TIBCO ActiveSpaces®

Developer's Guide

Software Release 2.1.4
August 2014

Two-Second Advantage®

Contents

Preface

TIBCO ActiveSpaces® is a distributed peer-to-peer in-memory data grid, a form of virtual shared memory that leverages a distributed hash table with configurable replication.

TIBCO ActiveSpaces® combines the features and performance of databases, caching systems, and messaging software to support large, highly volatile data sets and event-driven applications. It lets you off-load transaction-heavy systems and allows developers to concentrate on business logic rather than the complexities of developing distributed fault-tolerance.

TIBCO ActiveSpaces is available in three versions:

- **TIBCO ActiveSpaces® Enterprise Edition**—Provides C, Java, and .NET API sets and enables full cluster functionality. To enable remote clients, you must purchase licenses for the TIBCO ActiveSpaces Remote Client Edition.
- **TIBCO ActiveSpaces® Remote Client Edition**—Can be purchased in addition to the Enterprise Edition. Allows you to set up remote clients. Applications running on the remote clients can access the data grid and perform most ActiveSpaces operations.
- **TIBCO ActiveSpaces® Community Edition**—A developer version of the product. Provides limited functionality: one metaspace, four metaspace members and no TIBCO Rendezvous® capability. This version is downloadable from TIBCO Developer Network at http://developer.tibco.com.

This manual describes the basic features of ActiveSpaces and describes how to use the API set to develop applications that manage the data grid.

Topics

Related Documentation

This section lists documentation resources you may find useful.

TIBCO ActiveSpaces Documentation

The following documents form the TIBCO ActiveSpaces documentation set:

- *TIBCO ActiveSpaces Installation* Read this manual for instructions on site preparation and installation.
- *TIBCO ActiveSpaces Administration* Read this manual to gain an understanding of the product that you can apply to the various tasks you may undertake.
- *TIBCO ActiveSpaces C Reference* Read this manual for reference information on the C functions for developing an application that manages data grids.
- *TIBCO ActiveSpaces Release Notes* Read the release notes for a list of new and changed features. This document also contains lists of known issues and closed issues for this release.

Other TIBCO Product Documentation

You might find it useful to read the documentation for the following TIBCO products:

- TIBCO Rendezvous® software: TIBCO Rendezvous provides an optional transport that you can choose to use in place of the built-in Pragmatic General Multicast (PGM) multicast transport that TIBCO ActiveSpaces uses by default. TIBCO Rendezvous handles the transport of data and messages between member processes over the network.

  For information on TIBCO Rendezvous, see *TIBCO Rendezvous Concepts*, Chapter 8 “Transport,” for information about the service, network, and daemon parameters, which are used to configure discovery and listen transport in TIBCO ActiveSpaces.

Typographical Conventions

The following typographical conventions are used in this manual.

Table 1 General Typographical Conventions

Convention	Use
TIBCO_HOME	All TIBCO products are installed under the same directory. This directory is referenced in documentation as *TIBCO_HOME*. The value of *TIBCO_HOME* depends on the operating system. For example, on Windows systems, the default value is `C:\tibco`.
`code font`	Code font identifies commands, code examples, filenames, pathnames, and output displayed in a command window. For example: Use `MyCommand` to start the foo process.
`bold code font`	Bold code font is used in the following ways: • In procedures, to indicate what a user types. For example: Type **`admin`**. • In large code samples, to indicate the parts of the sample that are of particular interest. • In command syntax, to indicate the default parameter for a command. For example, if no parameter is specified, `MyCommand` is enabled: `MyCommand [`**`enable`**` | disable]`
italic font	Italic font is used in the following ways: • To indicate a document title. For example: See *TIBCO ActiveMatrix BusinessWorks Concepts*. • To introduce new terms For example: A portal page may contain several portlets. *Portlets* are mini-applications that run in a portal. • To indicate a variable in a command or code syntax that you must replace. For example: `MyCommand` *pathname*
Key combinations	Key name separated by a plus sign indicate keys pressed simultaneously. For example: Ctrl+C. Key names separated by a comma and space indicate keys pressed one after the other. For example: Esc, Ctrl+Q.
	The note icon indicates information that is of special interest or importance, for example, an additional action required only in certain circumstances.

Table 1 General Typographical Conventions (Cont'd)

Convention	Use
💡	The tip icon indicates an idea that could be useful, for example, a way to apply the information provided in the current section to achieve a specific result.
⚠	The warning icon indicates the potential for a damaging situation, for example, data loss or corruption if certain steps are taken or not taken.

Table 2 Syntax Typographical Conventions

Convention	Use
`[ ]`	An optional item in a command or code syntax. For example: `MyCommand [optional_parameter] required_parameter`
`|`	A logical `OR` that separates multiple items of which only one may be chosen. For example, you can select only one of the following parameters: `MyCommand para1 | param2 | param3`

`{ }`	A logical group of items in a command. Other syntax notations may appear within each logical group.
	
	For example, the following command requires two parameters, which can be either the pair `param1` and `param2`, or the pair `param3` and `param4`.
	
	`MyCommand {param1 param2} | {param3 param4}`
	
	In the next example, the command requires two parameters. The first parameter can be either `param1` or `param2` and the second can be either `param3` or `param4`:
	
	`MyCommand {param1 | param2} {param3 | param4}`
	
	In the next example, the command can accept either two or three parameters. The first parameter must be `param1`. You can optionally include `param2` as the second parameter. And the last parameter is either `param3` or `param4`.
	
	`MyCommand param1 [param2] {param3 | param4}`

How to Contact TIBCO Support

For comments or problems with this manual or the software it addresses, please contact TIBCO Support as follows.

- For an overview of TIBCO Support, and information about getting started with TIBCO Support, visit this site:

  http://www.tibco.com/services/support

- If you already have a valid maintenance or support contract, visit this site:

  https://support.tibco.com

  Entry to this site requires a user name and password. If you do not have a user name, you can request one.

Chapter 1 Introduction

This chapter introduces TIBCO ActiveSpaces and describes typical use cases for the software.

Topics

Product Overview

TIBCO ActiveSpaces is a peer-to-peer distributed in-memory data grid—a form of virtual shared memory that is replicated on distributed devices and applications.

ActiveSpaces provides an application programming interface (API) that allows developers to store and retrieve data and implement database and messaging functionality. ActiveSpaces also provides an administrative CLI tool and an administrative GUI that you use to create and administer the data grid. This makes it easy to create distributed applications that exchange and modify data shared between processes and across a network.

Benefits of TIBCO ActiveSpaces

ActiveSpaces provides:

- Coherent, in-memory, data storage and retrieval.
- Several network data transport options, including TCP, TIBCO Rendezvous, and TIBCO SmartPGM.
- An API that can be used to develop custom applications which utilize the features of ActiveSpaces. An API is available for the Java, C, and .NET programming languages.

ActiveSpaces facilitates and speeds up storage and retrieval of data in a distributed manner so that you can concentrate on writing business logic. You do not have to worry about where to store new data, where current data is stored, or if it is out-of-date.

In addition, ActiveSpaces:

- Combines database features with a simple, easy to use middleware management system.
- Supports many hardware and software platforms, so programs running on many different kinds of computers on a network can communicate seamlessly.
- Allows programmers to easily implement distributed processing of the data stored in ActiveSpaces while leveraging "data locality" using remote invocation functionality.
- Scales linearly and transparently when machines/peers are added. An increase in the number of peers in a space produces a corresponding increase in the memory and processing power available to the space.
- Allows your application to continue to function smoothly without code modification or restarts.

- Allows you to implement shared-all persistence or shared-nothing persistence to persist space data on local storage media.
- Ensures that any change in data is reflected on all nodes as it happens and that no node will deliver stale data when reading or querying data from any node.

From the programmer's perspective, the ActiveSpaces software suite provides the following benefits. ActiveSpaces:

- Is distributed for speed, resiliency, and scalability
- Simplifies distributed system development
- Provides location transparency: there is no need to worry about where or how to store or find data
- Decouples producers and consumers of data
- Allows applications to be notified automatically as soon as data is modified

TIBCO ActiveSpaces Features

ActiveSpaces combines some of the important features of a database and a messaging system in a single integrated interface.

Like a database

- Has a data definition language
- Has SQL-like `where` clause filters
- Can be made fault-tolerant using data
- Implements a form of horizontal partitioning
- Has locks for concurrent access control
- Has full ACID properties and includes support for transactions
- Supports Create, Read, Update, Delete (CRUD) operations.
- If you have purchased and installed the TIBCO ActiveSpaces Remote Client version, allows you to connect to TIBCO ActiveSpaces from remote clients

Like a messaging system

- Listeners give applications the ability to subscribe to changes in the data
- One or multiple recipients
- Changes to the data are immediately distributed to all intended recipients at the same time.
- Browsers let applications use a space as a queue

Additional features

Beyond the simplicity and convenience of having a single unified interface for both data storage features and messaging features, ActiveSpaces provides the ability to:

- Receive *initial values* when creating a listener.
- Run *continuously updated queries* in real-time.
- Trigger code execution transparently on the processes storing the data, either in parallel or using the distribution algorithm to direct a query directly to one of the nodes.

Usage Profiles

The ActiveSpaces data grid can solve a variety of application problems. The nature of the problem determines the best ActiveSpaces configuration. You should consider the:

- Size of the data
- Frequency of data updates versus reads
- Relative importance of update speed versus absolute consistency of data between members of the grid

The optimal architecture for your application also depends on whether the application is being built from scratch as a *space-based* application, or is being augmented for scalability, or is *space-enabled*.

Distributed Data Cache

You can use ActiveSpaces as a distributed data cache to store copies of data that is too expensive to fetch or compute. Data is distributed across multiple machines, so that cache size is limited only by the aggregate memory of all peers participating in the space. A *distributed database cache aside* architecture reduces database hits by caching database reads. Database updates invalidate the cache.

ActiveSpaces keeps the cache synchronized across any number of hosts, eliminating the costly operation of going to a disk. Data is fetched quickly from an in-memory data cache across a local network. The space handles coherency and locking.

In-Memory Operational Data Store

A real-time data store aggregates data from multiple sources to speed processing. Real-time data stores are often used to integrate real-time feeds such as market data, airline reservation data, or other business data, making the data instantly available for efficient processing. The data must be highly available, and the system must process large data sets and transient, volatile data.

Space-Based Architecture

When designing a new system from scratch, a space-based approach provides several advantages:

- Space-based architectures use a grid approach for both data storage and processing.
- Data storage and access is virtualized.
- The space takes care of both data communication and process coordination
- Processing units are loosely coupled, and run in parallel.
- Processes are coordinated through data and events.

Grid Computing

Grid computing refers to using multiple machines or nodes to solve a large computing problem. A complex problem is decomposed into smaller pieces that can be executed across many machines in parallel.

ActiveSpaces can improve analytical processing of large data sets because it allows you to co-locate and invoke processing of the data directly on the nodes that store the data. Because ActiveSpaces stores the data in a distributed manner over many machines, processing is naturally and transparently distributed.

Deployment Models

ActiveSpaces has two deployment models:

- Peer-to-Peer Deployment Mode
- Remote Client Deployment Mode

Peer-to-Peer Deployment Mode

You can deploy ActiveSpaces-enabled applications in a true peer-to-peer configuration where all processes are direct peers to each other and there are no "servers" or "clients," but rather *seeders* (contributing nodes) and *leeches* (non-contributing nodes).

This deployment mode yields the highest performance level, but requires all processes to establish bidirectional TCP connections with each other. In peer-to-peer mode, it is also recommended (although not absolutely required) that all the peers be physically interconnected by LANs (Local Area Networks) or MANs (Metropolitan Area Networks) rather than by WANs (Wide Area Networks).

You cannot use peer-to-peer deployment mode if there is network address translation between any of the peer machines.

Remote Client Deployment Mode

In this deployment mode, seeder processes (which can be seen as "servers") are full peers to each other and fully interconnected by TCP connections, as described above. Any number of applications can access the seeders as remote clients by making a single TCP connection to one of the ActiveSpaces agent processes, which act as proxies for the remote clients. Remote clients can connect to their ActiveSpaces agent proxies over any network topology that supports TCP connections, including WANs.

You can use this deployment mode even if there is a one-way firewall or network address translation between the remote client and the full peer proxi(es).

Chapter 2 TIBCO ActiveSpaces Concepts

This chapter explains basic terms used in ActiveSpaces® and discusses fundamental concepts.

Topics

Introduction to TIBCO ActiveSpaces Applications

ActiveSpaces applications are programs that use ActiveSpaces software to work collaboratively over a shared data grid. The data grid comprises one or more tuple spaces.

An *ActiveSpaces distributed application system* is a set of ActiveSpaces programs that cooperate to fulfill a mission (either using the administrative CLI tool or the ActiveSpaces API calls). Tuples are distributed, rather than "partitioned" across seeders (members that are configured to contribute memory and processing resources to a space).

ActiveSpaces automatically redistributes tuples when seeders join and leave the space. Unlike a horizontally partitioned database, where the allocation of items to nodes is fixed, and can only be changed through manual reconfiguration, ActiveSpaces data is automatically updated on all devices on the data grid and rebalanced transparently by using a minimal redistribution algorithm.

ActiveSpaces allows the distribution of data replicates on different peers for fault tolerance. The data access optimization feature of ActiveSpaces uses a replicate if one is locally available. If a seeder suddenly fails, the replicate is immediately promoted to seeder, and the new seeder creates new replicates. This optimizes system performance.

Basic ActiveSpaces Terms

This section defines basic terms for ActiveSpaces, which are used in the discussion of ActiveSpaces concepts later in this chapter.

Table 3 ActiveSpaces Terms

Term	Definition
Metaspace	A logical group of spaces—a cluster of hosts and processes that share the same metaspace name and set of discovery transport attributes. The hosts and processes in a metaspace work together by joining the same spaces.
Space	A shared, virtual entity that functions as a container for a collection of entries consisting of a tuple and associated metadata. Applications become members of a space in order to execute operations on the space. Spaces are contained in a metaspace.
Tuple	A sequence of named elements called fields (similar to the columns in a database table) that contain values of a specific type.
Seeder	A space member that can execute operations on spaces that it is connected to and which also plays an active role in maintaining the space by providing CPU and RAM. The service of storing the data contained in a space and handling requests to read and write this data is implemented in a distributed peer-to-peer manner by one or more seeders.
Leech	A space member that can execute operations on spaces that it is connected to but which does not contribute memory or CPU time to maintenance of the space.
Replication	An ActiveSpaces process that backs up data from one seeder to one or more additional seeders, to enable fault tolerance.
Persistence	An ActiveSpaces feature that allows you to persist data to disk storage and recover data if data loss occurs or there is a problem with cluster startup. ActiveSpaces allows the distribution of data replicates on different peers for fault tolerance. If a seeder suddenly fails, the replicate is immediately promoted to seeder, and the new seeder creates new replicates. This optimizes system performance
Browser	A mechanism to iterate through a series of tuples retrieved from a space using filters. Unlike a traditional iterator that works only on a snapshot of the data to be iterated through, the space browser is continuously updated according to the changes in the data contained in the space being browsed.

Table 3 ActiveSpaces Terms

Term	Definition
Listener	A mechanism that allows an application to monitor events that represent changes to the tuples stored in a space through a callback routines that are automatically called when specific events occur in a space.

What is a Metaspace?

A *metaspace* is a logical group of spaces—a cluster of hosts and processes that share the same metaspace name and set of discovery transport attributes. A metaspace:

- Is a virtual entity that contains spaces, which store the data used by applications.
- Is an administrative container for the spaces. A metaspace can contain:
  - *System spaces*—Spaces defined by ActiveSpaces.
  - *User spaces*—User-defined spaces.
- Consists of a *cluster* of application processes.

  The processes are usually deployed on multiple hosts interconnected by a network where ActiveSpaces is installed. ActiveSpaces applications can also be installed on a standalone host.

The hosts and processes in the cluster work together by joining the same spaces.

You can deploy multiple independent metaspaces over a single network, each with a different set of members and spaces, and each identified by a name and a set of network transport attributes.

Each metaspace should have a unique name, because an application cannot connect to two different metaspaces using the same metaspace name.

Space access is based on the combination of metaspace name and space name. Therefore, changes to a space called *clients* in a metaspace named *Dev* have no impact on a space named *clients* in a metaspace named *Prod*.

Metaspace Connection

To use ActiveSpaces, your application must first connect to a metaspace. For detailed information on connecting to a metaspace, see Connecting to the Metaspace, page 75.

When your application is connected to a metaspace, it can:

- Define and make use of any number of spaces.
- Connect to additional metaspaces; however, your application can only have a single connection for each metaspace.

When your application no longer needs access to a metaspace, you should disconnect from the metaspace.

For information on disconnecting from a metaspace, see Disconnecting from the Metaspace, page 80.

Metaspace Life Cycle

A metaspace is created when the first process connects to it, and disappears when the last process disconnects from it. The metaspace grows or shrinks automatically as members connect to it and disconnect from it.

Initially, a metaspace contains only system spaces. As users create spaces in the metaspace, the definition of those spaces (along with other administrative data) is stored in system spaces.

If you implement the ActiveSpaces data persistence feature, you can persist data to local storage. However, space and field definitions are not persisted in existing spaces after the last metaspace disconnects from it.

What is a Space?

Spaces are the main feature offered by ActiveSpaces. Together with metaspaces, spaces provide a distributed data grid. A space:

- Is a virtual entity that provides shared virtual storage for data.
- Is a container for a collection of entries that consist of a tuple and associated metadata.
- Is used concurrently by applications distributed over a network to store, retrieve, and consume data. Each application has the same view of the data contained in the space.

After connecting to a metaspace, your application can define, drop, join, and leave spaces, and also get an existing space's definition and list of members.

For information on joining a leaving a space, see Joining and Leaving a Space, page 100.

To enable the data grid, spaces:

- Distribute and synchronize data in a platform independent manner.
- Proactively notify applications of changes in the data contained in the space as changes happen (push model), and can therefore be used as a coordination mechanism for building distributed systems.

A space is distributed and implemented collaboratively by a group of processes located on multiple hosts and communicating over the network.

ActiveSpaces handles changes in the set of processes automatically: processes may join or leave the group at any time without requiring any user intervention. A space automatically scales up as the number of processes in the group increases, and scales down when processes suddenly disappear from the group or network. There is no negative impact on the data contained in the space when processes leave the space.

Space Contents

A space contains tuples and associated metadata:

- A tuple is a container for a collection of fields. A tuple is equivalent to a row in a database.
- Each field in the tuple has a name, a type and a value.

For more information on tuples and fields, see Tuples and Fields, page 18.

Differences Between ActiveSpaces and a Distributed Cache

ActiveSpaces is a distributed data grid that implements a distributed in-memory tuple space. There are important differences between the ActiveSpaces distributed data grid and a distributed cache:

- A cache can *evict* entries at any time if it needs to make room for new entries, but a tuple space data grid does not evict entries Therefore, a distributed cache (like all caches) can only be used in a *cache-aside* architecture to cache a system of record, and, unlike a data grid, can never be used as a system of record itself.

  Although it is possible to use ActiveSpaces as a distributed cache (in a cache-aside or in a cache-through architecture), the reverse is not true: a distributed cache cannot be used as a system of record.

- A distributed cache does not have a notification mechanism to proactively inform applications of changes in the data stored in the cache. Unlike ActiveSpaces, a distributed cache cannot be used for distributed process coordination.

Tuples and Fields

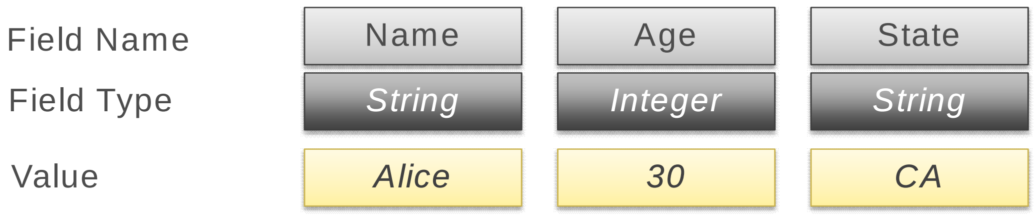
Actives Spaces spaces store data in tuples. A *tuple* is:

- A container for a sequence of field.
- Equivalent to a row in a database.
- Represents a set of related data.

A *field* is similar to a column in a database table. Each field has a specific name, type and value, as shown in the following figure.

Tuples and fields function like rows and columns in a traditional database.

Figure 1 Field Definition



A tuple can be seen as a kind of map in which fields can be put or removed. A tuple can also be seen as a self-describing message. Tuples are platform independent, and can be serialized and deserialized.

For information on defining tuple fields, see Defining Data Fields, page 95.

Field Type Conversion

When a tuple is stored into a space, the fields that it contains must match the names and types of the fields described in the space definition. If there is a type mismatch between a field contained in the tuple and the type of the field defined in the space field definition, then, if possible, ActiveSpaces performs an automated field conversion. If the conversion is not possible, the operation fails.

Table 4, Field Type Conversions shows which type conversions are supported. The letters in the table have the following meanings:

`x`: Conversion is supported with no loss of precision.

`l`: Conversion is supported, but with loss of precision.

N Conversion is not supported.

Table 4 Field Type Conversions

	Boolean	Short	Integer	Long	Float	Double	Blob	String	DateTime
Boolean	x	x	x	x	x	x	N	N	N
Short	l	x	x	x	x	x	N	N	N
Integer	l	l	x	x	l	x	N	N	N
Long	l	l	l	x	l	l	N	N	N
Float	l	l	l	l	x	x	N	N	N
Double	l	l	l	l	l	x	N	N	N
Blob	N	N	N	N	N	N	x	N	N
String	N	N	N	N	N	N	x	x	N
DateTime	N	N	N	N	N	N	N	N	x

There is a `Get` and `Put` method for each type of field. In Java, an overloaded `Put` method is also provided for convenience.

For general information on operations applied to tuples, see Getting the Name and Definition of a Space, page 110.

Key Fields and Indexes

This section discusses key fields and indexes.

Key Fields

You must define at least one of the fields in the space definition as a *key* field. ActiveSpaces uses key fields to build a key-value index for the space.

In the Java API, you can specify a set of fields to be used as key fields by using the SpaceDef's `setKey` method and passing it a number of strings containing the space names.

For detailed information on defining key fields, see Defining Key Fields, page 96.

Indexes

ActiveSpaces automatically builds a distributed, in-memory index of the tuples in the space when a space is created or loaded. Because indexes are stored in memory, queries locate matching records more quickly because the queries do not have to iterate through every record.

Using indexes, ActiveSpaces also allows you to query for any field of the records contained in the space, and the queries can be serviced faster if indexes are built on the fields used by the query filter statement.

Indexes can be either hash indexes (the default) or tree type indexes, and can contain one or more fields:

- A hash index is more efficient if the set of values to be stored is randomly distributed or the query is selecting for specific values rather than ranges of values.
- A tree index is more efficient when the query is selecting ordered ranges of values.

ActiveSpaces allows you to define as many indexes as you want on a space, as required, depending on the types of queries that will be run over the space. Indexes are part of the space's definition and are built on one or more of the fields that are defined for the space. You can build indexes on any of the fields defined for the space. Indexes have a type, which can be either “HASH” or “TREE.” Hash indexes speed up queries where the filter is an exact match ('=' operator) of a value to the field, e.g.: “field = value”. Tree indexes speed up queries where the filter is a range match ('>', '<', '>=', '<=' operators) of a value to the field, e.g. “field > value.”

If your query filter uses only one field, then you can speed it up by defining an index just on the field that it uses. If your query filter uses more than one field, then you can speed it up by creating a 'composite index' on the fields used in the filter. In this case the order of the fields when the index is defined matters when the TREE index type is used and the query filter contains both equality and range operators separated by 'AND': for example if the query is “field1 = value1 and field2 = value2 and field3 > value3” then in order to benefit from the index, it should be defined on fields “field1”,”field2,”field3” in that order (and only in that order).

A particular field can be used in more than one index, for example if two query filters such as “field = value” and “field > value” are to be used, then you could define two indexes on the field in question: one of type 'HASH' and the other one of type 'TREE,' and the ActiveSpaces query optimizer will automatically use the appropriate index depending on the query being issued.

There is always an index automatically created on the key fields of the space, this index is of type HASH by default (but can be changed to a TREE type if needed).

Basic Operations on Tuples

You can perform the following basic operations on tuples:

- **Put** Stores a tuple into a space.
- **Get** Retrieves the complete entry associated with provided key field(s).
- **Take** Performs an atomic "get and remove" action on the entry for provided key field(s).

  A take is a "consume" operation; therefore if two takes are initiated at the same time on the same entry, only one succeeds.
- **Lock** Performs an atomic "get and lock" action on the entry for provided key fields(). Can also lock a specific entry directly.
- **Update** Performs an atomic "compare and set" operation on the entry for provided key field(s). When used on a locked entry, also automatically unlocks it.

For information on performing puts, gets, and takes, see Reading and Writing in a Space, page 111.

Get, Put, Take, Lock, and Update are single entry operations. Two additional combination atomic operations are provided:

- **PutAndLock** Puts a tuple into a space and automatically locks it.
- **UpdateAndLock** Performs an atomic compare and set operation on a tuple and automatically locks it.

