=”cid:org.snia.stream.property@snia.org” xmlns:xop=”http://
www.w3.org/2004/08/xop/include”/>
            </xstream>
        </xstreams>
    </xset>
</xsets>
```

The XOP package for this would look like this:

```
Content-Type: Multipart/Related;boundary=MIME_boundary;
    type="application/xop+xml";
    start="<canonicalxset.xml@snia.org>";
    start-info="text/xml"
Content-Description: A sample XSet export document with one XStream

--MIME_boundary
Content-Type: application/xop+xml;    charset=UTF-8;    type="text/xml"
Content-Transfer-Encoding: 8bit
Content-ID: <canonicalxset.xml@snia.org>

[Sample XSet XML from above omitted]

--MIME_boundary
Content-Type: text/text
Content-Transfer-Encoding: text
Content-ID: <TOC>

Offset of XUID1234:org.snia.xam.stream.property@snia.org: 300

--MIME_boundary
Content-Type: image/jpeg
Content-Transfer-Encoding: binary
Content-ID: <org.snia.stream.property@snia.org>

// binary octets for png

--MIME_boundary--
```

The MIME_boundary text shall be any sequence of characters that does not appear in the binary data that follows.

8.8.5 Annotating the Canonical Format

The XSet canonical format shall allow optional attributes and elements to be included. These attributes and elements will allow vendors to annotate existing XSet properties and XStreams. In addition, they shall neither be XAM-defined attributes or elements nor considered as extensions to XAM.

Any vendor-added attributes or elements shall follow these rules:

- Import shall work even if the vendor-specific additions are not understood by the importing XSystem. An export-to-import sequence of operations shall be fully compliant with XAM, even if all of the additions are ignored and discarded on import. Ignoring vendor-specific additions shall not cause any API-visible behavior differences between the exporting XSystem and the importing XSystem.
- All added vendor annotations shall only be done in an added vendor namespace. Addition of annotations in any of the namespaces already used in the canonical format shall not be allowed.

8.9 XAM Jobs and XAM Job Control

Outside of XAM, a job typically refers to a sequence of one or more operation commands that are submitted to a system (typically in the form of a script), which are then run without any user interaction. Job control describes the mechanism by which these commands are submitted and how jobs are interacted with once they have been submitted (e.g., determining status or halting a running job).

In XAM, jobs and XAM job control follow this model. However, XAM jobs are targeted toward the sequence of commands as defined by the underlying XAM Storage System vendors. This sequencing allows the vendors to add both standard and non-standard operations to the XAM API, without requiring new API calls. An example of a job (and the only job that is standardized in this specification) is query. XAM jobs are VIM dependent. Note that standard XAM jobs (e.g., query) must have a deterministic outcome as defined in the functional specifications of that XAM job.

XAM job control uses XSets as the communication medium when running jobs, both for submitting the job and for determining status. Thus, the semantics defined for XSet manipulation can thus be re-used, providing that the interface is consistent around issues like persistence and data manipulation (e.g., field manipulation). Thus, fields on an XSet can be used to identify which command to run, to determine job status, and to control the job. Job control XSets need not be newly created; a XAM application can simply add the appropriate fields to an existing XSet and submit the job to the XSystem instance.

To run a XAM job, the XSet used for job submission must have certain standardized job input fields added to it. Then, the XSet can be submitted to the job control system using a XAM standard method (XSet.submitJob). The status of the job can be evaluated by examining the fields on the XSet using standard field access methods. Note that a XAM job can be terminated at any time by using an additional dedicated API method, XSet.haltJob.

The job XSet may be optionally committed before the job has been started or after it has completed, to enable persistent storage of job input parameters and/or results. Additionally, if an XSystem supports it, the job XSet may be committed at any point in the job lifecycle. This option enables persistent storage of the job results, even if the session to the XSystem is lost (e.g., normal or abnormal close of an XSystem, then reopen the XSystem later and examine the job XSet). The job operation, itself, is unchanged by the commit operation. Note that applications must be prepared to handle errors, when committing an XSet that contains a job; not all XSystems will allow such an XSet to be committed. Also note that such a failure to commit will not halt the job; an in-progress job will continue to run even when such an error occurs. An application can determine if the XSystem supports the committing of XSets that are bound to a running job by examining the following field on the XSystem (set to TRUE if commit is supported):

.xsystem.job.commit.supported

Note: The XAM Storage System is not guaranteed to continue to process the job while the XSystem is not open. An application can determine if the XSystem supports job continuance by examining the following field on the XSystem (set to TRUE if continuance is supported):

.xsystem.job.query.continuance.supported

8.9.1 Standardized Job Input Fields

A single standard field is used to identify the action that is to be taken by the job. This field must be created on an XSet before submitting the XSet to the job control system. The standard field is named as follows:

org.snia.xam.job.command

This field may be binding or nonbinding, at the discretion of the application. It should be of MIME-type "application/vnd.snia.xam.string" (and is thus a property field). It should contain a string that identifies which job should be run when the XSet is submitted to the job control system. Submitting a job that does not have this field defined, or is defined but is the wrong type, should result in a non-fatal malformed job command error. Submitting a job to a system that does not recognize the command string should result in a non-fatal unrecognized job command error. The field is not a readonly field by default. However, after the XSet is submitted to the job control system, the field will become readonly until the job is completed (either naturally or terminated through application action). Standard field errors will occur if the application tries to edit/delete the field while it is in a readonly state.

Note that some jobs may require additional fields. The definition of these fields is beyond the scope of the generic description of jobs and job control and should be defined by the specific job (e.g., query). As a matter of convention, it is expected that the string used to identify the command will be used as the prefix of any additional fields used by the job. Thus, if the command field *org.snia.job.command* has a value of "vnd.com.example.my_command", then it would be expected that all fields must begin with the prefix "vnd.com.example.my_command". For example, a "parameter" field would have the field name "vnd.com.example.my_command.parameter". Additionally, as a matter of convention, for fields that cannot change over the lifetime of a job, it is expected that the fields will change to readonly, while the job is operational.

8.9.2 Standardized Job Output Fields

Jobs can fail at various times during the job creation, submission, and run phases. The condition of a job can be determined by examining the standard fields that contain status, health and error information. The status field describes what step in the job lifecycle the job is in. The health field describes if the job is okay or if it has encountered an error. The error field will only be created, if the job has encountered an error, and will contain the specific error token associated with the job.

8.9.2.1 Job Status

To properly report status, applications will refer to different error return mechanisms. Errors encountered during job submission can be determined by examining the code returned by XSet.submitJob. Errors encountered after the successful submission of a job shall be reported on the job XSet using the properties defined here.

The XSystem creates the status field on the XSet when an application submits a job. The status field name is defined as follows:

.xset.job.status

Note that the status field is a "." prefaced field, and thus, the application cannot create it. The status field is a nonbinding field and is always in a readonly state. It is a property field of MIME-type "application/vnd.snia.xam.string". The string will be set to one of the following values:

- "NEW": The job was just submitted and has not yet been started by the system.
- "STARTING": The job has been started but is still being initialized and is not yet running,
- "RUNNING": The job is running.

- “SHUTTING DOWN”: The job is in the process of stopping for some reason.
- “COMPLETE”: The job is no longer running. It has run to the end.
- “SUSPENDED”: The job is no longer running. The system has temporarily ceased processing; however, it will resume.
- “HALTED”: The job is no longer running because the application stopped it.

  Note: When changing one or more binding fields of a “RUNNING” job XSet, which results in the creation of a new XSet (e.g., on commit, when it was already committed and had a XUID), then the new XSet will have the job state set to “HALTED”.

- “KILLED”: The job is no longer running because the application stopped it.

8.9.2.2 Job Error

The XSystem will create the errorhealth field on the XSet, if it encounters an error while running the job. The errorhealth field is defined as follows:

.xset.job.errorhealth

Note that the errorhealth field is a system field, and thus, the application cannot create it. The errorhealth field is a nonbinding field, and is always in a readonly state, if it exists. It is a property field of MIME-type “application/vnd.snia.xam.string”. The value shall either be “OK” or “ERROR”. If this field exists on an XSet that is submitted as a job, the XSystem will remove it.

The XAM Storage System shall create the error field (if not already present) and set it to contain the associated error token as appropriate to the job in question. Note that this field does not supersede errors that occur while submitting a job; job submission errors will be returned from the API. The error field name is defined as follows:

.xset.job.error

Note that the error field is a system field, and thus, the application cannot create it. The error field is a nonbinding field, and is always in a readonly state. It is a property field of MIME-type “application/vnd.snia.xam.string”. The field will contain an error token as appropriate to the job in question.

Additional job-specific fields may also be added, but they should follow the guidelines as per additional input fields and are otherwise beyond the scope of a generic discussion of jobs and job control.

8.10 Asynchronous Operations

Asynchronous versions of specific methods in the XSet data path were created because the synchronous versions of those methods may take an extended time to complete. This delay can cause the thread calling the method to block. For example, a method may require communication between the VIM and the back end of the XAM Storage System, and a thread calling the method will block until the XAM Storage System responds. The methods with an asynchronous version are defined in Table 52, “Methods with

Asynchronous Versions". Applications that do not wish to block should use the appropriate asynchronous version of the method from the table.

Table 52 – Methods with Asynchronous Versions

Methods with Asynchronous Analogs		Description
Synchronous	**Asynchronous**	
XSystem.openXSet	XSystem.asyncOpenXSet	Success shall instantiate an XSet instance. See Section 8.1, "XSet Behavior" for additional information.
XSystem.copyXSet	XSystem.asyncCopyXSet	Success shall instantiate an XSet instance. See Section 8.1, "XSet Behavior" for additional information.
<XAMHandle>.openXStream	<XAMHandle>.asyncOpenXStream	Success shall instantiate an XStream instance.
XStream.read	XStream.asyncRead	Read from the current byte offset position.
XStream.write	XStream.asyncWrite	Write to the current byte offset position.
XStream.close	XStream.asyncClose	Success shall release an XStream instance and all associated resources.
XSet.commit	XSet.asyncCommit	Persistently store the information in the XSet to the XSystem.

8.10.1 The XAsync Object

Once an application initiates an operation asynchronously, it creates an object instance, the XASync instance, to manage the operation. It uses the XAsync instance to determine if the asynchronous method has completed, to terminate the operation if it has not yet completed, and to get the output of the method after it has completed.

Table 53, "XAsync Methods" defines the methods that operate on an XAsync instance. For additional information on operating on the XAsync instance, see Section 8.10.2, "XAsync FSM".

Table 53 – XAsync Methods

Type of Method	XAsync Methods	Description
Operating on the XAsync instance while the operation is pending	XAsync.halt	Halt the asynchronous operation. Returns when the asynchronous operation has been successfully halted or completed.
Retrieving state of the object while the operation is pending or completed	XAsync.isComplete	Test to see if the asynchronous operation has completed.
	XAsync.getXOPID	Retrieve the XOPID.

Table 53 – XAsync Methods

Type of Method	XAsync Methods	Description
Retrieving output results of the asynchronous operation after the operation has completed	XAsync.getStatus	Retrieve the status of the asynchronous operation. If the operation has not completed, the returned status shall be pending.
	XAsync.getXSet	Returns the XSet handle output argument. Valid only if the asynchronous call was XSystem.asyncOpenXSet; otherwise, it returns a non-fatal error.
	XASync.getXStream	Returns the XStream handle output argument. Valid only if the asynchronous call was <XAMHandle>.asyncOpenXStream.
	XAsync.getXUID	Returns the XUID output argument. Valid only if the asynchronous call was XSet.commit; otherwise, it returns a non-fatal error.
	XAsync.getBytesRead	Returns the BytesRead output argument. Valid only if the asynchronous call was XStream.asyncRead; otherwise, it returns a non-fatal error.
	XAsync.getBytesWritten	Returns the BytesWritten output argument. Valid only if the asynchronous call was XStream.asyncWrite; otherwise, it returns a non-fatal error.
Destroying an XAsync instance	XAsync.close	Success shall release an XAsync instance and all associated resources.

When the XAsync instance is created, the XAM application can optionally specify a callback method, which will be called when the method completes, and an XOPID, which is an opaque 64-bit value that the application can retrieve at any time. The asynchronous method that is used to start the asynchronous operation returns a handle to the XAsync instance on completion. Note that method completion does not imply that the requested operation has completed. For example, if XSystem.asyncOpenXSet is called, the open of the XSet is not completed when the asynchronous call returns. Instead, the XAM application can use either a poll-based method or a callback-based method to determine if the operation has completed:

- Poll-based method: The XAM application periodically polls the XAsync instance, to determine the completion status of the requested operation. If the operation status is pending, then the operation has not completed. Once the operation has completed, the XAM application can retrieve the status and output arguments.
- Callback-based method: The XAM application provides a callback function, when the asynchronous operation is initiated. When the requested operation completes, the callback method is called. From within the callback method, the XAM application can retrieve the status and output arguments.

Once an asynchronous operation has begun, the XAM application can query whether the operation has completed (XAsync.isComplete) or can halt the operation (XAsync.halt). Both methods are blocking calls. XAsync.halt is a blocking method that will not return until the asynchronous operation has been successfully halted or completed. Thus, once XAsync.halt completes, the XAM object instance that was being asynchronously operated on (i.e., XSystem.openXSet; XSet.commit or <XAMHandle>.openXStream; and XStream.read, XStream.write, or XStream.close) has moved to its final end state. If the operation completed successfully, the output arguments can be retrieved. If not, only the output XAM status can be retrieved.

Each asynchronous XAM method shall require two additional input arguments, a callback method parameter and a XOPID parameter, and shall output an XAsync object instance. If the XAM application wishes to poll for the result, it does not provide a callback method. For a poll-based model, the XAM application is expected to call XAsync.isComplete repeatedly until the method returns TRUE for the isComplete result.

The XOPID shall always be specified by the XAM application, even if it is not used by the XAM application. A XAM application can use XAsync.getXOPID to retrieve the XOPID for a specific XAsync instance. The XAM application may use the XOPID to retrieve the application context about the operation. The mechanism for associating application context to the XOPID is the responsibility of the XAM application and is beyond the scope of this specification.

If a callback method is provided by the XAM application, then the callback method shall be called when the asynchronous method completes and shall pass the XAsync object or handle back as a parameter. This behavior is equivalent to transitioning to the completed state of the XAsync instance FSM, as shown in Figure 20, “XAsync Instance FSM”. If a callback method is not provided by the XAM application, then no callback method shall be called.

8.10.2 XAsync FSM

This section defines the FSM for the XAsync instance.

Note: No abandon method exists for the XAsync instance. Instead, the XAM application must first call XAsync.halt to halt the operation. This call will cause the parent object state to reach a determinate state, at which time the XAsync instance can be closed.

As mentioned previously, using the asynchronous version of the XAM methods that support asynchronous operations does not introduce new states to the associated FSMs. Instead, consider the XAsync instance FSM as being created when the method is called (i.e., the beginning of the arc in the parent FSM) and the parent arc as transitioning to the new state, when the XAsync FSM transitions to the completed state. Once the operation has completed, the XAsync instance must then be closed by the XAM application.

Entry to the XAsync instance FSM shall be when the FSM is instantiated, as defined in Section 8.10.1, “The XAsync Object”.

The FSM and the state transitions are shown in Figure 20, “XAsync Instance FSM”.

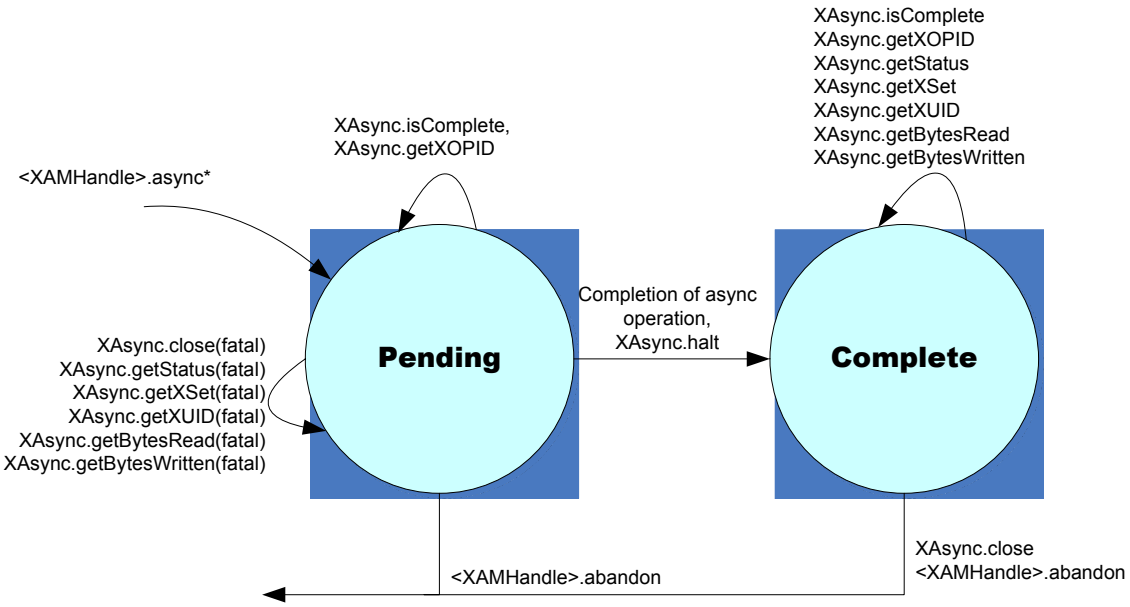

Figure 20 – XAsync Instance FSM

The XAsync instance FSM shall have the defined inputs, outputs, and transitions as shown in Table 54, “XAsync Instance FSM Transitions”.

Table 54 – XAsync Instance FSM Transitions

Start State	Transition	Final State	Comment
Outside of FSM (NULL)	<XAMHandle>.async*	Pending	The XAsync FSM shall be instantiated when the method returns successfully.
Pending	XAsync.close XAsync.getStatus XAsync.getXSet XAsync.getXUID XAsync.getBytesRead XAsync.getBytesWritten	Pending	Shall perform no action and always return a fatal error.
	XAsync.isComplete XAsync.getXOIP	Pending	Regardless of the completion status, the XAsync FSM shall stay in the pending state.
	Completion of asynchronous operation	Complete	This transition occurs due to the completion of the operation associated with this XAsync instance.
	XAsync.halt	Complete	When the parent FSM has completed its FSM transition, then the halt operation shall transition to the complete state.
	<XAMHandle>.abandon	NULL	Shall free the XAsync instance and release all resources associated with it.

Table 54 – XAsync Instance FSM Transitions

Start State	Transition	Final State	Comment
Abandon	XAsync.close	NULL	Shall free the XAsync instance and release all resources associated with it, regardless of whether the XAsync.close completed successfully with a fatal or non-fatal error.
	<XAMHandle>.abandon	NULL	Shall free the XAsync instance and release all resources associated with it.

Exit from the XAsync instance FSM shall cause the FSM to be uninstantiated and the associated resources to be returned to the operating environment. The XAsync instance FSM shall be uninstantiated through calling XAsync.close, and as previously noted, <XAMHandle>.abandon on any of its parents. If the XAsync.close completed successfully, no error occurred in closing the XAsync instance.

8.10.2.1 Effects on other FSMs

The XAsync instance FSM is not intended to affect the parent FSMs, where arcs support both asynchronous and synchronous arc transitions. This FSM includes the implementation of XAsync.halt. If an XSystem receives a halt request, it must transition the parent FSM to a valid state, before XAsync.halt returns, and transition the specific XAsync instance to the complete state. This implementation means that FSMs that include complex operations, such as XSet.commit, must behave the same way, regardless of whether the operation was called synchronously or asynchronously.

If the method is initiated through an asynchronous method, then the status is valid when the method has completed. At any point before the method completes, the status is pending. For FSM arcs that are initiated by an asynchronous method, the start of the arc occurs when the asynchronous method is called, which also causes the XAsync instance to be created. The end of the arc occurs when the asynchronous method completes (either successfully or with a fatal or non-fatal error). Note that completion of the asynchronous method does not destroy the XAsync instance. The XAM application must call XAsync.close to release the resources.

9 XSet Management

9.1 XSet Management Overview

An XSystem provides both XSet data access and XSet data management. XSet data access methods specify how to create, store, locate, retrieve, update, and delete an XSet that is contained within an XSystem. These methods are primarily used by data-consuming XAM applications. On the other hand, XSet data management methods specify how an XSystem manages an XSet until it is deleted. These methods are primarily used by data management XAM applications. In most cases, XAM applications will use a combination of both XSet data access and management methods.

Note: Application developers are strongly encouraged to understand XSet data management, as it differs greatly from data management that is available from familiar data access interfaces, such as file systems.

9.1.1 XSet Management Disciplines

XSet management is split into management disciplines: XSet retention, XSet hold, XSet deletion, and XSet storage.

- XSet retention uses retention time criteria to determine the time period(s) during which XSet deletion from the XSystem is prohibited.
  - — An XSystem shall not delete an XSet before the XSet retention time criteria are met, and any deletion tries (e.g., by a XAM application) shall generate non-fatal errors.
  - — After the XSet retention time criteria have been met, XSet retention shall no longer be a reason to prohibit XSet deletion.
- XSet hold enforces readonly XSet data access and prohibition of XSet deletion. While an XSet is on hold, an XSystem shall:
  - — Strictly enforce read-only access to the XSet.
  - — Prohibit XSet deletion.
- XSet deletion controls XSystem actions with respect to XSet deletion.
  - — An XSystem may automatically delete an XSet once the retention time and hold criteria have been met.
  - — A XAM Storage System may automatically shred or destroy the binary recording of an XSet that has been deleted.
- XSet storage controls storage management capabilities of the XSystem, such as resource, security, migration, virtualization, resiliency, and performance, all of which are outside the scope of XAM. XAM accommodates these capabilities by providing an XSet abstraction that requires the XSystem to adhere to the mutually agreed-to rules and behavior for data storage management.

XSet retention management, deletion management, and storage management apply to any XAM application that creates or deletes an XSet, as these disciplines mandate how an XSystem manages an XSet when it is created and until it is deleted. In contrast, XSet hold management applies to special-purpose XAM applications that single out XSets for readonly access and prohibit XSet deletion until the XAM application determines that the XSet hold is no longer required.

9.1.2 XSet Management Properties

XSet management properties specify the XSystem requirements for managing an entire XSet. A XAM application may express the desired XSet management behavior through two types of XSet management properties:

- Value management property: The value of the property shall directly mandate XSet management behavior.
- Policy management property: The value of this property shall be an XSet policy name and shall indirectly mandate XSet management behavior. For more information, see Section 8.7, “XSet Policy”.

Table 55 shows which XSet management property types are used by each XSet management discipline.

Table 55 – Management Discipline Property Types

XSet Management Discipline	Value Management Property	Policy Management Property
Retention	X	X
Deletion	X	X
Storage		X
Hold	X	

In addition to the discipline-specific properties, XAM defines a principal management policy property that shall specify the retention, deletion, and storage behaviors that apply in the absence of discipline-specific properties. This principal management policy property shall always be present on a committed XSet.

When a XAM application calls XSet.commit, the XSystem shall create the following XSet properties, if the property is not already present in the XSet instance:

- Principal policy management property
  - *.xset.management.policy*
- Retention value management properties