Batch Versions of Tuple Operations

The ActiveSpaces API provides batch versions of the basic tuple operations, which operate on a collection of tuples instead of on than just one tuple. Using the batch forms of the operations increases throughput by parallelizing the operations (including operations over the network).

For example, the Java API includes a single entry `Space.take` method that operates on a single tuple, and also a `Space.takeAll` method that operates on a collection of tuples. And the C API set includes a **`tibasSpace_Put()`** function that puts a single tuple into a space and also a **`tibasSpace_TakeAll`** operation that puts a collection of tuples into a space.

The Put Operation: Storing Data into a Space

Your application can store data into a space by using the space's `put` method and passing it a tuple as its argument. Once the tuple is in the space, it can be accessed by any other application using that space. Existing entries are replaced with new ones, which means that if there was already a tuple with the same key field values stored in the space, it is overwritten by the new tuple.

For information on performing a Put, see Performing a Put Operation—Storing a Tuple in a Space, page 112.

When a tuple is stored into a space, it is validated against the space definition as follows:

- Field names and types are checked against the fields defined for the space.

  If a tuple's field does not match the space's definition, ActiveSpaces attempts to automatically convert the field's value to the desired type as long as the field type is numerical (no lexical casting).
- Fields marked as `nullable` need not be present in the tuple, but if they are present, their type must match or be able to be upcasted.
- Fields present in the tuple that are not defined in the space's definition are not stored in the space

Batch Versus Blocking Operations

By default, spaces are distributed, which means that the servicing of requests and storage of entries for the space is implemented in a distributed manner by all of the space's seeders.

If seeders are distributed over a network, then some operations require at least one network round-trip to complete. Therefore, using the parallelized batch versions of the operations (or distributing space operations over multiple threads) rather than invoking the same blocking operation in a loop is the best way to achieve a high throughput of operations.

Retrieving Data from a Space

There are three ways to retrieve (or consume) data from a space:

- **Get Method** A tuple space implements the associative memory paradigm and allows the application to get a complete copy of the tuple associated with specific values of it's key fields.

  This is done by using the spaces's `get` method and passing a tuple containing appropriate key field values for that space. If a tuple with matching values for its key fields is currently stored in the space, the value of the status in the result object returned by the get method is equal to OK. If no tuple in the space has matching values for the key fields, the value of the status in the result object is NULL.

- **Callback Query Method** You can create listeners on a space that invoke a user query callback function as filtered initial data and new data are pushed from the space to the listeners. For more information on listeners, see Listeners on page 54.

- **Space Browser Query Method** You can also create space browsers on the space that let users retrieve filtered data initially stored in the space and retrieve new data tuple by tuple and on demand. For more information on space browsers, see Browsers on page 50.

Which method you use to retrieve data from a space depends on the application logic of your code:

- To retrieve a single tuple using an exact key match, use the `get` function.
- To retrieve and monitor either all or a filtered subset of the data contained in the space, both listeners and space browsers offer the same functionality. The choice of which method to use depends on whether your application needs a multi-threaded event-driven callback-oriented approach, or needs to iterate through the tuples at its own pace (i.e., on demand, using the space browser's `next` method).

The Take Operation: Consuming or Removing Data from a Space

You can remove tuples from a space by using the space’s `take` method and passing a tuple containing the appropriate key fields for that space. The `take` method behaves exactly like an atomic *get-and-remove*: If a tuple with matching values for its key fields is currently stored in the space:

- The status value of the result passed to the `take` operation is be equal to OK.
- The complete tuple is contained in the result, and at the same time removed from the space.

Otherwise (if there is no tuple with matching values for its key fields currently stored in the space), there is nothing to take, and the result's status is equal to `NULL.` Since ActiveSpaces provides immediate consistency, you have a guarantee that if two separate applications issue a take for the same entry at the same time, only one of them will see its take operation succeed; the other one will see its result's status be equal to `NULL`.

Unlike a simple delete operation that succeeds even if there is nothing to delete, you can use the take operation to effectively “consume” data from a space (for example, using a space browser), and your application can easily distribute workload using ActiveSpaces.

You can also perform a `take` operation on all or a filtered subset of the tuples contained in a space by using a *space browser*. For more information on space browsers, see Browsers on page 50.

Joining a Space: Members and Member Roles

Applications that need access to a space join the space and become space members. Your application can play two distribution roles when it joins a space:

- **Seeder** Plays an active role in maintaining the space by providing CPU and RAM resources.
- **Leech** Plays a passive role. Has access to space data but provides no resources.

For detailed information on joining a space, see Joining a Space, page 100.

Seeders

A seeder application participates in the storing of data in the space and can read and write data. When seeder applications join or leave the space, ActiveSpaces redistributes the data in the space as necessary to maintain even data distribution.

Leeches

A leech application participates passively in the space and does not read and write data or cause redistribution of space data when it joins or leaves the space.

Processing Characteristics of Seeders versus Leeches

You can consider seeders to be "servers" for the space, and leeches to be "clients." However, because applications can join a space as seeders, effectively embedding ActiveSpaces inside the application process, an application joining a space as a seeder is both a server and a client. Note also that the role played by an application is on a per space basis: a single application might be a seeder on one space and a leech on another space.

Using the as-agent Process as a Seeder

ActiveSpaces includes a utility called `as-agent`, which can join a space and function as a seeder.

The `as-agent` process provides:

- Scalability to a space, by automatically joining distributed spaces as a seeder and leveraging the resources of the machine where the agent is running.
- An access point to the metaspace for the remote clients.

Using the as-agent Process to Implement Remote Clients

A connection to a metaspace through a seeder or a leech is a direct connection to the metaspace. For applications running on hosts that are remote from the metaspace, or separated from it by a firewall, you can connect as a remote client.

You set up a remote client by running an `as-agent` that provides proxy access to the metaspace for remote clients. The as-agent, in effect, functions as a seeder for the remote client. The command line argument for the as-agent process specifies a “remote listen” URL that the agent uses to listen for data.

As-agents can also implement shared-nothing persistence.

For more information about as-agent, see Administrative Interfaces: AS-Admin, AS-Agent, and ASMM, page 65.

When to Join the Space as a Seeder or a Leech

Consider the following points when deciding when your application should join a space as seeder or as a leech:

- Even though ActiveSpaces has a true peer-to-peer architecture, rather than a client-server architecture, you can deploy applications as leeches (effectively, as clients of the space service) with as-agents acting as a server cluster.
- For some operations, an application that joins a space as a seeder experiences better performance than it would as a leech, but this comes at the expense of higher RAM and CPU usage.
- The entries in the space are stored randomly using a hash of the value(s) of the key field(s) (in practice as good as random), but are stored evenly between all of the seeders of the space. Seeders do not necessarily seed what they put in the space.
- The distribution role (seeder or leech) is only a level of participation—not a limitation on use. Leeches have access to the same set of space operations as seeders.
- You can also use the `as-agent` process to “keep the data alive” when all of the instances of an application have disconnected from the metaspace.
- When a seeder joins or leaves a space, there might be a temporary impact on space performance while redistribution is performed. On the other hand, leeches do not incur any impact when joining or leaving a space.

The choice of distribution role must be made on a per space basis: the best solution may be to join some spaces as a seeder and others as a leech.

Space Definition

You must define a space in the metaspace before it can be joined by applications and agents. The space is created when a member of the metaspace joins it and becomes the first member of the space. Conversely, the space is destroyed when the last member leaves it (and there are no more members of the space).

The space remains defined after all members have left, and can be reactivated if needed.

A space definition comprises two parts:

1. A set of space attributes and policies that define the space's behavior and mode of deployment.
2. A set of field definitions that describe the format of the data that will be stored in the space.

The space definition is contained in a `SpaceDef` object that is defined in the ActiveSpaces API set. The SpaceDef object is either created from scratch by invoking the `SpaceDef`'s `create()` method, or returned by the metaspace or space's `getSpaceDef` methods.

After a SpaceDef object has been created, you can set values for space attributes by specifying values for elements in the SpaceDef object.

Overview of Space Attributes and Policies

The attributes of a space define the space's behavior an mode of deployment. By calling the SpaceDef functions or methods provided in the ActiveSpaces API, you can specify:

- **Space Distribution** Specifies whether a space is distributed.

  For information on space distribution, see Distribution, page 30.
- **Space Capacity** Specifies the maximum number of entries per seeder.
- **Eviction Policy** (If a space capacity setting is specified, must be specified to set an eviction policy that is followed when the space capacity limit is reached.
- **Replication Count** Specifies whether replication is enabled, and if replication is enabled, specifies the number of seeders that are used to replicate data.

  For information on replication, see Replication, page 33.
- **Replication Type** If replication is enabled, specifies whether replication is synchronous or asynchronous.

For information on synchronous and asynchronous replication, see Synchronous and Asynchronous Replication, page 35.

- **Persistence** Specifies whether space data is persisted to permanent storage, and if so, what type of persistence is used.

  For information on persistence, seeSpace Storage Options and Persistence, page 37.

- **Routing** Specifies whether the space is routed.

  For information on implementing routing for a space, see ActiveSpaces Routing, page 67

- **Entry TTL** Controls how long a tuple can remain unmodifed before it is evicted from the space.

  For information on time to live and lock wait, see Expiration: Time to Live and Tuple Locking, page 45.

- **Lock TTL** controls how long a tuple remains locked after an application has locked it.

  For information on time to live and lock wait, see Expiration: Time to Live and Tuple Locking, page 45.

- **Lock Wait** How long an operation attempting to modify a locked tuple can block while waiting for a tuple lock to clear.

  For information on time to live and lock wait, see Expiration: Time to Live and Tuple Locking, page 45.

Field Definitions

You create field definitions in two steps:

1. By creating field definitions and specifying the data type for each field.
2. By associating the fields with a space definition.

Distribution

A space may be either distributed or non-distributed:

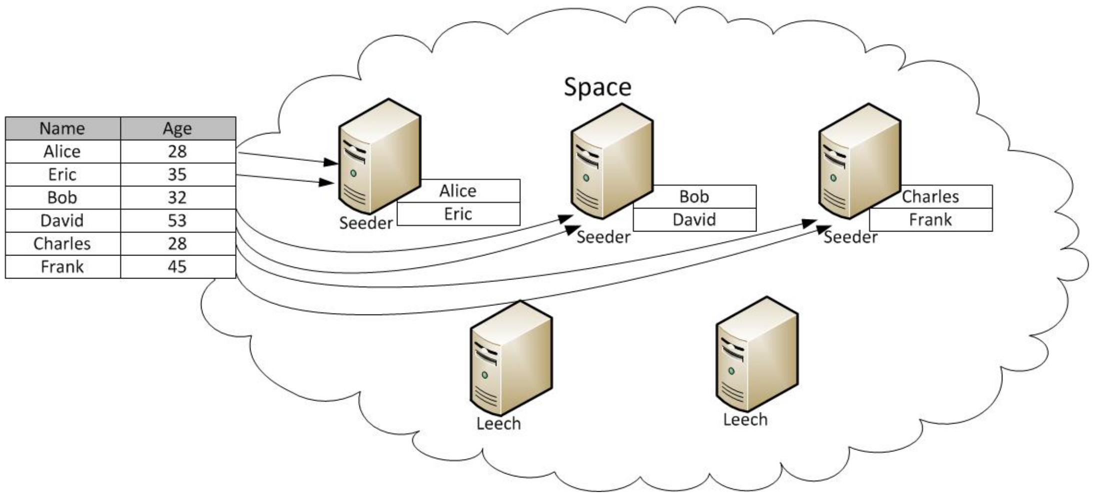
- **Distributed Spaces** With distributed spaces, management of the space data is shared among the seeders that have joined the space. Responsibility for storing the tuples is distributed evenly among all the seeders joined to the space.
- **Non-Distributed Spaces** With non-distributed spaces, a single seeder is responsible for all the tuples in the space (the responsibility for storing tuples in the space is assigned to one of the seeders joined to the space). However, other seeders may still store the tuples in the space, depending on the replication degree specified for the space. degree). (other seeders joined to the space may also replicate these tuples if a degree of replication is specified)

Distributed Space

By default, spaces are distributed. In a distributed space, management of the space's entries is distributed among the seeders that are members of the space, and the ActiveSpaces distribution algorithm ensures that entries are distributed evenly in the space.

Figure 2, Distribution of Entries in a Space shows how the entries for a space are distributed between seeders in the space. Each seeder has approximately the same number of entries.

Figure 2 Distribution of Entries in a Space



To ensure the best possible (most even) distribution of entries in a space regardless of the number of entries, the granularity of the ActiveSpaces distribution algorithm is a single key field's value. This means that an individual distribution decision is made for every entry stored in the space.

In a distributed space, management of the space's entries is distributed among the seeders that are members of the space:

- An efficient distributed hashing algorithm is used to ensure an even distribution of the entries among the seeders.
- The scalability of the space is limited to the number of entries that all the seeder nodes can manage.
- The ActiveSpaces coherence protocol ensures global ordering of the operations performed on values associated with a single key in a distributed space, and ensures that those changes are propagated as they happen. ActiveSpaces guarantees that every member of the space sees changes to the values associated with a particular key in the exact same order, regardless of the member's physical location or level of participation in the space.

Non-Distributed Space

A non-distributed space is entirely managed by a single member. The main reason for using non-distributed spaces is to get absolute view synchrony, so that changes are seen in the same order (as opposed to seeing changes in the same key in the same order).

At any time, one member of the space—the seeder, is in charge of managing the entries for the space. The scalability of the space is limited to the number of entries that the single seeder can manage.

Minimum Number of Seeders

It is possible to define a minimum number of seeders for a space. If this attribute is defined, the space is not usable until the required number of seeders have joined it. Since it is not possible to service any operation on a space until there is at least one seeder for it, there is always an implied default value of 1 for this setting.

Replication

To provide fault-tolerance and prevent loss of tuples if one of the seeders in a space suddenly goes down, you can specify that space data is replicated—backed up one or more seeders.

ActiveSpaces replication is performed in a distributed *active-active* manner:

- **Redistribution or Replication** Redistribution is performed when a new seeder *joins* the space, rereplication happens when a seeder *leaves* the space.
- **Distributed Replication** Replication is distributed over several seeders. Each seeder seeds some tuples and also replicates tuples assigned to other seeders. A given seeder does not have a designated backup that replicates all of the tuples that this seeder seeds; instead, the tuples that it seeds are replicated by any of the other seeders.
- **Active-active Mode** There are no “backup seeders” waiting for a “primary seeder” to fail to start before becoming active. Instead, all seeders are always active, and are both seeding and replicating tuples. This ensures efficient replication— if a seeder fails, there is no need to redistribute the tuples among the remaining seeders to ensure that the distribution remains balanced. ActiveSpaces simply rebuilds the replication data, which is less work than having to redistribute. This results in a lower performance impact when a seeder fails.

Degrees of Replication

When you configure a space you specify whether replication is enabled, and if it is enabled, specify the replication value using the `SpaceDef` object’s `setReplicationCount` or `getReplicationCount` methods. The replication value specifies whether replication is enabled, and if it is enabled, how may seeders participate in replication. You can configure:

- **Zero Replication** A value of 0 (the default value) specifies there is no replication. In this case, if one of the members that has joined the space as a seeder fails suddenly, the tuples that it was seeding disappear from the space. However, if the member leaves in an orderly manner by invoking a call to leave the space or call disconnect from the metaspace, there is no data loss.
- **Replication of Degree 1** This specifies that each tuple seeded by one member of the space is also replicated by one additional other seeder of the space. If a seeder suddenly goes down, the tuples that it was seeding are automatically seeded by the nodes that were replicating them, and no data is lost.

Seeders do not have designated replicating members; instead, all of the tuples seeded by a particular member of the space are evenly replicated by all the other seeders in the space. This has the advantage that even after a seeder failure, the tuples are still evenly balanced over the remaining set of seeder members of that space. It also means that ActiveSpaces' fault-tolerance is achieved in an active-active manner.

- **Replication degree 2 or higher** Higher replication degrees specify that each tuple seeded by a member of the space is replicated by two or more seeders in the space. This ensures that two or more seeder members of a space can go down (before the degree of replication can be re-built by the remaining members of the space) without any data being lost.
- **REPLICATE_ALL** If you specify REPLICATE_ALL, then all of the tuples in the space are be replicated by all the seeder members of the space. This allows the fastest possible performance for `Get` operations on the space, at the expense of scalability: each seeder member has a coherent copy of every single tuple in the space, and can therefore perform a read operation locally, using either its seeded or replicated view of the tuple.

Phase Count and Phase Ratio: Tuning Redistribution and Replication

You can control the pace and throttling of redistribution and replication be by using the following space definition settings:

- **Phase Count** The Java `setPhaseCount` method sets the number of phases that are used during redistribution and replication.

  - A phase count of 1 provides the fastest redistribution/replication time, but at the expense of incurring the most impact on ongoing space operations (no client requests are serviced until the redistribution/replication is completed).

  - The default value (-1) specifies that the total number of records that need to be redistributed or replicated is divided by 100,000 to compute the number of phases.

- **Phase Ratio** The Java `setPhaseCount` method specifies a percentage of the time spent doing redistribution/replication versus the time spent servicing client requests.

  By using the values that you specify for the Phase Count and the Phase Ratio, ActiveSpaces computes the amount of time to wait before starting the next phase, as follows:

  *Amount of time to wait before starting next phase = Amount of time spent during a phase * Phase ratio percentag*e

- The default value is 100 (%), which indicates that if, on average, a redistribution/replication phase takes *x* ms to complete, then redistribution/replication ise paused for 100% of *x* ms until the next phase starts.

Synchronous and Asynchronous Replication

You can also define whether replication is performed in synchronous or asynchronous mode for the space.

Asynchronous Replication

Asynchronous replication is the default behavior. It offers the best performance; however, it does not guarantee that the data modification is *n*-replicated to the desired degree by the time the operation completes.

Asynchronous replication does not degrade the coherency or consistency of the data view between members of a space. Under normal operating conditions, all e space members are notified of the change in the data at almost the same time when the operation returns.

Synchronous Replication

Synchronous replication offers the highest level of safety, at the expense of performance. With synchronous replication, an operation that modifies one of the tuples in the space only returns an indication of success when the modification has been replicated up to the degree of replication required for the space.

Comparison of asynchronous and synchronous replication

Asynchronous replication is more *permissive* than synchronous replication. It allows performance of operations that modify space data even when the configured degree of replication cannot be achieved because there are not enough seeders— as soon as a sufficient number of additional seeders have joined the space, the replication is performed automatically. Synchronous replication does not allow such operations.

Asynchronous replication provides a *best effort* quality of replication, while synchronous replication ensures a strict enforcement of the replication degree.

Host-Aware Replication

To enable host-aware replication, you can group seeders to help prevent the loss of replicated data. With host-aware replication, the data from the seeders in one group is replicated on seeders that reside in other groups.

For example, if you group all of the seeders that reside on one device into a group and that device goes down, no data loss occurs, because the replicated data is guaranteed to reside on seeders in another group that is not on the device that went down.

With *host-aware* replication, you can ensure that replicated data does not reside on the same system as the original data and therefore is not lost if that system goes down. Instead of increasing the replication degree to ensure that replicated data exists on other systems when more than one seeder resides on the same system, you can use host-aware replication instead.

With host-aware replication, you group seeders based upon their member names. To organize seeders into groups, use member names of the form:

<*group_name*>.<*member_name*>

ActiveSpaces groups all seeders with the same *group_name* together and their data is replicated on seeders outside of that group.

You can group any seeder in this way. You can set implement host-aware replication for ActiveSpaces applications run as seeders as well as as-agents that you start as seeders.

Space Storage Options and Persistence

ActiveSpaces provides several options for storing space data:

- RAM storage
- Storage on solid state drives
- Storage on magnetic disk

RAM Storage

ActiveSpaces data can be stored purely in RAM on machines running seeder processes. RAM is the fastest storage medium and provides the fastest reads and writes; however, it is the most ephemeral storage medium.

If the seeder process (or the machine running the process) goes down, the data held in the process' memory is lost. Although the ActiveSpaces mechanism enables recovery from individual process or machine failures, if all of the seeder processes of a space go down (because they all crash, or because the machine is in maintenance), then the data disappears from the space along with the last seeder.

Persistence

To avoid possible data loss, you can persist data to physical media. ActiveSpaces allows you to persist data to disk storage and recover data if data loss occurs or there is a problem with cluster startup.

You can persist space data to a storage system such as a database, a key-value store, or even a file system. When you define a space and specify that it is persisted, the space data is maintained in the persistence layer, and can be recovered at startup.

In addition, if the space is defined as persistent and you also specify a capacity value and an eviction policy of Least Recently Used (LRU), then you can use ActiveSpaces to cache access to the persistence layer in “cache-through” mode. In this case, applications can transparently access the data stored in the persistent layer through the space. If the data associated with a particular key field value is not in the space at the time of the read request (a “cache miss”), then it is transparently fetched from the persistence layer, and stored in the space such that a subsequent request for a `get` on the same key value can be serviced directly and much faster by the space (a “cache hit”).

When making a query on a space using a browser or a listener on a transparently cached space, there is a difference in behavior between the shared-nothing and the shared-all persistence modes of operation:

With the built-in shared-nothing persistence, the query can return ALL of the tuples stored in the space regardless of whether they are present in the cached records in RAM or on persistent storage. What is already cached is returned faster than what is evicted, but every matching record is returned. However, to do this, the fields being queried in the space MUST have indexes defined on them.

With external shared-all persistence, listeners and browsers only return the matching records that are present in the RAM-cached subset of the space, and will NOT return records that are only present in the persistence layer at the time the query is issued.

When a space is defined as persisted, it requires at least one persister or at list the minimum allowable number of seeders.

ActiveSpaces provides two types of persistence:

- **Shared-All Persistence** The implementation for external "shared-all" persistence is provided in the ActiveSpaces libraries. All nodes share a single persister or a set of persisters. Using the ActiveSpaces API, your application must provide an implementation of the persistence interface and interface to the shared persistence layer of choice.
- **Shared-Nothing Persistence** Shared-nothing persistence is built into the ActiveSpaces system, and provides a distributed back-up of space data. Each node that joins a space as a seeder maintains a copy of the space data on disk. Each node that joins as a seeder writes its data to disk and reads the data when needed for recovery and for cache misses. This type of built-in persistence is implemented by the ActiveSpaces libraries

When you implement persistence, you can use RAM to store either all of the data, or the most recently used data. The persistence layer holds all of the data stored in the space but the RAM of the seeder processes is used as a transparent in-line cache of a configurable size.

Shared-All Persistence

If you implement shared-all persistence, your application must provide code to handle reads to and writes from the external persistent storage medium. You can use a traditional RDBMS (or any other centralized disk-based data store) as the persistent storage medium.

With shared-all persistence, certain space members are designated as *persisters* — to provide the service of interacting with a persistence layer, just as some of the space members — the *seeders* — provide the basic space service.

With shared-all persistence:

- "Key operations," for example, Get and Take operations, transparently fetch entries that have been evicted from the space from the persistence layer.
- Queries only return matching records that are cached in RAM at the time the query is issued, but do not return records that have been evicted from the space.

Shared Nothing Persistence

When you use ActiveSpaces' built-in shared-nothing persistence, your application does not need to implement code to take care of persistence — ActiveSpaces seeders use any file system accessible to them (for example local solid state or disk drives) as the storage (and) medium.

When combined with in-memory indexing, shared-nothing persistence allows you to use ActiveSpaces as a distributed data store using local disks for persistent data storage and RAM as a *truly* transparent in-line caching layer.

With built-in shared-nothing persistence, if you define indexes on the fields used in a query, ActiveSpaces has a unique ability: because the key fields and indexes for all of the records in the data store are kept in RAM, queries return not just the matching records that are cached in RAM, but also records that have been evicted from the space.

Persistence Policy and Implementation

For both types of persistence, you can specify that the persistence is maintained synchronously or asynchronously.

Shared-Nothing Persistence

With shared-nothing persistence, each node that joins a space as a seeder maintains a copy of the space data on disk.

Where Is Persisted Data Stored?

When you configure shared-nothing persistence, you must use a unique name for each member joining the spaces, and you must specify an existing directory path for which ActiveSpaces has read and write access.

You can specify the directory path for data storage as follows:

- Through calls to the API operations.
- For as-agents, by using the command line arguments for `as-agent`.

- Through an environment variable.

The directory you specify is used as the root path under which ActiveSpaces creates its own subdirectory structure, using the format `metaspace/space/member`.

ActiveSpaces creates and manages persistence files automatically. You do not have to provide a filename for the stored data—the data store directory is used as the location to create and use the file.

For detailed information on implementing shared-nothing persistence, see Setting up Persistence, page 102.

For detailed information on implementing shared-all persistence, see Setting up Persistence, page 102.

Terms and Concepts for Persistence

The following terms and concepts are useful for understanding persistence:

- **Space State** Indicates whether the space can accept regular space operations or not. This happens only when the space is in READY state.
- **Persistence Type** Defines what type of persistence ActiveSpaces uses. Shared-all and shared-nothing are the supported types. Only one type of persistence can be configured on the same space at the same time.
- **Persistence Policy** Defines how the changes to space will be persisted—synchronously or asynchronously.
- **Member Name** A unique name to identify each node/seeder/member. Recommended if using shared-nothing persistence.
- **Data Store** The file system/directory location where ActiveSpaces stores the persistence files.
- **Data Loss** Data loss is detected by ActiveSpaces when the number of nodes (seeders) that either leave or fail (due to a crash) exceeds the count set for the space. In this situation, the space is marked as FAILED.
- **Space Recovery** ActiveSpaces recovers a space (based on user intervention) when the space state is FAILED either due to data loss or cluster startup.
- **Space Resume** ActiveSpaces resumes a space (based on user intervention) when the space goes into a SUSPENDED mode due to loss of a persister.

TIBCO ActiveSpaces Cluster Startup with Persistence

With shared-nothing persistence, when ActiveSpaces nodes are started for the first time and join the metaspace (and subsequently the defined space), ActiveSpaces creates new data store files, and since there are no old files to recover from, the space automatically enters the READY state and is available for space operations.

If any ActiveSpaces node is restarted either after a failure or as a new member, the space is available for space operations if none of the nodes in the space find files to load data from. If any node has an old file, the space state is set to WAITING (or INITIAL if starting nodes after failure), and your application must initiate a load action.

Space Recovery with Persistence

When you configure persistence, you have the option of configuring space recovery. Space recovery has two options that you can specify through the API functions or the recover command:

- **Recovery with Data** Use this option if data loss is not acceptable and you want to reload the data from persistence files into the space.
- **Recovery Without Data** If data loss is acceptable, then use recovery without data. This specifies that ActiveSpaces does not load data back into the space from persistence files.

You can perform recovery by using:

- API operations
- The Admin CLI

For detailed information on setting up recovery, see Setting up Recovery with Persistence, page 105.

Space Resume with Shared-All Persistence

When a space loses one of its persisters, the space is set to a SUSPENDED state, which means that no writes to persistence files can happen. In this case, you can resume the space.

Implementing Persistence

For details on how to set up persistence, see Setting up Persistence, page 102.

Space Life Cycle and Persistence

The space life cycle starts when the space is first defined in the metaspace and ends when the space definition is dropped from the metaspace.

A space can be in one of the following states:

- **INITIAL** The space has been defined and is waiting for the minimum number of seeders required by the space's definition to be reached, and for at least one persister to be registered if it is a persisted space.
- **LOADING** The space is a persisted space that has reached the required minimum number of seeders and has at least one registered persister. One of the persister's `onLoad` methods is being invoked and the space data is being loaded from the persistence layer.
- **READY** The space has the required minimum number of seeders and if persisted, data has been loaded and has at least one registered persister.

Space operations that read or write data in the space are only allowed when the space is in the READY state. The only exception to this rule is that the space's load method can be invoked when the space is in the LOADING state (typically by the registered persister `onLoad` method).

Your application can check that the space is in the READY state before attempting to use it by using the space's `isReady`() method.

Your application can also synchronize itself with the space's state by using the space's `waitForReady` method. This method takes a timeout that is the number of milliseconds for which it will block while waiting for the space to reach the READY state, and returns a boolean value indicating whether the timeout was reached or not (Java also has a convenient version of the method that does not take a timeout and just blocks until the space is ready).

Another way to synchronize an application with the space's state is to rely on the space definition's `SpaceWait` attribute: a configurable timeout that is used to block space operations when the space is not in the `READY` state until either the space becomes ready (at which point the operation is executed) or the `SpaceWait` timeout expires (at which point the operation will fail).

Persistence and Space Life Cycle

When a space needs to be persistent so that the data that is stored in it does not disappear after a disaster (all seeders have crashed) or a maintenance shutdown, you should define it as a persisted space.

Two choices are available for persistence: built-in shared-nothing persistence or external shared-all persistence.

At a high level, persistence is invoked at various steps in the life-cycle of a space:

- When a space is first reinstantiated after a complete shutdown or crash, the persistence “loading” phase is first invoked to “re-hydrate” the data (or rebuild indexes) from the persistent storage medium:
  - With built-in shared-nothing persistence, loading occurs in parallel on all the space's seeders at the same time.
  - With external shared-all persistence, the `onLoad` method of one of the registered persistence implementations is invoked.
- When a change is made to the data in the space (because of a put, a take or any other space action that modifies the data), this change needs to be reflected to the persistent storage medium.

  This is done either synchronously or asynchronously in a distributed manner by each seeder (including those that replicate the data). Data is persisted to it's designated local storage file folder in shared-nothing persistence, or by the persistence implementation's `onWrite` method in external shared-all persistence mode.

  Because in shared-nothing mode writes are automatically distributed between the seeders (taking into account the degree of the space) and are done to local disk on each seeder, write performance scales along with the number of seeders (just as for a non-persistent space). However, when you use shared-all external persistence is used, because the persistence layer is shared (is a centralized RDBMS, for example) the number of writes per second is ultimately limited by what the external persistence layer can handle and does not scale when more seeders are added to the space.
- When memory is used as a transparent in-line cache (rather than to store the entire data set), if there is a request to read an entry (as a result of a get, take or lock operation, for example) that is not currently in the part of the data cached in memory, then the entry is automatically loaded from the persistence layer either automatically by the seeders from their local persistent storage in shared-nothing mode or by the persistence's `onRead` method.
- When a query (rather than a single entry read) is issued on the space, then when external shared-all persistence is used, the query will only return the matching records that are in the in-memory cache at the time, while with shared-nothing persistence, when indexes are used, the query will return ALL matching records, including those that may have been evicted at the time the query was issued.

Write-Behind Caching

ActiveSpaces supports write-behind caching in addition to write-through caching to a back-end database. Write-behind caching provides asynchronous writes to the database for faster performance, and allows writes to be buffered in cache and written to the database later in case the database is down.

The writes to the database are asynchronous to the cache operation. The process is fault-tolerant of Persister failures and seeder failures of the space up to its replication degree.

To enable the write-behind feature, you must set the shared-all persistence policy to `ASYNC`.

To indicate that an update failed to persist to the back-end data store due to the database connection being down, your persister code must return a `PERSISTER_OFFLINE` error within the `onWrite` callback.

The `PERSISTER_OFFLINE` error indicates that the update is to be retried later. ActiveSpaces does not handle any other persister-related failures, continues with a warning, and removes the specific update that failed.

The following example shows how your code can return the `PERSISTER_OFFLINE` error message.

```
ActionResult.create().setFailed(new
   ASException(ASStatus.PERSISTER_OFFLINE, sqlException))
```

The `as-admin` utility can display statistics indicating the pending update count on each node. If a system is not able to catch up with incoming user updates, the `ToPersist` count shows the status of pending updates.

Because writes are now asynchronous, changes done in cache are not reflected immediately in the database. ActiveSpaces conflates multiple updates on the same key to a single update to reflect the last update. If the database cannot keep up with cache updates, then updates on different keys are aggregated in memory and potentially use more memory than expected. Running more Persisters might help scale out the workload.

If there are pending updates, caching with a defined limit cannot guarantee that the limit is respected. Once all updates have been flushed, the limit is applied and eviction takes place. Until then, the cache might grow and necessary tuning might be required.

Expiration: Time to Live and Tuple Locking

ActiveSpaces provides configuration settings that control:

- When data expires and can be evicted from a space
- How long a tuple remains locked when an application has locked it

To control eviction and how long tuples remain locked, ActiveSpaces allows you to specify the following values:

- **Entry TTL** Controls how long a tuple can remain unmodifed before it is evicted from the space.
- **Lock TTL** controls how long a tuple remains locked after an application has locked it.
- **Lock Wait** How long an operation attempting to modify a locked tuple can block while waiting for a tuple lock to clear.

Entry TTL

ActiveSpaces is a durable data store that can be used as a cache. Tuples stored into a space are not evicted from the space to make room for new tuples unless the space is specifically configured to evict tuples after a specified time. Therefore, you can use a space as a *system of record*.

In some cases you might want to have an automated *garbage collection* mechanism operate on the space to expire old, obsolete tuples from it. To do this, you can define a time-to-live (TTL) for tuples stored in the space. The TTL is the number of milliseconds that must elapse since the tuple was created or last modified before it is considered for expiration.

You can set or return the entry TTL using the `SpaceDef` objects's `setTTL` and `getTTL` methods, respectively.

Lock TTL

Applications that have joined a space can lock tuples in the space. By default, locked tuples remain locked until the application that created the lock clears it, or until that application disconnects from the metaspace, whether in an orderly manner or not.

To avoid potential deadlock situations, you can also possible set a maximum lock-time-to-live for a space, which specifies that if an application does not clear a lock on its own within a certain number of milliseconds, the lock is automatically cleared.

You can set or return `LockTTL` can be set or returned using the `SpaceDef` objects's `setLockTTL` and `getLockTTL` methods, respectively. It is expressed in milliseconds and defaults to `TTL_FOREVER`.

LockWait

While a tuple is locked, no space member besides the creator of the lock can perform a Put or Take operation on it. Therefore, distributed applications in which multiple instances of the application make concurrent modification to the data stored in a space should always ensure that they lock tuples for the shortest possible amount of time to maximize concurrency of the overall process.

If you expect that the locks on tuples in the spaces will have a very short duration, ActiveSpaces allows you to specify a `LockWait` value for the space. The `LockWait` value is the number of milliseconds an operation attempting to modify a locked tuple can block while waiting for the lock to clear.

If at the end of the `LockWait` period, the tuple is still locked, then the operation to modify that locked tuple throws an exception indicating that the operation could not be performed because the tuple is locked. The `LockWait` value of a space is also taken into consideration if a member attempts a non-transacted operation that conflicts with uncommitted data (for example, tuples about to be replaced by uncommitted operations are locked by that transaction).

You can set or get the `LockWait` value by using the `SpaceDef` object's `setLockWait` and `getLockWait` methods, respectively. Lock wait value is expressed in milliseconds. The default value is `NO_WAIT`, indicating that this feature is not used, and that an operation attempting to modify a locked tuple will immediately return with a failure indicating that the tuple is currently locked.

For detailed information on the procedure for setting a LockWait value, see Specifying a LockWait Value for a Put, page 112.

Concurrently Updating Data in a Space

When multiple processes concurrently get and update tuples in a space, two processes might try to update the same tuple at the same time. In that case, it is often necessary to serialize updates. The classic example of this scenario is that of a bank account balance: if a deposit and a debit to the same bank account are being processed at the same time, and if each of these operations follows the pattern "get current account balance, add/remove the amount of the transaction, and set the new account balance," both transactions might start at the same time and get the same current account balance, apply their individual changes to the account value, but the application that is last to set the new account balance overwrites the other applications's modification.

There are two ways to solve this problem using ActiveSpaces:

1. An *optimistic* approach is best when the likelihood of having a collision is low. In this case, you should make use of the space's `update` method, which is an atomic *compare and set* operation.

   This operation takes two parameters, one representing the *old* data that was retrieved from the space, and another one representing the *new* version of that data. If the old data is still in the space at the time this operation is invoked, then the operation will succeed. However, if the data in the space was changed in any way, the operation will fail, which indicates that your application should refresh its view of the data and re-apply the change.

2. A *pessimistic* approach to the concurrent update problem is best when there is a high likelihood of more than one process trying to update the same tuple at the same time. In this case, application programmers should first attempt to lock the tuple, and only apply their update to it after having obtained the lock. Locking is described in the following section.

Locking Data in a Space

ActiveSpaces allows users to lock records and keys in the space. The granularity of the locking in ActiveSpaces is a key, meaning that any possible key that could be used in the space can be locked, regardless of whether a tuple is actually stored in the space.

The space's lock function takes a tuple representing the key as an input parameter and can optionally return what is stored in the space at that key (if there is anything) just as a get operation allows you to lock tuples in the space. The space's `lock` method is an atomic *get and lock*, and takes the same argument as the `get` method.

After a key is locked, it is read-only for all other members of the space except for either the process or the thread that issued the lock command. The lock's scope (the thread or the process) can be specified when the space's lock method is invoked.

If a thread or process other than the locking thread or process tries to do a put, take, lock, or any operation that would modify whatever is stored for the locked key, that operation may block until the lock is cleared.

A locked key is *read-only* for all space members except the member that has locked it. Only one member can lock a specific key at any given time. If a member other than the lock owner tries to overwrite, take, update, or lock a locked key, that operation may block until the lock is cleared. If you want to implement this behavior, set a lock wait value using the space's `LockWait` attribute.

After a key is locked, the owner of the lock can unlock it.

You can also iteratively lock all or a filtered subset of the tuples in a space by using a space browser.

Finally, you can specify a maximum time to leave for locks in a space: if a lock is held for longer than the value specified in the space's `LockTTL` attribute, it is then automatically cleared. Locks are also automatically cleared when the application that has created the lock leaves the metaspace or crashes.

Results

Most batch space operations return *results* (or collections of results). Results objects contain information about the result of the operation and are always returned by those operations regardless of the operation being a success or a failure.

A result object contains a status value indicating whether the operation completed successfully, and if so, whether or not an entry is contained in the result, or whether the operation failed and the result contains an exception object.

See Appendix A, Result and Status Codes for more detailed information on results.

Browsers

ActiveSpaces provides another method of interacting with spaces—space browsers. You can use space browsers when working with groups of tuples, rather than with the single tuple key lookup of the space’s `get` method. Space browsers allow you to iterate through a series of tuples by invoking the space browser’s `next` method. However, unlike a traditional iterator that works only on a snapshot of the data to be iterated through, the space browser is continuously updated according to the changes in the data contained in the space being browsed.

Changes happening to the data in the space are automatically reflected on the list of entries about to be browsed as they happen: a space browser never gives the user outdated information. For example, if an entry existed at the time the space browser was created, but it gets taken from the space before the space browser’s user gets to it, then this entry will not be returned by the space browser.

Do not forget to invoke the stop method on the browser once you are done using it in order to free the resources associated with the browser.

Space Browsers and the Event Browser

There are two main types of browser:

- **Space Browsers** Allow your application to not only retrieve the next tuple in a series of tuples, but also to operate directly on the tuple. You can implement: three types of space browser:
  - **Get Browser** Retrieves the next tuple in a series of tuples.
  - **Take Browser** Retrieves the next tuple in a series of tuples and consumes it.
  - **Lock Browser** Retrieves the next tuple in a series of tuples and locks it.

For information on coding a space browser, see Implementing a Space Browser: Querying the Space, page 121

- **Event Browsers** Allow you to iterate through the stream of events (changes) occurring in the space.

For information on coding an event browser, see Using Event Browsers, page 123

Here are some additional differences between space browsers and event browsers:

- Space browsers and event browsers both have two methods, `next()` and `stop()`. However, a space browser's `next()` method returns a `SpaceEntry`, while the event browser's `next()` method returns a `SpaceEvent`.

- A space browser also has a `getType()` method, which the event browser does not have.
- A space browser's `next` method will do a `get`, `take`, or `lock`, according to the browser's type: `GetBrowser`, `TakeBrowser`, or `LockBrowser`.
  - — The Get Browser’s `next()` method does a `get` on the next tuple to browse (very much like a regular iterator).
  - — The Take Browser’s `next()` method atomically retrieves *and* removes the next tuple currently available to take from the space.
  - — The Lock Browser’s `next()` method atomically retrieves *and* locks the next tuple currently available to lock in the space).
- The Event Browser’s `next` method returns a `SpaceEvent` rather than a tuple.
- The `SpaceEvent` objects returned by the event browser’s `next` method optionally include the initial values, that is, what was in the space at the time the event browser was created.
- The initial values are presented as a continuously updated string of `PUT` events preceding the stream of events that happen *after* the creation of the event browser. Event browsers allow you to see deletions and expirations of tuples they have already iterated through.

Space browsers deliver the tuples (and initial PUT events) for the initial values in no particular order, and the order might change from one instance of a space browser to another.

Since a space browser is continuously updated, it does not have a `next()` method; instead, it has a timeout: the amount of time the user is willing for the next call to block in the event that there is nothing to `get`, `take`, or `lock` at the time it is invoked (but there may be in the future).

Continuously updating tuples means that if multiple `TAKE` browsers created on the same space are used to take tuples from the space using `next()`, a particular tuple is only taken by one of the space browsers, effectively allowing the use of a space as a tuple queue.

Scopes of a Space Browser

A space browser can have either *time scope* or *distribution scope*, which are defined by setting the values of fields in the browser’s `BrowserDef` object:

Time Scope The time scope can be used to narrow the period of time of interest.

- *snapshot* means that the browser starts with all the tuples in the space at the time the browser is created (or *initial values*), but is not updated with new tuples that are put into the space after that moment.

Note that the browser's timeout value is ignored when the time scope is `snapshot`, because in this case the browser will only iterate through a finite set of tuples (only those that are present in the space at the time of the browser's creation).

- *current* Allows client applications to create queries that return large result sets using less resources. Note the following points regarding the `current` setting:
  - When the query returns a very large result set, the amount of memory required by the seeders and the querying application to provide true snapshot functionality may become a problem source; in that case, you can use the `current` time scope instead.
  - The current time scope is a lightweight, best effort, version of the snapshot time scope that requires almost no extra memory, and very little initial processing, but where changes in the space done after a browser was created with a current time scope may or may not be visible.
  - It is advisable to use the `current` time scope with key indexes of type TREE. However, the `current` time scope setting also works with HASH indexes.
  - The HASH index might return duplicate values if there is a need to grow the hash index used with a seeder. If duplicate values are an issue, it is advisable to switch to the TREE index type.
  - If there is a seeder join/leave/drop on the space while the browser is being created or is in use, then the `current` time scope throws an exception, because the iterated data would not be correct. Losing a seeder or having a new seeder causes redistribution of entries, and this violates the time scope requirements of the `current` setting.
- *new* means that the browser starts empty, and is updated only with tuples (or associated events) put into the space after the moment of the browser's creation.
- *all* means that the browser starts with all the tuples in the space, and is continuously updated with new tuples.
- *new_events* is applicable only to event browsers, and means that the browser starts empty and is updated with all the events generated by the space after the moment of the browser's creation (unlike *new*, which would only deliver events associated with entries put in the space after the browser's creation time)

Distribution Scope The distribution scope can be used to narrow down the set of tuples or events being browsed.

- *all* is used to browse over all the tuples (or associated events) in the space
- *seeded* is used to browse only over the tuples (or associated events) actually distributed to the member creating the browser

Listeners

ActiveSpaces can proactively notify applications of changes to the tuples stored in a space. Users can invoke the metaspace or space's `listen` method to obtain a listener on spaces for receiving event notifications. There are five types of listeners:

1. **PutListener** The PutListener's `onPut` method is invoked whenever a `SpaceEntry` is inserted, updated, or overwritten in the space.
2. **TakeListener** The PutListener's `onTake` method is invoked whenever a `SpaceEntry` is removed from the space.
3. **ExpireListener** The PutListener's `onExpire` method is invoked whenever a `SpaceEntry` in the space has reached its time to live (`TTL`) and has expired.
4. **SeedListener** The PutListener's `onSeed` method is invoked whenever there is redistribution after an existing seeder leaves the space and now the local node is seeding additional entries. This is only applicable if the listener distribution scope is `SEEDED`.
5. **UnseedListener** The PutListener's `onUnseed` method is invoked whenever there is redistribution after a new seeder joins the space and now the local node stops seeding some of the entries. Only applicable if the listener distribution scope is `SEEDED`.

In the ActiveSpaces Java API, listeners must implement at least one of the listener interfaces shown above. Listeners are activated using the `listen` method of the Metaspace or Space class.

- The `PutListener` interface requires an `onPut(PutEvent event)` method.
- The `TakeListener` interface requires an `onTake(TakeEvent event)` method.
- The `ExpireListener` interface requires an`onExpire(ExpireEvent event)` method.
- The `SeedListener` interface requires an `onSeed(SeedEvent event)` method.
- The `UnseedListener` interface requires an `onUnseed(UnseedEvent event)` method.

In the C API, you must call the `tibasListener_Create` function and specify a single callback function that is invoked for all event types. The new `tibasListener` object created by `tibasListenerCreate` is then activated using the `tibasMetaspace_Listen` or `tibasSpace_Listen` functions. The callback function is passed a `tibasSpaceEvent` object whose type can be determined by invoking the `tibasSpaceEvent_GetType` function.

ActiveSpaces generates space events of type:

- `TIBAS_EVENT_PUT` when a tuple is inserted, overwritten, or updated.
- `TIBAS_EVENT_TAKE` when a tuple is taken or removed.
- `TIBAS_EVENT_EXPIRE` when a tuple reaches the end of its time to live and expires from the space.
- `TIBAS_EVENT_SEED` when there is redistribution after a seeder joins or leaves, and the local node is seeding or unseeding. This is only applicable if the listener distribution scope is `SEEDED`.
- `TIBAS_EVENT_UNSEED` when there is redistribution after a seeder joins or leaves, and the local node is seeding or unseeding. This is only applicable if the listener's distribution scope is `SEEDED`.

You can also specify that a current snapshot of the entries stored in the space (sometimes referred to as *initial values*) is prepended to the stream of events. In this case, the initial values of all the tuples contained in the space at the listener's creation time are seen as space events of type `PUT` preceding the current stream of events.

Filters

ActiveSpaces supports the application of filters to both listeners and browsers, as well as the ability to evaluate a tuple against a filter. Filters allow your application to further refine the set of tuples it wants to work with using a space browser or event listener.

A filter string can be seen as what would follow the *where* clause in a **`select * from Space where…`** statement.

Examples

```
field1 < (field2+field3)
state = "CA"
name LIKE ".*John.*" //any name with John
```

Filters can make reference to any of the fields contained in the tuples. s do not provide any ordering or sorting of the entries stored in the space.

Operators Supported in Filters

Table 5 shows the operators that are supported in the ActiveSpaces filters:

Table 5 Operators for ActiveSpaces Filters

Operator	Meaning
`>, >=`	`greater than`
`NOT or ! or <>`	not
`*`	`multiply`
`=`	`equal`
`!=`	`not equal`
`ABS`	`absolute value`
`MOD`	`modulo`
`NOTBETWEEN`	`not between`
`BETWEEN`	`between`
`||`	`string concatenation`
`NOTLIKE`	`not like (regex)`

Table 5 Operators for ActiveSpaces Filters

Operator	Meaning
`LIKE`	`like (regex)`
`<, <=`	`less than`
`+`	`addition`
`OR`	`or`
`-`	`subtraction`
`AND`	`and`
`IN`	`range, as in " age in (1,3,5,7,11)`
`NULL`	`does not exist`
`NOT NULL`	exists
`IS`	only used with `NULL` or `NOT NULL`, as in "`x IS NULL`" or "`x IS NOT NULL`"
`NOR`	nor, as in "`age NOT 30 NOR 40`"
`/* ....`	comments
`// ....`	comments

Specifying a String Value in a Filter

If you specify a string value in a filter, then the filter value must be enclosed in double quotes; for example:

```
value = "Jones"
```

See ASQuery (Java Only), page 181 for examples of filter queries that utilize strings enclosed within double quotes.

Regex Syntax for Filter Values

Table 5 indicates several filter formats for regular expressions (regex values). For regular expressions, ActiveSpaces uses the syntax for Perl Compatible Regular Expressions (PCRE).

For general information on PCRE, see the PCRE website at the following URL:

http://www.pcre.org/

For detailed documentation on PCRE, see the text version of the man pages for PCRE at the following URL:

http://www.pcre.org/pcre.txt

Formats for Filter Values

Table 6 shows the formats for values used in filters.

Table 6 Formats for Filter Values

`octal value`	\oXXX
`hexadecimal value`	\xXXX
`exponents (as in 1E10 or 1E-10)`	XXXEYY
`date time`	YYYY-MM-DDTHH:MM:SS
`date`	YYYY-MM-DD
`time`	HH:MM:SS:uuuu+/-XXXXGMT
`true`	TRUE
`false`	FALSE

You must enclose datetime values in single quotes (not double quotes, as with strings).

Remotely Invoking Code over a Space

ActiveSpaces allows space members to remotely invoke code on other members of the space. This feature allows the code to be co-located with the data for optimal performance.

Execution of the Invocable interface is triggered by an application that can be running on the same member or a different member of the space. In ActiveSpaces, this is referred to as remote invocation.

The invocable code is executed either on the member that contains specified data (if the Invocable interface is used) or on specified members (if the MemberInvocable interface is used). If you use the MemberInvocable interface, your application specifies which members should execute the interface.

Compare the two approaches to updating the value of a field on all of the entries stored in a space.

- One approach is to create a browser of distribution scope `all` on the node to serially retrieve and update each entry in the space one entry at a time.

  This represents a non-distributed process, as a single node is actually doing the updating. It incurs a fair amount of network traffic, since retrieving an entry might require a round-trip over the network, and updating that entry might require another round-trip. The latency induced by those network round-trips has a negative impact on the overall throughput.

- Another approach is to use remote invocation to invoke a function on all of the Space seeders in parallel. The remote function creates a browser of distribution scope `seeded`, to iterate only over the entries that it seeds and, therefore, avoid incurring a network round-trip.

  For each entry, the invoked function updates the field and the entry the same way as described for the non-distributed process, with the difference that the entry updates will be performed much faster since they do not incur a network round-trip.