  - *.xset.retention.list.base*
  - *.xset.retention.base.enabled*
  - *.xset.retention.base.starttime*
  - *.xset.retention.list.event*
- Hold value management property
  - *.xset.hold*

The associated value and attribute settings for the properties above are described in the property specifications. All XSet value and policy management properties have a readonly attribute of TRUE, and therefore, shall only be created, modified, or removed using the XSet management methods described in this chapter. These changes will be made to the XSet instance before persistence, and thus, will be reflected in the XSet.

For the retention, deletion, and storage disciplines, once an XSet value or policy management property exists in a committed XSet, it shall not be removed from the XSet.

Certain XSet operations may generate a new, uncommitted XSet from an existing committed XSet. For detailed information on XSet operations, see Section 8.4, “XSet Methods” and Section 8.5, “XSet Instance Finite State Machine (FSM)”. When these XSet operations occur, management properties for the retention, deletion, and storage disciplines shall be transferred from the existing XSet to the new, uncommitted XSet instance. Hold discipline properties shall not be transferred from the existing XSet to the new, uncommitted XSet instance, and when committed, the new XSet shall not be on hold.

If a XAM application wants management properties removed from the new, uncommitted XSet instance instead of inherited from the existing XSet, the application should reset the management fields and set the desired management properties before committing the XSet instance.

9.2 XSet Retention and Deletion Value Management Properties

XSet retention defines the criteria that an XSystem shall use to prohibit XSet deletion. XSet deletion defines the criteria that may be used by an XSystem to automatically delete an XSet, once all retention and hold time criteria are satisfied. XSet deletion also defines the XSet shredding criteria that a XAM Storage System may use to destroy the binary recording of a deleted XSet. The values of XSet retention and deletion value management properties shall be preserved by export and import activities that transfer XSets to other XSystems.

9.2.1 XSet Retention

Retention management uses time criteria to determine the time period(s) during which XSet deletion from the XSystem shall be prohibited. XSet retention criteria shall be specified by:

- A retention criteria identifier: a XAM application-specified string that shall unify the retention criteria
- A retention enablement flag: a Boolean value indicating if retention shall be (or was) enforced
- A starting time and time duration: the start time, when used together with duration, indicates when retention shall no longer be enforced

A XAM application may create multiple sets of retention criteria on a single XSet, where each set is unified by its retention identifier. When it tries to delete an XSet, the XSystem shall evaluate all such retention criteria and return a non-fatal error, if any such retention criteria has not been met.

In addition to any XAM application-defined retention criteria identifiers, all XSets shall have retention criteria identifiers of “base” and “event”. All XSet retention criteria shall operate independently of the XSet on-hold status (see Section 9.4, “XSet Hold Properties”), and a XAM application may determine if an XSet is currently subject to any XSet retention criteria by using XSystem.isXSetRetained.

Examples of “base” retention and all other retentions are shown in Figure 22 and Figure 23. Figure 23 is an example of combining “base” retention and “other” retention. These figures show retention criteria

applied to an XSet in a manner such that XSet deletion is continuously prohibited until XSet retention criteria is met.

"base" XSet Retention Management

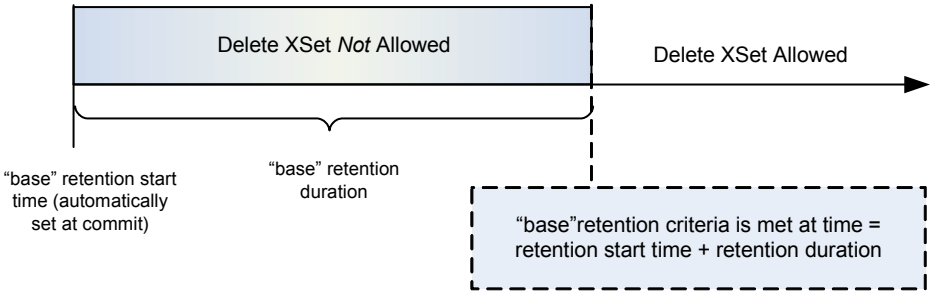

Figure 21 – "base" Retention Criteria

"other" XSet Retention Management

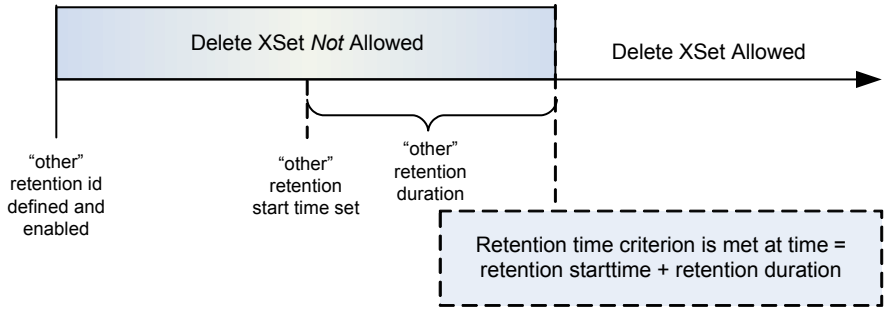

Figure 22 – "other" Retention Criteria

Combined XSet Retention Management

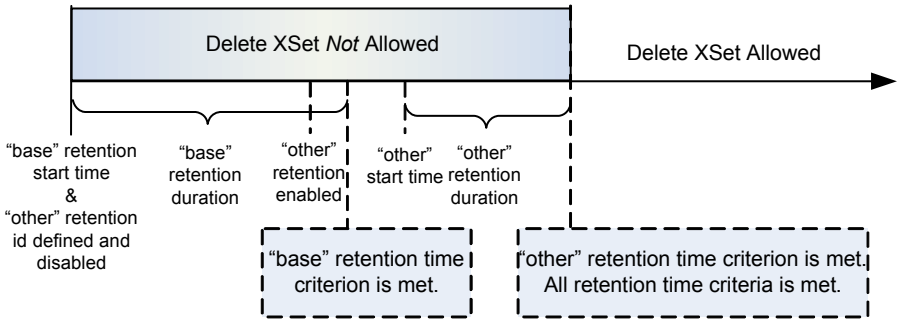

Figure 23 – Combined "base" and "other" Retention

Figure 24 is an example of how to establish time-based retention with an “other” retention identifier.

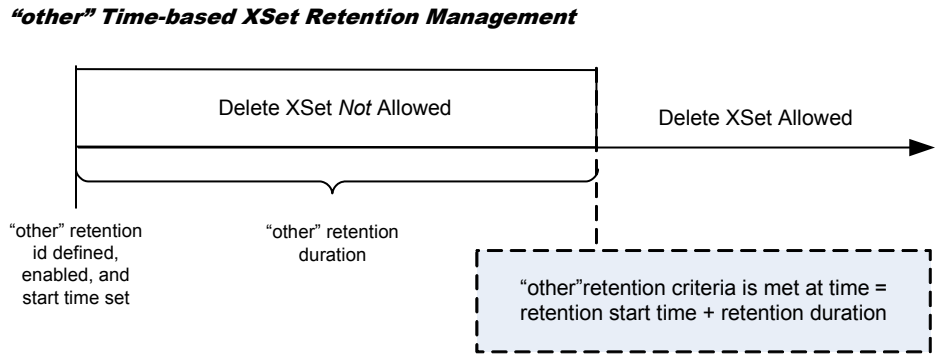

Figure 24 – Time-based Retention

Table 56 lists the retention value management properties, which are described in the paragraphs following the table. Note that the retention identifier that provides scope to the retention criteria (indicated as <retention id>) shall be encoded into the field name.

Table 56 – Retention Value Management Properties

Field	stype	Binding	Read-only	Update Rules
.xset.retention.list.base	xam_string	TRUE	TRUE	Shall only be set when created
.xset.retention.list.event	xam_string	TRUE	TRUE	Shall only be set when created
.xset.retention.list.<retention id> (where <retention id> is other than “base” or “event”)	xam_string	Application specified	TRUE	Shall only be set when created
.xset.retention.base.enabled	xam_boolean	TRUE	TRUE	Shall be set to a value of TRUE and shall not be changed to FALSE. Shall be set only if *.xset.retention.list.base* exists
.xset.retention.<retention id>.enabled (where <retention id> is other than “base”)	xam_boolean	Application specified	TRUE	Value shall not be changed from TRUE to FALSE. Shall be set only if *.xset.retention.list.<retention id>* exists
.xset.retention.<retention id>.duration	xam_int	Application specified	TRUE	Value shall not be decreased. Shall be set only if: • *.xset.retention.list.<retention id>* exists and • *<retention id>* enabled flag is TRUE or FALSE

Table 56 – Retention Value Management Properties

Field	stype	Binding	Read-only	Update Rules
.xset.retention.base.starttime	xam_datetime	TRUE	TRUE	Shall only be set when created with the value of *.xset.time.xuid* Shall be set only if: • *.xset.retention.list.base* exists and • "base" enabled flag is TRUE and • "base" duration is a positive integer or -1
.xset.retention.<retention id>.starttime (where <retention id> is other than "base")	xam_datetime	Application specified	TRUE	Shall only be set when created Shall be set only if: • *.xset.retention.list.<retention id>* exists and • <retention id> enabled flag is TRUE and • <retention id> duration is a positive integer or -1

.xset.retention.list.<retention id>

contains the retention criteria identifier on the XSet. This identifier shall provide a scope for all other retention criteria having the same identifier. Note that the property string value shall be equivalent to the retention identifier section of the field name and that *.xset.retention.list.base* and *.xset.retention.list.event* shall be binding properties. If *.xset.retention.list.base* and *.xset.retention.list.event* are not present in the XSet instance, a successful XSet.commit shall create and set these properties. Once *.xset.retention.list.<retention id>* is created and set, it shall not be set again.

Methods - XSet.createRetention, XSet.setBaseRetention, XSet.applyBaseRetentionPolicy

.xset.retention.<retention id>.enabled

indicates if the retention criteria under the given scope should be evaluated when determining if XSet deletion shall be prohibited. A value of TRUE indicates that XSet deletion shall be prohibited until the retention time criteria, *.xset.retention.<retention id>.starttime* plus *.xset.retention.<retention id>.duration*, is met. A value FALSE indicates that XSet deletion shall be allowed and the retention time criteria shall be ignored. Once the value is TRUE, it shall not be changed to FALSE. Note that *.xset.retention.base.enabled* shall be a binding property and shall have a value of TRUE. If *.xset.retention.base.enabled* is not present in the XSet instance, a successful XSet.commit shall create and set this property. *.xset.retention.<retention id>.enabled* shall be set only when *.xset.retention.list.<retention id>* exists on the XSet.

Methods - XSet.setRetentionEnabledFlag, XSet.setBaseRetention, XSet.applyBaseRetentionPolicy

.xset.retention.*<retention id>*.duration

when combined with *.xset.retention.<retention id>.starttime,* indicates the end time for which the XSystem shall prohibit deletion of the XSet for the retention criteria under the given scope. This value shall be specified as a positive number of milliseconds, or as “forever,” which is represented by a value of -1. This value shall only be allowed to increase and never decrease; thus, once “forever” is specified, it shall not be set again. The *.xset.retention.<retention id>.duration* property shall be set only when *.xset.retention.list.<retention id>* exists on the XSet and the policy-determined or property value of *.xset.retention.<retention id>.enabled* is TRUE or FALSE.

Methods - XSet.setRetentionDuration, XSet.setBaseRetention

.xset.retention.base.starttime

indicates the start time for the retention time criteria for the “base” retention identifier. The *.xset.retention.base.starttime* property shall be created only when: *.xset.retention.list.base* exists on the XSet; the policy-determined or property value of *.xset.retention.base.enabled* is TRUE; and the policy-determined or property value of *.xset.retention.base.duration* is a positive integer or -1. If *.xset.retention.base.starttime* is not present in the XSet instance, a successful XSet.commit shall create and set this property as a binding property with a value that is the same as *.xset.time.xuid.* Once *.xset.retention.base.starttime* is created and set, it shall not be set again.

.xset.retention.<retention id>.starttime

indicates the start time for the retention criteria under the given scope other than “base”. The *.xset.retention.<retention id>.starttime* property shall be created only when: *.xset.retention.list.<retention id>* exists on the XSet; the policy-determined or property value of *.xset.retention.<retention id>.enabled* is TRUE; and the policy-determined or property value of *.xset.retention.<retention id>.duration* is a positive integer or -1. Once *.xset.retention.<retention id>.starttime* is created and set, it shall not be set again.

Method - XSet.setRetentionStarttime

9.2.1.1 XSet Retention Value Management Property Methods

XSet retention value management properties shall not be created or set using the standard field create and set operations. Therefore, to allow applications to create and set the retention value management properties, the following methods are provided.

These methods generally provide a one-to-one correspondence with the retention properties, with the exception of a utility method that shall create the “base” retention criteria. This utility method creates multiple properties, but the actions performed by this method shall be performed as a group, so that intermediate states are not visible through the XAM API.

XSet.setBaseRetention

This method will take a xam_int containing a duration to set as the value for *.xset.retention.base.duration*. It shall create the binding field *.xset.retention.list.base*, with a value of “base”, if it does not exist in the XSet instance. It shall create the binding field *.xset.retention.base.enabled,* with the value set to TRUE, if it does not exist in the XSet instance. Note that the XSystem will create/set the value of *.xset.retention.base.starttime* to the value of *.xset.time.xuid* at the time of first commit. An additional flag shall allow the application to choose if *.xset.retention.base.duration* is a binding or a nonbinding field.

XSet.createRetention

This method shall take an application-specified xam_string, which shall not be “base”, as the retention criteria identifier. It shall create *.xset.retention.list.<retention id>* with a value of *<retention id>*, where

<retention id> shall be the application-specified xam_string. An application shall also specify a xam_boolean to select, if *.xset.retention.list.<retention id>* shall be binding or nonbinding.

XSet.setRetentionEnabledFlag

This method shall take an application-specified <retention id>, which shall not be “base”, and xam_boolean that indicates if the <retention id> retention criteria is enabled or not. It shall create and/or set *.xset.retention.<retention id>.enabled* with the xam_boolean value that is specified by the application. An application shall also specify a xam_boolean to select, if *.xset.retention.<retention id>.enabled* shall be binding or nonbinding.

XSet.setRetentionDuration

This method shall take an application-specified <retention id>, which shall not be “base”, and xam_int that specifies the number of milliseconds after *.xset.retention.<retention id>.starttime*, which the XSystem shall no longer prohibit XSet deletion, or negative one (-1), if the XSystem shall prohibit XSet deletion forever, whenever *.xset.retention.<retention id>.enabled* is TRUE. It shall create *.xset.retention.<retention id>.duration,* with the xam_int value passed in by the application. An application shall also specify a xam_boolean to select, if *.xset.retention.<retention id>.duration* shall be binding or nonbinding.

XSet.setRetentionStarttime

This method shall take an application-specified <retention id>, which shall not be “base”, and create *.xset.retention.<retention id>.starttime* with the value of the current time on the XSystem (i.e., the value of *.xsystem.time*). An application shall also specify a xam_boolean to select, if *.xset.retention.<retention id>.starttime* shall be binding or nonbinding.

XSystem.isXSetRetained

This method shall evaluate all retention criteria that exists on a given XSet, specified as a xam_xuid, and shall return TRUE if retention criteria exists that would prohibit XSet deletion. This method shall return FALSE if all XSet retention criteria, i.e., retention criteria associated with the “base”, “event”, and *.xset.retention.list.<retention id>* identifiers, have been met. This method shall not evaluate the on-hold status. A non-fatal error shall be returned if the specified XUID is improperly formatted, does not exist in the XSystem, or if the application is not authorized to access the XSet.

9.2.1.1.1 Retention Value Management Methods and the Open XSet FSMs

The methods that create or alter XSet retention value management properties shall have the following effects on the open XSet finite state machines (FSMs) that are specified in Section 8.5, “XSet Instance Finite State Machine (FSM)”:

- If the XSet retention value management property does not exist in the XSet, then creating the property shall cause the same FSM effects as creating any other field. The field shall be set as binding or nonbinding by the appropriate field creation method. See Section 6.4, “Methods that Operate on Fields” for more information.
- If the XSet retention value management property exists in the XSet, altering the property value shall cause the same FSM effects as XSet.set<stype>.
  - — If the existing XSet retention value management property is nonbinding, it shall cause the FSM effects of a nonbinding modification.
  - — If the existing XSet retention value management property is binding, it shall cause the FSM effects of a binding modification.

If the XSet retention value management property exists in the XSet, altering the property's binding attribute shall cause the same FSM effects as XSet.setFieldAsNonbinding or XSet.setFieldAsBinding, whichever is appropriate.

9.2.1.2 XSet Retention Management FSM

The FSM for XSet retention management is normatively defined in this section. This FSM is based on creating and changing individual retention elements that are represented as fields and are under the scope of a single retention identifier. An XSet can have multiple retention identifiers defined, and thus, can have multiple retention FSMs associated with it. This FSM does not try to describe how the fields are created or modified. Any additional semantics that are overlaid onto the open XSet FSMs are described above. This FSM (see Figure 25) will describe how retention management is defined for an XSet that is under the scope of a single retention identifier.

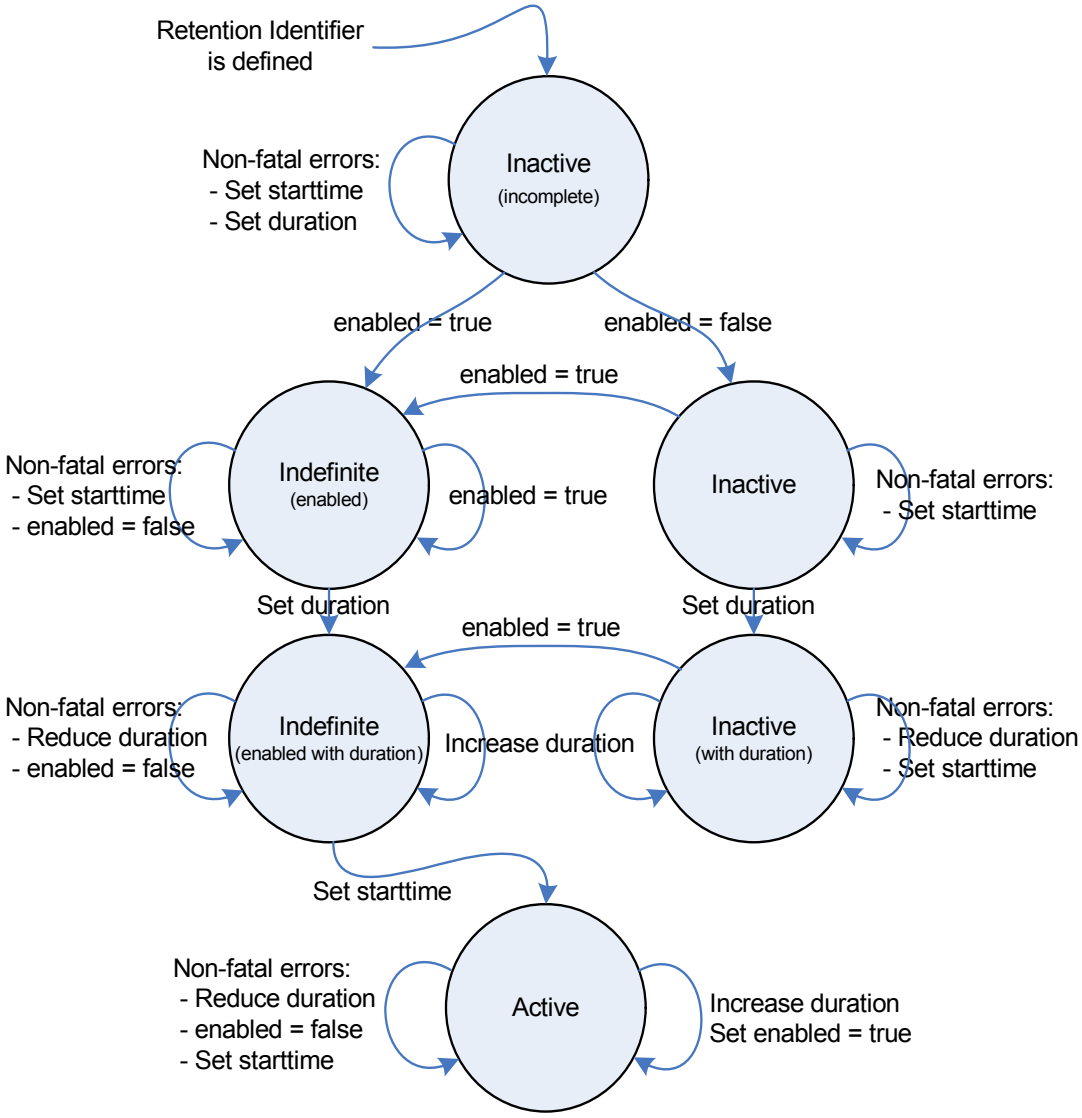

Figure 25 – The Retention Finite State Machine (FSM)

Three states are controlled by the retention criteria scoped by a retention identifier: indefinite, active, and inactive. Before retention identifiers are created, the XSet is not under retention management.

- The indefinite state is a state in which any attempt to delete the XSet shall result in a non-fatal error. Indefinite retention is characterized by a valid retention id, but incomplete retention criteria (i.e., one of following fields are not present in the XSet: the enabled flag, the duration, and the start time).
- Retention is said to be active for a given scope, when the retention identifier is set and the retention criteria is scoped with the retention identifier (enabled flag, duration, and start time), and all are present and valid.
- When retention in a given scope is inactive, it shall have no effect when deleting the XSet. An XSet can be inactive when the enabled flag is set to FALSE. It may also be inactive when the criteria that are scoped by the retention identifier is incomplete or invalid.

  Note: A correctly implemented state machine does not allow entry into the inactive retention state due to invalid fields; however, an inactive retention state can be entered by importing improperly formatted canonical XSet data. In such cases, the inactive state can only be left by resetting all management fields in the XSet.

The tables in this section define the normative transitions for the FSM. Refer to Figure 25 above for an illustration of those normative transitions and states. XSet.applyBaseRetentionPolicy, XSet.applyRetentionEnabledPolicy, and XSet.applyRetentionDurationPolicy are equivalent to XSet.setBaseRetention, XSet.setRetentionEnabledFlag, and XSet.setRetentionDuration in the transition columns of the retention FSM tables (see Table 57 through Table 61).

Table 57 – Entrance XSet Retention FSM; Setting the Retention Identifier

Start State	Transition	Final State	Comment
Outside of FSM No retention	XSet.createRetention	Inactive (incomplete)	This method creates a retention identifier on the XSet, providing the scope for the complete retention criteria to be applied to the XSet.
	XSet.setBaseRetention	Active	This method creates the “base” retention identifier, sets the enabled flag to TRUE, and sets the duration. The start time is set to *.xset.time.xuid* at the time that the XSet instance is committed.
	XSet.setRetentionEnabledFlag	Outside of FSM	A non-fatal error occurs when this method is called before the retention identifier is created.
	XSet.setRetentionDuration	Outside of FSM	A non-fatal error occurs when this method is called before the retention identifier is created.
	XSet.setRetentionStarttime	Outside of FSM	A non-fatal error occurs when this method is called before the retention identifier is created.

Once the retention identifier is defined, then XSet retention is in an inactive state until the corresponding enabled flag is set. To move from this state to other substates, the enabled flag must be set, as described in Table 58.

Table 58 – Setting the Retention Enabled Flag

Start state	Transition	Final state	Comment
Inactive (incomplete)	XSet.setRetentionEnabledFlag (TRUE)	Indefinite (enabled)	Transitions the FSM to the Indefinite (enabled) substate.
	XSet.setRetentionEnabledFlag (FALSE)	Inactive	Transitions the FSM to the inactive state.
	XSet.setRetentionDuration	Inactive (incomplete)	A non-fatal error occurs when this method it called before the enabled flag is set.
	XSet.setRetentionStarttime	Inactive (incomplete)	A non-fatal error occurs when this method it called before the enabled flag is set.

After the retention identifier is defined and the enabled flag is set (note that the value may come from the management policy for the XSet, scoped by the retention identifier), the duration of the retention must be set, as described in Table 59.

Table 59 – Setting the Duration

Start state	Transition	Final state	Comment
Indefinite (enabled)	XSet.setRetentionDuration (initial)	Indefinite (enabled with duration)	Initial duration value can be any valid positive integer or -1, indicating that the retention duration is forever.
	XSet.setRetentionEnabledFlag (TRUE)	Indefinite (enabled)	No change.
	XSet.setRetentionEnabledFlag (FALSE)	Indefinite (enabled)	A non-fatal error occurs.
	XSet.setRetentionStarttime	Indefinite (enabled)	A non-fatal error occurs when this method is called before the duration is set.
Inactive	XSet.setRetentionDuration (initial)	Inactive (duration)	Initial duration value can be any valid positive integer or -1, indicating that the retention duration is forever.
	XSet.setRetentionEnabledFlag (TRUE)	Indefinite (enabled)	Transitions the FSM to the Indefinite (enabled) substate.
	XSet.setRetentionEnabledFlag (FALSE)	Inactive	No change.
	XSet.setRetentionStarttime	Inactive	A non-fatal error occurs when this method is called before the duration is set.

Once the retention identifier is set and the enabled flag and duration are set, then retention end time can be established by setting the retention start time, as shown in Table 60.

Table 60 – Setting the Start Time

Start state	Transition	Final state	Comment
Indefinite (enabled with duration)	XSet.setRetentionStarttime	Active	Transitions to the active state.
	XSet.setRetentionDuration (increase)	Indefinite (enabled with duration)	No change.
	XSet.setRetentionDuration (decrease)	Indefinite (enabled with duration)	A non-fatal error occurs when reducing the duration.
	XSet.setRetentionEnabledFlag (TRUE)	Indefinite (enabled with duration)	No change.
	XSet.setRetentionEnabledFlag (FALSE)	Indefinite (enabled with duration)	Once set to TRUE, setting enabled to FALSE results in a non-fatal error.
Inactive (with duration)	XSet.setRetentionEnabledFlag (TRUE)	Indefinite (enabled with duration)	Transitions to the Indefinite (enabled with duration) state.
	XSet.setRetentionStarttime	Inactive (with duration)	A non-fatal error occurs when setting the start time before retention is enabled.
	XSet.setRetentionDuration (increase)	Inactive (with duration)	No change.
	XSet.setRetentionDuration (decrease)	Inactive (with duration)	A non-fatal error occurs when reducing the duration.
	XSet.setRetentionEnabledFlag (FALSE)	Inactive (with duration)	No change.

Once the retention is in an active state, only the duration shall be changeable, and then only when the duration is increased. The start time and enabled flag shall generate not-fatal errors when an application tries to change it, as shown in Table 61.

Table 61 – Increasing the Retention Duration on an Active Retention Scope

Start state	Transition	Final state	Comment
Active	XSet.setRetentionStarttime	Active	A non-fatal error shall occur when setting the start time after it had already been set.
	XSet.setRetentionDuration (increase)	Active	No change.
	XSet.setRetentionDuration (decrease)	Active	A non-fatal error occurs when reducing the duration.
	XSet.setRetentionEnabledFlag (TRUE)	Active	No change.
	XSet.setRetentionEnabledFlag (FALSE)	Active	Once set to TRUE, setting enabled to FALSE results in a non-fatal error.

9.2.1.3 Examples of Multiple XSet Retention Identifiers

When using multiple retention criteria, XAM applications may create an XSet retention time gap, during which deleting an XSet is allowed. Figure 26 shows an example of an XSet retention time gap, with “base” and “other” retention criteria, whereby an XSet is vulnerable to deletion. XAM applications should be aware of this vulnerability when setting the XSet retention criteria.

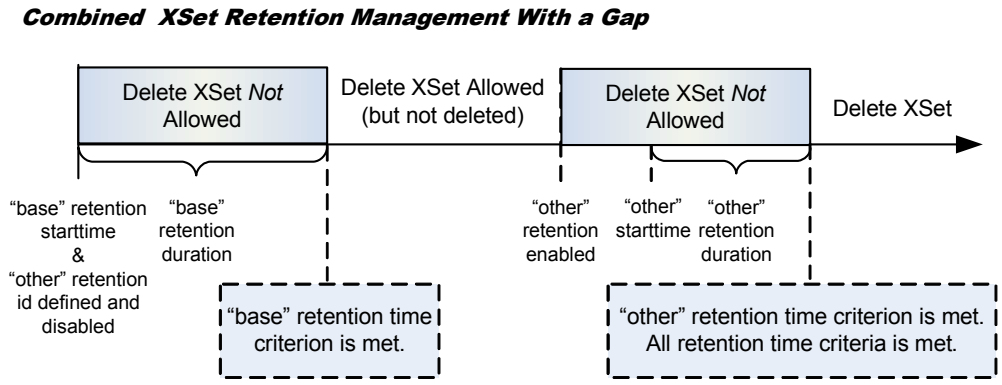

Figure 26 – Combining Retention with a Gap

9.2.2 XSet Deletion

XSet deletion properties shall determine whether an XSystem may delete an XSet, once an XSystem is no longer required to prohibit XSet deletion. An XSystem may delete an XSet, once retention and hold time criteria are met. For hold time criteria, see Section 9.4, “XSet Hold Properties”. XSet deletion properties also shall determine whether a XAM Storage System may shred or destroy the binary recording of a deleted XSet.

Figure 27 shows examples of automatic XSet deletion, once all retention criteria are met, both with and without holds on the XSet.

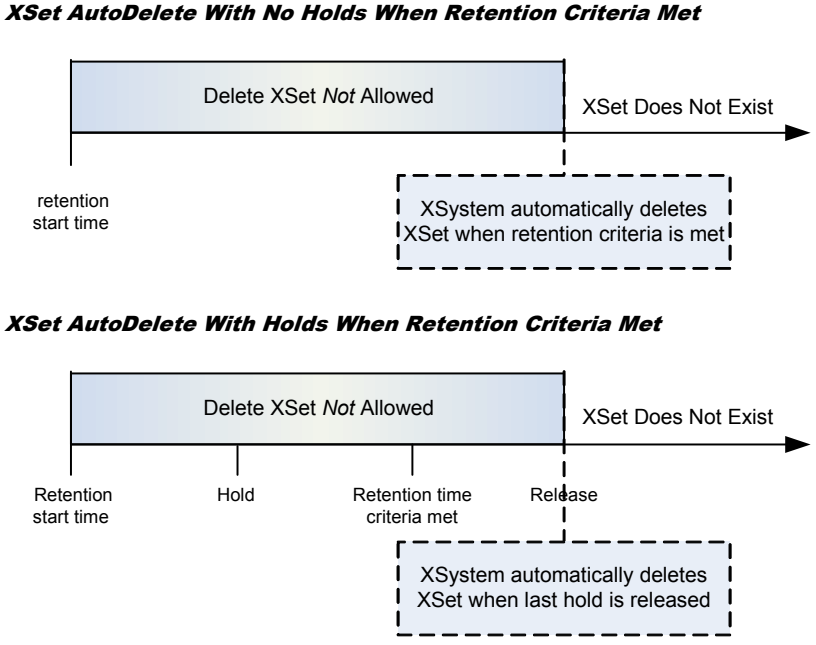

Figure 27 – AutoDelete Behavior With and Without Holds

Table 62 lists the deletion value management properties, which are described in the paragraphs following the table.

Table 62 – Deletion Value Management Properties

Field	stype	Binding	Readonly	Comments
.xset.deletion.autodelete	xam_boolean	Application specified	TRUE	XSet autodelete shall occur if set to TRUE and *.xsystem.autodelete* is set to TRUE.
.xsystem.deletion.autodelete	xam_boolean	FALSE	TRUE	See Section 7.1, “XAM Library” and Section 7.2, “XSystem”.
.xset.deletion.shred	xam_boolean	Application specified	TRUE	XSet shred shall occur if set to TRUE and *.xsystem.shred* is set to TRUE.
.xsystem.deletion.shred	xam_boolean	FALSE	TRUE	See Section 7.1, “XAM Library” and Section 7.2, “XSystem”.

.xset.deletion.autodelete

indicates that an XSystem shall automatically, and asynchronously, delete the XSet once the XSet retention time criteria has been met and the XSet is not on hold. If this value is TRUE, then the XSet shall automatically be deleted; otherwise, it shall not.

An XSystem may ignore *.xset.deletion.autodelete*. Applications shall determine if an XSystem supports an *.xset.deletion.autodelete* value of TRUE, if *.xsystem.deletion.autodelete* has a value of TRUE. If *.xsystem.deletion.autodelete* is FALSE, then an XSet shall only be deleted by a XAM application by using XSystem.deleteXSet. Setting *.xset.deletion.autodelete* to TRUE when *.xsystem.deletion.autodelete* is FALSE shall not result in a non-fatal or fatal error. XAM applications that set *.xset.deletion.autodelete* to TRUE should anticipate the possibility that the XSystem may be enabled for automatic XSet deletion, when the XSet retention time criteria are met and the XSet is not on hold.

Method – XSet.setAutoDelete

.xset.deletion.shred

indicates that a XAM Storage System shall automatically, and asynchronously, shred or destroy the binary recording of an XSet once the XSet is deleted from the XSystem. If this value is TRUE, then the binary recording of the deleted XSet shall automatically be shredded; otherwise, it shall not.

The method or algorithm that a XAM Storage System uses to shred the binary recording of a deleted XSet is outside the scope of XAM.

An XSystem may ignore *.xset.shred*. Applications shall determine if an XSystem supports an *.xset.deletion.shred* value of TRUE, if *.xsystem.deletion.shred* is TRUE. If *.xsystem.deletion.shred* is FALSE, then the binary recording of a deleted XSet shall not be shredded by the XAM Storage System. Setting *.xset.deletion.shred* to TRUE when *.xsystem.deletion.shred* is FALSE shall not result in a non-fatal or fatal error. XAM applications that set *.xset.deletion.shred* to TRUE should anticipate that the XSystem may be enabled for automatic XSet shredding when the XSet is deleted.

Method – XSet.setShred

9.2.2.1 Deletion Value Management Methods and the Open XSet FSMs

The methods that create or alter XSet deletion value management properties (XSet.setAutoDelete and XSet.setShred) shall have the following effects on the XSet finite state machines (FSMs) specified in Section 8.5, “XSet Instance Finite State Machine (FSM)”:

- If the XSet deletion value management property does not exist in the XSet, then creating this property shall cause the same FSM effects as creating any other field. This field shall be set as binding or nonbinding by the appropriate field creation method. See Section 6.4, “Methods that Operate on Fields” for more information.
- If the XSet deletion value management property exists in the XSet, then XSet.set<deletion value management property> that alters the property value shall cause the same FSM effects as an XSet.set<stype> method.
  - — If the existing XSet deletion value management property is nonbinding, then XSet.set<deletion value management property> shall cause the FSM effects of a nonbinding modification.
  - — If the existing XSet deletion value management property is binding, then an XSet.set<deletion value management property> shall cause the FSM effects of a binding modification.
- If the XSet deletion value management property exists in the XSet, then XSet.set<deletion value management property> that modifies the property’s binding attribute shall cause the same FSM effects as XSet.setFieldAsNonbinding or XSet.setFieldAsBinding, whichever is appropriate.

9.2.3 XSystem Clock/Time Management

The accuracy and integrity of the XSet properties for retention start times depend on the accuracy and integrity of the XSystem clock that is used to set their values. Equally important is the relative accuracy and integrity of the XSystem clock, which determines if XSet retention duration has elapsed, to the XSystem clock, which sets the start time property. Relative time differences between these two clocks can lead to undesirable retention and deletion management behavior.

For example, an XSet is created in an XSystem at time 0 with *.xset.retention.base.duration* of 8 years and *.xset.deletion.autodelete* of TRUE. At time 1 year, the XSystem clock is adjusted forward to 9 years. Now, because the XSystem time is 9 years, the XSet retention time criterion is satisfied, even though only 1 year has actually elapsed. And since *.xset.deletion.autodelete* is TRUE, the XSystem automatically deletes the XSet.

The specifications for accuracy and integrity of XSystem time keeping is not within the scope of XAM. However, to prevent undesirable XSet retention and deletion management consequences, XSystems are strongly encouraged to maintain accurate clock time, with zero or minimal deviation to clock integrity.

9.3 XSet Policy Management Properties

The XSet policy management properties are XSet policy properties as described in Section 8.7, “XSet Policy”. Included in XAM is an XSet policy management property that represents an abstraction for XSystem storage management capabilities, which are outside the scope of XAM. The storage policy property allows XAM applications to use such XSystem storage management capabilities for XSet management. XAM also includes XSet policy management properties that govern retention and deletion management in a manner that is consistent with the governance of the retention and deletion value management properties described in Section 9.2, “XSet Retention and Deletion Value Management Properties”. In other words, the XSet policy management properties shall be used to determine the actual values of retention durations, retention enablement, automatic deletion, and shredding, in the absence of the corresponding value management property in the XSet. Lastly, the XSet principal management policy

property is used by the XSystem to determine the actual value of the retention, deletion, and storage management properties, value or policy, when the corresponding XSet management property in an XSet is absent. A successfully committed XSet shall always have a principal policy management property.

The XSet policy management properties, together with the corresponding retention and deletion value management properties, have a hierarchical precedence relationship. This hierarchy eliminates ambiguity of the actual governing XSet retention and deletion management value in cases where multiple XSet policy and value management properties exist in an XSet and when each property can determine the same governing value.

The XSet policy management properties are XSet policy properties (see Section 8.7, “XSet Policy”), and therefore, may not be interoperable across XSystems. To accommodate XSystem interoperability, the getActual methods get the value management property or policy management property that is the determined (actual) value of the retention and deletion value management properties. The getActual methods may also be used by XAM applications to analyze the net effect of altering XSet policy management properties.

9.3.1 Storage Management Policy

Per the XSet storage management discipline, storage management properties pertain to XSystem storage management capabilities, which are outside the scope of XAM, e.g., XSet storage performance, resiliency, and virtualization. As a result, such XSystem-specific XSet storage management capabilities are abstractly specified as a storage policy management property.

9.3.2 Retention and Deletion Management Policy

Retention and deletion policy management properties are used by an XSystem to determine XSet retention time criteria, autodelete, and shred behavior in the absence of the corresponding value management properties. The XSet value management property behaviors that a XAM application may indirectly express through retention and deletion policy management properties are:

- *.xset.retention.<retention id>.enabled*
- .xset.retention.<*retention id*>.duration
- *.xset.deletion.autodelete*
- *.xset.deletion.shred*

Figure 28, “An Example Policy Management Property” shows an XSet that contains two management properties. The first property is *.xset.deletion.shred* with a xam_boolean value of TRUE. The second property is *.xset.deletion.shred.policy* with a xam_string value of “confidential”. The XSystem policy named “confidential” states that the value that the XSystem uses in the absence of *.xset.deletion.shred* is the

xam_boolean of TRUE. In either case, the XSystem recognizes the actual value of TRUE for XSet shredding.

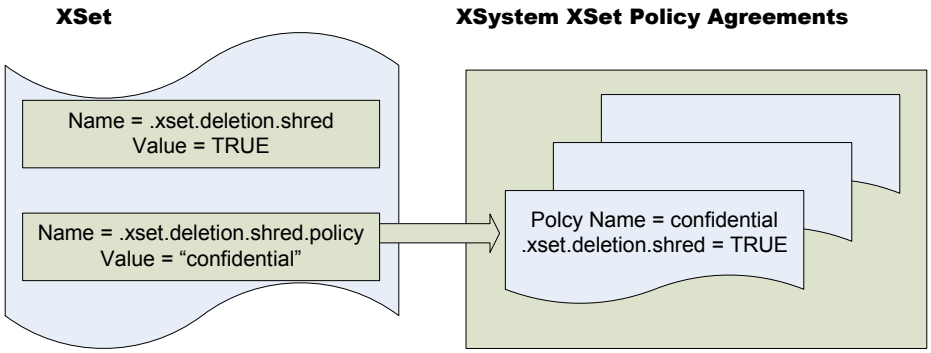

Figure 28 – An Example Policy Management Property

An XSystem shall be subject to the same update rules (see Table 56, "Retention Value Management Properties") in updating a retention actual value in a retention policy agreement as the corresponding retention value management property. Likewise, a XAM application shall be subject to the same update restrictions with respect to the actual values as the corresponding retention value management property, when updating a retention policy management property.

9.3.3 XSet Management Policy

The principal management policy property, or XSet management policy, shall be used by an XSystem to determine XSet retention criteria, autodelete, and shred behavior in the absence of both value and policy management properties for any of the retention and deletion value management properties mentioned previously. An XSystem shall also use the XSet principal management policy property to determine the storage management behavior in the absence of the XSet storage policy management property.

An XSystem shall be subject to the same update rules (see Table 56, "Retention Value Management Properties") in updating a retention actual value in a management policy agreement as the corresponding retention value management property. Likewise, a XAM application shall be subject to the same update rules with respect to the retention actual values as the corresponding retention value management property, when updating a management policy property.

Table 63 lists the XSet policy management properties, which are described in the paragraphs following the table.

Table 63 – XSet Policy Management Properties

Policy Management Property Name [Binding-Application Specified, Readonly-TRUE]	Policy Management Property Method
.xset.management.policy	XSet.applyManagementPolicy
.xset.retention.<retention id>.enabled.policy	XSet.applyRetentionEnabledPolicy
.xset.retention.base.duration.policy	XSet.applyBaseRetentionPolicy
.xset.retention.<retention id>.duration.policy	XSet.applyRetentionDurationPolicy
.xset.deletion.autodelete.policy	XSet.applyAutoDeletePolicy

Table 63 – XSet Policy Management Properties

Policy Management Property Name [Binding-Application Specified, Readonly-TRUE]	Policy Management Property Method
.xset.deletion.shred.policy	XSet.applyShredPolicy
.xset.storage.policy	XSet.applyStoragePolicy

.xset.management.policy

shall be used by an XSystem to determine XSet retention criteria, autodelete, and shred behavior in the absence of both value and policy management properties for any of the following value management properties:

- *.xset.retention.<retention id>.enabled*
- *.xset.retention.<retention id>.duration*
- *.xset.deletion.autodelete*
- *.xset.deletion.shred*

The XSystem shall also use *.xset.management.policy* to determine the XSet storage management behavior in the absence of *.xset.storage.policy*.

When XSet.commit creates a new XSet and *.xset.management.policy* does not exist, XSet.commit shall create *.xset.management.policy* and shall set its value to the policy name found in *.xsystem.management.policy.default.*

An XSystem shall be subject to the same update rules (see Table 56) in updating a retention management value in a retention management policy agreement as the corresponding retention value management property. Likewise, a XAM application shall be subject to the same update rules with respect to the policy-determined retention values as the corresponding retention value management property, when updating a retention management policy property.

Method - XSet.applyManagementPolicy

.xset.retention.<retention id>.enabled.policy

shall be used by an XSystem to determine the actual value of an XSet's retention enablement for a given scope (as specified by the retention id) in the absence of *.xset.retention.<retention id>.enabled*.

An XSystem shall be subject to the same update rules (see Table 56) in updating a retention management value in a retention management policy agreement as the corresponding retention value management property. Likewise, a XAM application shall be subject to the same update rules with respect to the policy-determined retention values as the corresponding retention value management property, when updating a retention management policy property.

The XSystem policy list for *.xset.retention.<retention id>.enabled.policy* shall have XSystem property names of *.xsystem.retention.enabled.policy.list.<policy name>*.

Method - XSet.applyRetentionEnabledPolicy

.xset.retention.base.duration.policy

shall be used by an XSystem to determine the actual value of an XSet's retention duration for a given scope (as specified by the retention id) in the absence of *.xset.retention.base.duration*.

An XSystem shall be subject to the same update rules (see Table 56) in updating a retention management value in a retention management policy agreement as the corresponding retention value management property. Likewise, a XAM application shall be subject to the same update rules with respect to the policy-determined retention values as the corresponding retention value management property, when updating a retention management policy property.

The XSystem policy list for *.xset.retention.base.duration.policy* shall have XSystem property names of *.xsystem.retention.duration.policy.list.<policy name>.*

Method - XSet.applyBaseRetentionPolicy

.xset.retention.<retention id>.duration.policy

shall be used by an XSystem to determine the actual value of an XSet's retention duration for a given scope (as specified by the retention id) in the absence of *.xset.retention.<retention id>.duration*.

An XSystem shall be subject to the same update rules (see Table 56) in updating a retention management value in a retention management policy agreement as the corresponding retention value management property. Likewise, a XAM application shall be subject to the same update rules with respect to the policy-determined retention values as the corresponding retention value management property, when updating a retention management policy property.

The XSystem policy list for *.xset.retention.<retention id>.duration.policy* shall have XSystem property names of *.xsystem.retention.duration.policy.list.<policy name>.*

Method - XSet.applyRetentionDurationPolicy

.xset.deletion.autodelete.policy

shall be used by an XSystem to determine the actual value of XSet autodelete in the absence of *.xset.deletion.autodelete*.

Method - XSet.applyAutoDeletePolicy

.xset.deletion.shred.policy

shall be used by an XSystem to determine the actual value of XSet shred in the absence of *.xset.deletion.shred*.

Method - XSet.applyShredPolicy

.xset.storage.policy

shall be used by an XSystem to determine how to manage an XSet with respect to the storage management capabilities of the XAM Storage System that are outside the scope of XAM, e.g., storage performance, resiliency, and virtualization.

Method - XSet.applyStoragePolicy

9.3.3.1 Policy Management Property Methods and the Open XSet FSMs

The methods that create or alter the XSet policy management properties shall have the following effects on the XSet finite state machines (FSMs) specified in Section 8.5, “XSet Instance Finite State Machine (FSM)”:

- If the XSet policy management property does not exist in the XSet, then creating the property shall cause the same FSM effects as creating any other field. The field shall be set as binding or nonbinding by the appropriate field creation method. See Section 6.4, “Methods that Operate on Fields” for more information.
- If the XSet policy management property exists in the XSet, then XSet.apply<management policy> that alters the property value, i.e., policy name, shall cause the same FSM effects as an XSet.set<stype> method.
  - If the existing XSet policy management property is nonbinding, then XSet.apply<management policy> shall cause the FSM effects of a nonbinding modification.
  - If the existing XSet policy management property is binding, then an XSet.apply<management property> shall cause the FSM effects of a binding modification.

If the XSet policy management property exists in the XSet, then an XSet.apply<management policy> that alters the property’s binding attribute shall cause the same FSM effects as XSet.setFieldAsNonbinding or XSet.setFieldAsBinding, whichever is appropriate.

9.3.4 XSet Policy Management Hierarchy

The XSet policy management properties, together with the retention and deletion value management properties, are conceptually organized in a three-level hierarchy, with the retention and deletion value management properties as the third level:

- Level 1: *.xset.management.policy*
  - Level 2: *.xset.retention.<retention id>.duration.policy*
    - Level 3: *.xset.retention.<retention id>.duration*
  - Level 2: *.xset.retention.<retention id>.enabled.policy*
    - Level 3: *.xset.retention.<retention id>.enabled*
  - Level 2: *xset.deletion.autodelete.policy*
    - Level 3: *.xset.deletion.autodelete*
  - Level 2: *xset.deletion.shred.policy*
    - Level 3: *.xset.deletion.shred*
  - Level 2: *.xset.storage.policy*

Per the definitions of the policy management properties, the precedence order that a XSystem shall use for XSet retention, deletion, and storage management is:

1. Retention and deletion value management properties (Level 3)
2. Retention, deletion, and storage management discipline policy properties (Level 2)
3. Principal management policy property (Level 1)

Given this order of precedence, an XSet-creating XAM application may control XSet retention and deletion management by setting the retention and deletion value or policy management properties (level 3 or level 2) when the application creates the XSet.

The management policy hierarchy has relevance when a XAM application creates or updates any of the retention, deletion, storage, or principal policy management properties in the hierarchy. An XSet policy management property, which has or would have precedence for a level 3 property, shall not be changed or created if that property setting would violate the update rules (see Table 56) that apply to the specific level 3 value management property.

For example, the *.xset.retention.<retention id>.duration* property has an update rule that it can only be updated to a larger value. If the actual value for XSet base retention duration is 2 years, as determined by *.xset.retention.foo.duration.policy* (where retention id = foo), an attempt to create and set *.xset.retention.foo.duration* to 1 year shall generate a non-fatal error. Whereas, an attempt to create and set *.xset.retention.foo.duration* to 3 years is successful, with precedence given to *.xset.retention.foo.duration*. *.xset.retention.foo.duration.policy* remains in the XSet, but this property shall no longer be used to determine the actual retention duration value for the “foo” retention scope.

The XSet policy management hierarchy places a restriction on the XSystem policy agreements that correspond to level 1 and level 2 XSet policy properties. If the level 1 and level 2 XSet policy values are identical (i.e., they have the same policy name), then they shall also, in effect, represent the same policy. Specifically, the actual value determined from a level 2 retention or deletion policy property shall be the same actual value determined from the level 1 policy property.

For example, “foo” is a valid policy name for both *.xset.management.policy* and *.xset.retention.bar.duration.policy* (where retention id = bar). The “foo” retention duration policy for the “bar retention scope is defined to be 2 years. An XSet that has *.xset.retention.bar.duration.policy* = “foo” and does not have *.xset.retention.bar.duration*, has an actual base retention duration value of 2 years. Likewise, an XSet that has *.xset.management.policy* = “foo” and does not have *.xset.retention.bar.duration.policy* or an *.xset.retention.bar.duration*, also has an actual retention duration value of 2 years.

9.3.5 XSet Management Policy Default

As described in Section 9.2, “XSet Retention and Deletion Value Management Properties” and previously in this section, XSet retention and deletion value and policy management properties may be created, and possibly altered, throughout the life of the XSet. If a XAM application creates a new XSet and elects to not define the retention, deletion, and storage management properties, a default mechanism shall determine the governing retention and deletion values and storage policy.

When XSet.commit creates a new XSet and *.xset.management.policy* does not exist, XSet.commit shall create *.xset.management.policy*, the principal management policy property, and shall set its value to the policy name found in *.xsystem.management.policy.default*. The XSystem policy list, *.xsystem.management.policy.list.<policy name>,* shall be non-empty and *.xsystem.management.policy.default* shall contain a management policy name found on the XSystem management policy list. Therefore, *.xset.management.policy* shall exist in every committed XSet.

9.3.6 getActual Methods for Retention and Deletion Value Management Properties

The getActual methods shall return the actual values determined from the XSet policy management hierarchy for the retention and deletion value management properties. If the value management property exists in the XSet, then the value returned by the getActual method shall be the same value obtained using the XSet.get<type> method. If the value management property does not exist in the XSet, then the value returned by the getActual method shall be the policy-determined value. If a policy-determined value is not established, then the getActual method shall return a non-fatal error.

The getActual methods are:

- XSet.getActualRetentionDuration
- XSet.getActualRetentionEnabled
- XSet.getActualAutoDelete
- XSet.getActualShred

The XSet.get<management actual value> methods above shall cause the same XSet finite state machine effects as an XSet.get<stype> method as specified in Section 8.5, “XSet Instance Finite State Machine (FSM)”.

9.4 XSet Hold Properties

A XAM application may place an XSet on hold. When an XSet is on hold, XAM applications shall be subject to failures or unexpected state changes on XSet operations, which would otherwise be successful if the XSet was not on hold. An XSystem shall maintain an on-hold XSet in readonly mode with respect to the application XSet access and shall prohibit XSet deletion, either automated or explicit. XAM applications shall tolerate these XSet on-hold failures or state changes.

An XSet shall be placed on hold via XSystem.holdXSet and shall be taken off hold via XSystem.releaseXSet (see Figure 29). The hold string identifier specification shall bind the hold/release pair together, and thus, implicitly defines the hold/release time criteria.

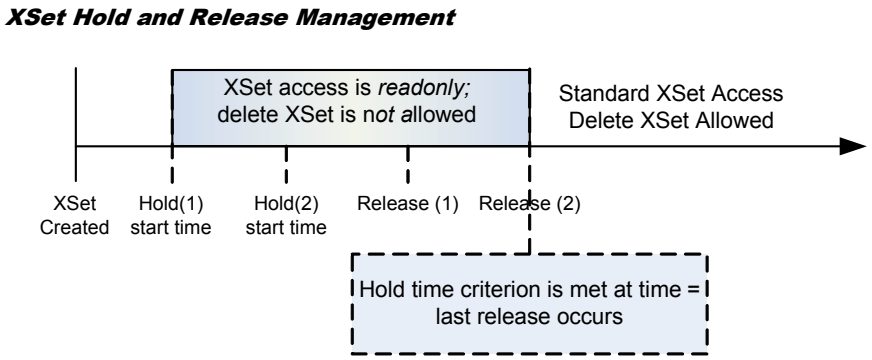

Figure 29 – XSet Hold and Release Management

The hold property, *.xset.hold*, shall be added to the XSet with a value of FALSE, indicating that the XSet is not on hold on initial successful commit of a new XSet.

When a XAM application subsequently places an XSet on hold with XSystem.holdXSet, *.xset.hold* shall be updated with a value of TRUE, indicating that the XSet is on hold and the specified hold identifier shall be added to the hold list as the*.xset.hold.list.<hold id>* property with value *<hold id>*. When a XAM application releases a hold on an XSet using XSystem.releaseXSet, *.xset.hold.list<hold id>* shall be removed from the XSet. And if the removed hold list property is the last one and the hold list becomes empty, then *.xset.hold* shall be set to FALSE, to indicate that the XSet is no longer on hold. A XAM application may determine the on-hold status of an XSet by the *.xset.hold* property.

A XAM application may use the XIterator mechanism to discover the hold list.

Table 64 lists the hold properties, which are described in the paragraphs following the table.

Table 64 – Hold Properties

Property Name	stype	Binding	Readonly	Comments
.xset.hold	xam_boolean	FALSE	TRUE	Added at XUID creating commit and set to FALSE
.xset.hold.list.<hold id>	xam_string	FALSE	TRUE	Value=<hold id>

.xset.hold

shall indicate the on-hold status of an XSet. When XSet.commit creates a new XSet and *.xset.hold* does not exist, XSet.commit shall create *.xset.hold* and shall set its value to FALSE. When a hold list is created on the XSet, the value of *.xset.hold* shall be set to TRUE. A release that removes the last *.xset.hold.list.<hold id>* property shall update *.xset.hold* to FALSE. The *.xset.hold* property shall not be created on the XSet instance that is created by XSystem.copyXSet and shall be removed from the XSet instance on a binding modification.

Methods - XSystem.holdXSet, XSystem.releaseXSet

.xset.hold.list.<hold id>

shall identify the hold with a xam_string hold identifier when an XSet is placed on hold. *.xset.hold.list.<hold id>* shall be added with a value of *<hold id>*, and *.xset.hold* shall be set to TRUE. A release that specifies the identical *<hold identifier>* shall remove *.xset.hold.list.<hold id>* and update *.xset.hold* to FALSE, if the last *.xset.hold.list<hold id>* is removed.

.xset.hold.list<hold id> shall not be created on the XSet instance that is created by XSystem.copyXSet and shall be removed from the XSet instance on a binding modification.

Methods – XSystem.holdXSet, XSystem.releaseXSet

9.5 Reset Management Fields

As mentioned in Chapter 8, “XSet Operations”, certain XSet operations may create a new, uncommitted XSet instance from an existing XSet. When this happens, the new, uncommitted XSet instance shall inherit all retention, deletion, and storage value and policy management properties that exist in the existing XSet. Furthermore, hold properties shall not be propagated into the new, uncommitted XSet instance. The on-hold property shall be created and set to a value of FALSE, on initial successful commit of the new XSet. If a XAM application wants to remove the value and policy management properties, rather than have them inherited by the new uncommitted XSet instance, then the XAM application may use XSet.resetManagementFields to remove all value and policy management properties from the XSet instance.

Method - XSet.resetManagementFields

XSet.resetManagementFields shall cause the same XSet finite state machine effects as a binding modification, as specified in Section 8.5, “XSet Instance Finite State Machine (FSM)”.

10 Query

10.1 Overview of Query

The general purpose XAM API is intended to provide a vendor-independent method for storing and retrieving application data. In addition to specific API methods to be used for accessing data, a query-based interface is also included. This interface enables an application to access data using content-based criteria. These criteria are expressed as relationships between XSet properties and, in some cases, content queries of some XSet streams.

The XAM query language is based on a subset of the SQL standard. Specifically, XAM queries look like the select statement of SQL. An example XAM query is shown as follows:

```
select ".xset.xuid" where "com.example.subject" = 'Electronics Service, Unit #16'
```

The XAM query function is designed around the common concepts and capabilities provided by the XAM API. A query is performed by creating a query XSet, then submitting it as a XAM job. As results are found matching the query, they are stored into the same XSet. If the query XSet is committed, the query runs asynchronously from the application, and the results are persisted across application sessions. In response to a query, the XAM query system will return the XUID information to the application. The application can use the XUIDs to reference the selected XSets to read, change, or process according to the application's business logic.

XAM query should not be confused with the more general purpose SQL relational databases. XAM query is not intended to provide the same performance guarantees seen in a mature relational database management system. XAM Storage Systems are generally designed to be archives of data, rather than relational databases. Example uses of query include locating the following types of records:

- Archived medical data records for a patient
- A collection of telephone data records referencing some phone number
- A computer backup data set containing a named file

Refinements of these basic searches can be extended using the XAM query relational operators to narrow the search. These search strategies might include modifiers to searches to include date ranges, user identifiers, etc. Metadata contained in XSets is flat with respect to the XSet, and relationships beyond 'in the XSet' should not be expected. An application will be unable to store data in a table form or perform SQL-style, row-level, comparisons between data types.

XAM supports two levels of query, level 1 and level 2. Level 1 allows query on simple metadata, while level 2 allows query into stream content.

- The level 1 query capability restricts the *where* clause to relationships between property fields in an XSet. A storage vendor shall implement level 1 query capability within an XSystem.
- The level 2 query capability, in addition to level 1 expressions, allows for expressing relationships between the content of some XStreams. A storage vendor may implement level 2 query capability within an XSystem.

In both cases, a single language and grammar describe all of the capabilities. Both levels of query are accessed through a single, defined job type that all XAM Storage Systems shall support.

10.2 Query Goals

The XAM query capability has several goals:

- Provide a primary way of locating XSets appropriate to an application-defined criteria
- Provide a sufficiently rich expressive grammar to allow applications to describe a wide range of data collections
- Leverage concepts from a well known query language (SQL)

10.3 Introduction to the Query Language Grammar

The XAM Query Language (XAM QL) is modeled on the SQL select statement. Two parts to the statement allow the application writer to control the contents of the query. Consider the query example:

```
select ".xset.xuid" where ".xam.time.xuid" > date('2006-01-01T00:00:00.0')
```

The first part (the select clause) specifies that the application is requesting a list of XUID values. Unlike SQL, the return value ".xset.xuid" is required and shall be the only allowable value. The second part (the where clause) allows specification of a subset of XSets to be returned in the results. For XAM 1.0, the select clause shall be present and contain only the phrase "select '.xset.xuid'".

Because XAM QL allows any characters to be used as field names, all instances of field names in the XAM query string shall be quoted. Quoting of XAM field names is accomplished by using double quotes, whereas, string literals use single quotes.

The second part of the query, the where clause, is optional and provides the greatest amount of application control. This optional clause allows an application to use the simplest form of a XAM query:

```
select ".xset.xuid"
```

In this example, the results stream will contain a list of every XSet that is readable at the time of the query. More typical examples are like the previous example, which specifies all XSets created after January 1, 2006.

10.4 Level 1 Query: Where Clause Operators

The operators listed in this section are defined to operate on XSet properties and field attributes, also known as a level 1 query. All XAM Storage Systems shall implement level 1 query. This type of query is restricted to comparisons between:

- XSet properties and literal values
- Field attributes and literal values

XSets with named properties of differing types, as specified by the literal expression, shall not be included in the query results.

Table 65 shows which field and literal types can be validly compared.

Table 65 – Valid Comparisons

Literal	Field Type				
	xam_int	xam_double	xam_string	xam_datetime	xam_xuid
int	*	*[1]			
double	*[2]	*			
string			*		
datetime				*	
XUID					*

1. This comparison may be narrowed to only xam_double properties, by specifying the property type equal to ‘application/.xam.snia.xam.double’ in the where clause.
2. This comparison may be narrowed to only xam_int properties, by specifying the property type equal to ‘application/.xam.snia.xam.int’ in the where clause.

Table 66 shows which comparison operators may be used, depending on the field type.

Table 66 – Comparison Operators

Field Type	Operators						
	=	<>	<	<=	>	>=	like
xam_boolean	*	*					
xam_int	*	*	*	*	*	*	
xam_double	*	*	*	*	*	*	
xam_string	*	*	*	*	*	*	*
xam_datetime	*	*	*	*	*	*	
xam_xuid	*	*					

Table 67 describes the valid operators in the level 1 query.

Table 67 – Operator Descriptions

Operator	Description
=	Equality test between values
<>	Inequality test between values Note: XAM query allows ’!=’ as a synonym for ’<>’.
<	Less than
<=	Less than or equal
>	Greater than

Table 67 – Operator Descriptions

Operator	Description
>=	Greater than or equal
AND, OR, NOT	Logical operators
Like[1]	Simple pattern matching on strings

1. This operator is similar to SQL, but is limited to the '%' operator, which matches zero or more characters. This operator does not support any other pattern matching models typical in SQL. See the examples shown in Table 68.

The like operator compares literal strings against xam_string properties. The string literals in the comparison is a pattern which may contain a wildcard character '%'. This wildcard character shall match zero or more characters and is the only wildcard expression allowed in the string literal.

Table 68 summarizes the results of various like expressions given a property "com.example.prop" containing the value 'abcdefg'.

Table 68 – Summary of "like" Operator

Expression	Result
"com.example.prop" like 'abc%'	TRUE
"com.example.prop" like '%efg'	TRUE
"com.example.prop" like '%def%'	TRUE
"com.example.prop" like 'ab%fg'	TRUE
"com.example.prop" like 'xb%'	FALSE
"com.example.prop" like '%'	TRUE

10.4.1 String Operators

Applications are given a great amount of freedom in how strings are generated in UTF-8 [RFC 3629]. Because of this freedom, applications are expected to follow UTF-8 rules and limitations. XAM shall validate UTF-8 correctness for string-valued properties. String literals supplied to the XAM query system shall be verified for UTF-8 correctness. Any non-conforming UTF-8 string literals shall generate a XAM non-fatal query syntax error.

Some of the complications that could affect the query string comparisons include:

- Lack of canonical representations for some glyphs
- Non-printable characters
- Single vs. multiple glyph characters
- Different string representations for the same data from multiple applications

Solving all of these issues is outside the scope of XAM query and XAM. XAM query string comparison operators shall operate on a byte-by-byte basis. For relational operators (e.g., “>”, “<”, etc.), the relationships shall be defined by the byte values. XAM query string comparisons shall be case sensitive. The XAM Storage System shall not try to normalize textual values to ignore case, nor are any functional values supplied to do so. All data normalization and UTF-8 conformance shall be the responsibility of the application. To maximize application interoperability, applications are encouraged to use the IDN profile of stringprep as defined in [RFC 3491] and [RFC 3454].

String literals shall be written using single quotes. For example:

```
select ".xset.xuid" where "com.example.string-property" = 'My string literal'
```

To specify a single quote in a string, the single quote shall be written into the string using the backslash quote notation. For example:

```
select ".xset.xuid" where "com.example.ownership" = 'Tom\'s'
```

The length operator shall return the number of bytes used by the string. It should not be used to compare the number of characters, as this comparison depends on the character encoding being used. String properties in XAM are limited in length (see Section 6.3, “XAM Fields”), and string literals longer than this shall generate a XAM non-fatal query syntax error.

10.4.2 Numeric Property Value Comparisons

The query system allows specification of numeric literal values for comparison, requiring the query system to compare numeric value property fields. For purposes of comparisons, all numeric-valued properties shall be comparable with one another. When comparing a xam_double with an xam_int, the xam_int value shall be promoted to a xam_double for the purposes of comparison. The promotion of the numeric value shall be for comparison purposes only and shall have no effect on any XSet property value. Numeric promotion shall occur whenever the type of the literal and a property value differ, regardless of other subclauses in the where expression. Literal values are typed by how they are represented. Integer literal values shall contain no decimal point, and double literal values shall contain the appropriate decimal point. Table 69 summarizes this behavior.

Table 69 – Query Numeric Comparisons of Different types

Type of Property	Type of Literal	
	xam_int	xam_double
xam_int	=	Promote property value
xam_double	Promote literal value	=

When required, the comparisons may be restricted to a specific type by using the typeof() function. This restriction allows the query to specify tighter control on the comparisons. The following example shows how a query can be restricted to only integer-valued properties. In this example, only XSets with a xam_int valued com.example.property will be considered:

```
select ".xset.xuid" where "com.example.property" = 12 and typeof("com.example.property") =
   'application/vnd.snia.xam.int'
```

Any XSets containing a double-valued property named com.example.property shall fail the comparison and will not be included in the result set.

A query may also restrict the numeric comparison to double, as illustrated in this example:

```
select ".xset.xuid" where "com.example.property-two" = 124.0 and
    typeof("com.example.property-two") = 'application/vnd.snia.xam.double'
```

In this case, any XSets with integer-valued properties named com.example.property-two will fail the comparison and will not be included in the result set. Also note that this example specifies a double literal, which causes a double-to-double comparison.

10.4.3 Numeric Comparisons with IEEE-754 Exception Values

The IEEE-754 Floating Point Numeric [IEEE754] standard defines two exceptional values which can be stored into a XAM Storage System by an application. These values are:

- NaN - An IEEE-754 value representing a “non-number”
- Inf - An IEEE-754 value representing “infinity”

The XAM query system shall compare these values as defined below, but unlike other systems, the XAM query system shall not generate an exception during the processing of the query.

All comparisons with NaN shall fail by returning a FALSE condition. For instance, consider the query with a property named com.example.not-number, containing the value NaN.

```
select ".xset.xuid" where "com.example.not-number" > 12
```

This query will never return any results. In other words, the result stream in the query job XSet will contain zero XUID values when the query has completed. This is because the comparison Nan > 12 is always going to fail.

The value Inf shall not fail and shall generate correct comparisons. For instance, the following query with a property named com.example.+big, contains the value of +Inf.

```
select ".xset.xuid" where "com.example.+big" > -99
```

The XSet which has com.example.+big with the value of +Inf will be included in the results because +Inf is larger than -99.

Nan, +Inf, and -Inf shall be acceptable values as numeric literal values and shall be considered to be xam_double values.

10.4.4 Field Attribute Accessor Functions

The field attribute accessor functions are exists(), typeof(), readonly(), binding(), and length(). Descriptions and examples of these functions are as follows:

- exists() tests for the existence of an XSet field as specified by field name (either property or stream). An XSet containing the named field shall evaluate this function to TRUE, and an XSet that does not contain the named field shall evaluate this function to FALSE. For example,

  ```
  select ".xset.xuid" where exists("com.example.name")
  ```

- typeof() returns the MIME type of the named XSet field. This function is only suitable as part of a string comparison. For example,

```
select ".xset.xuid" where typeof("com.example.data") = 'text/plain'

select ".xset.xuid" where typeof("com.example.data") like 'text%'
```

The typeof function may be used whenever an application could use a field reference to a string-valued property. Comparisons with any non-string literal value shall generate a non-fatal error.

- readonly() evaluates to TRUE when the field is marked as readonly. For example,

  ```
  select ".xset.xuid" where readonly("com.example.flag"")

  select ".xset.xuid" where not readonly("com.example.name")
  ```

- binding() evaluates to TRUE when the field is marked as binding. For example,

  ```
  select ".xset.xuid" where binding("com.example.case_id")

  select ".xset.xuid" where not binding("com.example.subject")
  ```

- length() returns the length, in bytes, of the named field. This function is more useful for streams, but is also defined for properties. The length() function used on property fields returns the length as defined in Table 5, “stypes”. The result of length() on a stream being updated is unspecified. For example,

  ```
  select ".xset.xuid" where length("com.example.data") > 1024
  ```

10.4.5 Logical Operators

Subclauses within the where expression may be combined and modified by using the logical operators not, and, and or. These operators function the same as those defined in SQL.

- not negates the Boolean expression. For example,

  ```
  where not binding("com.example.property")[1]
  ```

  Only selects XSets with nonbinding properties named ’com.example.property’

  ```
  where not ("com.example.property1" < 12 or "com.example.property2" > 100)
  ```

  Only selects XSets with property1 >= 12 AND property2 <= 100

- and requires both comparisons to be TRUE before including the XSet in the results.

  ```
  where typeof("com.example.stream") = 'image/jpeg' and length("com.example.stream") >
     1024
  ```

  Only selects XSets containing Jpeg images larger than 1024 bytes

1. Note that normal Boolean rules apply. This example may also be written as ’where binding(com.example.property) == FALSE

- or evaluates to TRUE if either subclause evaluates to TRUE

  ```
  where typeof("com.example.stream") = 'image/jpeg' or typeof("com.example.stream") =
      'image/gif'
  ```

  Only selects XSets containing the named stream of image type Jpeg or GIF

These logical operators allow the query author to combine property relationships in ways that are unique and useful to the application.

10.4.6 Selector Functions for XUID and Date-Time Properties

The query language accepts selector functions to allow a specification of some values that may violate other parts of grammar. These value types are date-time and XUID-valued literals.

The function date takes a properly formed date-time value, specified as a string. The value shall be consistent with the date-time specification being used by XAM (see Section 6.3.3, “Properties”). An improperly formed data-time value shall generate a non-fatal error during the parsing of the query.

The function xuid takes a printable format of a XUID (base64 encoded) and is specified as a string. The value shall be consistent with the XUID specification. An improperly formed XUID literal shall generate a non-fatal error during the parsing of the query.

10.5 Level 2 Query: Where Clause Content Search Operators

Optionally, a vendor may choose to implement the where clause XStream search operators. These functions allow an application to perform searches through appropriate XStreams. For instance, an application can select XSets where an XStream contains the word “SNIA.” An application may determine if level 2 queries are supported by reading the Boolean property *.xsystem.job.query.level2.supported* from the XSystem instance (see Section 6.3, “XAM Fields”). If this Boolean property is TRUE, then the application may use level 2 constructs.

Level 2 query expressions are part of the XAM QL and, if level 2 is supported, the vendor shall support level 2 as part of the standard XAM query job type. Vendors may implement query extensions via vendor-specific job types, but these shall not be considered interoperable with different vendor’s XAM Storage Systems.

This functionality is intended to apply to any XStream content type (as specified by the stream’s MIME type) that is amenable to string-based analysis. Examples include ’text/plain’, ’application/msword,’ and so on. Any system that supports level 2 query shall support query against XStream types of ’text/plain.’ The ability to perform textual search with additional XStream types is vendor specific.

Vendor systems may search text slightly differently, depending on the character sets used to store the content. For the following discussion, a token is a categorized block of text, usually consisting of indivisible characters. For most European character sets, a token is a collection of characters separated by white space. For idiographic languages, a token may be a series of octet values, since white space is implied.

The following where clause operators may be used in a query submitted to a level 2 compliant system.

- contains – Indicates that the specified stream contains the indicated token. This operator shall accept a single token. For example,

  ```
  select ".xset.xuid" where "com.example.mystream" contains('foo')
  ```

  This example indicates that the word ’foo’ is contained somewhere in the stream ’mystream’.

- before – Indicates that one token is before the other specified token in terms of location. This operator shall accept two tokens as arguments. For example,

  ```
  select ".xset.xuid" where "com.example.mystream" before('first', 'second')
  ```

  This example indicates that the word 'first' is before an occurrence of 'second.'

- after – Indicates that one token is after the other specified token in terms of location. This operator shall accept two tokens as arguments. For example,

  ```
  select ".xset.xuid" where "com.example.mystream" after( 'last', 'first' )
  ```

  This example indicates that the word 'last' is after an occurrence of 'first.'

- within – Indicates that the specified stream contains the indicated tokens near each other. This operator shall accept three arguments, two string tokens, and the token distance allowed. There shall be no implied ordering for the two token. For example,

  ```
  select ".xset.xuid" where "com.example.mystream" within( 'word1', 'word2', 7 )
  ```

  This example indicates that the word 'word1' occurs within 7 tokens of 'word2.'

10.6 Complete Grammar

The following ABNF [RFC 4234] grammar represents the XAM QL. This grammar uses the rules, "char" and "digit" from the core rules of the ABNF RFC. The XAM QL is case insensitive and uses the US-ASCII [RFC 1345] character set. String literals and field names specified in a query expression shall be full UTF-8.

```
XAM-query           = "select" xuid-property-name [ "where" where-expression ]
xuid-property-name  = dq ".xset.xuid" dq  ; as in ".xset.xuid"
property-name       = dq *(char) dq
where-expression    = term
term                = [ "not" ] (factor / "(" term ")" )  ["and" term / "or" term]
factor              = ( property-name binary-op literal-expression ) /
                      ( attribute-bool-fn ) /
                      ( attribute-bool-fn binary-op literal-expression ) /
                      ( attribute-function binary-op literal-expression ) /
                      ( property-name level2op )
binary-op           = "=" / "<>" / / "!=" / "<" / "<=" / ">" / ">=" / "like"
attribute-bool-fn   = attribute-bool-name "(" property-name ")"
attribute-bool-name = "exists" / "binding" / "readonly"
attribute-function  = attribute-fn-name "(" property-name ")"
attribute-fn-name   = "typeof" / "length"
level2op            = "contains" "(" xam-string-literal ")" /
                      "before" "(" xam-string-literal "," xam-string-literal ")" /
                       "after" "("  xam-string-literal "," xam-string-literal ")" /
                       "within" "(" xam-string-literal "," xam-string-literal ","
                                    xam-integer-literal ")"
literal-expression  = ( ["+" / "-"] (xam-integer-literal / xam-double-literal) /
                       xam-string-literal / xam-boolean-literal /
                                    xam-date-literal/
```

```
                        xam-xuid-literal )
xam-boolean-literal = "TRUE" / "FALSE"
xam-integer-literal = 1*digit
xam-double-literal  = ( *digit) "." (*digit) [exponent] /
                      (*digit) [exponent] /
                      "NaN" / ["+" / "-"] "Inf"
exponent            = ("e" / "E") ["+" / "-"] 1*digit
xam-string-literal  = sq *(char) sq; String as in 'abc def ghi'
xam-date-literal    = "date" "(" xam-string-literal ")"; date( '2006-04-01T00:00' )
xam-xuid-literal    = "xuid" "(" xam-string-literal ")"; xuid ( 'AAAAAwAbB68AAH' )
sq                  = %x27       ; This is a single quote, as in 'foo'
dq                  = %x22       ; This is a double quote, as in "bar"
```

10.6.1 Reserved Key Words and Operator Precedence

The only reserved key words shall be select, where, and, or, not, like, exists, binding, readonly, typeof, length, TRUE, FALSE, before, after, contains, and within.

Operator precedence shall be as follows:

1. Attribute functions and exponent signs: exists, readonly, typeof, length, binding, +, -[1]
2. Binary operators: =, <>, >, >=, <, <=, like
3. not
4. and
5. or

Operators of the same precedence shall be evaluated left to right within the query. Operator precedence may be overridden using parentheses. The following example illustrates how parentheses change the meaning of a query:

```
select ".xset.xuid" where not "com.example.bool-prop" and
   "com.example.int-prop" = 42
```

Compared with:

```
select ".xset.xuid" where not ("com.example.bool-prop" and
   "com.example.int-prop" = 42)
```

In the first example, the not operator applies to the com.example.bool-prop property, while in the second example, the not operator applies to the expressions com.example.bool-prop and com.example.int-prop = 42.

1. Plus (+) and minus (-) operators may only appear as part of numeric literals.

10.6.2 Specifying String Literals and Field Names with Special Characters

Table 70 describes the escape sequences for quoted field names and strings.

Table 70 – Escape Sequences for Quoted Field Names and Strings

Escape Sequence	Meaning	Notes
\\	Backslash (\)	
\'	Single quote (')	
\”	Double quote (“)	
\uxxxx	Character with value xxxx	xxxx is a hexadecimal Unicode code only.

If the field name contains double quote characters, each double quote character shall be escaped by using the backslash notation. For example, the field name for the xam_boolean property com.example.”qstring” should be represented in the query as:

```
select ".xam.xuid" where "com.example.\"qstring\"" = TRUE
```

For field names containing a backslash character, the backslash itself shall be escaped. For example, the field name for the xam_double property com.example.file\ratio would be represented in the query as:

```
select ".xam.xuid" where "com.example.file\\ratio" = 100.1
```

String literals which contain a single quote character shall be escaped using the backslash single quote notation. For example, the string literal of ’a quoted string’ would be represented in a query as shown in the following example.