Remote space invocation is available for all language bindings. It only takes care of function or method invocation and does not take care of distributing the code to the space members; for example, the function or method being invoked must be available in the `CLASSPATH` of all space members.

Invocation Patterns

public interface Invocable

With the Invocable interface, the application indicates the key of an entry stored in the space. ActiveSpaces determines which space member stores the element associated with the key or which space member would be used to store the element, if the element does not exist in the space. Execution of the Invocable interface will then occur on that space member.

The code implementing the Invocable interface needs to be included in the CLASSPATH for each member of the space.

The following remote space invocation services are available:

- **invoke** Invokes a method only on the member seeding the key passed as an argument to the call.
- **invokeMember** Invokes a method only on the Space member being specified as an argument to the call.
- **invokeMembers** Invokes a method on all of the Space members.
- **InvokeSeeders** Invokes a method on all of the seeder members of the Space.

All of those calls also take as arguments:

- A class on which the method implementing the appropriate Invocable interface is invoked
- A context tuple that gets copied and passed as is to the method being invoked.

The `invoke` method takes a key tuple, which gets passed to the method implementing the `Invocable` (rather than `MemberInvocable`) interface in the class; the method gets invoked regardless whether an entry has been stored in the space at that key.

Both the `Invocable` and the `MemberInvocable` interfaces return a tuple, but the remote space invocation methods return either an `InvokeResult` (`invoke` and `invokeMember`) or an `InvokeResultList` (`invokeMembers` and `invokeSeeders`), from which the Tuple can be retrieved using the `getResult` (or `getResults`) method.

The methods being invoked using remote space invocation should always be idempotent; in case there is a change in the membership (or seedership) of the space while the Remote Space Invocation is being performed, the Space can retry and re-invoke the methods once the change has happened.

Transactions

ActiveSpaces Enterprise Edition allows you to atomically perform sets of space operations using transactions. Transactions can span multiple spaces, but not multiple metaspaces. A transaction starts when an individual thread in an application creates a transaction, and terminates when either `commit` or `rollback` is invoked, at which point all space operations performed by that thread are either validated or canceled. Pending transactions may be rolled back automatically if they exceed an optional `TTL` (time-to-live) threshold, or when the member creating them leaves the metaspace.

Transactions can also be moved from one thread to another using the `releaseTransaction()` and `takeTransaction()` methods.

In ActiveSpaces 2.0, the only supported read isolation level is `READ_COMMITTED`. This isolation level applies only to your view of data modified by the transactions of other applications and threads. This means that whether in a transaction or not, you will not see uncommitted data from other transactions, but if you yourself are in a transaction you will see your own uncommitted data.

ActiveSpaces has an implied write isolation level of `UNCOMMITTED`, meaning that any entry potentially modified by a pending transactional operation appears to be locked for other users of the space (in which case the space's `LockWait` attribute will apply).

Deployment

ActiveSpaces is a peer-to-peer distributed in-memory tuple space. This means that the tuples are stored in the memory of a cluster of machines working together to offer the storage of tuples. There is no central *server* used for the coordination of operations, but rather any number of *peers* working together to offer a common service.

To store tuples, ActiveSpaces uses a distributed hashing algorithm applied on the values of the key fields of the tuple to distribute the *seeding* of the tuples as evenly as possible (that is, their storing and management) over a set of peers. This means that:

- Given the current set of *seeders* (any process joined to a particular space as a seeder), any participating member of the space knows where to find the tuple associated with a particular set of key field values.
- The more seeders there are for a space, the larger the amount of data that can be stored in that space.
- There has to be at least one seeder joined to a space in order for the space to be functional. A space with no seeders joined to it does not contain any data and can not have any tuple put into it until at least one seeder has joined it.

By specifying its role as a seeder, a process indicates its willingness to lend some of its resources—memory, CPU, and network resources—to the storing of tuples in the space. This is the means by which a space is scaled up. ActiveSpaces also allows applications to use spaces as leeches, which means that, while retaining full access to the service provided by the seeders, the application is not willing to lend any of its resources to the storing of tuples in the space. Adding or removing seeders from a space can incur performance costs by necessitating redistribution of the entries in the space, while leeches can join and leave spaces without impacting performance.

Before being able to join spaces, applications must first connect to a metaspace, which is an administrative domain in which users can create and use any number of spaces, but which also represents the cluster of machines and applications being able to communicate with each other.

Networking Considerations

Applications can connect to the metaspace either as full peers to the other peers of the metaspace, at which point they will need to be able to establish and receive TCP connections from all the other full peers of the metaspace (regardless of their role in individual spaces), or as 'remote clients' that connect to the metaspace through establishing a single TCP connection to a proxying ActiveSpaces agent process (itself a fully connected peers). Fully connected peers will always experience lower latency of execution of the space operations than remote clients, and remote clients will always be limited to join spaces as leeches (rather than be able to join spaces a seeders).

Before establishing TCP connections to each other, the full peers of a metaspace need to 'discover' each other. Discovery can be done by using a reliable multicast protocol (either the built-in PGM protocol stack, or, optionally using the TIBCO Rendezvous messaging system) or directly with TCP by listing a set of well known IP addresses and ports. From a configuration standpoint, the easiest option is to use the default built-in PGM reliable multicast protocol, but this assumes that all of the full peers of the metaspace are able to exchange multicast packets with each other over the network.

In this default deployment scenario, metaspace members must be able to both receive each other's multicast transmissions and establish TCP connections to each other. To enable this, firewall settings on the host may have to be adjusted to allow sending and reception of UDP multicast packets on the port specified through the multicast URL used to connect to the metaspace, and to allow incoming and outgoing TCP connections to the ports specified in the listen URL used by the members of the metaspace to connect to it.

Also, if the host has multiple network interfaces, care must be taken to ensure that the member binds its multicast and listen transports to the appropriate interface, that is, to the network that can be used to send or receive UDP multicast packets and establish or accept TCP connections with the other members of the metaspace. The interface to use for each transport can be specified in the associated URL used to connect to the metaspace. If no interface is specified, the ActiveSpaces transport libraries will default to using the default interface for the host such as the interface pointed to by 'hostname').

For more information see Joining a Space or Metaspace: Special Considerations on page 64.

Joining a Space or Metaspace: Special Considerations

A single process can connect to a given metaspace or join a given space only once:

- A single application can only connect once to the same metaspace. In other words, you cannot invoke `connect` on the same metaspace twice and have two connections to the same metaspace from the same process. However, you can connect simultaneously to several different metaspaces, and it is possible to get a copy of a currently connected metaspace object by using the `ASCommon` object's methods.
- When a process joins a space through its metaspace connection, it will only join a space once. If you call `getSpace` twice, the process will join the space the first time you call it, but the second `getSpace` call will return to you a new reference to the previously created space object. If you specify a different role the second time you call `getSpace`, then it will adjust your role on that space, but this does not mean that you have joined the same space twice.
- The space object is reference-counted and the space will actually be left by the process only when the space object's `leave` method is invoked an equal number of times to the Metaspace's `getSpace` method for that particular space.

  The role on the space (seeder or leech) is also automatically adjusted when leave methods are being invoked on those space objects.

Administrative Interfaces: AS-Admin, AS-Agent, and ASMM

The amount of data that can be stored in a space depends on the number of seeding members of that space. It can be necessary to add seeders to a space to scale it up. **as-agent** is a pre-built process that users can run on any host whose sole purpose is to join all distributed spaces in the specified metaspace as a seeder and therefore to add the resources of the machine it is running on to the scalability of the spaces it joins. Agents can also be used to ensure that the desired degree of specified for a space can be achieved.

For more information, see “Using as-agent” in *TIBCO ActiveSpaces Administration*.

Using Remote Clients

If you have purchased a license for the Remote Client Edition of TIBCO ActiveSpaces, then you can implement ActiveSpaces on remote clients.

A remote client acts as a node without actually being a a member of the cluster. Instead of being directly connected to the space, it is connected through a proxy—typically through an as-agent.

If you are connecting to a space through a firewall, the applications that connect *must* connect as remote clients.

Remote clients can:

- Run applications that use the API operations, and perform all of the basic space operations—Get, Put, Take, Lock, and so on.
- Run the Admin CLI to allow operators to issue administrative commands.
- Use the same notification mechanisms that are used by nodes within the cluster.

How Remote Client Communication Works

Remote clients are implemented as follows:

1. You set up an `as-agent`, which plays the role of a seeder and a proxy server for remote clients.
2. Using the `as-agent`, you issue a `remote_listen` command to contact the remote client and accept incoming remote client connections.
3. Using the C API or the Java API, you perform remote discovery to discover the remote client, and you specify a list of well known IP addresses and ports of proxy as-agents to remotely connect to.

Remote Clients and Seeding Distribution Scope

A remote client application can never become a seeder on a space. If an application that is remotely connected requests to join a space as a seeder, it will remain a leech. However, remote client applications can still create browsers or listeners with a distribution scope of “seeded.” In this case, the seeded scope becomes the scope of the proxying member of the metaspace through which the remote client is connecting (i.e,. the tuples that the proxy seeds on the space if any).

ActiveSpaces Routing

The ActiveSpaces routing feature is implemented by means of a callback function, similar to the callback function used with shared-all persistence. Using routing, your application forwards updates to another site, and receives a status message in return. The operation is blocked until the status is returned.

A space does not require persistence to be enabled to enable routing. If persistence is also enabled, both the persister and the router receive callbacks before completing an operation.

This section describes:

Implementing ActiveSpaces Routing

Implementing routing for a space consists of these steps:

Enabling Routing in the Space Definition

To enable routing for a specified space, connect to a metaspace, create a SpaceDef object, and then call the `tibasSpaceDef_SetRouted()` function (C API) or the `setRouted` method (Java API).

The `tibasSpaceDef_SetRouted()` function is defined as follows:

```
tibas_status tibasSpaceDef_SetRouted(
    tibasSpaceDef spaceDef,
    tibas_boolean routed);
```

where:

- **spaceDef** Specifies the `SpaceDef` object returned to your application by the `tibasSpaceDef_Create()` function.
- **routed** Specifies whether the node data is routed. To route data, specify `TIBAS_TRUE`; otherwise, specify `TIBAS_FALSE`.

Creating the Router Object

To create a Router Object using the C API, call the `tibasRouter_Create()` function. The `tibasRouter_Create()` function is defined as follows:

```
tibas_status tibasRouter_Create(
    tibasRouter* router,
    tibas_onRoute onOpen,
    tibas_onRoute onWrite,
    tibas_onRoute onClose,
    tibas_onRoute onAlter);
```

where:

- **router** Returns a router object that can be associated with a space.
- **onOpen** Specifies the function to be invoked when the tibasSpace_SetRouter() function is called to set the router for a space.Your application is responsible for making the necessary connections to the routed site.
- **onWrite** Specifies the function to be invoked when the there is a Put or Take operation on the node to which data is routed. You function is responsible for performing the Put or Take actions that take effect on the other site.
- **onClose** Specifies the function to be invoked when the connection to another node is terminated (the `tibasRouter_Free()` function is called).
- **onAlter** Specifies the function to be invoked when a space definition is altered.

The callback definition for a router does not provide an `OnRead` function. You cannot perform Get operations over a routed connection.

Declaring a Callback Function

Declare a callback that conforms to the `tibas_onRoute` typedef (C API) or the Router interface (Java API).

The `tibas_onRoute` callback has the following function prototype:

```
typedef void (TIBAS_CALL_API *tibas_onRoute) (
    tibasRouter router,
    tibasAction action,
    tibasActionResult result);
```

where:

- **router** Specifies the router object returned by the `tibasSpace_RouterCreate()` function.
- **action** Returns the action that occurred over the routed connection.
- **result** Returns the result of the action.

In Java there is no need to declare a function prototype: the Router interface provides for methods referenced in the callback.

Setting the Router Object on the Space

After you have configured routing, you must set the Router object for the space.

To set the Router object for the space, call the `tibasSpace_SetRouter()` function (C API) or the `setRouter` method (Java API).

The `tibasSpace_SetRouter()` function is defined as follows:

```
tibas_status tibasSpace_SetRouter(
    tibasSpace space,
    tibasRouter router);
```

where:

- **space** Is a valid space object.
- **router** Specifies the router object returned by the `tibasRouter_Create()` function.

Java Implementation:

The Java `setRouter` method has the following signature:

```
Router setRouter (Router router) throws ASException;
```

Freeing a Router

When the routed connection has been terminated, you should free the router object.

To free the router object using the C API, call the `tibasRouter_Free()` function. The tibasRouter_Free() function is defined as follows:

```
tibas_status TIBAS_COMMON_API TIBAS_CALL_API tibasRouter_Free(
    tibasRouter* router);
```

where `router` specifies the Router object that was used to create the router.

Java Implementation:

Using the Java API, you can free the router by calling the stopRouter method. The stopRouter method has the following definition:

```
void stopRouter (Router router) throws ASException;
```

Performance Monitoring

The ActiveSpaces utility programs, as-admin and as-agent, support performance monitoring. When performance monitoring is enabled, as-admin users can display performance statistics by entering commands such as the following:

```
show member "<member_agent_name>" stats
```

where *member_agent_name* is the name of the member or agent for which you are querying performance statistics.

To display statistics for all members, users can enter:

```
show system stats
```

By default, performance monitorin is not active. For information on enabling performance monitoring, see Enabling Performance Monitoring, page 124.

For information on the performance monitoring commands and command output, see the *TIBCO ActiveSpaces Administration* document.

Miscellaneous Topics

The ASCommon Object

The `ASCommon` class provides a set of static methods for managing metaspaces. `ASCommon` can be used to connect to a metaspace, retrieve a list of currently connected metaspaces, and set a global log level.

`ASCommon` can be very convenient to use; for example, to obtain a copy of the currently connected Metaspace object (using ASCommon.getMetaspace(by name) since a process can only be connected once to a particular metaspace and therefore can not have parts of it's code invoke `Metaspace.connect` twice in a row for a particular metaspace.

Chapter 3 Performing Basic TIBCO ActiveSpaces Tasks

This chapter describes how to perform the most common programming tasks used to develop TIBCO ActiveSpaces® applications.

Topics

Connecting to the Metaspace

Typically, one of the first things an ActiveSpaces process does is connect to a metaspace.

Before an application can do anything with ActiveSpaces, it must be connected to a metaspace.

As mentioned in Chapter 2, TIBCO ActiveSpaces Concepts, there are two modes of connection to a metaspace—as a full peer or as a remote client. Regardless of the connection mode, the API calls to use to connect to a Metaspace are the same.

The `Metaspace.connect()` method—or, in C, the `tibasMetaspace_Connect()` call—has two input parameters:

- name
- a `MemberDef` object

The `tibasMetaspace_Connect()` function returns a metaspace object that you can use to define a space or join a space. The input parameters are explained in more detail in the sections that follow.

Metaspace Name

The metaspace name is a string containing the name of a particular metaspace instance. The name cannot start with a **$** or _, and cannot contain the special characters **.**, **>**, or *****. If null or an empty string is given as argument to the connect call, the default metaspace name of **ms** will be used.

MemberDef Object

The `MemberDef` object contains the attributes of the connection to the metaspace:

- discovery attribute
- listen attribute
- MemberName attribute

Discovery Attribute

The discovery attribute specifies how this instance of the metaspace discovers the current metaspace members.

The format of the discovery attribute determines the mode of connection to the metaspace (either as a full peer or as a remote client).

When your application is connecting as a full peer, the discovery attribute specifies how this instance of the metaspace is used.

Discovery can be unicast (TCP) or multicast (PGM —Pragmatic General Multicast or TIBCO Rendezvous—RV).

There are three different discovery protocols:

- PGM discovery

  See PGM (Pragmatic General Multicast) URL format, page 76.
- TIBCO RV Discovery

  See TIBCO Rendezvous Discovery URL format, page 77.
- TCP discovery

  See TCP Discovery URL format, page 78.

To become members of the same metaspace, all intended members of a metaspace must use compatible discovery URLs. If the members do not specify compatible discovery URLs, then they are not connected to the same metaspace.

For example, if two metaspace members attempt to connect to the same metaspace using different discovery protocols, two different metaspaces are created and the two members are in different metaspaces. For example, if one application connects to a metaspace named `ms` using `tibrv` as its discovery URL, and another connects to a metaspace named `ms` using `tibpgm` as its discovery URL, since `tibrv` and `tibpgm` are two incompatible discovery protocols, two independent metaspaces, with the same name, are created instead of a single one.

PGM (Pragmatic General Multicast) URL format

With PGM, discovery of the current metaspace members is done by using reliable IP multicast. The attributes of this discovery mechanism are expressed in the form of an URL in the following format:

```
tibpgm://[dport]/[interface];[discovery group
address]/[option=value;]*
```

where

- **`dport`** specifies the destination port used by the PGM transport protocol. If not specified, the default value of **`7888`** is used.
- **`interface;discovery group address`** specifies the address of the interface to be used for sending discovery packets, and the discovery group address to be used. If not specified, it will default to the default interface and discovery address, **`239.8.8.8`**.

- **`optional transport arguments`** a semicolon-separated list of optional PGM transport arguments. For example:
  - — `source_max_trans_rate=100000000` (in bits per second) would limit the PGM transport to limit its transmission rate to 100 megabits per second.
  - — By default, the PGM transport is tuned to provide the best performance according to the most common deployment architectures, and the values of those optional arguments should only be changed when necessary, and with care as inappropriate values could easily result in degraded performance of the product.

Creating raw PGM packets (as opposed to UDP encapsulated PGM packets) requires the process to have root privileges on Unix-based systems.

PGM discovery does not work on wireless networks.

TIBCO Rendezvous Discovery URL format

The discovery URL for use with TIBCO Rendezvous has the following format:

```
tibrv://[service]/[network]/[daemon]
```

The syntax indicates that Rendezvous is used as the discovery transport, and that the optional *service*, *network*, and *daemon* Rendezvous transport creation arguments can be specified, separated by slashes, as shown.

- **`service`** specifies the Rendezvous service number (UDP port) that will be used. If not specified, it will default to **`7889`**.
- **`network`** specifies the interface and the discovery group address that will be used to send the Rendezvous discovery packets. The format is:

  ```
  interface;discovery_group_address
  ```

  If not specified, ActiveSpaces uses the default interface and discovery group address **`239.8.8.9`** (so the URL will be equivalent to **`tibrv://7889/;239.8.8.9/`**). If an interface is specified (by IP address, hostname, or by interface name) do not forget to also specify a discovery group address otherwise Rendezvous will revert to using broadcast rather than discovery (for example, to specify usage of the interface identified by IP address **`192.168.1.1`** use the URL: **`tibrv:///192.168.1.1;239.8.8.9/`**).
- **`daemon`** specifies where to find the Rendezvous daemon. If not specified, it will try to connect to a local daemon on port **`7500`**.

- For more information on these parameters, see Chapter 8 of *TIBCO Rendezvous Concepts*.

TCP Discovery URL format

When multicast discovery is not desirable or possible, you can use pure TCP discovery. In this case, a number of metaspace members are designated as the “well known” members of the metaspace, and all metaspace members must specify this exact same list of well known members in their discovery URL. At least one of the members listed in the discovery URL must be up and running for the metaspace to exist.

Each well known member is identified by an IP address and a port number. This address and the port are those specified by the well known member's Listen URL (if the member did not specify a Listen URL then the discovery process will use it's default IP address and the first free TCP port it can acquire from the OS (starting at port 50000 and above). See the following section for more information on the Listen URL).

The discovery URL to use well known address TCP discovery has the following format:

```
tcp://ip1[:port1];ip2[:port2],...
```

Where any number of ip[:port] well-known addresses can be listed. If no port is specified, the default port number value of 50000 is assumed.

ALL of the metaspace members (including the well-known members themselves) must use the same Discovery URL string when TCP discovery is used.

At least one of the well-known members listed in the discovery URL must be up and running for the metaspace to exist; if none of the well known members listed in the discovery URL is up, other members regularly try to connect and print an advisory message stating that they are waiting for one of the discovery nodes to come up.

Discovery URL format for remote clients

Remote clients connect to the seeder that is running as-agent by calling the Java `setremoteDiscovery` method.

The discovery URL format is the same as the discovery URL for the listen attribute:

```
tcp://interface:port
```

Connecting as a remote client:

The discovery URL format for connecting to a metaspace as a remote client is

tcp://IP:port?remote=true

Where IP is the IP address and port is the TCP port number of a member of the metaspace that is connected as a full peer AND offering remote client connectivity (though the "remote listen" attribute of its metaspace connection).

For information on how remote clients connect to the seeder that is running as-agent, see Steps for Connecting a Remote Client, page 127.

Listen Attribute

Regardless of the mechanism used for the initial metaspace member discovery phase, the members of the metaspace always establish TCP connections to each other. The `listen` attribute lets the user specify the interface and the TCP port that the process will use to listen for incoming connections from new members to the metaspace, and specified in the form of a URL.

Listen URL format

To use a listen URL, use a string of the form:

```
tcp://[interface[:port]]
```

This syntax specifies that the member should bind to the specified interface and the specified port when creating the TCP socket that will be used for direct communication between the members of the metaspace. If not specified, it will default to **0.0.0.0** (`INADDR_ANY`) for the interface and to the first available port starting from port **5000** and above.

A successful connection to the metaspace will return a valid instance of a `Metaspace` object, which can then be used to define, join or leave spaces.

See the entry for **Metaspace** in the *TIBCO ActiveSpaces Java API Reference* for more information about transport arguments.

MemberName Attribute

The `MemberName` attribute specifies a string that indicates the member name. Each member must have a unique name in the metaspace.

If no member name is provided at connection name, then a globally unique name is generated automatically. If a member name is provided but there is already another member of the metaspace connected with that name, then the connection fails.

Disconnecting from the Metaspace

When your application terminates or no longer needs to use the spaces in a particular metaspace, it should disconnect from it using the metaspace's `close` method. The `close` method causes the application to properly leave all of the spaces to which it may still be joined in that metaspace, destroys any listeners or space browsers that may still exist on those spaces, and ultimately severs all network connections with the other members of the metaspace.

Each `getSpace()` function call increments the use count on an object. This use count ensures that a metaspace leave does not happen when there is a valid user space/browser/listener active. The user needs to leave/stop all these space/browsers/listeners to ensure that when `Metaspace.close` is used, metaspace disconnect happens as expected. This is the case both for Java and for C APIs.

You can forcibly close a metaspace connection even if there are still valid Space objects by using the `closeAll` method instead of the `close` method.

Metaspace Membership

The `Member` object is returned by the `Metaspace` object's `getSelfMember` method. The `Member` object has only two methods: `getName` and `getRole`. The `getName` method returns a string representing a globally unique name for that particular member. One of the members of the metaspace takes on the role of the membership manager for that member in the metaspace. The `getRole` method returns the role of that member in the metaspace (`MEMBER` or `MANAGER`) for metaspace group membership management purposes.

Getting the Connection’s Self Member Object

When connecting to a metaspace, each application is automatically assigned (or provides) a unique member name within that metaspace. Your application can determine this name by invoking the metaspace’s `getSelfMember` method, which returns a `Member` object.

Getting the List of User-defined Space Names

You can retrieve the list of names of the user spaces that are currently defined in the metaspace by using the metaspace's `getUserSpaceNames` method, which in Java returns a `String[]` and in C returns a `String`List object (`tibasStringList *`).

Configuring Logging

Using a set of methods in the Java API set, you can control file logging, and also set up rolling log files.

Rolling Log Files With rolling log files enabled, you specify the size of each log file and the number of log files to be maintained. When the current log files reach their maximum configured size, logging is rolled over to a new log file. The number of log files is also configurable.

In the C API, logging configuration is specified by two functions, one that specifies log file level, and another function that controls rolling log file configuration.

Java API

The `FileLogOptions` class in the Java API provides the following methods:

- **setFile** Specifies the log file to be used.

  ```
  public abstract FileLogOptions setFile(java.io.File logFile)
  ```

  The `logFile` parameter specifies the name of the log file.

- **setLimit** Specifies the file size limit for the log file (in bytes).

  ```
  public abstract FileLogOptions setLimit(int limit)
  ```

  The limit parameter sets the file size limit. The default value (-1) specifies unlimited) file size.

  The `count` parameter specifies the total number of log files.

- **setAppend** Specifies whether log fils can be appended to.

  ```
  public abstract FileLogOptions setAppend(boolean append)
  ```

  The `append` parameter specifies whether to append to existing files.

- **setLogLevel** Specifies the log level to be used.

  ```
  public abstract FileLogOptions setLogLevel(LogLevel level)
  ```

  The level parameter specifies the log level to be used. You can specify the following values:

  — **ERROR** Error level logging output of errors

  — **FATAL** Error level logging of fatal errors

  — **FINE** Outputs debug information

  — **FINER** Outputs debug information

  — **FINEST** Outputs detailed debug information,

  — **INFO** Outputs debug information. The default log level is `INFO`.

  — **NONE** Specifies no logging

  — **WARN** Warning level logging outputs warnings

Querying Log Settings

The Java API includes a set of methods that query the log file settings:

- **getFile** Returns the log file name.
- **getLimit** Returns the configured log file size.
- **getFileCount** Returns the configured number of log files.
- **isAppend** Indicates whether appending to log files is configured
- **getLogLevel** Returns the configured level of file logging.