```
select ".xset.xuid" where "com.example.string-prop" = '\'a quoted string\''
```

The only acceptable escape characters shall be those listed in Table 70; any other escape sequence shall generate an error. All quoted strings shall always escape any contained quote characters.

10.7 Job Control and API Methods

A query to a XAM Storage System is run as a job. As discussed in Section 8.9, “XAM Jobs and XAM Job Control”, the input to the query jobs is an XSet. To successfully submit a query job, the query XSet requires two input fields:

- *org.snia.xam.job.command* - Set to ‘xam.job.query’.
- *xam.job.query.command* - This text XStream (MIME type of ‘text/plain; charset=utf-8’) shall contain the XAM query expression string to be processed.

A query job XSet may optionally be committed before or during the execution of the job, but this commit is not required. Failing to commit a job before closing the XSet shall cause the results of the query job to not be persistently stored. The uncommitted XSet will likely terminate the associated job when the XSet is closed. An application may determine if a commit of running jobs is allowed by checking the Boolean property *.xsystem.job.commit.supported* in the XSystem instance (see Section 7.2.3, “XSystem Fields”).

10.7.1 Query Job Specific XSet Fields

Table 71 shows the specific fields created as part of the query job but does not show all the generic job fields. See Section 8.9.2, “Standardized Job Output Fields” for more information.

Table 71 – Query Job-Specific Fields

Field Name	Type	Binding	Readonly
org.snia.xam.job.command	xam_string	Application choice	TRUE when running
xam.job.query.command	text/plain; charset=utf-8	Application choice	TRUE when running
xam.job.query.results	application/ vnd.snia.query.xuid_list	FALSE	TRUE
xam.job.query.results.count	xam_int	FALSE	TRUE
xam.job.query.level	xam_string	FALSE	TRUE

org.snia.xam.job.command

is a xam_string property indicating what job this XSet represents. The value of this string shall be ‘xam.job.query’.

xam.job.query.command

is a UTF-8 text stream (MIME type of ‘text/plain; charset=utf-8’) and shall contain the query string itself and shall be required for the query job.

xam.job.query.results

is a stream (MIME type of ‘application/vnd.snia.query.xuid_list’) containing the binary XUID values resulting from the evaluation of the query.

xam.job.query.results.count

is a xam_int property indicating the number of XUID result values in the stream ‘xam.job.query.results’.

xam.job.query.level

is a xam_string property indicating the level of the query which matches the results. The value shall be either ‘org.snia.xam.job.query.level.1’ or ‘org.snia.xam.job.query.level.2’.

10.7.2 Runtime Behavior of the Query Job

Once the required job control fields are set (*org.snia.xam.job.command* and *org.snia.xam.job.query. command*), the job can be executed successfully by invoking XSet.submitJob. Failure to set these fields shall result in the standard job error field *.xam.job.error* being added to the job XSet with the value set to either ‘org.snia.xam::not_a_job’ or ‘org.snia.xam::unspecified_command’.

Note that when formatting a query command, the calling entity, such as an application, is responsible for verifying the level of query that is supported (1 or 2) and submitting the properly formulated query string. A level 2 type where clause submitted to a XAM system that only supports level 1 query syntax shall result in a non-fatal error field as above, with the value set to:

```
'xam.job.query::level_not_supported'
```

Other syntax errors in the query command string shall result in the error field being created with the value of:

```
'xam.job.query::invalid_command_syntax'
```

All syntax errors shall be non-fatal errors with respect to the query job and shall cause the job to abort.

When XSet.submitJob is successfully executed, the query job shall store the query results in the job XSet in the form of an XStream. This field is named *xam.job.query.results*.

The XStream shall contain a list of XUIDs of the XSets that match the query specification. The XStream shall have a MIME type of 'application/vnd.snia.xam.query.xuid_list' and the XUID values in the stream shall be written using their binary format.

The following fields shall also be added to the XSet by the job. These fields will contain the number of results first and then the query level at which the job was run.

- *xam.job.query.results.count*
- xam.*job.query.level*

The results stream will contain a number of XUIDs at any given time, which is reflected in *xam.job.query.results.count*. This property shall be updated as results are entered into the results XStream. During query processing, this update allows the application to provide confirmation that results are being processed (if needed). In the end, it provides a quick way of determining the count of the number of XUIDs in the results. Because this property is dynamic, it shall be nonbinding and readonly. Updating *xam.job.query.results.count* is not required to be atomic. However, to ensure good application behavior, the XAM Storage System shall add new XUID values to the result stream before updating *xam.job.query.results.count*.

The results of a query run in a level 2 system that are exported to a level 1 compliant system are still legal results. An application may process those results, assuming that all other required XSets have also been imported into the level 1 system. Running the level 2 query in a level 1 system shall result in a non-fatal error as described above.

Applications may determine if the query job has completed by examining the job properties that indicate if the job has completed. See Section 8.9.2.1, "Job Status" for more information.

10.7.3 Query Job Error Codes

A job's run state may be determined by examining the value of *.xam.job.status*. See the job description of this document in Section 8.9.2.1, "Job Status". An XSystem that restarts while active query jobs are being processed shall ensure that, for committed query jobs, the appropriate error code is set or the job is run. The XStorage System vendor may choose to terminate these jobs, restart them, or continue from the point of interruption.

The query job may set the following error codes into *.xam.job.error*. These errors are described in Table 72, “Query Job Error Codes”.

Table 72 – Query Job Error Codes

Error Code	Description
`xam.job.query::level_not_supported`	Query-specific non-fatal error when a specific XSystem is unable to support the level required by the query command.
`xam.job.query::invalid_command_syntax`	A non-fatal syntax error occurred when parsing the query command.
`xam.job.query::insufficient_permission`	The currently authenticated user does not have sufficient permission to execute a query job.
`xam.job.query::insufficient_resources`	The XSystem does not pose sufficient resources to complete the query job.

10.7.4 Result Stream Format

The result stream shall be typed as ‘application/vnd.snia.xam.query.xuid_list’. This stream shall contain binary XUID values that are the result of the query job. Each XUID value in the stream shall be stored in a binary format, as part of an 80-byte record (see Figure 30, “Result Stream”). If the resulting XUID is shorter than 80 bytes, the record shall be zero padded, as shown in Figure 31, “Result Stream with Variable Length XUID Values”. While processing the query, the XAM Storage System shall never write a partial XUID to the result stream. Applications shall read all 80 bytes to ensure that an entire XUID value is returned from the stream. Partial XUID values passed to other XAM API methods shall result in a non-fatal error.

XUID values obtained from the query stream shall be acceptable to all XAM functions taking binary XUIDs as arguments. The application may ignore the zero padding and treat all XUID values in the stream as 80-byte values. Other XAM functions requiring XUID arguments shall accept these values and ignore the zero byte padding.

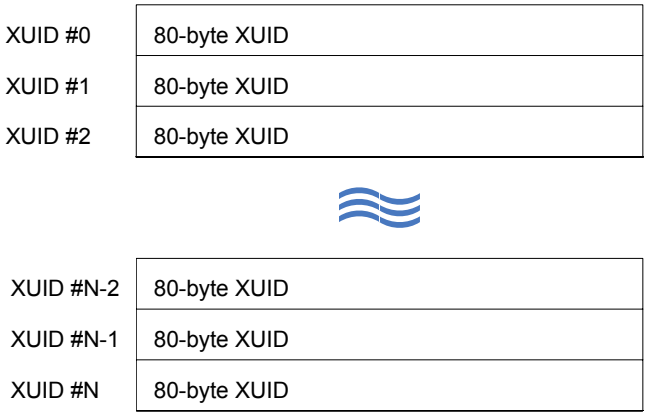

Figure 30 – Result Stream

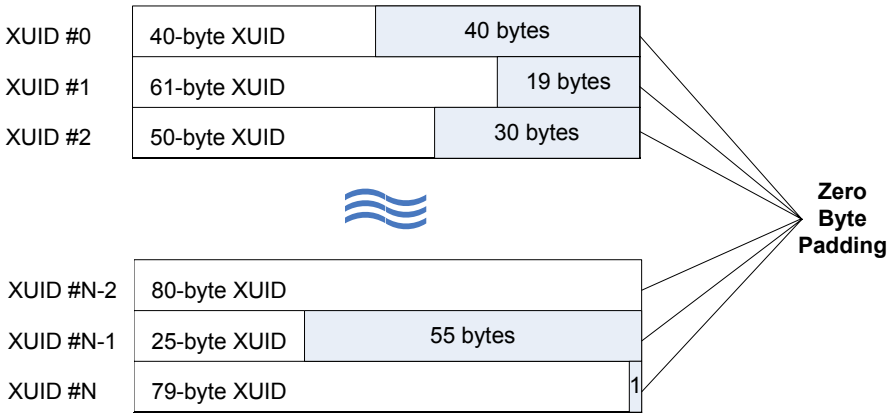

Figure 31 – Result Stream with Variable Length XUID Values

The XUID Iterator, as described in Section A.1.2, “XUIDIterator”, shall present a sequential interface to this stream, but applications may perform random seek operations in the stream to access the results non-sequentially. An application randomly accessing XUIDs within the stream shall perform the appropriate offset calculations, assuming 80-byte XUID values throughout the stream.

10.7.5 Scope of Query

Any XSet property value that is accessible to an application program via the XAM Library shall be capable of being queried by the XAM QL. This requirement shall be TRUE for all standard, vendor-implemented, and application-defined properties.

10.7.6 Runtime Caveats

Applications using the XAM QL should note the following:

- Even though the XAM QL is similar to SQL and supports a modified subset of SQL features, the XAM Storage System may not have been implemented using a database. To ensure interoperability between XAM Storage Systems, the application shall not assume any particular implementation for the XAM query.
- A query does not alter the behavior of the XAM Storage System; no locking or suspend operations are supported. A query result is, therefore, not an instantaneous snapshot of the system. Data stored while the query job is running may or may not be included in the query results.
- The application may evaluate the results of a query job at any time, even before the job has completed. Note that reaching the end of the data in the result XStream does not imply that the result set is complete; completeness of the result can only be evaluated when the job has finished.
- Query jobs have a finite duration and shall complete without intervention. However, a job may be terminated prematurely (halted) using an API call. When a query job is halted, the appropriate value shall be set on the standard job health and status fields (*.xam.job.errorhealth* and *.xam.job.status*, respectively).

10.7.7 Result Stream State After a Job Halt

When halted, the XAM Storage System may place zero or more complete XUIDs in the result XStream of the query XSet. Previously existing rows of the result XStream shall not be modified. Note that in all cases, writing partial XUIDs shall be prohibited. An application shall always see complete XUID values in the

result stream. Halting a query job shall be considered a permanent condition on the query job. After halting a query, an application may resubmit the query job. Continuation of query processing on a halted query job shall not be possible.

10.7.8 Reading Results of In-Process Queries

Result XStreams produced by query jobs may be consumed by applications before the XAM Storage System has finished storing all of the result rows. If, while reading results, the application has consumed all of the available data and the query job is still running, XStream.read shall return zero bytes but not yet indicate an end of file. If an application wishes to wait until new XUID values are written into the result XStream, XStream.asyncRead, specifying a timeout value, should be used.

10.7.9 What Is / Is Not Included in a Query Result

The completeness of a query job can be evaluated by evaluating the health and status fields of the job XSet.

A query result is a transient, point-in-time snapshot of the XAM Storage System, which implies that there is no “locking” of the system during a query. A direct result of this statement is that since the system is changing during a query, certain XSets will be included and others will not.

The general rule is that any XSet that has been stored before the query initiation shall be valid for the search criteria, and any XSet that has been stored after the query has completed shall not be included in the search. XSets that are stored during the actual query job execution will have nondeterministic results—they may or may not be included. Inclusion depends on how the query result set is constructed, and there are no assumptions. Moreover, this rule shall also apply to deletions and expiration of XSets in the XAM Storage System. Thus, any changes during the query will result in nondeterministic results.

The only guarantee is that an XSet, which was stored before the query initiation and not modified before the query was completed, shall qualify as a target for the search criteria.

XAM query guarantees that XSets stored:

- before submitting the query are considered for inclusion in the result set.
- after the query has completed are not considered for inclusion in the result set.
- while the query is being processed may or may not be considered for inclusion in the result set.

The same rules apply to the query result set. A query result set may include XUIDs of XSets that are no longer on the system or an attribute that was modified after the query store. In other words, even though the query result set met the search criteria at the time of the query, due to a change, it no longer qualifies.

For example, the following query has a where clause specifying a nonbinding property:

```
select ".xset.xuid" where "com.example.propertyXYZ" = 5
```

Assume XUID 795 meets this search. After the query stores the XUID, an application changed com.example.propertyXYZ to 9. Later, rerunning the query will not include XUID 795. In other words, the results of a query shall be considered to be valid for the time the query was run, but the results are likely to be different if the same query is run later.

10.7.10 Query and Permissions

With respect to permissions, the assumed behavior is that a query shall operate within the roles/permissions granted to the connection. That means the query results shall only include those XSets that

are visible and accessible to the application, at least from a read perspective, to the role under which the query is executed.

The procedures outlined in Chapter 11, “Security” determine which XAM users can submit and process queries. The query job authorization granule controls the ability to submit queries, and the job-query-commit authorization granule controls the ability to commit an XSet that represents a running query, thereby allowing it to continue after the XAM session ends. Roles that authorize these granules permit the corresponding actions. See Chapter 11, “Security” for more details.

10.8 XAM Query Examples

The semantic examples in this section illustrate XAM query and show the expected behavior from a XAM 1.0 compliant system. For the sake of discussion, assume an XSystem contains the following collections of XSets. As shown in Table 73, the XUID values are “XSET1”, “XSET2”, “XSET3”.

Table 73 – Query Example XSets

XSET1		XSET2		XSET3	
Property	**Value**	**Property**	**Value**	**Property**	**Value**
com.example.foo	1	com.example.foo	77	com.example.foo	6
com.example.bar	’string’	com.example.bar	42		
com.example.num	123.55	com.example.num	100	com.example.num	200

10.8.1 All XSets

This query selects all XSets in the above example:

```
select ".xset.xuid"
```

Result set:

- XSET1
- XSET2
- XSET3

10.8.2 A Subset of XSets

This query selects a subset of the XSets in this system. This example demonstrates the restriction due to the where clause.

```
select ".xset.xuid" where ("com.example.foo" > 0) and ("com.example.foo" < 50)
```

Result set:

- XSET1
- XSET3

10.8.3 Heterogeneous Properties

This query demonstrates the behavior of a where clause when properties are heterogeneous with respect to type. For this example, look at the property “com.example.bar”.

```
select ".xset.xuid" where ("com.example.bar" > 0) and ("com.example.bar" < 100)
```

Result set:

- XSET2

Note: XSET3 does not contain “com.example.bar.” A non-existent property cannot participate in the where clause, so XSET3 is restricted from the result set. XSET1 has the property, but it is an stype string. This string is not an appropriate type for participating in the where clause for this example, so XSET1 is excluded from the result set.

10.8.4 The exists() Function

This query demonstrates the behavior of the exists() function.

```
select ".xset.xuid" where exists("com.example.bar")
```

Results set:

- XSET1
- XSET2

The exists operator only tests for the presence of an XSet field. In this case, all the XSets containing a field named com.example.bar will be returned into the result set.

10.8.5 The String like Operator

This query demonstrates the behavior of the string like operator:

```
select ".xset.xuid" where "com.example.bar" like '%ing%'
```

Result set:

- XSET1

The like operator has the same behavior as SQL. Note that because XSET2 does not have a string-valued property, it is not eligible for inclusion in the result set. XSET3 is not eligible because com.example.bar does not exist.

10.8.6 Numeric Comparisons When Promoting a xam_literal

This query demonstrates numeric comparisons when promoting a xam_int literal:

```
select ".xset.xuid" where "com.example.num" >= 124
```

Result set:

- XSET3

XSET3 is included in the result because the value of property com.example.num, 200, is greater than the literal 124. XSET1 is not included in the result because the value of property com.example.num, 123.55, is

not greater than or equal to literal 124.0 (promoted). XSET2 is not included because the value of property com.example.num, 100, is not greater than or equal to 124.

10.8.7 Numeric Comparisons When Promoting a xam_int Property

This query demonstrates numeric comparisons when promoting a xam_int property:

```
select ".xset.xuid" where "com.example.num" >= 124.6
```

Result Set

- XSET3

XSET3 is included in the result because the value of property com.example.num, 200.0 (promoted), is greater than the literal 124.6. XSET1 is not included in the result, because the value of property com.example.num, 123.55, is not greater than or equal to 124.6. XSET2 is not included in the result set because the value of property com.example.num, 100.0 (promoted), is not greater or equal to 124.6.

10.8.8 Numeric Comparisons When Restricting a Property Type

This query demonstrates numeric comparisons when restricting to a particular type of property:

```
select ".xset.xuid" where ("com.example.num" >= 123) and
    typeof("com.example.num") = 'application/vnd.snia.xam.int')
```

Result Set

- XSET3

XSET3 is included in the result because the value of property com.example.num, 200, is greater than the literal 123. XSET1 is not included in the result because, even though the value of property com.example.num, 123.55, is greater than or equal to 123.0 (promoted), the type of the property is not a xam_int. XSET2 is not included in the result set because the value of property com.example.num, 100, is not greater than or equal to 123.

11 Security

11.1 XAM Security Overview

XAM security functionality consists of three security disciplines: XAM application authentication, XSystem authorization, and XSet access control.

- **XAM application authentication**. The SASL (Simple Authentication and Security Layer) framework [RFC 4422] enables an application that uses the XAM API to authenticate to the XSystem as part of connecting to the XSystem. SASL may also establish an authorization identity for authorization and access control purposes.

- **XSystem authorization.** When connecting to an XSystem, authorization determines which functional elements of the API may be called during the resulting XAM session. The SASL authorization identity may be an input to authorization decisions.

- **XSet access control.** An XSet may have an access control policy applied to it that determines whether an API method is permitted. XSet access permissions can forbid modifications or deny access to some or all XAM applications. A XAM policy name is used to refer to the XSet access control policy.

XAM separates policy from mechanism for XSystem authorization and XSet access control. XAM specifies the enforcement mechanisms for XSystem authorization and XSet access control, but does not specify how to determine what access restrictions are to be enforced. The latter area, including configuration and operation of XSystem authorization and XSet access control policies, is outside the scope of this standard. XAM specifies how a XAM application can determine what controls are being enforced (i.e., the permitted vs. denied methods) without attempting to call the affected methods. XAM also specifies some inputs to authorization and access control decisions (e.g., SASL authorization identity or XSet access control policy name), but does not specify how these inputs are used, whether other inputs are used, or how the decisions are made. XAM specifies a standard set of roles for authorization identities, but does not require that this set be used. The operation and configuration of XSystem authorization and XSet access control policies are outside the scope of this specification. These policies may consider inputs other than what is specified by the XAM standard in making decisions.

XAM does not specify cryptographic integrity or confidentiality for either communications or stored data. Cryptographic integrity detects unauthorized changes to communication and/or stored data via means outside the XAM API. Confidentiality protects communications and/or stored data from unauthorized disclosure via means outside the XAM API. A VIM or XAM Storage System may implement cryptographic integrity and confidentiality for communications or stored data (e.g., by using a security protocol such as Transport Layer Security (TLS) or by encrypting the data before storing it in the XAM Storage System). The means of applying such services to XSets are vendor specific (e.g., a storage policy in *.xset.storage.policy* may be used, or this may be specified as part of the connect string). A VIM or XAM Storage System may participate in the SASL authentication, so that it can link these security measures to the authenticated identity of the XAM application.

XSet visibility is a security-related concept. As discussed in Section 7.2, “XSystem”, an XSystem is a logical container of XSets, an XSystem may contain more than one XSet, and an XSet may be contained by more than one XSystem. An XSet is only visible through the XAM API via the XSystem or XSystems that contain the XSet; the XSet shall not be visible through any XSystem that does not contain the XSet. A XAM session is created by connecting to an XSystem (the term XAM session is synonymous with XSystem instance). A XAM session shall only access XSets contained in the XSystem to which the session is connected. An important consequence of this principle is that the XAM session used to launch a query job scopes the query job to the XSystem to which the XAM session is connected. The query is directed to that XSystem and shall not return results for XSets that are not contained by that queried XSystem. Similarly, if an attempt is made to open an XSet that is not part of the XSystem to which the XAM

session (on which the XSet open attempt occurs) is connected, that open attempt shall return a non-fatal error and shall not open the XSet.

11.2 XAM Application Authentication and SASL

XAM uses the SASL framework to provide connection-oriented authentication of applications, including replaceable SASL authentication mechanisms. A XAM application may authenticate on its own behalf or may pass authentication through from the actual user of the XAM application. This distinction is not visible at the XAM API, as the authentication identity is considered to represent the XAM application at the XAM API in both cases. A SASL authentication dialogue shall be conducted by passing text-based SASL tokens back and forth across the XAM API in an iterated challenge/response sequence until the authentication succeeds or fails. Authentication failure is a non-fatal error that does not create a XAM session. The SASL framework also supports data security services, but that support shall not be used with XAM. SASL's mechanism replacement support enables authentication mechanisms to be added without change to the XAM API, but a XAM application is responsible for generation and interpretation of SASL tokens for any SASL mechanism that the application uses.

The SASL framework contains two identities, an authentication identity and an authorization identity.

- The authentication identity represents the XAM application or the user of the XAM application.
- The authorization identity determines what that XAM application or user is permitted to do.

Authorization identity support is not present in every SASL authentication mechanism. Authorization identity presentation and usage shall be optional for every SASL authentication in XAM.

After SASL authentication is complete, the two SASL identities shall be made accessible via the following XSystem fields for the duration of the XAM session:

- *.xsystem.auth.identity.authentication* - SASL authentication identity
- *.xsystem.auth.identity.authorization* - SASL authorization identity. This field shall contain the null string, if no SASL authorization identity is used.

These two XSystem identity fields shall be of type xam_string, shall be readonly, and shall be nonbinding.

Auditing (including auditing use of privileged authentication and authorization identities) should be performed, but the means for doing so are outside the scope of the XAM API standard.

A SASL authorization identity may be used to assert a role in support of Role Based Access Control (RBAC), but a full RBAC implementation involves significant authorization management functionality and process restrictions that are outside the scope of the XAM API (see [RBAC]).

11.2.1 XAM Application Authentication Approaches

XAM application authentication (and optional establishment of the authorization identity) consists of proving the identity of the XAM application to the XAM Storage System. The XAM Storage System may check that authentication directly or may delegate this responsibility to a third-party authentication server. Any delegation of this authentication check responsibility shall provide security measures sufficient to assure the correctness and integrity of the check. Using TLS [TLS] to communicate with an LDAP [LDAP] server that has a PKI certificate is one example of how this requirement can be met. In all cases, the XAM application authentication shall be checked as part of XAMLibrary.connect, and this check shall be performed in a fashion that prevents tampering by the application or other entities not trusted by the XAM Storage System. For example, if the VIM is in the same address space as the application and there is no protection boundary that prevents application modifications to VIM code or state, then this authentication

check shall not be performed in the VIM. The XAM Library is not involved in authentication; the XAM Library passes the SASL tokens between the XAM application and the VIM without change.

In the reverse direction, authentication of the XAM Storage System to the XAM application is not supported, due to an asymmetry in the XAM implementation architecture. In contrast to XAM application authentication, where the application using the API authenticates to the VIM or something beyond it, the VIM does not need to be authenticated to the XAM application. The VIM is code running in the XAM Library framework, possibly in the same address space as the XAM application, and hence has to be implicitly trusted by the XAM application. Configuration controls on what VIM code can be loaded (e.g., dynamically) and called through the XAM API are platform specific and outside the scope of the XAM API. If authentication of the XAM Storage System is necessary, that authentication may need to be checked by the VIM on behalf of the XAM application (e.g., if TLS is used between the VIM and the XAM Storage System, only the VIM can reliably check the certificate presented by the XAM Storage System, as the binding of the certificate to the TLS connection is not visible through the XAM API).

The flexibility of the SASL framework enables use of a number of authentication implementation approaches, because the text-based SASL tokens are independent of both interfaces and protocols. Three possible authentication implementation approaches are:

- Simple pass-through - The XAM application authenticates to the XAM Storage System; SASL tokens flow through the VIM without change.
- VIM mediation - The SASL dialogue is between the XAM application and the VIM. The VIM uses the SASL dialogue to call another mechanism (that need not employ SASL) to complete the authentication. For example, a password-based SASL mechanism may be used to establish access to a certificate and its private key (or other credentials) that the VIM uses to authenticate to the XAM Storage System on behalf of the XAM application.
- VIM responsibility - The SASL dialogue does not perform any authentication. Instead, the VIM handles authentication on behalf of all applications that use the VIM. The SASL EXTERNAL mechanism is among the ways to realize this approach.

In addition, a third-party authentication server may be used with XAM authentication in multiple ways, for example:

- The XAM application may contact a third-party authentication server to obtain security credentials for use with SASL.
- The VIM may contact a third-party authentication server to obtain credentials, either for itself or for the XAM application.
- The XAM Storage System may contact a third-party authentication server (e.g., via RADIUS) to validate the authentication.

The above approaches may be used individually or in combination. For example, a VIM may obtain authentication credentials on behalf of the application, and the XAM Storage System may use a third-party authentication server to validate authentications based on those credentials.

11.2.2 SASL Profile and Requirements for XAM

The details of how XAM employs SASL are specified in Section 7.3, “XAM Session”. This section covers the SASL profile that specifies how SASL is applied to XAM. Section 4 of [RFC 4422] specifies the (profile)

elements that are required in a standard that employs the SASL framework. The XAM specification of these required elements is shown in Table 74.

Table 74 – XAM Requirements for SASL

Element	XAM Specification
Service name	The service name shall be "snia-xam", and a request will be made to register it in the IANA GSSAPI registry of service names, when a stable public version of the XAM API is available. The use of a hyphen ('-') in the service name should avoid conflict with other service names.
SASL mechanism negotiation	The XAM application retrieves a list of supported SASL mechanisms via the API, selects one, and uses it. The list of mechanisms shall be in the *.xsystem.auth.SASLmechanism.list.<mechanism>* XSystem fields, and the default SASL mechanism that should be used in the absence of a specific reason to use a different mechanism shall be in the *.xsystem.auth.SASLmechanism.default* field (see Section 7.2.3, “XSystem Fields”. The list of supported SASL mechanisms should be retrieved prior to each SASL authentication exchange, as the list may change during the lifetime of a XAM session.
Authentication exchange messages	The API methods for initiating authentication (including providing the name of the selected SASL mechanism), transferring challenges and responses, and indicating the outcome of the negotiation are specified in Section 7.3.1, “Authentication State Machine”, [XAM-C-API], and [XAM-JAVA-API].
Authorization identity syntax and semantics	An authorization identity shall be a displayable Unicode string that satisfies the SASLprep [RFC 4013] requirements for user names. Authorization identities shall be case-insensitive. The UTF-8 character-encoding format [RFC 3629] should be used unless the SASL mechanism specifies a different format. The authorization identity prefix "xam-" shall be reserved for authorization identities defined by this standard. The semantics of authorization identities are defined in Section 11.3, “XSystem Authorization and XSet Access Control”.
Support for aborting an authentication exchange	This support is provided by the XSystem.abandon method (see Section 7.3.3.3, “Closing/ Abandoning XAM sessions”). Note that while XSystem.abandon is generally safe to use during an initial authentication, the warning in Section 7.3.3 about possible data loss applies to use of XSet.abandon during reauthentication of a XAM session that has dirty uncommitted data.
When security layers take effect	Not applicable. SASL security layers shall not be negotiated for or used with XAM.
Ordering of security layers with respect to other security services	Not applicable, as XAM forbids SASL from negotiating security layers (see previous element).
Effect of multiple authentications	Multiple authentications shall be supported for the purpose of re-authenticating an existing session. The effect of multiple authentications is specified in Section 7.3.3.

As specified in Section 7.3, “XAM Session”, all XAM implementations shall support the SASL ANONYMOUS and PLAIN mechanisms and are encouraged to support at least one stronger authentication mechanism. Use of the ANONYMOUS and PLAIN methods shall not be required, and a XAM Storage System may disable them by default. The SASL ANONYMOUS mechanism provides no authentication, so the resulting level of security is roughly equivalent to anonymous FTP.

The SASL PLAIN mechanism passes a password in the clear, and hence, should not be used over a network, unless the password is protected by a secure protocol providing confidentiality (e.g., TLS) or some other means of preventing unauthorized use (e.g., the password is a one-time password that cannot

be reused). Use of the SASL PLAIN mechanism, without additional protection of the password, may be a serious security flaw; therefore, in accordance with [RFC 4616], by default, implementations are strongly discouraged from advertising and strongly discouraged from using the PLAIN mechanism across a network, unless adequate data security services are in place. The SASL PLAIN mechanism may be used to provide authentication credentials that VIM uses to obtain access to more powerful credentials (e.g., PKI certificate and associated private key) on behalf of the application, but the password should not be transmitted across a network, unless the password is encrypted or otherwise protected.

SASL authorization identities shall be supported but shall be optional to use in the XAM API. SASL mechanism support for authorization identities varies (e.g., the ANONYMOUS mechanism does not support authorization identities, but the PLAIN mechanism does), and use of authorization identities is generally optional when supported by a SASL mechanism. XAM implementations shall support authorization identities, and in particular, shall not reject a SASL authentication with an error solely because the authentication contains a SASL authorization identity. Determining whether authorization identities are required is a vendor- and site-specific security policy decision; authentications that do not present authorization identities may be rejected with a non-fatal error.

11.3 XSystem Authorization and XSet Access Control

XAM distinguishes XSystem authorization from XSet access control by the objects to which their restrictions apply:

- XSystem authorization shall be instantiated when a XAM session is established by connecting to an XSystem. XSystem authorization controls what methods are permitted vs. prohibited on that XAM session, independent of the XSet that is the target of the methods (e.g., whether creation of new XSets or modification of application fields is permitted).
- XSet access control is associated with an individual XSet. XSet access control determines whether an individual XSet can be accessed in a specific fashion (e.g., read vs. write). This determination may be independent of the XAM session (e.g., modifications to an XSet may be forbidden for all sessions).

XSystem authorization and XSet access control enforcement are based on grouping API methods into granules; each authorization or access control decision either permits or denies all the methods in the granule. Some methods are subject to both XSystem authorization and XSet access control. Any such method shall be permitted or denied as follows:

- A method shall be permitted only if it is permitted by both XSystem authorization and XSet access control.
- A method shall be denied if it is denied by either XSystem authorization or XSet access control.

For example, if XSystem authorization causes a XAM session to forbid modifications, any attempt to modify any XSet field does not perform the modification and returns a non-fatal error, even if XSet access control permits modifications. A second example is that if an XSet is set to forbid modifications by XSet access control, any attempt to modify any XSet field does not perform the modification and returns a non-fatal error, even if modifications are authorized for the XAM session by XSystem authorization.

The interfaces for XSystem authorization and XSet access control granule permissions and prohibitions and the means by which they are associated with a XAM session are outside the scope of the XAM standard. XSystem authorization permissions shall be associated with a XAM session as a consequence of successful authentication. XAM provides XSystem fields that enable a XAM application to determine what granules (and hence methods) its XSystem authorization permissions permit vs. deny on a XAM session. XAM provides an XSet policy name field that enables an XSet access control policy to be applied to an XSet and provides the XSystem.accessXSet method to report which accesses are permitted vs. denied on an XSet.

XAM employs SASL, so any XAM session with an XSystem has a SASL authentication identity and may have a SASL authorization identity. Both identities may be used to make XSystem authorization and XSet access control decisions (i.e., they are acceptable inputs to XSystem authorization and XSet access control decisions).

11.3.1 XSystem Authorization

XSystem authorization determines what methods may be performed on a XAM session with an XSystem. Actual XSystem authorization policy is opaque; the XAM API provides means of determining what XSystem authorization restrictions are in place, but not why. The specification, management, and application of XSystem authorization policies are outside the scope of the XAM API.

11.3.1.1 XSystem Authorization Elements

XSystem authorization controls the methods that are authorized as a set of groups. This section specifies these groups, which are called authorization granules. XSystem authorization implementations shall not make finer grain distinctions within a granule.

XAM authorization is based on control of effects that are independent of how they are performed, but the authorization granules are specified in terms of API methods that are permitted vs. denied for concreteness. The XAM API provides multiple means of achieving some effects (e.g., creation of a new XSet), each of which is listed with the granule that controls that effect; hence, these method lists contain a number of entries. Each granule specification begins with a short description of the effects that it is intended to control.

Creation and modification of XSets each involve two API operations; the first operation creates or modifies the XSet instance, and the second operation commits the XSet instance (XSet.commit), to make the results persistent. Authorization control requirements for these steps are slightly different. An operation is controlled by authorization, when attempting an unauthorized creation or modification results in a non-fatal authorization error. Operations that modify XSets and their contents should be controlled by authorization, and any operation whose effects are visible outside the XSet instance or after the XSet instance is closed shall be controlled by authorization. To make this concrete, any operation that creates or opens an XSet shall be controlled by authorization, and XSet.commit shall be controlled by authorization. Operations on XSet instances, where the modifications will be discarded if the XSet instance is not committed, should be controlled by authorization. If XSet.commit is not called, the modifications are discarded; otherwise, authorization is enforced on XSet.commit if it has not been previously enforced. Authorization shall control any operation on an XSet instance that immediately affects the actual XSet, independent of whether XSet.commit is subsequently called (e.g., XSystem.holdXSet).

11.3.1.1.1 XSystem Authorization Granule (Component) Specification

The granules are: read, write-application, write-system, create, delete, job, job-commit, hold, and retention-event. The methods each granule permits shall be as follows, and for any method that has an asynchronous version (e.g., XSystem.asyncOpenXSet is the asynchronous version of XSystem.openXSet), the asynchronous version shall be permitted whenever the original method is permitted:

- **Read**: Read XSet contents. The read granule shall always be permitted on any XSet that is visible in a XAM session. In the XSet state diagrams in Chapter 8, the read granule shall permit all states and methods in the master finite state machine (FSM), except import. The read granule shall also permit the readonly mode FSM (i.e., only the clean XUID state) in Section 8.5.3. The read granule shall permit the following methods:
  - — XSystem.openXSet in readonly mode
  - — XSet.close, XSet.abandon, and XSet.containsField on any open XSet.

— Any method that reads XSet contents (any XSet fields) and any XAM fields, as long as no modification is made. For XStreams, this granule permits opening the XStream in readonly mode, plus the read, seek, tell, abandon, and close methods. For management properties, the permitted methods include the getActual methods (see Chapter 9, “XSet Management”).

— Creation of an XIterator (<XAMHandle>.openFieldIterator) and all XIterator methods.

— XSystem.accessXSet and XSystem.getXSetAccessTime

— Exporting XSets via XSystem.openExportXStream, plus XStream.read, XStream.tell, and XStream.close on the resulting XStream. XStream.seek is prohibited on an export XStream.

— An XSet.commit that makes no modifications to the XSet and that does not create a new XSet

- **Write-application**: Write XSet application contents. The write-application granule shall permit opening an XSet in restricted mode and the applicable states and methods of the restricted mode FSM, i.e., the clean XUID and dirty XUID states as shown in Figure 13, “Restricted Open XSet FSM”. Nonbinding modifications to the application portion of XSets shall be permitted, but binding modifications to XSets shall require the create granule. The write-application granule shall consist of the following methods for application fields, in addition to the methods in the read granule:

  — XSystem.openXSet in restricted mode.

  — Any method that makes a nonbinding modification to XSet application fields, including creating, setting, and deleting nonbinding application XSet fields. These methods include XSet.setFieldasBinding and XSet.setFieldasNonBinding, when no change to the binding attribute occurs.

  — All XStream operations on XStreams that are nonbinding application fields. In addition to the methods permitted by the read granule, this adds XSet.createXStream, opening an XStream in writeonly or appendonly mode, and XStream.write.

  — XSet.commit, when no binding modification has been made to the XSet, so that XSet.commit does not create a new XSet.

- **Write-system**: Write XSet system contents. This granule shall permit XSystem.openXSet in restricted mode and operations that make nonbinding modifications to system fields, except for retention and hold functionality. The general field operations specified in Section 6.4, “Methods that Operate on Fields” cannot be used to set or delete system fields. Any system field that can be set or deleted has a specific method for that purpose, and with the exception of retention functionality, all such methods shall be part of this granule when they make nonbinding modifications. Binding modifications shall require the create granule.

- **Create**: Create new XSets. This granule shall permit opening an XSet in unrestricted mode, copying an XSet, and importing an XSet. It shall include the states and methods of the unrestricted mode FSM in Section 8.5.3.2.3. The create granule also permits the import state in the master XSet FSM in Section 8.5.2. The create granule requires both write-application and write-system functionality. Therefore, if the create granule is permitted, both write-application and write-system granules shall be permitted. The create granule does not control whether XSet.commit can be called on an XSet that represents a job (e.g., a query job); that method call shall be controlled by the job-commit granule. The create granule shall permit the following methods, in addition to those permitted by the read, write-application, and write-system granules:

  — XSystem.createXSet, XSystem.openXSet, and XSystem.copyXSet in any mode

  — Any method that makes a binding modification to the XSet, including creating, setting, and deleting binding XSet fields. These methods include XSet.setFieldasBinding and

XSet.setFieldasNonBinding, when a change to the binding attribute occurs, and any API that changes the binding attribute of a system field or deletes a system field.

— All XStream operations on XStreams that are binding fields. In addition to the methods permitted by the read granule, this adds XSet.createXStream, opening an XStream in writeonly or appendonly mode, and XStream.write.

— Importing XSets via XSet.openImportXStream, plus XStream.write, XStream.tell, and XStream.close on the resulting XStream. (Note that XStream.seek is prohibited on writeonly XStreams, including import XStreams).

— XSet.commit, when a binding modification has been made to the XSet, so that XSet.commit creates a new XSet.

- **Delete:** Delete XSets. This granule shall permit XSystem.deleteXSet.
- **Job**: Run jobs, including the query job (see Section 8.9, “XAM Jobs and XAM Job Control”). This granule does not cover committing jobs or their results. This granule shall include:

  — XSystem.createXSet to create an XSet to run a job (see Section 11.3.1.1.3, “The Job Granule and XSet Creation” for a special case when the create and job-commit granules are not permitted).

  — Creation and modification of job input fields, including *org.snia.xam.job.command* and *xam.job.query.command.* See Section 10.7.1, “Query Job Specific XSet Fields” and input fields for any additional vendor-specific jobs controlled by this granule. This permission is in addition to the applicable permissions from the write-application and create granules, so that the job granule provides sufficient permission to run jobs in the absence of the write-application and create granules.

  — XSet.submitJob and XSet.haltJob on an uncommitted XSet that represents a query job

- **Job-commit**: The ability to commit jobs and their results, including query jobs. This granule shall permit:

  — XSet.submitJob and XSet.haltJob on a committed XSet that represents a job

  — XSet.commit on an XSet that represents a running job

  The job-commit granule requires job, create, write-user, and write-system functionality, so these four granules shall be permitted whenever the job-commit granule is permitted.

- **Hold**: Add and release XSet holds. This granule shall permit XSystem.holdXSet and XSystem.releaseXSet.
- **Retention-event**: The ability to set retention start times and create new retention identifiers. This granule is independent of the write-system and create granules that are required to modify XSet system fields; one of these two granules must be permitted, in addition to retention-event, so that the permissions granted by retention-event can be effective. This results in three cases:

  a If neither the write-system nor the create granule is permitted, the retention-event granule shall have no effect.

  b If the write-system granule is permitted, the retention-event granule shall permit XSet.createRetention and XSet.setRetentionStarttime, when they make nonbinding modifications.

  c If the create granule is permitted, the retention-event granule shall permit XSet.createRetention and XSet.setRetentionStarttime in all cases.

11.3.1.1.2 Authorization Requirements for Common XSet Operations

Based on the granules specified above, permission to perform XSystem.openXSet shall require:

- The read granule to open the XSet in readonly mode. The read granule is always permitted, so open in readonly mode shall always be permitted.
- The write-application or write-system granule to open the XSet in restricted mode.
- The create granule to open the XSet in unrestricted mode. The create granule implies the write-application and write-user granules, and hence, shall permit opening an XSet in any mode.

XSystem.createXSet and XSystem.copyXSet both create a new XSet; hence, permission to perform each of these methods shall require the create granule.

Based on the granules specified above, permission to access the contents of an XSet involves:

- The read granule for read access. This access is always permitted.
- The write-application or write-system granule for nonbinding modifications to application and user fields, respectively.
- The create granule for binding modifications of any form. The reason for requiring the create granule is that a new XSet is created by XSet.commit of an XSet instance on which a binding modification has been performed. The create granule implies the write-application and write-system granules, and hence, permits nonbinding modifications.

XSet.commit authorization is subtle. The granule that shall be required to perform XSet.commit depends on the effects of XSet.commit:

1 If the XSet represents a running job that becomes disconnected from the XAM session because of XSet.commit, the job-commit granule shall be required.

2 If the XSet.commit creates a new XSet (i.e., the XSet instance is the result of XSystem.createXSet, XSystem.copyXSet, or a binding modification to an open XSet), the create granule shall be required.

3 If only nonbinding modifications have been made to the open XSet, the write-application and/or write-system granules shall be required, according to whether modifications have been made to application and/or system fields.

4 If XSet.commit does not create a new XSet or modify the existing XSet, then the read granule shall be required; this granule is always permitted, and hence, this operation is always permitted.

Some errors caused by lack of sufficient authorization for XSet.commit can be detected at an earlier stage. The XSet.commit authorization checks specified above shall always be performed. In addition, authorization checks should be made on all operations that modify an XSet instance, so that an operation that creates an uncommitable XSet instance (e.g., a new binding field created in an existing XSet without holding the create granule) returns a non-fatal error at the point where the XSet instance would become uncommitable (vs. waiting for XSet.commit to discover that the open XSet instance cannot be committed). For example, in item **3** above, if the appropriate write granule is not held, any earlier field modification operation should have returned a non-fatal error indicating that modifications are not authorized.

11.3.1.1.3 The Job Granule and XSet Creation

The job granule permits XSystem.createXSet for the purpose of creating an XSet to run a job. If the job granule is permitted, but the job-commit and create granules are not permitted, that XSet can never be committed (see items **1** and **2** above). In this situation, a special job-only XSet results when XSystem.createXSet is called in restricted mode. This XSet behaves as follows:

- Creation and modification of job input fields shall be permitted for all jobs, both standard XAM jobs (i.e., the query job) and jobs (if any) defined by the XAM Storage System vendor. Any method that creates or modifies a field, other than a job input field, should return a non-fatal error.
- XSet.commit shall return a non-fatal error indicating that the XSet can only be used to run jobs.
- The XSet shall occupy only the clean no XUID and dirty no XUID states of the restricted mode XSet FSM as defined in Section 8.5.3.2.2, “Restricted Open XSet FSM”.

11.3.1.1.4 XSystem Authorization Properties

For a XAM session, the permissions and restrictions imposed by XSystem authorization can be determined by examining the values of Boolean XSystem properties whose names have the form *.xsystem.auth.granule.list.<granule>*, e.g.,:

- *.xsystem.auth.granule.list.read*
- *.xsystem.auth.granule.list.write-application*
- *.xsystem.auth.granule.list.write-system*
- *.xsystem.auth.granule.list.create*

Attempting to call a function that is not permitted on a XAM session shall return a non-fatal error.

XSystem authorization may be extended to vendor-specific functional additions by defining vendor-specific authorization granules. The vendor-specific Boolean XSystem properties that indicate whether a vendor-specific authorization granule is permitted vs. denied shall have names of the form *<reversed-dns>.authz.<granule>*, e.g., *.com.example.xamsys.authz.funky* where *com.example.xamsys* is the vendor-specific reversed DNS prefix and *funky* is the name of the authorization granule (see Section 6.3.1, “Field Namespace”). Vendor-specific authorization granules shall not affect the behavior of authorization granules defined in this standard, and in particular, shall not provide additional control over methods that are covered by a granule defined in this standard.

11.3.1.2 XSystem Authorization Roles

A XAM role shall be a named collection of authorization granules. XSystem authentication causes a role to be applied to the created XAM session, determining what actions are authorized on that session. Roles are passed across the XAM API as SASL authorization identities. The prefix "xam-" indicates a standard XAM role and shall only be used for roles defined in this document. Other roles may be defined and used. The standard XAM roles and their component granules shall be:

- xam-read-only: read. This role provides read access without permission to perform modifications for situations in which it is possible to determine the information to be accessed (i.e., the XUIDs of the XSets) via means outside the XAM API. This role is appropriate for read access via a XAM application that maintains its own index to information that the application has stored.
- xam-read-job: read, job. This role provides read access plus the ability to run XAM jobs (e.g., queries). For more information, see Section 8.9, “XAM Jobs and XAM Job Control”. This role is appropriate for a XAM application that browses stored information by using queries to determine what has been stored.
- xam-restricted: read, job, write-application, write-system. In addition to read access, this role provides the ability to modify nonbinding fields of stored XSets. It is appropriate for a XAM application that annotates stored information without modifying it, when the stored information that is not to be modified has been stored in binding fields of XSets.

- xam-create: read, write-application, write-system, create, job. This role provides general access to an XSystem to read, write, and create, but not delete XSets.
- xam-job-commit: read, write-application, write-system, create, job, job-commit. This role adds the ability to run a background (detached) job (e.g., a query job) to the xam-create role. It is a separate role, so that limits can be imposed on consumption of XAM Storage System resources by background (detached) jobs.
- xam-create-delete: read, write-application, write-system, create, delete, job. This role adds the ability to delete XSets to the xam-create role.
- xam-hold: read, job, hold. This role is intended for a XAM application that manages holds on XSets, e.g., for litigation support purposes.
- xam-event: read, job, RetentionEvent. This role is intended for a XAM application that manages retention events on XSets; see Section 9.2.1, “XSet Retention”.
- xam-super: All granules shall be included. Support for this role shall not be required, but if any role providing permission to call all methods is supported, this role shall be supported. Some environments require a super user role that can do anything.

Auditing (including auditing use of important or sensitive XSystem authorization roles) should be performed, but the means for doing so are outside the scope of the XAM API standard.

11.3.2 XSet Access Control Policy

XSet access control policy is the rough analog of control over which users are permitted to read and write files. XSet access control policy governs three method groups via the read, write-application, and write-system granules:

- Read: Read and export XSet contents. Read shall always be permitted.
- Write-application: Write XSet application fields and create, modify, or remove application fields.
- Write-system: Write XSet system fields and create, modify, or remove system fields.

These granules include the same methods as the XSystem authorization granules with the same names specified in Section 11.3.1.1 and shall be the granularity of XSet access control permissions. All methods within each granule (read, write-application, or write-system) shall be permitted or forbidden as a unit; finer grain distinctions shall not be made.

XSystem.accessXSet enables a XAM application to determine whether each of these method groups is permitted; it indicates that a method group is permitted, if XSystem authorization and XSet access control both permit it. If an XSet cannot be read via a XAM session, it shall not be visible through that XAM session. XSystem.accessXSet for read access shall indicate whether the XSet is visible. If XSystem.accessXSet successfully returns "FALSE", the XSet does not exist in the context of the XAM session.

In contrast, XSystem.openXSet shall return a non-fatal error, when read access is not permitted, just as a non-fatal error is returned, when XSystem.openXSet is called with a XUID for an XSet that does not exist within the XAM Storage system. The same non-fatal error shall be used for both cases; this prevents a XAM application that is unable to read an XSet from determining whether the XSet exists based on which non-fatal error is returned.

XSet access control shall be an XSet policy (see Section 8.7, “XSet Policy”). Access control is accomplished indirectly through named policies, the details of which are specific to the implementation. Policy names shall be strings chosen by means outside the scope of the XAM standard. The

.xset.access.policy field shall contain the name of the access control policy, if one has been applied; this field shall be a xam_string. This policy name shall be an input to XSet access control policy decisions that determine whether the methods included in each of the above granules are permitted or denied on the XSet. Table 75 lists the XSet access control policy property and the policy property method, which are described in the paragraphs following the table.

Table 75 – XSet Policy Management Properties

Policy Property Name [Binding: Application-Specified, Readonly: FALSE]	Policy Property Method
.xset.access.policy	XSet.applyAccessPolicy

XSet.applyAccessPolicy (see Table 76) shall be used to apply a policy and specify whether the access control policy field is binding. The access control policies that may be applied to an XSet shall be contained in the *.xsystem.access.policy.list.<name>* fields of the XSystem; these fields shall be an XSystem policy list (see Section 8.7, “XSet Policy”). If this list is empty, then any call of XSet.applyAccessPolicy in the corresponding XAM session shall return a non-fatal error. The *.xsystem.access* field of the XSystem indicates whether the XSystem supports XSet access control.

If *.xset.access.policy* does not exist for an XSet, then no additional access controls exist beyond those imposed by XSystem authorization. An authorized application may remove access control using XSet.resetAccessFields to remove *.xset.access.policy*. In contrast to the top-level *.xset.management.policy*, which is required to exist in every committed XSet, *.xset.access.policy* shall not be required to exist in any XSet.

Table 76 – XSet Access Control Policy Methods

Method	Input	Output	Comment
XSet.applyAccessPolicy	policy name	-	Establishes the access control policy for the XSet by creating or setting the *.xset.access.policy* field, including whether the policy value is binding on the XSet.
XSet.resetAccessFields	-	-	Resets the access control fields for the XSet by removing the *.xset.access.policy* field if it exists.

The authorization granule (see Section 11.3.1.1) required to call these methods depends on whether the access control policy value is binding. If the new or existing value, if any, of *.xset.access.policy* is binding, the create granule shall be required. If the new or existing value, if any, of *.xset.access.policy* is nonbinding, the write-system granule shall be required.

These methods shall have the following effects on the XSet finite state machines (FSMs) specified in Section 8.5:

- If *.xset.access.policy* does not exist in the XSet, then XSet.applyAccessPolicy shall cause the same FSM effects as XSet.create<stype>, and XSet.resetAccessFields has no FSM effects.
- If *.xset.access.policy* exists in the XSet, then XSet.applyAccessPolicy shall cause the FSM effects of a binding modification or a nonbinding modification, whichever is performed by the method. If the field is nonbinding, both before and after the modification, the modification is nonbinding; otherwise, the modification is binding.

- If *.xset.access.policy* exists in the XSet, then XSet.resetAccessFields shall cause the FSM effects of a binding modification or a nonbinding modification, whichever is performed by the method. If the field is nonbinding before it is removed by this method, the modification is nonbinding; otherwise, the modification is binding.

All access control is accomplished via indirection. For example, the XAM API cannot directly forbid modifications to an XSet. Instead, an access control policy, e.g., "NoChanges", is defined that forbids modifications (i.e., does not permit either of the write granules). The policy is made available by creating the corresponding XSystem's *.xsystem.access.policy.list.*<*name*>, and the policy is applied to the XSet via XSet.setAccessPolicy. This design decision insulates the XAM API from access control details.

XSet access control policies shall be optional to support. If an XSystem does not support XSet access control policies, then there shall be no *.xsystem.access.policy.list.<name>,* and *.xsystem.access* shall have the value FALSE. In this situation, all calls of XSet.applyAccessPolicy to any XSet in the XSystem shall return a non-fatal error.

Auditing (including auditing use of important or sensitive XSet access control policies) should be performed, but the means for doing so are outside the scope of the XAM API standard.

Annex A
(normative)
XAM Toolkit

This annex describes a toolkit of normative (required) functionality to be provided, in addition to the standard API methods.This toolkit shall be implemented using only the standard API methods and shall not require additional methods to be specified for VIMs to correctly operate.

Most of the functionality in this toolkit is present as a recognition that common operations need to be easy to perform.

A.1 Query

The XAM query facility is expected to be used frequently. The toolkit provides convenience functions to make the functionality easier to use. These toolkit functions are built on the base XAM API and shall not require new VIM interfaces.

A.1.1 XAMQuery

Invoking and checking on the progress of a query are operations which will take place often. Functions are introduced here to make submission and query termination easier. Table A.1, “XAMQuery Methods” details the method query toolbox calls.

Table A.1 – XAMQuery Methods

Method	Input	Output	Comment
query	query_string	XSet	Creates an XSet, fills in the appropriate jobs fields, and calls XSet.submitJob. The *query_string* argument shall be compatible with the query job command.
queryCompleted	XSet	xam_boolean	Tests the query XSet job to determine, if the job is still running. This method shall return FALSE, if the job status represents a terminal state, and TRUE in all other cases.

A.1.2 XUIDIterator

The XAM query system produces a stream of XUID result values, which are attached to the query job XSet. To process the XUID values, the application program needs to read through them. An application may read the stream directly, but the XUIDIterator utility functions may be used for ease of programming.

An XUIDIterator is similar to a cursor in the stream. XUIDIterator code shall be re-entrant, and multiple XUIDIterators shall be permitted on the same result XStream. The application programmer shall be responsible for allocating and deallocating the XUIDIterators. Table A.2, “XUIDIterator Methods” details the methods available to the XUIDIterator.

Table A.2 – XUIDIterator Methods

Method	Input	Output	Comment
start	XStream	-	Initializes the XUIDIterator. The application passes in the XStream of the query results job.
hasNext	-	xam_boolean	Returns FALSE if the associated query result stream is complete and all XUIDs have been returned; otherwise, this method shall return TRUE.

Table A.2 – XUIDIterator Methods

Method	Input	Output	Comment
nextXUID	-	xam_xuid	Returns the next XUID from the result stream. If the job has not yet completed, this method shall block. An unblock shall occur when new XUID values are placed into the stream. This method shall return non-padded XUID values.
close			Closes the XUIDIterator, allowing the VIM to release any resources associated with the XIterator.

A.2 Base64 Translator

To store XUID values in printable formats, it is recommended that applications base64 encode it. This encoding algorithm is detailed in Section 6.8 of [RFC 2045]. Table A.3, “Base64 Methods” details the Base64 encoding and decoding methods.

Table A.3 – Base64 Methods

Method	Input	Output	Comment
XUIDToString	XUID	xam_string	Base64 encodes the bytes of the XUID. This method shall ignore padding bytes presented in the XUID.
stringToXUID	xam_string	XUID	Decodes the Base64-encoded XUID. This method shall produce an unpadded XUID.

A.3 XUID Padding

XUID values may be encountered that have been padded with extra zero value bytes. For ease of reading, the query job result stream (see Chapter 10, “Query”) pads XUID values to present fixed-length records in the result XStream. To preserve the significant bytes of the XUID, padding bytes should be removed. Table A.4, “XUID Padding Methods” details this tool.

Table A.4 – XUID Padding Methods

Method	Input	Output	Comment
stripPadding	byte_array	XUID	Removes padding bytes from a byte array that contains a padded XUID.
addPadding	XUID	byte_array	Adds padding bytes to a XUID to produce a fixed-length record of 80 bytes.

A.4 Property vs. XStream Field Determination

An application may wish to easily determine if a field is either a property or an XStream. While an application may check by examining the MIME type of a field, these methods are more convenient for a certain amount of ‘future proofing’.

Table A.5 – Field Determination Methods

Method	Input	Output	Comment
fieldIsProperty	xam_string	xam_boolean	Returns TRUE if the named field’s MIME type is one of stypes.
fieldIsStream	xam_string	xam_ boolean	Returns FALSE if the named field’s MIME type is one of stypes.

Annex B
(normative)
Canonical XSet Interchange Format

B.1 Introduction

This annex describes the XSD (XML Schema Definition) for the XML manifest part of the XSet canonical format. This XSD defines the XSet XML manifest that contains the complete definitions and content of properties and definitions of XStreams for the XSets being imported or exported. The contents of the XStreams are found in the MIME attachments that are contained in the same XOP package as the XML manifest.

B.2 XSD