C API

The C API provides two functions that control logging:

- **tibas_EnableFileLogging()** Specifies the log directory, the log file name, and the log level to be displayed.

  ```
  tibas_status tibas_EnableFileLogging(
      const char* logDir,
      const char* fileName,
      tibas_logLevel logLevel);
  ```

- **tibas_EnableFileLoggingEx()** Sets the values in the `tibasfileLogOptions` structure, which controls the values for rolling log file.

The tibasFileLogOptions structure is defined as follows:

```
struct _tibasFileLogOptions {
    const char* filePath;
    tibas_logLevel level;
    tibas_int limit;
    tibas_int fileCount;
    tibas_boolean append;

};
```

Defining a Space

You must first define space in the metaspace before it can be joined by applications and agents.

There are two ways to define a user space within a metaspace: through the Admin CLI tool, or by using API calls.

If the space definition does not exist (that is, it was not defined earlier using the Admin CLI tool, or by another application using `defineSpace`), the space definition is created and stored in the metaspace (more specifically, it is stored in the system spaces).

Space Definition Through the Admin CLI

To use the Admin CLI tool to define a space, you must first connect to the desired metaspace using the `connect` command, and then use the `define space` or `create space` command.

The following example shows the use of the `define space` CLI command:

```
define space name 'myspace' (field name 'key' type 'integer', field
name 'value' type 'string') key ('key')
```

Space Definition Through the API

By calling the SpaceDef functions or methods provided in the ActiveSpaces API you can specify the basic attributes and policies for a space.

Using the SpaceDef object In the Java API, defining a space is done using the `defineSpace` method of the `Metaspace` object, which takes a `SpaceDefinition` object as its sole parameter. In the C API, you define a space by calling the `tibasSpaceDef_Create()` function.

If the space was already defined in the metaspace, then `defineSpace` compares the space definition that was passed to it as an argument with the space definition currently stored in the metaspace; if the definitions match then `defineSpace` returns successfully, otherwise an error is thrown.

Using the admin object execute method A space can also be defined through the API by using the admin object's `execute` method to execute a **`define Space`** admin language command.

Example creating a space using the admin language:

```
define space name 'myspace' (field name 'key' type 'integer', field
name 'value' type 'string') key ('key')
```

Space names are case-sensitive.

Getting a Space Definition

You can get the space definition for a space that has been previously defined in the metaspace by using the `Metaspace` object's `getSpaceDef` method, which takes a space name as a parameters and returns either a copy of that space's `SpaceDef` object or throws an exception if no space of that name is currently defined in the metaspace.

Dropping a Space

You can delete a space's space definition from a metaspace by invoking the `dropSpace` method of the `Metaspace` object. This call will only succeed if there are no members to that space at the time this method is invoked. It is also possible to drop a space using the Admin tool.

Configuring Distribution Policy

You can set or get a space's distribution policy by using the `SpaceDef` object's `setDistributionPolicy` and `getDistributionPolicy` respectively. The value of the distribution policy argument can be either `DISTRIBUTED` (which is the default value) or `NON_DISTRIBUTED`.

Applications that require strict global view synchrony should use non-distributed spaces.

Defining Capacity

You can define a capacity for the space to control the amount of memory used by the seeders for storing tuples in the space. The capacity is expressed in number of tuples per seeder and defaults to `-1`, which means an infinite number of tuples per seeder.

If a capacity is specified, then you must specify an eviction policy that is used to indicate the outcome of an operation that would result in an additional tuple being seeded by a seeder that is already at capacity. The two choices for the eviction policy are `NONE`, which means that the operation will fail with the appropriate exception being stored in the Result object, or **`LRU`**, which means that the seeder will evict another tuple using the Least Recently Used (LRU) eviction algorithm, where the least recently read or modified tuple will be evicted from the space.

Specifying a capacity and an eviction policy of LRU for a space means that the space can effectively be used as a cache, and when used in conjunction with persistence, allows access to a persistent data-store in a “cache-through” mode of operation.

If you specify a capacity setting, then ActiveSpaces enforces the capacity limitation at one second intervals, and carries out any eviction policies that are configured.

Setting Up Host-Aware Replication

With host-aware replication, you group seeders based upon their member names. To organize seeders into groups, use member names of the form:

<group_name>.<member_name>

ActiveSpaces groups all seeders with the same *group_name* together and their data will is replicated on seeders outside of that group.

You can group any seeder in this way. You can set implement host-aware replication for ActiveSpaces applications run as seeders as well as as-agents that you start as seeders.

You can set up host aware replication in several ways:

1. By using the TIBCO ActiveSpaces API functions in your application to set up a MemberDef that specifies a member name using the host-aware replication naming convention.

   See Using the ActiveSpaces API Set to Implement Host-Aware Replication, page 92

2. By using the as-admin utility.
3. By starting as-agents that run as seeders and using the `as-agent` **`-name`** parameter to set up member names that use the host-aware replication naming convention.

   For more information on setting up host-aware replication using *as-agent*, see Host-Aware Replication on page 6 in the *TIBCO ActiveSpaces Administration Guide*.

Using the ActiveSpaces API Set to Implement Host-Aware Replication

The following examples show how to set the member name in the MemberDef object for each of the API sets:

Java API

MemberDef memberDef = MemberDef.Create();

memberDef.setMemberName = “mymachinename.seeder_n”;

C API

tibasMemberDef memberDef;

tibasMemberDef_Create(&memberDef);

tibasMemberDef_SetMemberName(memberDef, “mymachinename.seeder_n”);

.NET API

MemberDef memberDef = MemberDef.Create();

memberDef.MemberName = “mymachinename.seeder_n”;

If using the ActiveSpaces examples, the member name is specified on the command line using the **`-member_name`** command line parameter.

The type of replication for a space can be set or queried using the `SpaceDef` object’s `setSyncReplicated` and `isSyncReplicated` methods, respectively. Those methods take and return a boolean and the default value is false, that is, *asynchronous* replication.

Configuring EntryTTL, LockTTL, and LockWait

The entry TTL can be set or returned using the `SpaceDef` objects's `setTTL` and `getTTL` methods, respectively. The default value (`DEFAULT_ENTRY_TTL`) is `TTL_FOREVER`, which means that tuples in the space never expire. When a tuple expires from the space, an event of type `EXPIRE_EVENT` is automatically generated and can be caught by applications by using a listener or event browser on the space.

You can set or return the `LockTTL` value by using the `SpaceDef` objects's `setLockTTL` and `getLockTTL` methods, respectively. The value is expressed in milliseconds and defaults to `TTL_FOREVER`.

The `LockWait` value can be set or gotten using the `SpaceDef` object's `setLockWait` and `getLockWait` methods, respectively. It is expressed in milliseconds. The default value is `NO_WAIT`, indicating that this feature is not used, and that an operation attempting to modify a locked tuple will immediately return with a failure indicating that the tuple is currently locked.

Defining Data Fields

This section describes how to define data fields and indexes.

Field Definitions

Field definitions describe the format of the data that will be stored in the space. A valid space definition must contain at least one field definition. Field definitions are created by the `FieldDef`'s `create()` method, and can be put (or taken) from space definitions. Field definitions can also be reused and put into as many space definitions as needed (for example when using some fields as *foreign keys* to correlate tuples stored in different spaces).

A field definition is created by using the `FieldDef`'s `create()` method. A field definition has two mandatory attributes which are provided to the `create()` method: a *name* and a *type*.

The field name is a string and must start with a letter (upper or lower case) character and then contain any combination of letters (upper or lower case) and numbers, or special characters or symbols such as "-" or "_" or "$".

Note that field names are case-sensitive.

The field type must be one of those described in the following table.

Type	Description
`BLOB`	A `BLOB` (binary large object) type is an array of 8 bit values.
`BOOLEAN`	The `BOOLEAN` type has one of two values, true or false, represented by the integers `1` or `0`, respectively.
`CHAR`	Char type represents a char.
`DATETIME`	Datetime type represents a date. Two date formats are supported: • **Julian Calendar** Any date 64-bit time value • **Proleptic Gregorian Calendar** 1 A.D. to 64-bit time value.
`DOUBLE`	Double type represents double.

Type	Description
`FLOAT`	Float type represents float.
`INTEGER`	Integer type represents a int.
`LONG`	Long type represents a long.
`SHORT`	Short type represents a short.
`STRING`	String type represents string.

A field definition's name and type can be retrieved using the `FieldDef` object's `getName` and `getType` methods.

Beyond the field's name and type, a field definition also has the following optional boolean attribute:

- **Optional (nullable) field** You can use the `FieldDef` object's `setNullable` method to specify if the field can be null or not (defaults to false), which marks that field as optional, and use the `isNullable` to test if the field is optional or not. The equivalent functions in the C API are `tibasFiledDefSetNullable()` and `tibasFieldDefIsNullable()`.

Defining Key Fields

For the space definition to be valid, at least one of the defined fields must to be used as a *key field*. In the Java API, you can specify a set of fields to be used as key fields can be specified by using the SpaceDef's `setKey` method and passing it a number of strings containing the space names. (this is repeated in the concepts chapter)

In the C API, you can call the `tibasKeyDefCreate()` function to specify a single string containing a comma-separated list of field names to use as a key.

It is also possible to get a list of the name of the fields marked as key fields in a SpaceDef. In Java, by using the SpaceDef's `getKey` method. And in C, by using the `tibasKeyDefGetFieldNames()` function.

If all key fields are marked as nullable it then becomes possible to store a single tuple with no key field values in the Space.

Defining Key Fields for Distribution (Affinity)

When you define key fields for a tuple, ActiveSpaces allows you to define the key fields in a way that controls their distribution over seeders. This feature of ActiveSpaces is called “affinity.”

By specifying that certain key fields are distribution fields, your application can ensure that tuples that have the same value for a particular field or fields, are stored on the same seeder.

The ActiveSpaces API provides functions and methods for each API set to enable distribution based on affinity:

- **Java** `setDistributionFields()`
- **C** `tibasSpaceDef_SetDistributionFields()`
- **.NET** `SetDistributionFields`

Each function or method takes as its arguments the space definition for the space that is to be defined and a list of fields within quotation marks, separated by commas.

For more information on the C API function, see the reference article for the `tibasspaceDef_SetDistributionFields()` function in chapter 5 of the TIBCO ActiveSpaces C Reference, “SpaceDef.”

For more information on the Java method, see the JavaDoc entry for `setDistributionFields().`

Each API set also provides a function to retrieve the distribution key setting for a specified space. For example, the Java API provides the `getDistributionFields()` method.

Admin CLI Support for Distribution Fields

When you define a space using the **`define | create space`** command in the ActiveSpaces Admin CLI, you can specify the `distribution_def` parameter to set specified keys for distribution fields. And when you issue the **`show spaces`** command, the command output indicates any fields that are set up as distribution fields.

For more information, see the reference article for **`define | create spac`**e in chapter 2 of the *TIBCO ActiveSpaces Administration Guide*, “Administering ActiveSpaces with the Admin CLI.”

SpaceWait

No operations on a space are possible unless the space is in the `READY` state. To help synchronize applications with the space state, each space has a `SpaceWait` attribute. This attribute is the number of milliseconds a space operation will block for and wait for the space to reach the `READY` state if it is not in that state at the time the operation is invoked.

Adding Fields to a Previously Defined Space

ActiveSpaces allows you to alter the fields in a space that is already defined by using the `as-admin` utility or by calling `Metaspace.alterspace()`. The C API and the .NET API provide equivalent operations. There is no disruption in service when you alter the space.

Any new fields that you add must be nullable.

If the space has not yet been defined or the space definition is incompatible with the one that is defined (for example, has new fields that are not nullable), ActiveSpaces generates an exception describing what was incorrect.

Adding and Dropping Indexes

ActiveSpaces allows you to add indexes to a space that is already defined or drop indexes from the space. You can add or drop indexes by using the `as-admin` utility or by calling `Metaspace.alterspace()`. The C API and the .NET API provide equivalent operations. There is no disruption in service. ActiveSpaces builds the new indexes in the background, and when they are ready, sues them automatically to optimize queries.

You cannot modify an existing index's fields or index type. If you want to modify the fields or index type, you must first drop the index, then alter the space, add the index and alter it again. Also, when you add a new index, you cannot use fields from an existing index.

Adding and dropping an index using the Java API

The following example shows how to add and drop an index using the Java API.

```
SpaceDef spaceDef = metaspace.getSpaceDef(“test”);
    spacedef.removeIndexDef(“index1”);
    spaceDef.addIndexDef(…)
    spaceDef.addIndexDef(…)
    spaceDef.putFieldDef(FieldDef….)
metaspace.alterSpace(spaceDef);
```

Joining and Leaving a Space

Joining a Space

After a space has been defined in the metaspace, applications can join the space—and as-agents started in the network automatically join the space if the space is distributed.

With the Java API, you join a space by invoking the `getSpace` method in the `Metaspace` class.

The getSpace method has the following signature:

```
public abstract Space getSpace(java.lang.String spaceName,
             Member.DistributionRole distributionRole)
                         throws ASException
```

The `getSpace` method has the following parameters:

- **spaceName** Must be the name of a space defined in the metaspace
- **DistributionRole** Specifies the role for the application. Can be SEEDER or LEECH.

In Java, there is a second signature for the `getSpace` method which does not take a role argument, and can be used to join a space with a distribution role of leech.

In the C API, the equivalent function is `tibasMetaspace_GetSpace()`.

Spaces are also automatically joined (with a role of leech) as needed when creating a listener or a browser on the space from the metaspace.

A successful invocation of the metaspace's `getSpace` method returns a reference to a `Space` object that can then be used to interact with the space.

No space operation—other than getting the space's name and its definition, or creating a listener or browser on the space—will succeed until the space is in the READY state.

Note also that the space object is reference-counted, meaning that successive calls to `getSpace` do not result in the space being joined more than once. However. the process' role for the space can be adjusted by multiple calls to `getSpace`: a first call may cause the process to join the Space as a Leech, and a following call may change the role to Seeder (and the role will revert back to Leech when leave is called on the space object reference returned by the second `getSpace` call). An application's role for a space can also be adjusted using the Space's `setDistributionRole` method.

Leaving a Space

Your application can leave a space by invoking the space object's `close()` method (Java API), by calling the `tibasSpace_Free`() function (C API), or by stopping the listeners or browsers that may have joined the space automatically when created.

Because the space object is reference-counted, when created from a metaspace object, the space will actually be left by the process only when the space object's `close()` method is invoked an equal number of times to the metaspace's `getSpace` method for that particular space. Just as the application's distribution role in the space can be adjusted by subsequent calls to `getSpace` with different roles, the application's distribution role in the space is adjusted when `close()` is invoked on the instances of the Space object returned by each call to `getSpace`.

If browsers or listeners were created and not terminated when `close()` was invoked, the process does not leave the space until the browsers or listeners are terminated. You can forcefully leave the space and terminate any existing browsers and listeners on the space by invoking the `closeAll()` method.

Setting up Persistence

When you set up persistence for a space, you specify:

- **Persistence Type:** You can set persistence to NONE (no persistence, shared all persistence, or shared-nothing persistence).
- **Persistence Policy:** You can set up asynchronous persistence or synchronous persistence.

Persistence Type

ActiveSpaces provides two types of persistence:

- **Shared-Nothing Persistence** Each node that joins a space as a seeder maintains a copy of the space data on disk. Each node that joins as a seeder writes its data to disk and reads the data when needed for recovery and for cache misses
- **Shared-All Persistence** All nodes share a single persister or a set of persisters.

Persistence Policy

For both shared-nothing persistence and shared-all persistence, you can specify that ActiveSpaces uses either synchronous or asynchronous communication to maintain persistence.

API Operations for Setting up Persistence

You can set up persistence for the space using the following API operations:

- **C API** By using the `tibasSpaceDef_SetPersistenceType()` function and the `tibasSpaceDef_SetPersistencePolicy()` function.
- **Java API** By using the `PersistenceType` and `PersistencePolicy` methods of the **SpaceDef** class.
- **.NET API** By using the `PersistenceType` and `PersistencePolicy` methods of the SpaceDef class

For descriptions of the ActiveSpaces example programs used to set up persistence, see ASPersistence, page 185 and Shared-Nothing Persistence, page 189.

Using the tibasSpaceDef_SetPersistenceType() Function

The ActiveSpaces C API provides the `tibasSpaceDef_SetPersistenceType()` Function. When you call this function, you can specify:

- **TIBAS_PERSISTENCE_NONE** Do not persist objects in the space.
- **TIBAS_PERSISTENCE_SHARED_ALL** Use shared-all persistence
- **TIBAS_PERSISTENCE_SHARED_NOTHING** Use shared-nothing persistence.

Using the tibasSpaceDef_SetPersistencePolicy() Function

The C API provides the `tibasSpaceDef_SetPersistenceType()` function specifies the persistence policy to use on the space. You can specify `PERSISTENCE_SYNC` or `PERSISTENCE_ASYNC`.

Using the PersistenceType and PersistencePolicy Methods

In the Java API and the .NET API, the SpaceDef class provides the following methods:

- **PersistenceType Method** Lets you set persistence to NONE, SHARE_ALL, or SHARE_NOTHING.
- **PersistencePolicy Method** Lets you set the persistence policy to ASYNC or SYNC.

Setting up Shared-Nothing Persistence

To set up and configure shared-nothing persistence:

1. Use a unique name for each member that is joining the space. This is used to uniquely identify the persistence files.
2. Define a data store for each node that is joining as a seeder.

   The data store must be an existing directory name (with full path) that ActiveSpaces has permissions to read and write to.

You can do this by:

— Using the MemberDef object and calling the `tibasMemberDef_SetDataStore()` function.

— Setting the `AS_DATA_STORE` environment variable to the directory required before starting the client program or `as-agent`.

— With the as-agent, using the `-data_store <directory name with path>` CLI command.

The data store directory can be different or same for each node (seeder). If it is the same, ActiveSpaces creates the required subdirectory structure based on the metaspace name, space name, and member name. Each file has the member name and a timestamp embedded in its name.

Do not rename the files that are created by the persistence operation.

For code examples showing how to set up shared-nothing persistence, refer to `ASPersistence2.c` in the `/examples` directory.

Setting up Shared All Persistence

Because ActiveSpaces is a true peer-to-peer distributed system, it provides API operations that let you enable shared all persistence. With shared all persistence, some of the space members — the *persisters* — are responsible for interacting with a persistence layer, just as some of the space members — the *seeders* — provide the basic space service.

Applications can register their ability to provide the persistence service on a space by invoking the `setPersister` method of the Space object. In Java, this method takes an instance of a class that implements the Persister interface. In C it takes a `tibasPersister` object, which itself can be created using the `tibasPersister_Create` function where pointers to the functions required by the persister interface are provided by the user. It is therefore necessary for the application to first have joined the space (as a seeder or as a leech) before it can register itself as a persister.

Applications can also indicate their desire to stop providing the persistence service by invoking the space's `stopPersister` method in Java and `tibasPersister_Free` in C.

Interaction with the persistence layer is implemented by classes (or sets of functions in C) that implement the `Persister` interface. It is up to the user to provide implementations of this interface to perform persistence to their persistent storage of choice (for example a database, or a key-value store, or a file system).

Applications able to provide the persistence service register an instance of a class implementing the `Persister` interface for a space using the space object's `setPersister` method, and indicate their willingness to stop providing the persistence service for a space using the space object's `stopPersister` method.

The `Persister` interface consists of five methods:

- **onOpen** Invoked when the persister object is registered with the space
- **onClose** Invoked when the persister object is stopped for the space
- **onWrite** Invoked when a tuple stored in the space is modified (due to a put, take, or update operation) and is intended to perform the steps necessary to reflect the change in the space onto the persistence layer.
- **onRead** Invoked if the space has a capacity set and a capacity policy of `EVICT`, and if a request to read, lock, or take a tuple by key value did not result in a matching tuple being found in the space.
- **onLoad** The onLoad callback is made to the first persister instance.

  It is invoked as soon as the space has reached the minimum number of seeders. If the space has a capacity set and a capacity policy of `EVICT` it is not required to do anything but can still be used to pre-load the caching space with some of the tuples stored in the persistence layer.

Setting up Recovery with Persistence

You can set up recovery with persistence by:

- Using the API Operations
- Using the Admin CLI

Setting up Recovery Using the API Operations

The C API provides the following recover API settings, which are specified using the metaspace object, space name, and recovery options and passing a struct:

```
struct _tibasRecoveryOptions {
    tibas_boolean recoverWithData;
};
```

The default value for `recoverWithData` (TIBAS_TRUE) specifies recovery with data; if you specify TIBAS_FALSE, recovery without data is set.

You can also use `tibasMetaspace_RecoverSpaceEx(metaspace, spaceName, recoveryOptions)`.

The Java and .NET APIs provide similar APIs.

Setting up Recovery Using the CLI

To set up recovery using the Admin CLI:

Connect to the metaspace and enter recovery commands as shown in the following example:

```
as-admin> recover space "myspace" with data

Space myspace recovery started

as-admin> show space "myspace"
```

Using Tuple Methods

The simplest way to look at a tuple is as a map of fields, that is, an object in which you can *put*, *get*, and *remove* fields. A tuple can also be seen as a self-contained, self-describing set of fields. Tuples are self-describing in that you can get a list of the names of all the fields contained in a tuple (although not in any specific order). A tuple does not have any reference to any metaspace or space in particular. A copy of the tuple is made inside an tuple during a space put operation.

It is not necessary to be connected to any metaspace or joined to a space in order to create and use a tuple. You can, for example, extract a tuple out of the tuple that was returned by a `get` from one space, modify it by changing field values (or adding or removing fields), and then to `put` that tuple into another space (as long as the fields in the tuple are compatible with that space's field definitions). And you can `put` a tuple into a space and then re-use the same tuple object by updating one of the field values and using it to do another `put` in the same space.

A tuple contains fields that are identified by names, each of which is unique within the tuple. A field name must start with a letter (upper or lowercase) or the underscore character (_), followed by any combination of letters (upper or lower case) and numbers. Field names are case-sensitive.

Fields also have a value and a type (see Field Definitions on page 95 for the list of field types supported by ActiveSpaces). The tuple object has a `put` method for each one of the supported field types.

Examples

```
tuple.putString(“Name”,”John Doe”)
tuple.putInt(“Age”,40)
```

The Java API also has an overloaded generic `put` method that uses the type of the value being passed to establish the type of the field, for example:

```
tuple.put(“Name”,”John Doe”)
tuple.put(“Age”,40)
```

The tuple object also has a set of `get` methods used to get fields of the appropriate type. For example, `tuple.getString(“Name”)` will return a string.

Tuple fields

In Java, getting a field of a scalar type will return an object rather than the primitive type for the field. For example, `tuple.getInt(“Age”)` will return an object of type Integer (rather than an int).

This is because if there is no field by the name requested in the `get` method in the tuple, a null is returned, rather than an exception being thrown. It is therefore important, especially when using autoboxing, to always check if the `get` method returned a null when dealing with tuples that may or may not contain the requested field, for example, when a space is defined with some `Nullable` (optional) fields. Trying to get a field that exists in the tuple but is of a different type than expected will, however, throw a runtime exception. Because objects are returned for all field types, the Java API also has a generic `get` method that returns an object of corresponding type for the requested field, or null if there is no such field in the tuple.

When using the C API, which unlike the Java methods, returns a status code—a `tibasTuple_Get`… function call will return `NULL` if there is no field by the requested name, and will return `TIBAS_INVALID_TYPE` if a field of that name exists in the tuple but is of a different type than what is being requested.

Automatic lossless type upcasting is however supported when getting fields from a tuple. This means that, for example, if the tuple contains a field named '**`A`**' of type `short`, doing a **`getLong("A")`** will succeed and return a `long` that is set to the value of the field. The supported automatic type conversions are shown in Table 7.

Table 7 Automatic Type Conversions

Upcasting Supported to...	Type of Field in Tuple
Short	Short, Int, Long, Float, Double
Int	Short, Int, Long, Float, Double
Long	Short, Int, Long, Float, Double
Float	Short, Int, Long, Float, Double
Double	Short, Int, Long, Float, Double
Blob	Blob, String

You can test for the presence of a named field inside a particular tuple by using either the `exists` method or the `isNull` method, which both return a boolean indicating the presence or absence of the field in the tuple.

You can also find out the type of a particular field in the tuple using the `getFieldType` method, which can return a field type value of `NULL` (`TIBAS_NULL` in C).

The choice of which method to use is a matter of preference. Programmers more familiar with messaging systems may prefer to test for the *existence* of a field, while programmers more familiar with database systems may prefer to test whether a field *is NULL*. In ActiveSpaces, a *NULL field* is actually a non-existing field.

A tuple can also be seen as a self-describing data structure in that it offers methods that can be used to inspect the fields contained in the tuple:

- The `size` method returns the number of fields in the tuple
- The `getFieldNames` method returns a `String[]` in Java, while in C it returns a pointer to a `tibasStringList` object.

You can serialize or deserialize a tuple into a platform-independent stream of bytes using the `serialize` and `deserialize` methods.

And finally a convenience `toString` method can also be used to generate an XML representation of the tuple into a string.

Getting the Name and Definition of a Space

You can get the name and the space definition of the space represented by the `Space` object by invoking the `getName` and `getSpaceDef` methods respectively.

Reading and Writing in a Space

This section describes how to read data from a space and write data to a space.

Getting or Taking a Single Tuple from the Space

Retrieving or taking a single tuple from a space is done using the space's `get` or `take` methods. These methods take a tuple as an argument, which must contain fields of the same name and type as the fields marked as key in the space's *space definition*. If there is a tuple in the space containing a tuple whose key field values match exactly the values in the tuple passed as argument stored in the space, these methods return a copy of that tuple inside a Result object. (In the case of the `take` method, it also atomically removes that tuple from the space.)

Performing a Put Operation—Storing a Tuple in a Space

To store a tuple in a space, call the space's `put` method, which takes the tuple to put as an argument and returns what ever was overwritten in the space (if anything was) by the put.

The put operation is an "upsert" operation: like both the INSERT and UPDATE SQL statement (like a MERGE statement); it always overwrites what was stored previously in the space. When you perform a Put:

- If the tuple contains fields that do not match the names of the fields in the space's definition, then these fields are stripped from the tuple stored in the space
- If there is already a tuple containing a field with the same key field values stored the space at the time the `put` is invoked, then the old tuple is replaced by the new one. If this behavior is not desired, and you want to avoid overwriting an existing tuple by mistake, then you should use a special form of the space's update method.
- When the time the put is invoked, if there is already a matching tuple in the space and that tuple is locked, the method might block for the amount of time specified in the space definition's `LockWait` attribute. After LockWait time is reached, if the stored tuple is still locked, the Put fails, and the failure scenario will be indicated by the Result object's Status field (or the return value of the method in C) having a value of `LOCKED`.

Specifying a LockWait Value for a Put

If you expect that the locks on tuples in the spaces will have a very short duration, ActiveSpaces allows you to specify a `LockWait` value for the space. The `LockWait` value is the number of milliseconds an operation attempting to modify a locked tuple can block while waiting for the lock to clear.

If at the end of the `LockWait` period, the tuple is still locked, then the operation to modify that locked tuple throws an exception indicating that the operation could not be performed because the tuple is locked. The `LockWait` value of a space is also taken into consideration if a member attempts a non-transacted operation that conflicts with uncommitted data (for example, tuples about to be replaced by uncommitted operations are locked by that transaction).

You can set or get the `LockWait` value by using the `SpaceDef` object's `setLockWait` and `getLockWait` methods, respectively. Lock wait value is expressed in milliseconds. The default value is `NO_WAIT`, indicating that this feature is not used, and that an operation attempting to modify a locked tuple will immediately return with a failure indicating that the tuple is currently locked.

Updating a Tuple in a Space

Put operations always overwrite (and return) whatever was stored before them in the space. But when more than one application updates the same data concurrently, it can be necessary to use "compare and set" type operations.

ActiveSpaces has two compare and set operations:

- **compareAndPut** This is a conditional put operation to atomically change the values associated with a key from one value to another (or fail). It takes an old tuple, and a new tuple as inputs, and returns the current tuple (meaning the new tuple if the compare was a success and the current value stored in the space otherwise). Note that you can pass NULL as an old tuple if you want to make sure your put is an insert rather than an overwrite.
- **compareAndTake** This is a conditional take operation that succeeds only if the tuple that was stored in the space matches the old tuple value it takes as input.

For an example of code that works with tuples, see the documentation on the ASOperations example (ASOperations, page 173).

Locking and Unlocking

You can make a key in a space read-only by locking it. When a key is locked, only the lock owner has the ability to modify the data that is stored with that key. Locking is enforced regardless of the presence or not of a record for the key being locked.

The owner of the lock is either the thread that created it, or the process. You can define this lock ownership scope when acquiring the lock. Locks are automatically cleared if the member that created them leaves the metaspace.

Locking protects the data stored in the space, you can even lock an empty key to prevent anyone else from inserting a record there. Any operation to modify the key (put, take, and so on) will be denied if it is issued outside of the lock's scope. It is also possible to make those operations (not just other attempts to lock) block for a period of time using the space's LockWait attribute.

The lock function automatically returns the tuple that was stored at the key (or null is there was none) upon lock acquisition. If the lock could not be acquired then an exception is thrown (or LOCKED is returned in the status in C).

You can optimize performance by combining locking and unlocking with many space operations using options.

Using Transactions

ActiveSpaces supports transactions to keep data consistent, even in cases of system failure. This section describes how transactions work in ActiveSpaces.

Creating and Committing or Rolling Back Transactions

Since in ActiveSpaces transactions can span operations on more than one space, users need to specify the start and the end of a transaction using the metaspace's `beginTransaction`, `commit`, or `rollback` methods. A transaction is associated with the thread that created it: at the end of the transaction all of the space operations invoked by that thread will be either committed (if none of those space operations failed) or rolled back (if one of those operations failed), according to the method invoked. In other words, the *prepare* phase of the transaction is the invocation of operations on a space by a thread that has invoked `beginTransaction` first.

Obviously, only space operations that modify the data stored in a space are affected by the outcome of the transaction, but a `get` operation within a transaction will always return the uncommitted version of the tuples that may have been modified in that transaction.

Space Operation Options

Many space operations can be combined with various options. Options can be used to optimize performance of the space operations, by either combining two operations together or indicating whether the return value is unwanted.

Most operations for example can be combined with a lock or an unlock options. For example to simultaneously update the value and unlock a record you can pass the unlock option to your put operation. And if you do not care about the return of the put operation, you can specify the forget option.

Using Batch Operations

Because in most installations, ActiveSpaces is deployed with seeders on many machines on the same network, some of the operations might need a complete network round trip to complete. Therefore, large improvements in throughput can be achieved by parallelizing operations using the *batch* versions of the space operations whenever possible.

For example it is always faster to use the `putAll` method than to do a series of repeated individual puts in a loop. This throughput improvement is due to the fact that the individual operations of the batch are executed asynchronously and are therefore parallelized, providing improvement in overall throughput for the application. Batch versions of the space methods are named adding the 'All' suffix to the method's name (e.g., `putAll`) and return a `ResultList` object.

The `ResultList` contains a multiple ways to get the individual `Result` objects for each of the operations contained in the batch, as well as convenience methods such as `hasException()` which is true if any of the operations failed, or methods to get lists of `SpaceEntries` for operations that returned `OK` rather than `NOT_FOUND`.

Using Listeners

Listeners are used to monitor events that represent changes to the tuples stored in a space. Listeners are callback-driven, unlike space `EventBrowsers`, where the user decides when to get the next event by invoking the `EventBrowser`'s `next` method. This means that a method of a callback class (or a callback function in C) provided by the user will be automatically invoked by the ActiveSpaces library code whenever a change happens to one of the tuples in the space being monitored.

In Java, to use a listener, users provide a class to the listen method that implements one or more of the listeners interfaces.

- The `PutListener` interface requires a `onPut(PutEvent event)` method.
- The `TakeListener` interface requires a `onTake(TakeEvent event)` method.
- The `ExpireListener` interface requires a `onExpire(ExpireEvent event)` method.
- The `SeedListener` interface requires a `onSeed(SeedEvent event)` method.
- The `UnseedListener` interface requires a `onUnseed(UnseedEvent event)` method.

In C, users provide a callback function to the `tibasListener_Create` function or one of the additional listener creation functions that will be invoked for all event types. This user callback function will be passed a single tibas_spaceEvent object that they can pass to the `tibasSpaceEvent_GetType` function to determine the type of the event.

- SpaceEvents of type `TIBAS_EVENT_PUT` are generated whenever a tuple is inserted, overwritten or updated.
- SpaceEvents of type `TIBAS_EVENT_TAKE` are generated whenever a tuple is taken or removed.
- SpaceEvents of type `TIBAS_EVENT_EXPIRE` are generated whenever a tuple reaches the end of its time to live and expires from the space.
- SpaceEvents of type `TIBAS_EVENT_SEED` are generated whenever there is redistribution after a seeder leaves the space and the local node is now seeding additional tuples. This is only applicable if the listener distribution scope is `SEEDED`.
- SpaceEvents of type `TIBAS_EVENT_UNSEED` are generated whenever there is redistribution after a seeder joins the space and the local node is no longer seeding some of the tuples. Only applicable if the listener distribution scope is `SEEDED`.

You can optionally specify a filter string when creating a listener. A filtered listener will only return events that match the specified filter. This is done by adding a filter argument when you invoke the `listen()` method on the metaspace. The filter argument contains a string in the language described in Filters on page 56. Note that some of the filtering may be executed in a distributed manner by ActiveSpaces in order to optimize performance.

In addition to the basic `tibasListenerCreate()` function, the C API provides functions that allow you to create additional types of listener:

- **tibasSpaceMemberListener_Create()**—Creates a space member listener.
- **tibasMemberListener_Create()**—Creates a member event listener.
- **tibasSpaceRemoteMemberListener_Create()**—Creates a remote space member listener.
- **tibasRemoteMemberListener_Create()**—Creates a remote member listener.
- **tibasSpaceStateListener_Create()**—Creates a space state listener.
- **tibasSpaceDefListener_Create()**—Creates a space definition listener.

A listener can have either *time scope* or *distribution scope*, defined by setting the values of fields in the listener's `ListenerDef` object:

Time scope The time scope can be used to narrow down the period of time of interest.

- *snapshot* means that the listener contains only `PUT` events corresponding to the tuples stored in the space at creation time.
- *new* means that the listener starts empty and is updated only with events related to new or overridden tuples in the space.
- *new events* means that the listener starts empty, and is updated with all events that occur in the space after creation time. Unlike the time scope *new* described above, this time scope includes events (such as `TAKE` or `EXPIRE` events) related to tuples already contained in the space at creation time.
- *all* means that the listener starts with all the tuples currently in the space at creation time (which will be presented as an initial set of `PUT` events) and then is continuously updated according to changes in the space.

Distribution scope The distribution scope can be used to narrow down the set of tuples or events being browsed.

- *all* is used to listen to events related to all tuples in the space.
- *seeded* is used to listen only to events associated with the tuples in the space that are seeded by this member. It will be empty unless the member is a seeder on the space.

When the listener's distribution scope is set to *seeded*, two additional types of events may be generated:

- **SEED** Generated when the member that created the listener starts seeding an tuple.
- **UNSEED** Generated when the member that created the listener no longer seeds an tuple.

For a description of the ASListener example program, which illustrates how to set up a listener, see ASListener, page 203.

Using SpaceEvent Objects

`SpaceEvent` objects are either passed as the argument to methods of a class implementing the `SpaceListener` interface or returned by an `EventBrowser`'s `next` method. `SpaceEvent` objects implement the following methods:

- **getSpaceTuple** Returns the tuple associated with the space event (that is, the tuple that was put or updated, the tuple that was taken or removed, the tuple that expired, or the tuple that is now seeded or is no longer seeded).
- **getSpace** Returns the space object representing the space on which this event was generated (this is a convenience function so that the users can then easily interact with the space in reaction to the event).
- **getType** Returns the type of the event (useful when using an event browser) which can be either `EVENT_PUT`, `EVENT_TAKE`, or `EXPIRE_EVENT`. If the distribution scope of the listener or browser is `SEEDED`, the type can also be `EVENT_SEED` or `EVENT_UNSEED`.

Implementing a Space Browser: Querying the Space

Although you can use a listener to query the data in a space, in practice, you will almost always use space browsers: a browser is used to iterate through the contents of a space.

You create a browser by calling the browse method of a Space or Metaspace object. Parameters of this method include the type of browser and a BrowserDef object, which contains configuration settings for the browser.

A browser has two only methods: next() and stop(). What the next() method does, and what it returns, depends on the type of the browser. There are three types of browser:

- **GET** A browser of type GET's next method gets and returns the next unread tuple.
- **TAKE** A browser of type TAKE's next method takes and returns the next unread and unlocked tuple.
- **LOCK** A browser of type LOCK's next method locks and returns the next unread and unlocked tuple.

To run queries on the space, you can specify a filter string when creating the browser: a filtered browser will only return tuples that match the specified filter. The browser's filtering criteria is expressed as a string in the language described in the section on filters: see Filters on page 56. Note that some of the filtering may be executed in a distributed manner by ActiveSpaces in order to optimize performance.

A tuple browser can have either *time scope* or *distribution scope*, defined by setting the values of fields in the browser's `BrowserDef` object:

Time scope The time scope can be used to narrow down the period of time of interest.

- *snapshot* means that the browser starts with all the tuples in the space at the time the browser is created (or *initial values*), but is not updated with new tuples that are put into the space after that moment.
- *new* means that the browser starts empty, and is updated only with tuples put into the space after the moment of the browser's creation.
- *all* means that the browser starts with all the tuples in the space, and is continuously updated with new tuples.

Distribution scope The distribution scope can be used to narrow down the set of tuples or events being browsed.

- *all* is used to browse over all the tuples (or associated events) in the space

- *seeded* is used to browse only over the tuples (or associated events) actually distributed to the member creating the browser

The `BrowserDef` object for a tuple browser can include a *timeout* value. The timeout is the amount of time for which a tuple browser's `next()` method can block while waiting for something new in the space to '`next()`' on. If there is still nothing new for the browser to'`next()`' on at the end of the timeout, the `next()` method will return `null`.

The browser's timeout value is ignored when the time scope is set to **`snapshot`**. In this case any invocation of the `next` method on the browser once all of the tuples in the snapshot have been iterated through will return **`null`** right away.

Browser prefetch A query in ActiveSpaces consists of two steps:

1. The space's seeders filter the data according to the criteria of the filter.
2. The matching tuples must be retrieved from those seeders to the application issuing the query.

By default, the matching records will be mostly retrieved on demand, one by one, each time `next()` is called on the browser. Due to network latency there is a minimum amount of time potentially required by a `next()` operation, and it is therefore much more efficient in many cases (for example, when the resulting dataset is large) to pre-fetch some of the tuples in parallel with the processing of the result tuples.

The prefetch is expressed in number of records per seeder that are prefetched in parallel with the browser's thread. A special value of ALL (-1) indicates that the whole result set should be pushed at once to the client.

If any form of fault-tolerance is desired, prefetch should not be used on TAKE browsers.

When the ALL and NEW timescopes are used ActiveSpaces automatically maintains the coherency of the prefetched tuples with the space.

The browser's next() call is thread-safe: you can have multiple processing threads call next() concurrently on the same browser safely.

For a description of the `ASBrowser` sample program, which shows how to implement a browser, see ASBrowser, page 199.

Using Event Browsers

An event browser is the iterative version of a listener's `next()` method and returns the next event (the same kind of object you would get in a listener) in the space's event stream. Just like any other browser or listener, it can take a filter and has the same time and distribution scopes. Please see the documentation about Listeners to learn more about events.

For a description of the `ASEventBrowser` sample program, which shows how to implement an event browser, see ASEventBrowser, page 201.

Enabling Performance Monitoring

To enable the performance monitor from your application, call the function or method provided with the API set that you are using. The performance monitor activates several commands in the as-admin utility that allow you to monitor system performance:

```
show member stats
show system stats
```

By default, this feature is not turned on.

C API

With the C API, to enable performance monitoring, call the `tibas_EnablePerformanceMonitor()` function with the argument specified as TIBAS_TRUE:

```
tibas_EnablePerformanceMonitor(TIBAS_TRUE)
```

Java API

With the Java API, to enable performance monitoring, invoke `the AsCommon.setEnablePerformanceMonitor` method, as follows:

```
ASCommon.setEnablePerformanceMonitor(true)
```

.NET API

With the .NET API, to enable performance monitoring call the `AsCommon.EnablePerformanceMonitor` method as follows:

```
ASCommon.EnablePerformanceMonitor = true
```

Using Remote Space Invocation

With TIBCO ActiveSpaces 2.0.0, the Java API and the .NET API allow applications to remotely invoke code execution over the space to other members.

For a method in a class to be able to be invoked remotely, it needs to fulfill two criteria:

- The method needs to implement one of the two invocable interfaces: `Invocable` or `MemberInvocable`
- The class needs to be present in all of the members' `CLASSPATHs` (or preregistered for C applications)

The remote invocation can then be triggered by any space member using either a single member invocation calls `invoke`, which will invoke the method on the node seeding the key passed as an argument); or `invokeMember`, which will invoke the method on the member listed in the argument.

In addition, a parallel distributed invocation of the method on multiple space members can be triggered using the calls `invokeMembers` (which invokes it on all of the space's members, regardless of role), and `invokeSeeders` (which invokes it on all of the space's seeders).

You can very easily create distributed processing of data stored in ActiveSpaces leveraging in-process "data locality" by invoking a class on all the seeders of a space where the invoked class creates browsers of distribution scope "seeded." This means that each seeding process can iterate at very high speed (in-process latency, memory bus bandwidth) through it's subset of the data, like a "map" in a map/reduce processing architecture, and either store the updated tuples back in the space (still a local operation) or return the results back to the invoker in the result tuple (or store them in another space) where they can be 'reduced'.

Remote invocation in ActiveSpaces is completely cross-platform. The class name is what is used to identify what is going to be invoked. This means that you could have applications in one platform invoke classes implemented in another platform. You could even have a mix and match of platforms implementing the same class on a space, as long as the class names match (and of course as long as they expect the same kind of tuples as context and return).

Using a Space as a Cache

To use a space as a cache, you should configure it with the following attributes:

- **Capacity** Assign a capacity (the maximum number of tuples per seeder that can be stored in the space).
- **Eviction Policy** Set up an eviction policy; for example, set up a Least Recently Used (LRU) policy to determine what happens when the space is full and a request to insert a new tuple is made. In that case, and in that case only, the space will evict one of the tuples to make room for the new one. By default, spaces do not have a capacity, meaning that the capacity is unlimited. Also, by default, spaces have no eviction policy, meaning that if the space has a capacity, an attempt to insert an extra tuple will fail due to the capacity being reached.

Working with Remote Clients

If you have purchased the TIBCO ActiveSpaces Remote Client in addition to the Enterprise Edition, then you can implement applications that run on remote clients and which communicate with a space running on the core ActiveSpaces cluster.

Java applications that connect remotely to the metaspace need to use the MemberDef's `setRemoteDiscovery` call to specify the list of well known addresses to connect to.

C applications need to set the Discovery URL attribute in the memberDef object to specify the remote listen URL that was specified when the as-agent was started on a seeder device in the core cluster.

Remotely connected client applications have access to the same features as any other applications, however they can never be seeders on a space. If an application remotely connected asks to get a space as a seeder it will get a handle on the space but will remain a leech. The SEEDED distribution scope is the scope of the proxy client they are connected to.

Also, while remotely connected clients can invoke methods on the space members, on can not invoke methods remotely on a remote client application.

Steps for Connecting a Remote Client

The basics steps for connecting a remote client are:

1. On the device that will act as the full peer proxy to the remote clients, run the as-agent program, and specify the discovery URL, the listen URL, and the remote listen URL.

   The remote listen URL specifies on which IP address and TCP port this proxy metaspace member will be listening for remote client connections.
2. From an application running on the remote client, call the member functions to discover the seeder and establish communication.

Starting as-agent to Listen for a Remote Client

Start an `as-agent` with remote_listen parameter that points to a URL and specific port.

Enter the **as-agent** command as follows:

```
as-agent -metaspace <“metaspace_name”> -discovery
<“discovery_url"> -listen <“listen_url”> -remote_listen
<“remote_listen_url”>
```

For example, assuming that the IP address of the machine where you will run as-agent to set up a seeder is 10.98.200.194, enter:

```
as-agent -metaspace “agent007” -discovery
“tcp://10.98.200.194:5000” -listen “tcp://10.98.200.194:5000”
-remote_listen “tcp://10.98.200.194:5001”
```

This opens up a port on 5001 for the remote client program to communicate with a member in the metaspace, in this case the as-agent.

Connecting to the Seeder from the Remote Client

After the `as-agent` is running on the seeder, remote clients can connect to a metaspace and a space on the proxy machine.

For an application to connect to a metaspace as a remote client, when connecting to the metaspace, all it needs to do is pass a Discovery URL of the format:

tcp://ip:port?remote=true

Where *IP* is the IP address and *port* is the TCP port used by a proxying full peer member of the metaspace to listen for remote client connections (specified in the “remote client listen” URL of that proxying process).

You can now create spaces and perform the normal space operations on the space that is running on the core cluster.

Chapter 4 Implementing ActiveSpaces Security

TIBCO ActiveSpaces® provides a comprehensive solution for security of the ActiveSpaces hardware and the data stored in the data grid. This chapter provides an overview of ActiveSpaces security and describes how to set up security.

Topics

Overview of ActiveSpaces Security

TIBCO ActiveSpaces allows you to secure the information stored in the data grid by protecting both transport data and stored data. TIBCO ActiveSpaces security is provided for metaspaces that use TCP for discovery.

TIBCO ActiveSpaces security provides the following security features:

- Transport security
- Data encryption
- Authentication, Authorization and Accounting (AAA)

With TIBCO ActiveSpaces security you can:

- Encrypt information stored in the data grid
- Encrypt data stored in shared-nothing persistence stores
- Secure data during transmission within the data grid
- Prevent unauthorized access to data in the grid
- Prevent unauthorized recovery of persisted data
- Restrict user access to metaspaces, spaces, or data within a space
- Trace and log security related actions

ActiveSpaces Security Architecture

TIBCO ActiveSpaces security utilizes the concept of a security domain. A security domain defines specific security behavior for:

- Transport security
- Restricting transport access
- Data encryption
- User authentication
- User access control

You can define different security domains where each domain defines a different set of security-related behaviors. You specify security settings that define the security behavior for a security domain in a security policy file. See Security Policy Files, page 135 for more information on security policy files.

A security domain also specifies which metaspaces the security behavior of the security domain can be applied to. To apply security to a metaspace:

1. Associate the metaspace with a security domain.
2. Start a security domain controller for the metaspace.

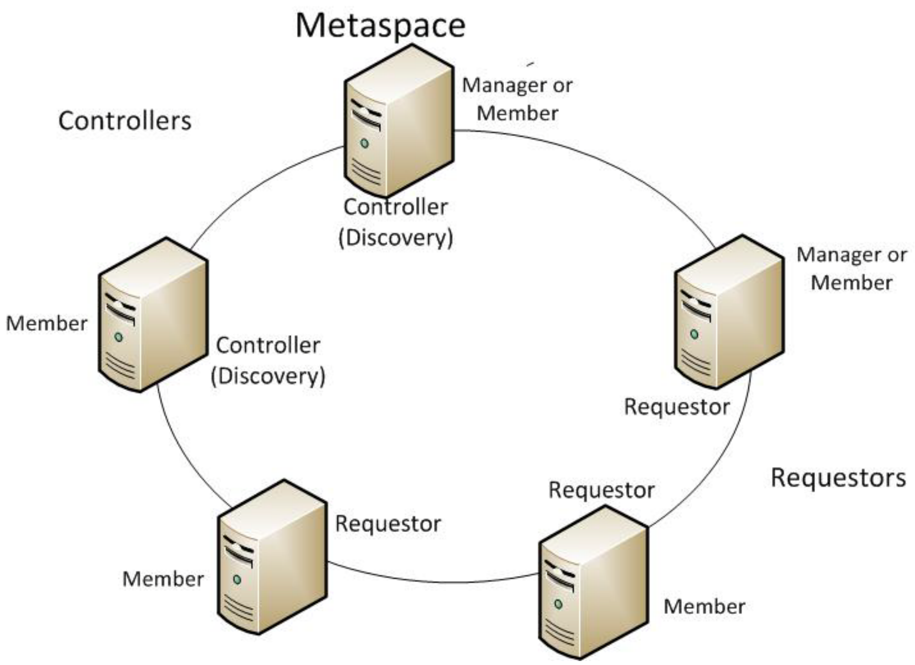
If you want to apply the same security settings to more than one metaspace, you can associate the same security domain with multiple metaspaces. However, only one security domain can be associated with each metaspace. See Metaspace Access List, page 141 for more information on associating metaspaces with security domains.

To utilize security within a metaspace, one or more nodes in the metaspace are initialized as a security domain controller for the metaspace. A security domain controller enforces the security domain's defined behavior for a metaspace. A security domain controller can be the manager of a metaspace or just a member of a metaspace. See Security Domain Controllers, page 133 for more information

Nodes in the metaspace that request security services from the security domain controllers are called security domain requestors. Similar to security domain controllers, a security domain requestor can be a manager of a metaspace or a member of a metaspace. These security components allow you to set up a secured ActiveSpaces cluster.

Figure 3 shows the two node types in the TIBCO ActiveSpaces security architecture.

Figure 3 TIBCO ActiveSpaces Security Architecture



The two node types in the ActiveSpaces security architecture are:

- **Security Domain Controllers** TIBCO ActiveSpaces nodes that are dedicated to enforcing a security domain's defined security behavior for a metaspace associated with the security domain. Security domain controllers are the only discovery nodes in a metaspace.

  For more information, see Security Domain Controllers, page 133.

- **Security Domain Requestors** Nodes that require access to the data in the data grid, such as a seeder or a leech, and which need to be authorized by a controller. Requesters can never be used a discovery nodes.