```
<?xml version=”1.0” encoding=”UTF-8”?>
<!--
   XAM Architecture Specification
   Canonical XSet Export/Import Format

   All contents copyright 2008 Storage Networking Industry Association
   Please see the SNIA XAM Architecture Specification for details on
   how this document may be used
-->
<xs:schema xmlns=”http://www.snia.org/2007/xam/export” xmlns:xs=”http://www.w3.org/2001/
XMLSchema” xmlns:xop=”http://www.w3.org/2004/08/xop/include” targetNamespace=”http://
www.snia.org/2007/xam/export” elementFormDefault=”qualified”
attributeFormDefault=”unqualified” version=”1.0.0”>
    <xs:import namespace=”http://www.w3.org/2004/08/xop/include” schemaLocation=”http://
www.w3.org/2004/08/xop/include”/>
    <xs:import namespace=”http://www.w3.org/2001/XMLSchema” schemaLocation=”http://
www.w3.org/2001/XMLSchema”/>
    <!-- The canonical XSet Export/Import format is organized as follows
   <xsets>
    <version>
    <policies>
     <policy [attributes]><[value type]>[value]</[value type]>
      <property [attributes]><[value type]>[value]</[value type]></property>
     </policy>
    </policies>
    <xset>
     <properties>
      <property [attributes]><[value type]>[value]</[value type]></property>
     </properties>
     <xstreams>
      <xstream [attributes]>
       <xop:Include />
      </xstream>
     </xstreams>
    </xset>
   </xsets>

    value type can be one of string, integer, boolean, date, float
-->
    <!-- Define the field attributes for properties and streams-->
```

```
    <xs:attributeGroup name="FieldAttrs">
        <xs:attribute name="name" type="xs:string" use="required"/>
        <xs:attribute name="type" type="xs:anySimpleType" use="required"/>
        <xs:attribute name="binding" type="xs:boolean" use="required"/>
        <xs:attribute name="readOnly" type="xs:boolean" use="required"/>
        <xs:attribute name="length" type="xs:long" use="required"/>
        <xs:anyAttribute namespace="##any" processContents="lax"/>
    </xs:attributeGroup>
    <!--
  Define a property type
    -->
    <xs:complexType name="PropertyType">
        <xs:sequence>
            <xs:choice minOccurs="1" maxOccurs="1">
                <xs:element minOccurs="1" maxOccurs="1" name="string" type="xs:string"/>
                <xs:element minOccurs="1" maxOccurs="1" name="integer" type="xs:int"/>
                <xs:element minOccurs="1" maxOccurs="1" name="boolean" type="xs:boolean"/
>
                <xs:element minOccurs="1" maxOccurs="1" name="date" type="xs:dateTime"/>
                <xs:element minOccurs="1" maxOccurs="1" name="float" type="xs:float"/>
            </xs:choice>
        </xs:sequence>
        <xs:attributeGroup ref="FieldAttrs"/>
    </xs:complexType>
    <!--
  Define the format of each policy entry
 -->
    <xs:complexType name="policy_type">
        <xs:sequence>
            <xs:element name="policy" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                    <xs:sequence>
                    <xs:element minOccurs="1" maxOccurs="1" name="string" type="xs:string"/>
                    <xs:element name="property" type="PropertyType" minOccurs="0"
maxOccurs="unbounded"/>
                    </xs:sequence>
                    <xs:attributeGroup ref="FieldAttrs"/>
                </xs:complexType>
            </xs:element>
        </xs:sequence>
    </xs:complexType>
    <!--
  Define the format of each xset entry
 -->
    <xs:complexType name="xset_type">
        <xs:sequence>
            <xs:element name="properties">
                <xs:complexType>
                    <xs:sequence>
                    <!-- XAM defines several required properties, so there will definitely be at
least one (minOccurs=1) -->
                    <xs:element name="property" type="PropertyType"
maxOccurs="unbounded"/>
                    </xs:sequence>
                </xs:complexType>
```

```
            </xs:element>
            <xs:element name=”xstreams”>
                <xs:complexType>
                    <xs:sequence>
                    <!-- An XSet may have 0 XStreams -->
                    <xs:element name=”xstream” minOccurs=”0” maxOccurs=”unbounded”>
                    <xs:complexType>
                    <xs:sequence>
                    <xs:element ref=”xop:Include”/>
                    <xs:any namespace=”##any” processContents=”lax” minOccurs=”0”/>
                    </xs:sequence>
                    <xs:attributeGroup ref=”FieldAttrs”/>
                    </xs:complexType>
                    </xs:element>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
        </xs:sequence>
    </xs:complexType>
    <!--
  Define the xsets entry
 -->
    <xs:element name=”xsets”>
        <xs:annotation>
            <xs:documentation>XAM Canonical XSet Export/Import Format
    </xs:documentation>
        </xs:annotation>
        <xs:complexType>
            <xs:sequence>
                <!-- There is exactly one version element -->
                <xs:element name=”version” type=”xs:string”/>
                <!-- There is exactly one policies element -->
                <xs:element name=”policies” type=”policy_type”/>
                <!-- There is at least one XSet in the document -->
                <xs:element name=”xset” type=”xset_type” maxOccurs=”unbounded”/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```