  For more information, see Security Domain Requestors, page 138.

Security Domain Controllers

A security domain controller is a TIBCO ActiveSpaces node that is dedicated to enforcing a security domain's defined security behavior for a metaspace associated with the security domain. For example, a security domain controller enforces the level of transport security used for communication between nodes of a metaspace and interfacing to an LDAP system when LDAP user authentication is used.

You must use TCP discovery to apply security to metaspaces. A security domain controller must be a “well known” member of a metaspace. This means that the member's listen URL IP address and port must be part of the discovery URL for the metaspace.

A TIBCO ActiveSpaces application becomes a security domain controller for a metaspace when it connects to a metaspace using all of the following:

1. A security policy file that sets up a security access binding for the metaspace and its discovery list.
2. The discovery list in the binding includes the application's listener address (in ip_address:port or hostname:port format).

See Security Policy Files, page 135 for detailed information on creating and using security policy files with security domain controllers.

The security domain controller for a metaspace must be running before security domain requestors are allowed to complete their connection to the metaspace. See Security Domain Requestors, page 138 for more information on security domain requestors.

When choosing an application to be a security domain controller, consider that the highest load on a security domain controller occurs when members are joining a metaspace. This processing is done in a separate thread from your application's thread. If you require quick response times when members join a metaspace, consider running an application that serves only as a security domain controller and does not connect to any spaces.

The `ASDomainController` example program demonstrates how to handle a domain controller. The `ASDomainController` example only connects to a metaspaces as a security domain controller.

For more information on the `ASDomainController` example program, see ASDomainController, page 213 in Chapter 5, Using the Example Code.

If servicing a lot of metaspace joins is not a concern, you can also use an `as-agent` as a security domain controller. The following example shows how to start `as-agent.exe` as a security domain controller:

```
as-agent.exe -metaspace "examplems"
-discovery "tcp://192.168.0.5:50000;192.168.0.10:50000"
-listen "tcp://192.168.0.5:50000"
-security_policy "mypolicy.txt"
```

You should set up multiple security domain controllers for each metaspace to provide fault tolerance for the security of each metaspace. If the security domain controllers go down for some reason, you lose your security for the metaspace.

Setting Up a Node as a Security Domain Controller

To set up a TIBCO ActiveSpaces node as a security domain controller:

1. Create a security policy file using the Admin CLI.
2. Edit the Metaspace Access List for the security domain in the security policy file.
3. Ensure that there is a `metaspace_access` entry with the metaspace name and discovery URL for the metaspace the node will connect to, in the Metaspace Access List.
4. Review the Transport Security, Restricted Transport Access, Authentication, Data Encryption, and Access Control settings to ensure they are set to meet your security requirements.
5. Save your changes to the security policy file.
6. Validate your security policy file using the Admin CLI.
7. Securely join the metaspace by using the TIBCO ActiveSpaces security API and passing in the security policy file. See the `ASDomainController` example for each supported programming language to see how the security API is used to connect to a metaspace as a security domain controller.

For some features of TIBCO ActiveSpaces security, the settings in the security policy file work in conjunction with calls to the security API in your application. Depending upon the type of security feature you implement, you might need to modify your application to use the features provided by the security API. See the sections for the individual security features to learn how to implement a particular security feature.

Security Policy Files

A security policy file contains the security settings for one or more security domains. Security domain requestors use a security token file that you generate from a security policy file to connect to a metaspace contained in the Metaspace Access List for a security domain that is defined in the security policy file.

The basic steps for creating and using a security policy file are:

1. Create a security policy file using the Admin CLI.

   See Creating a Security Policy File, page 136 for information on how to create a security policy file.

2. Open the security policy file with a text editor.
3. Edit the settings for each security domain to define the specific security behavior desired. See Security Domain Settings, page 136 for information on defining each type of security behavior.
4. Save and close the security policy file.
5. Validate the security policy file using the Admin CLI. See Validating a Security Policy File, page 137 for information on how to validate a security policy file.
6. If required for your transport security requirements, generate one or more security token files from the security policy file using the Admin CLI. See Security Token Files, page 139 for detailed information on security token files.
7. Modify your TIBCO ActiveSpaces application to enable it to function as a security domain controller for a metaspace, and use the security API to have the application connect to the metaspace using the security policy file.
8. See the `ASDomainController` example program for each supported programming language to see how the security API is used to connect to a metaspace using a security policy file.
9. Save, build and run your application.

Applications that connect to a metaspace listed in the Metaspace Access List for a security domain in a security policy file use the security policy file to become security domain controllers for the metaspace.

Applications that connect to a metaspace listed in the Metaspace Access List for a security domain in a security policy file, but do not use the security policy file to connect to the metaspace, become security domain requestors for the metaspace.

In general, you should require the security domain requestor to connect to a metaspace using a security token file that is generated from the security policy file. See 'Security Domain Requestors, page 138 and Security Token Files, page 139 for more information on security domain requestors and security token files.

You can also choose the less secure method of allowing connections without a security token. This is a weaker security solution, but is easier to deploy.

Creating a Security Policy File

You generate security policy files using the Admin CLI. You then edit the settings for each security domain within the security policy file to fit your particular security needs. The following example shows the Admin CLI command to create a security policy file for a policy named `mypolicy` and a security domain named `mydomain`:

```
as-admin> create security_policy
policy_name "mypolicy/mydomain"
policy_file "mypolicy.txt"
```

If you do not specify a domain name, ActiveSpaces creates a domain named AS-DOMAIN in the security policy file.

See Chapter 2, "Administering ActiveSpaces with the Admin CLI" in the *TIBCO ActiveSpaces Administration Guide* for information on the **`define | create security_policy`** command.

Security Domain Settings

A security policy file contains the following security settings for one or more security domains:

- Metaspace Access List
- Transport Security
- Restricted Transport Access
- User Authentication
- Data Encryption
- User Access Control

Some of the security settings work in conjunction with the TIBCO ActiveSpaces security API. For a more detailed discussion of how to use each setting, see the section for the setting in this chapter.

Validating a Security Policy File

You validate security policy files using the Admin CLI. After you have finished editing the security settings for the security domains included in the security policy file, validate the file to make sure that your edits to the file seem reasonable before you try to actually use the file. The following example shows the Admin CLI command to validate a security policy file:

```
validate policy_name "mypolicy" policy_file "mypolicy.txt"
```

Security Policy File Keys and Certificates

For each security domain, the security policy file also contains:

- A private key and public certificate that security domain requestors use to verify the identity of the security domain controller when establishing transport connections with the security domain controller. See Transport Security, page 143 for more information on the private key and public certificate.
- A data encryption key that is used by each node in a metaspace to encrypt data that resides in memory or is locally persisted. See Data Encryption, page 146 for more information on the data encryption key.

Security Domain Requestors

A security domain requestor is a TIBCO ActiveSpaces node that requests security services from a security domain controller for a metaspace. Typically, a security domain requestor is just a normal TIBCO ActiveSpaces application that uses the security API when connecting to a metaspace associated with a security domain.

The security domain settings in the security policy file for the security domain controller define the security applied to a security domain requestor.

Depending on your application's security requirements, the application might be required to connect to a metaspace using a security token file that has been generated from a security policy file. Consider requiring a security token file when a security domain requestor connects to a metaspace to ensure:

- Security domain requestors are restricted to connecting to specified metaspaces.
- The identity of security domain controllers the security domain requestor tries to connect to.
- The identity of security domain requestors when establishing secure transport connections.
- Only certain security domain requestors are allowed to connect to a metaspace.

See Security Token Files, page 139 for more information on how connecting to a metaspace with a security token file can affect a security domain requestor.

Connecting to a Metaspace Without Using a Security Token File

A a security domain requestor can also connect to a metaspace without specifying a security token file. In this scenario, an empty string (`"none"`) is used as the name of the security token file when connecting to the metaspace.

When a security domain requestor does not use a security token file, the following occurs:

- Connections to any metaspace in any security domain are allowed.
- A transport security level of `encrypted_normal` is used by default.
- If the security domain's transport security level is `encrypted_strong`, the security domain requestor's transport security level is automatically upgraded to `encrypted_strong`.

See Transport Security, page 143 for more information on transport security levels.

Security Token Files

When a security domain requestor uses a security token file to connect to a metaspace, ActiveSpaces uses the contents of the security token to:

1. Restrict the metaspaces to which a security domain requestor can connect.
2. Ensure the identity of the security domain controller.
3. Determine the level of transport security the security domain requestor should use for TCP communication.

The same token file can be shared by different security domain requestors. If you use the same token file for different requesters, consider the following:

- If the token does not have an ID, the tokens used by different requestors on the same metaspace will probably look the same.
- However, if the tokens do have an ID, you should avoid sharing it as their certificates will be the same.

Creating a Security Token File

You generate a security token file from a security policy file using the Admin CLI.and an existing security policy file. The following example shows the Admin CLI common for generating a security token file:

```
as-admin> create security_token
domain_name "mydomain"
policy_file "mypolicy.txt"
token_file "mytoken.txt"
```

This command generates a security token file that contains the following information from the specified security domain in the security policy file:

- The Metaspace Access List
- The Transport Security setting
- The public certificate of the security domain

See Chapter 2, “Administering ActiveSpaces with the Admin CLI “in the *TIBCO ActiveSpaces Administration* document for information on the **`define | create security_token`** command.

Limiting Metaspace Access

Typically, you do not need to edit a security token file. The one case where you might want to edit a security token file is when a security domain is associated with more than one metaspace, but you want to make sure that a security token file can only be used to connect to a specific metaspace.

When a security domain is associated with more than one metaspace, the Metaspace Access List for the security domain contains multiple `metaspace_access` entries in the security policy file. When you generate a security token file from the security policy file, multiple metaspaces are listed in the security token file. To restrict the metaspaces that can be connected to using this security token file, remove the `metaspace_access` entries for connections that should not be allowed.

See Metaspace Access List, page 141 for more information about the format of the Metaspace Access List in security policy files.

Validating a Security Token File

You validate security token files using the Admin CLI. After you have finished generating or editing a security token file, you should validate the file to make sure that the token file is valid before you try to actually use it. The following example shows the Admin CLI command to validate a security token file:

```
as-admin> validate token_file "mytoken.txt"
```

Security Token File Keys and Certificates

When you generate a security token file from a security policy file, the public certificate of the domain identity in the security policy file is copied to the security token file. When a security domain requestor attempts to connect to a metaspace using the security token file, the connection fails if the public certificate in the security token file does not match the security domain controller's identity certificate.

By default, security token files do not contain a private key and public certificate for establishing the identity of the security domain requestor. Thus, when a security domain requestor attempts to connect to a metaspace, a temporary private key and public certificate are dynamically created for the security domain requestor to establish secure connections with. This key and certificate are valid for the duration of its connection to the metaspace.

Optionally, when you generate a security token file you can specify creation of a private key and public certificate. The following example shows the Admin CLI command to generate a security token file with a private key and public certificate for establishing a security domain requestor's identity for secure transport connections:

```
as-admin> create security_token
domain_name “mydomain”
policy_file “mypolicy.txt”
          create_identity
             token_file “mytoken.txt”
```

See Restricting Transport Access, page 144 for information on how generating a private key and public certificate in the security token file can be used to restrict access in a security domain to only certain security domain requestors.

Metaspace Access List

Each domain defined in a security policy file contains a Metaspace Access List. The Metaspace Access List restricts the security behavior defined by the settings for its security domain to only those metaspaces specified in the list. A metaspace can only belong to one security domain.

Each item in the Metaspace Access List must follow the format:

```
metaspace_access=metaspace=<metaspace name>;discovery=<discovery
URL>
```

where:

- *metaspace name* is the name of the metaspace (no quotes)
- *discovery URL* is the TCP discovery URL of the metaspace (no quotes)

For example, to add the metaspace 'examplems' with a discovery URL of 'tcp://192.168.0.10:50000' to the Metaspace Access List for the domain 'mydomain' in the security policy file `mypolicy.txt`:

1. Open `mypolicy.txt` in a text editor.
2. Find the Metaspace Access List for the security domain `mydomain`.
3. Modify the following line in the Metaspace Access List:

   ```
   metaspace_access=metaspace=mydomain-ms1;
   discovery=tcp://127.0.0.1:50000
   ```

   to read:

   ```
   metaspace_access=metaspace=examplems;
   discovery=tcp://192.168.0.10:50000
   ```

4. Save `mypolicy.txt.`
5. To add additional metaspaces to the metaspace access list add, another metaspace_access item after the first metaspace_access item. For example:

```
metaspace_access=metaspace=examplems;
discovery=tcp://192.168.0.10:50000

metaspace_access=metaspace=examplems2;
discovery=tcp://192.168.0.11:50001
```

When you generate a security token file from a security policy file, the Metaspace Access List for the specified security domain is copied from the security policy file into the security token file. A security domain requestor using the security token file is allowed to connect to any of the metaspaces in the Metaspace Access List. To further restrict which metaspaces can be connected to, you should edit the security token file and remove any undesired metaspaces from the Metaspace Access List.

Transport Security

TIBCO ActiveSpaces allows you to protect data being transported by preventing:

- Alteration of traffic
- Eavesdropping
- Exchange of data between untrusted parties

When security is used for a metaspace, any transmission of messages within the data grid occurs on a secure transport. A security domain's `transport_security` setting controls the level of security used for communication within the data grid.

The available settings for `transport_security` are:

- **encrypted_normal** Use secure transport with 128 bit symmetric key encryption (default).
- **encrypted_strong** Use secure transport with 256 bit symmetric key encryption.
- **integrity** Use secure transport without encryption.

Restricting Transport Access

TIBCO ActiveSpaces security allows you to restrict transport connections within a security domain to only “trusted” nodes.

To restrict transport connections within a security domain:

1. Open the security policy file for the domain in a text editor
2. Go to the line that reads `transport_access=false;cert_file=`
3. Edit the line to read:

   transport_access=true;cert_file=<*trusted_certs_file*>

   where *trusted_certs_file* is the filename for a trusted certificate file that you will create in step 8.
4. Save the security policy file.
5. Use the **`validate policy_file`** Admin CLI command to validate the security policy file.
6. Use the Admin CLI to generate a security token file from the security policy file, which contains its own private key and public certificate. This key and certificate are used to verify the identity of a node using the security token file when it tries to initiate any transport connections. For example,

   ```
   as-admin> create security_token
   domain_name "mydomain"
   policy_file "mypolicy.txt"
   create_identity
   token_file "mytoken.txt"
   ```
7. Use the **`validate token_file`** Admin CLI command to validate the security token file.
8. Create an empty trusted certificates file to hold the public certificates of the nodes to allow transport connections from.
9. Copy and paste the public certificate of the local token identity from the security token file into the trusted certificates file.

The public certificate is everything in the security token file between and including

```
-----BEGIN CERTIFICATE-----
-----END CERTIFICATE-----
```

10. Save the trusted certificates file.
11. Start a security domain controller using the security policy file name when connecting to the metaspace.

Metaspace communication within the security domain is now restricted to only security domain controllers and security domain requestors that connect to the metaspace using a security token file whose public certificate is contained in the trusted certificates file.

Data Encryption

The TIBCO ActiveSpaces security API allows you to define encrypted fields in a space.

When data is put into a field that is defined to be encrypted, the data is encrypted while it resides in memory in the data grid and when it is persisted with shared-nothing persistence.

Certain types of fields in a space should not be encrypted. Do not encrypt fields that are used:

- As keys or indexes
- In filters for searching through the data in a space

Suppose that you need to protect the social security number of patients admitted to a hospital. You could store the social security number in an encrypted field to ensure that the social security number cannot be accidently read while it is stored in memory in the data grid or stored using shared-nothing persistence. You can use the patient's name or admission ID as a key for the space and search for their name or admission ID to later retrieve their social security number.

To allow encryption to be used when defining the fields of a space using any of the TIBCO ActiveSpaces language APIs, set the following for the security domain in the security policy file:

```
data_encryption=true
```

If you try to define an encrypted field in a space when the `data_encryption` setting is set to `false`, ActiveSpaces throws an exception.

The `data_encryption` setting in a security domain is used in conjunction with the following methods to specify that the contents of a field should be encrypted:

- **Java** `FieldDef.setEncrypted(boolean)`
- **C** `tibasFieldDef_SetEncrypted(tibasFieldDef fieldDef, tibas_boolean secured)`
- **.NET** FieldDef.Encrypted

The data stored in an encrypted field is encrypted with a symmetric data encryption key that is generated when a security policy file is created. The data encryption key is always unique for each security domain and is stored encrypted under the domain's identity.

The security domain controllers pass the data encryption key to each security domain requestor so that all security domain requestors can encrypt and decrypt the data of encrypted fields in a space.

Security Tracing and File Logging

ActiveSpaces outputs security-related messages output as trace messages and to the TIBCO ActiveSpaces log file. You can control the level of the security messages, as with normal TIBCO ActiveSpaces tracing and logging. The following API methods control the level of security messages that are output to both the console and log file:

- **Java API** `ASCommon.setSecurityLogLevel(LogLevel)`
- **C API** tibasSetSecurityLogLevel(LogLevel)
- **.NET API** ASCommon.SecurityLogLevel

User Authentication

TIBCO ActiveSpaces security allows you to authenticate the users of security domain requestors.

If the security domain for a metaspace has been configured to perform user authentication, then user authentication occurs when a security domain requestor tries to connect to the metaspace.If authentication fails, the connection to the metaspace fail.s

The `authentication` setting in the security policy file used by the security domain controller for a metaspace controls how users are authenticated. You can specify two types of user authentication:

- Username and password authentication
- Certificate-based authentication against an LDAP server

Username and password authentication can be performed using the operating system's authentication services or an LDAP server. Certificate-based authentication can only be done using an LDAP server.

The basic format of the `authentication` setting in the security policy file is:

```
authentication=<none(default)|userpwd|x509>;[source=<system|ldap>;
<source property>;...;hint=<string>]
```

If you specify `userpwd` or `x509` for the `authentication` setting. you must specify `source` settings to enable the security domain controller to connect to the system that performs the authentication:

- `source=system` specifies that the security domain controller should use operating system services to authenticate users.
- `source=ldap` indicates that the security domain controller should connect to and use an LDAP server for authentication.

See the following sections for more detailed information on how to configure the `authentication` settings for operating system or LDAP authentication:

- Operating System User Authentication, page 149
- LDAP User Authentication, page 150
- LDAP Certificate Authentication, page 151

For each type of authentication, TIBCO ActiveSpaces prompts the user of the security domain requestor to enter the appropriate information needed for authentication (for example, user name and password, location of PKCS#12 file, and the password of the private key in the file).

You can override the default behavior for retrieving authentication information by using a callback mechanism. If a callback function is available, then when a security domain requestor tries to connect to a metaspace, ActiveSpaces uses the callback function to retrieve the user's authentication information instead of using the default behavior provided by TIBCO ActiveSpaces. See Authentication Callback, page 152 for more detailed information.

Operating System User Authentication

User name and password authentication can be done using the operating system to authenticate the user. When operating system authentication is used and a security domain requestor first tries to connect to a metaspace, TIBCO ActiveSpaces prompts the user to enter their:

1. Login domain name (on Windows systems)
2. Login user name
3. Login password

For example, suppose you normally log into Windows using “AcmeInc\brady” for your domain and user names and “abc123” for your password. You should enter:

1. “AcmeInc” when prompted for the domain
2. “brady” when prompted for the user name
3. “abc123” when prompted for the password

The logon information entered is passed to the security domain controller, which tries to perform user authentication with the operating system.

When operating system based user authentication is configured:

- Pluggable Authentication Modules (PAM) is used on UNIX and Linux systems
- NTLM/Kerberos is used on Windows systems

To configure the security policy file to perform user name and password authentication using the operating system, set the `authentication` setting as follows:

```
authentication=userpwd;source=system;service=login;hint=<message
to display to user>
```

The `service` setting specifies the operating system application to use for authentication. Currently this setting is ignored for Windows and is only used for UNIX systems. Specifying `service=login` causes the UNIX "login" system access application to be used to authenticate security domain requestor users. You can use the `service` setting to redirect PAM authentication requests to other local authentication applications.

LDAP User Authentication

You can configure user authentication to use an LDAP server to perform user name and password authentication. When user name and password authentication is used with LDAP and a security domain requestor first tries to connect to a metaspace, TIBCO ActiveSpaces prompts the user to enter their:

1. Login user name
2. Login password

The logon information entered is passed to the security domain controller, which tries to connect to the LDAP server configured in the security policy file and use the LDAP server to authenticate the user.

To configure the security policy file to perform user name and password authentication with an LDAP server using an unsecure connection to the LDAP server, the `authentication` setting uses the following format:

```
authentication=userpwd;source=ldap;name=<LDAP object name>;
host=<LDAP server name>;plainPort=<port number>;
baseDN=<DN of parent>;hint=<message displayed to user>
```

where the unsecure LDAP connection parameters are:

- **name** Name of the object to query LDAP for (for example, `cn` for common name, `uid` for unique ID).
- **host** The fully qualified domain name of the LDAP server (for example, `ldapsrvr.com`).
- **plainPort** The port on which the LDAP server listens for clear text TCP/IP connections (default: 389).
- **baseDN** The distinguished name of the parent of the LDAP subtree (for example: `dc=users,dc=com`).
- **hint** A message to be displayed to the user as a hint of what they should enter.

If connecting to the LDAP server requires a secure connection using SSL/TLS, the `authentication` setting uses the following format:

```
authentication=userpwd;source=ldap;name=<LDAP object name>;
```

```
host=<LDAP server name>;securePort=<port number>;trustStore=<LDAP
keystore>;
baseDN=<DN of parent>;hint=<message displayed to user>
```

where the secure LDAP connection parameters are:

- **name** Name of the object to query LDAP for (for example, `cn` for common name, `uid` for unique ID).
- **host** The fully qualified domain name of the LDAP server (for example, `ldapsrvr.com`).
- **securePort** The port on which LDAP clients should connect to the LDAP server using SSL/TLS (default: 636)
- **truststore** A file that contains the secure LDAP server's certificate chain
- **baseDN** The distinguished name of the parent of the LDAP subtree (for example: `dc=users,dc=com`).
- **hint** A message to be displayed to the user as a hint of what they should enter.

The security domain controller uses the contents of the truststore to authenticate the LDAP server when establishing a connection to the LDAP server. The truststore format can be a p7b file containing only certificates and certificate chains. If the LDAP server certificate is self-signed, the truststore can be a .pem certificate file or a binary DER format file.

LDAP Certificate Authentication

Security domain controllers can be configured to perform certificate authentication using an LDAP server. When certificate authentication is used and a security domain requestor attempts to connect to a metaspace, the user will be prompted to enter the following:

1. Path to a PKCS#12 (.p12) file to use for authentication.
2. Password for the private key inside of the PKCS#12 file.

The authentication information is passed to the security domain controller, which tries to authenticate the user against the LDAP server configured in the security policy file. To configure the security policy file to perform LDAP certificate authentication with an LDAP server, the `authentication` setting uses the following format:

```
authentication=userpwd;source=ldap;name=<LDAP object name>;
host=<LDAP server name>;securePort=<port number>;trustStore=<LDAP
keystore>;
baseDN=<DN of parent>;hint=<message displayed to user>
```

where the secure LDAP connection parameters are:

- **name** Name of the object to query LDAP for (for example, `cn` for common name, `uid` for unique ID).
- **host** The fully qualified domain name of the LDAP server (for example, `ldapsrvr.com`).
- **securePort** The port on which LDAP clients should connect to the LDAP server using SSL/TLS (default: 636)
- **truststore** A file that contains the secure LDAP server's certificate chain
- **baseDN** The distinguished name of the parent of the LDAP subtree (for example: `dc=users,dc=com`).
- **hint** A message to be displayed to the user as a hint of what they should enter.

When LDAP certificate authentication is used, a secure LDAP server must always be used. The security domain controller uses the contents of the truststore to authenticate the LDAP server when establishing a connection to the LDAP server. The truststore format can be a p7b file containing only certificates and certificate chains. If the LDAP server certificate is self-signed, the truststore can be a .pem certificate file or a binary DER format file.

Authentication Callback

Sometimes you might not want to use ActiveSpaces' default behavior for retrieving user authentication information. For example, if you want your users to use a smart card or USB drive to hold their authentication information and automatically authenticate security domain users without their being aware that the authentication has taken place, you can override the default behavior.

To override the default behavior of TIBCO ActiveSpaces for retrieving user authentication information, the ActiveSpaces API provides a callback mechanism. If a security domain requestor tries to connect to a metaspace using an authentication callback, the callback is used to retrieve the user's authentication information.

See ASUserAuthenticator, page 216 for information on each supported programming language to see how the security API is used to implement an authentication callback.

User Access Control

TIBCO ActiveSpaces security provides user access control to the operations on a metaspace or space. User access control allows you to control the types of TIBCO ActiveSpaces functionality a user is allowed to perform. User access can be allowed or denied for the following permissions:

- **read** Allows reading the contents of a space. The tuple_get operation requires read access. Get browsers also require read access.

  Read is the minimum permission required on any scope, because it implies the right to connect to a metaspace.
- **write** Allows writing data to a space. Examples of operations that require write access are:
  - put
  - take
  - lock
  - unlock

  Take browsers, lock browsers, and transaction also require write access.

  Write permission implies that a user also has read permission, because most ActiveSpaces operations that write to a space must first be able to read data from the space. For example, to take a tuple from a space, an application must first do a read to find the tuple before it can remove the tuple.
- **invoke** Perform remote invocations on a a space
- **seeder** Allows the user to seed tuples.
- **encrypt** Allows the user to encrypt tuples.

You can grant or deny a user all of the above permissions by specifying one of the following:

- `grant_all`
- `deny_all`

You can also arrange users into groups and apply permissions to all users in a group or to the users in several groups. And you can specify which spaces in a metaspace the permissions for users and groups should be applied to.

Permissions can be applied for:

- A single metaspace and space
- All spaces in a metaspace

- A particular space name in any metaspace
- All metaspaces and spaces

User access control works in conjunction with user authentication. Do not enable authorization (access control) unless user authentication is enabled

See User Authentication, page 148 for information on how to configure the security policy file to enable user authentication.

Enabling User Access Control

The following example shows the format of the `access_control` setting in the security policy file:

```
access_control=<true|false(default>;default=<deny|grant>
```

After you have configured user authentication in the security policy file, you then need to enable user access control in the security policy file by specifying the following setting:

```
access_control=true;default=deny
```

The default setting specifies whether a user should be denied any access or granted all access permissions, if no access permissions have been defined for a user.

Access Control Groups

To group users so that permissions can easily be applied to multiple users, you must define each group of users that you would like to apply permissions to in the security policy file. Locate the `groups` heading in the security policy file and add a line after it for each group of users. For example, specify the following:

```
groups
   group1=davidl,robertb,tomd
   group2=susanh,joannd,nicolem
   group3=group1,miket,joew
```

A group name can consist of any combination of letters and numbers but can only be defined once. A group can be assigned to other groups.

A user name is either a user's logon user name, if user name/password authentication is used, or the common name of the user's leaf certificate when LDAP certificate authentication is used.

Access Control Permissions

Once you have defined your user groups, you can now apply permissions to each group of users or to single users. Locate the `permissions` heading in the security policy file and add a permissions declaration after the `permissions` heading for each metaspace or space that you want to control the access to. A permissions declaration has the following format:

```
<<metaspace name>|<space name>|<metaspace name>/<space name>>
<<user name>|<group name>>=<permission>,...
```

where `permission` can be any of the following:

- `grant_all`
- `deny_all`
- `read`
- `write`
- `invoke`
- `seeder`
- `encrypt`

For detailed information on the permissions, see User Access Control, page 153.

You can use a wildcard character (*) for the metaspace name or space name. A single wildcard character (*) can replace both the metaspace name and space name to designate that the permissions will apply to all metaspaces and all spaces. For example:

```
// Examples:
// domain1-ms1/* group1=read, seeder
// domain2-ms4/sp1 group2=write, encrypt
// */sp2 group1=write, invoke
//
permissions
ms/* group1=seeder,read,write,encrypt
```

Permissions Precedence

Permissions precedence is based on the following evaluation rules:

1. Scope rule
2. Denial rule
3. Ambiguity rule
4. Owner rule

5. Order rule

The permissions rules work as follows:

- **Scope rule** When an access control list (ACL) is enabled, any connection request to a metaspace must be associated with a valid space-level permission entry, which implicitly grants access to the metaspace.

  If there is no space-level permission, the client's connection to the controller fails even if authentication is successful. The only exception is when the user cannot be mapped to the group list and the ACL's default access is grant_all. In other words, any successful connection to a metaspace requires that one or more permissions with a corresponding scope exist in the permissions table.

  The minimum permission required on any scope is `read`, which implies the right to connect to a metaspace.

- **Denial rule** A `deny_all` declaration for a user or group takes ultimate precedence over any other permission declaration that might apply to the same user or group.

- **Ambiguity rule** If there are multiple permissions that can be applied to a metaspace or space, then the permissions declaration that explicitly names the metaspace or space takes precedence over any permissions declarations that use a wildcard character (*).

- **Owner rule** If there are multiple permissions that can be applied to a user, the permissions declaration that explicitly names the user takes precedence over any permissions declarations applied to a group that the user is a member of.

- **Order rule** If after applying the above rules there is still more than one permission that applies to the authenticated user's context, the effective privilege is retrieved from the most recent (lowest) matching permission in the table.

Chapter 5 Using the Example Code

This chapter describes how to use the example programs that are provided with TIBCO ActiveSpaces.

Topics

- SpaceStateMonitor, page 209
- SpaceMemberMonitor, page 211
- ASDomainController, page 213
- ASUserAuthenticator, page 216

Overview

TIBCO ActiveSpaces provides examples programs that demonstrate how to use the various features of the product. Some of the examples (such as `ASOperations`) provide boilerplate code templates that you can copy and paste directly into your application; others demonstrate the use of specific API functions.

The example programs are one of the most important resources to use when learning to program with ActiveSpaces. It is highly recommended that any developer new to ActiveSpaces reserve some time to study the examples.

The Examples Directory

The TIBCO ActiveSpaces examples are provided in the *AS_HOME*`/examples` directory. This directory contains examples for each API set included with the product—Java, C, and .NET. The examples for each API set are in the following directories:

- *AS_HOME*/**`examples/c`**—C API examples
- *AS_HOME*/**`examples/java`**—Java API examples
- *AS_HOME*/**`examples/dotnet`**—.NET API examples

Different examples are available depending upon the language API. For example, the `ASPaint` example is only available for the Java and .NET API sets.

Building the Examples

The following sections describe how to build the TIBCO ActiveSpaces examples for each API set:

- Building the Java API Examples, page 160
- Building The C API Examples, page 160
- Building the .NET API Examples, page 161

Building the Java API Examples

The Java examples are provided in the *AS_HOME*`/examples/java` directory. To facilitate building the Java examples, a `build.xml` file for use with Apache Ant is provided.

See `http://ant.apache.org` for more information on using Apache Ant for building Java applications.

Complete these steps to build the Java examples:

1. Ensure that the `/bin` directory for `ant` is in your path.
2. Change to the directory containing the ActiveSpaces Java examples.

   `cd` *AS_HOME*`/examples/java`
3. Enter the following:

   `ant`

The Java compiler compiles the examples and two jar files are created:

- `Examples.jar` contains the class files for all of the examples except for `ASPaint`
- `ASPaint.jar` contains the class files for only the `ASPaint` example.

Building The C API Examples

The C examples are provided in the *AS_HOME*`/examples/c` directory. Makefiles are provided for the platform you are installing on.

Building the C Examples on Windows

For the Windows platform, TIBCO ActiveSpaces provides a `Makefile` that works with Microsoft Visual C++.

Complete these steps to build the C examples:

1. Ensure the *AS_HOME* environment variable has been set to the location where TIBCO ActiveSpaces has been installed.
2. Ensure the *TIBRV_HOME* environment variable has been set to the location where TIBCO Rendezvous has been installed.

This step is only required if TIBCO Rendezvous will be used as the network transport for ActiveSpaces.

3. Ensure that your environment is set up for building with Microsoft Visual Studio. For example, on Windows 7 64-bit with Visual Studio 2010, do the following:

   ```
   set VCINSTALLDIR=c:\Program Files (x86)\Microsoft Visual Studio 10.0\VC
   ```

   ```
   "%VCINSTALLDIR%"\vcvarsall.bat x86_amd64
   ```

4. Enter the following commands:

   ```
   cd AS_HOME/examples/c
   ```

   ```
   nmake
   ```

The compiler generates executable files for the example programs.

Building the C Examples on Non-Windows Platforms

For platforms other than MS Windows, two makefiles are provided:

- `Makefile`
- `Makefile.mk`

`Makefile.mk` is included by the `makefile` file, and contains platform and compiler settings for the target platform.

To build the C examples on a non-Windows platform:

1. Edit the settings in the `Makefile.mk` file as required.
2. Run your compiler against the `makefile` to generate the example code.

Building the .NET API Examples

The .NET API examples can be found in the *AS_HOME*`/examples/dotnet` directory. To facilitate building the .NET examples, the file `build.cmd` is provided. A Microsoft Visual Studio solutions file is also provided.

This section describes the following tasks:

- Building the .NET Examples from the Windows Command Line, page 162
- Building the .NET Examples from MS Visual Studio, page 163

Building the .NET Examples from the Windows Command Line

Complete these steps to build the examples from a Windows command window:

1. Ensure that the *AS_HOME* environment variable is set to the TIBCO ActiveSpaces installation directory.

   For example:

   ```
   set AS_HOME=c:\tibco\as\2.0
   ```

2. Ensure that your environment has been set up for building with Microsoft Visual Studio.

   For example, on a Windows 7 64-bit machine with Visual Studio 2010, execute the following commands:

   ```
   set VCINSTALLDIR=c:\Program Files (x86)\Microsoft Visual Studio
   10.0\VC
   ```

   ```
   "%VCINSTALLDIR%"\vcvarsall.bat x86_amd64
   ```

3. Ensure that *AS_HOME*`/lib/TIBCO.ActiveSpaces.Common.dll` has been installed into your Global Assembly Cache (GAC).

This should have automatically been done during installation, but you can verify it using the following command:

```
gacutil -l TIBCO.ActiveSpaces.Common
```

4. If `TIBCO.ActiveSpaces.Common.dll` is not present in your GAC, enter the following:

   ```
   cd AS_HOME/lib
   ```

   ```
   gacutil -i TIBCO.ActiveSpaces.Common.dll
   ```

5. To build the .NET examples, enter:

   ```
   cd AS_HOME/examples/dotnet
   ```

```
build.cmd
```

To build the .NET examples for 32-bit when installed on a 64-bit platform, add the following C# compiler command option to the csc commands in build.cmd:

```
/platform:x86
```

For example,

```
  csc /r:%AS_HOME%\lib\TIBCO.ActiveSpaces.Common.dll /platform:x86
ASOperations.cs ASExampleBase.cs
```

Building the .NET Examples from MS Visual Studio

The ActiveSpaces installation provides a Visual Studio solutions file for building the .NET examples.

Complete these steps to build the examples using MS Visual Studio:

1. Ensure that the *AS_HOME* environment variable has been set to the TIBCO ActiveSpaces installation directory.

   For example:

   set *AS_HOME*=**c:\tibco\as\2.0**

2. Ensure that *AS_HOME*`/lib/TIBCO.ActiveSpaces.Common.dll` has been installed into your Global Assembly Cache.

This should have automatically been done during installation, but you can verify it using the following command:

```
gacutil -l TIBCO.ActiveSpaces.Common
```

3. If `TIBCO.ActiveSpaces.Common.dll` is not present in your GAC, do the following:

   cd *AS_HOME*/**lib**

   ```
   gacutil -i TIBCO.ActiveSpaces.Common.dll
   ```

4. Invoke the Microsoft Visual Studio solutions file for the TIBCO ActiveSpaces .NET examples:

   cd *AS_HOME*/**examples/dotnet**

   ```
   Examples.sln
   ```

5. In Visual Studio, right click on **Solution Examples** and select **Build Solution**.

To build the .NET examples for 32-bit when installed on a 64-bit platform, set the Platform Target to x86 in the Build properties for the project.

Running the Examples

As long as your *PATH/LD_LIBRARY_PATH* and *CLASSPATH* environment variables are correctly set up, it should be straightforward to run the examples.

Since many examples are distributed, you should run multiple instances of the example applications at the same time to see how easy it is to create a coherent and consistent distributed system using ActiveSpaces.

Because ActiveSpaces is platform independent, the C, Java, and .NET versions of the examples are completely interoperable; you can, for example, run one instance of an example application in C and one in Java and they will interact just as if all instances were C (or Java).

The following sections contain information on setting up your environment and the commands to use for running the ActiveSpaces examples for each of the language APIs.

Running the Java API Examples

Before running the Java examples, ensure that your environment variables have been set up for running ActiveSpaces as discussed in 'Chapter 2 Setting Environment Variables' of the document 'TIBCO ActiveSpaces Installation'.

`Examples.jar` or `ASPaint.jar` must be appended to your *CLASSPATH* depending upon which example you are running. For example, on a Microsoft Windows platform, you might set your `CLASSPATH` variable as follows:

```
C:\tibco\as\2.1\examples\java\Examples.jar;
C:\tibco\as\2.1\examples\java\ASPaint.jar
```

If any additional jar files need to be appended to your *CLASSPATH* for running an example, the section on that example will have the additional information.

After your *CLASSPATH* is set appropriately, you can invoke all of the Java examples by using the following basic command:

`java` *example_name command_line_arguments*

For example, to run `ASOperations` with the default settings, enter:

```
java ASOperations
```

See Command Line Arguments, page 166 for information on the command line arguments that you can use when invoking the examples.

Alternatively, in Java, you can launch the example programs using the `java` command with the `classpath` (`-cp`) parameter.

```
java -cp Examples.jar class_name command_line_arguments
```

where *class_name* is the name of the example class. For example:

```
java -cp Examples.jar ASChat
```

`ASOperations` is the default class for the jar file and can be launched directly using

```
java -jar Examples.jar
```

Running the C API Examples

To run a C API example program:

Windows

1. Set your `PATH` environment variable to the directory where you have compiled the example programs (e.g. *AS_HOME*`/examples/c`).
2. Ensure *AS_HOME*`/lib` is set in your library path environment variable (e.g. PATH, LIBPATH, LD_LIBRARY_PATH, etc.).
3. Enter the name of the executable file for the example you wish to run, followed by any command line arguments. For example, to run `ASChat`, enter **`ASChat`**.

The following section on command line arguments applies to all three API sets.

UNIX/Linux/AIX/HP UX

1. Use a text editor to edit the `Makefile.mk` file provided with the TIBCO ActiveSpaces C example programs.

   The Makefile is located in the `/AS_HOME/examples/c` directory.
2. Locate the line that reads:

   `CC=`*<compiler>*

   For UNIX, this line reads `CC = CC`; for Linux, `CC = gcc`; for AIX, it reads `CC = xlC`; for HP-UX, `CC = aCC`.
3. Change the line to read:

   `CC= <`*`path`*`>/<compiler>`

   where *path* is the path to the C compiler that you are using and compiler is the compiler name.
4. Save the file.

5. Enter the name of the executable file for the example you wish to run, followed by any command line arguments. For example, to run `ASChat`, enter **`ASChat`**.

Running the .NET API Examples

To run a .NET API example program:

1. Set your PATH environment variable to the directory where you have compiled the example programs (e.g. *AS_HOME*`/examples/dotnet`).
2. Ensure *AS_HOME*`/lib` is also set in your PATH environment variable.
3. Enter the name of the executable file for the example you wish to run, followed by any command line arguments. For example, to run `ASChat`, enter **`ASChat`**.

The following section on command line arguments applies to all three API sets.

Command Line Arguments

Many of the examples use the same basic set of command line arguments, which you can use to change the default settings used by the examples. For example, you might want to use a different metaspace name so you will not run into conflicts with other users who are running the ActiveSpaces examples on the same network.

When an example does not support certain command line arguments or supports additional command line arguments, those argument differences will be described in the sections relating to that particular example.

The following sections contain detailed information about the basic command line arguments supported by most of the examples.

Command Line Help

Invoking any example with the following argument displays the list of possible command line arguments for that example:

`-help`

Metaspace Command Line Arguments

All of the ActiveSpaces examples default to using the same metaspace.

You can use the following command line arguments to change metaspace settings when invoking the examples:

`-metaspace` *metaspace_name*   `(`default`: ms)`

`-discovery` *discovery_URL* (default: `tibpgm`)

`-listen` *listen_URL* (default: `tcp`)

`-member_name` *member_name* `(Must specify a unique string identifying the member;` default: `auto-generated member name)`

SpaceMemberDef Command Line Arguments

The following command line arguments can be used to change the SpaceMemberDef settings when invoking the examples:

`-role leech | seeder (default: leech)`

`-persistence shared_nothing | shared_all`

SpaceDef Command Line Arguments

Most of the ActiveSpaces examples default to using the same SpaceDef.

When an example uses other settings, this is indicated in the section covering that example. The following command line arguments can be used to change the SpaceDef setting when invoking the examples:

`-space` *space_name* (default: `myspace)`

`-capacity` *entries_per_seeder* where *entries_per_seeder* specifies the number of entries per seeder; -1= infinite (default)

`-eviction none | lru` (default: `none)`

`-data_store` *directory_path* `(default:` *AS_HOME*`/examples/data_store)`

`-_count` *num_copies* `(default: 0)`

`-min_seeders` *num_seeders* `(default: 1)`

Default Metaspace Name

The default metaspace name used by all of the examples is `ms`. Since the default discovery URL is `tibpgm`, conflicts might occur when multiple users are on the same network and using the default metaspace `ms`. Therefore, it is recommended that when running the examples, you specify a unique metaspace name on the command line using the `-metaspace` command line option.

Default Discovery URL

The default discovery URL used by all of the examples is `tibpgm`. This means that the PGM transport is used by ActiveSpaces applications (peers) to "discover" each other. If multiple versions of ActiveSpaces are being used on the same network, message deserialization errors can occur, because the message formats are typically not compatible between the various versions of ActiveSpaces.

When `tibpgm` is specified as the discovery URL, the default destination port used is 7888. To prevent conflicts when multiple versions of ActiveSpaces are being used with PGM as the discovery transport, it is recommended that a unique discovery URL be specified on the command line using the `-discovery` command line option.

For example,

```
-discovery tibpgm://7900
```

This discovery URL indicates that the destination port used by the PGM transport will be port 7900. As long as no other version of ActiveSpaces is run using the same port, you will not get discovery message deserialization errors.

Default Space Fields

Unless otherwise noted, the default space (`myspace`) that is defined for use by the examples has a definition consisting of three fields:

- A mandatory key field called `key` of type `integer`
- An optional field called `value` of type `String`
- An optional field called time of type `DateTime`

Adding Security

Most of the examples can be run using the security features of TIBCO ActiveSpaces. To help you get started, an example security policy file and security token file can be found in the `/security` subdirectory of the examples for each language API.

For detailed information on the ActiveSpaces security features, see Chapter 4, Implementing ActiveSpaces Security.

To run an example with security applied to it, you must first start up a security domain controller for the metaspace the example will use. You can use as-agent as a security domain controller.

To run `as-agent` as a security domain controller using the example security policy file, change to the example directory that contains `example_policy.txt` and invoke `as-agent` as follows:

Java Invocation

```
java [-d64] -jar AS_HOME/lib/as-agent.jar -security_policy
example_policy.txt -discovery tcp://127.0.0.1:50000 -listen
tcp://127.0.0.1:50000
```

C Invocation

```
AS_HOME/bin/as-agent.exe -security_policy example_policy.txt
-discovery tcp://127.0.0.1:50000 -listen tcp://127.0.0.1:50000
```

.NET Invocation

```
AS_HOME\bin\Agent.NET.exe -security_policy example_policy.txt
-discovery tcp://127.0.0.1:50000 -listen tcp://127.0.0.1:50000
```

These commands start up `as-agent` using the default metaspace name of `ms`. When security is used, the TCP transport must be used. So we have used the loopback IP address of tcp://127.0.0.1 and a port of 50000 for both the discovery and listen URLs. This will keep the network connections on your local machine.

If you need to change the metaspace name or discovery URL, you will also need to modify the metaspace name or discovery URL in the `metaspace_access` entry in the example security policy file and regenerate the example security token file.

To see how to use the security API to code a security domain controller, see ASDomainController, page 213.

Once you have started a security domain controller, to apply security to most of the examples you just need to add the following command line option when starting the example:

```
-security_token exdomain_token.txt
```

Refer to the section for each example to see whether security can be applied to it.

The following sections explain how the example security policy file and security token file were created and which security features are enabled when you use them.

Example Security Policy File

The security policy file, `example_policy.txt`, was created using the Admin CLI by issuing the following command:

as-admin> **`create security_policy policy_name "example/exdomain" encrypt false policy_file "example_policy.txt"`**

This generates a security policy file named `example_policy.txt` containing the following:

1. One security domain named `exdomain`.
2. A metaspace access list with a default `metaspace_access` entry of:

   ```
   metaspace_access=metaspace=ms;discovery=tcp://127.0.0.1:50000
   ```
3. An unencrypted private key and public certificate for the security domain's identity.
4. A data encryption key for the security domain.

When you use security, you must use TCP transport. Notice that the discovery URL in the `metaspace_access` entry is `tcp://127.0.0.1:50000`. This URL designates the loopback IP address of 127.0.0.1 and port 50000 as the discovery IP address and port, which will keep metaspace discovery messages on your local machine.

The generated policy file, `example_policy.txt`, was edited as follows:

1. The data_encryption setting was changed from `false` to `true`.
2. The group entry, `group1 = user1`, was added under Access Control Groups.
3. The following permission entry was added under Access Control Permissions:

   ```
   ms/* group1=tuple_get,tuple_put,tuple_encrypt,space_browse
   ```

The example security policy file uses the default metaspace name of `ms` in the `metaspace_access` list. If you wish to use a different metaspace name, you must change the metaspace name in the following line:

```
metaspace_access=metaspace=ms;discovery=tcp://127.0.0.1:50000
```

Without any modifications, the example security policy file can be used to ensure secure transports are used for communication throughout a metaspace.

It can also be used to run most of the examples with an encrypted data field added to the space used by example.

See User Authentication, page 148 and User Access Control, page 153 for information on how to change the example security policy file to turn on user authentication or user access control.

Example Security Token File

The security token file, **`example_token.txt`**, was created from the example security policy file by using the following Admin CLI command:

as-admin> **`create security_token domain_name "exdomain" policy_file "example_policy.txt" token_file "exdomain_token.txt"`**

This generates a security token file named `exdomain_token.txt`, which is based upon the security settings for the security domain named `exdomain` in the security policy file `.xample_policy.txt`. The example security token file contains a copy of the following from the security policy file:

1. The metaspace access list. This restricts the metaspaces that can be connected to using this security token file.
2. The transport security setting. This determines the type of security to be applied when making transport connections within a metaspace.
3. The public certificate of the security domain. This is used to establish a secure transport connection with the security domain controller.

This security token file can be used by the examples to connect to a security domain controller for the default metaspace named `ms`. A connection to the security domain controller for a metaspace is established when an example tries to connect to the metaspace using the example security token file.

The security domain controller for the `ms` metaspace takes care of ensuring that security is applied to any example that connects to the metaspace using the example security token file. Other than using a security token file when connecting to a metaspace, there is nothing else that an application needs to do to have security applied to it, unless the default mechanism for obtaining user credentials for user authentication is not desired.

See ASUserAuthenticator, page 216 for an example of how to use the security API callback mechanism to override how user credentials can be retrieved for user authentication.

ASOperations

The first example program you should examine is the `ASOperations` example. `ASOperations` defines and joins (as a leech) a space called `myspace` that has a very basic definition consisting of three fields.

You should launch more than one instance of ASOperations at the same time to see the different instances interacting with each other over the same space (for example, you can try the lock or compare and put operations).

Overview

`ASOperations` is a basic example that demonstrates how to connect to a metaspace, define and join a space, and invoke the basic methods of the API for performing operations on the space, such as put and get. `ASOperations` is the first example that you should examine and run to get familiar with ActiveSpaces. `ASOperations` is also sometimes run with other examples, so it is a good idea to become familiar with it first.

The following are some of the features of ActiveSpaces that ASOperations exercises:

- **Metaspace** Connect, close, begin transaction, commit transaction, rollback transaction.
- **Space** Define, close, get, put, putAll, take, compare the previous tuple value and if still the same do a put, get the number of entries in a space, display the entries in the space, lock, unlock.

Starting ASOperations

The following examples show how to invoke `ASOperations` for each of the API sets:

Java Invocation

```
java ASOperations -metaspace examplems -member_name op
```

C Invocation

```
ASOperations -metaspace examplems -member_name op
```

.NET Invocation

```
AS_HOME\examples\dotnet\ASOperations.exe -metaspace examplems
-member_name op
```

`ASOperations` by default joins a space (`myspace`) as a leech. If you try to run `ASOperations` using the default settings, you will get the following message asking you to start a seeder:

```
waiting for the space to be ready...
please start 1 seeder nodes (ex. as-agent)
```

You have two options:

- Start an as-agent before running `ASOperations`.
- Run `ASOperations` and specify `-role seeder` as a command line option.

Starting ASOperations With Security

The following is an example of the command line options that you can use when starting `ASOperations` to have it join the security domain `exdomain` and to use a space with an additional encrypted field:

```
-discovery tcp://127.0.0.1:50000 -member_name op -security_token
exdomain_token.txt -encrypt_field
```

These command line options start `ASOperations` using the default metaspace named `ms` and allow it to connect to a security domain controller that has been started using the example security policy file `example_policy.txt`. When you use `ASOperations` to put data into the space, you will see the following additional prompt:

```
Put: Enter the value to be encrypted (string):
```

The value you enter at this prompt is stored into a field named `secure_value` that is defined as a string. Any string value that you enter in response to this prompt will be encrypted when it is stored in the space or when it is persisted. See the section Data Encryption, page 146 in Chapter 4, “Implementing ActiveSpaces Security” for information on encrypting data fields.

Using ASOperations

After `ASOperations` initializes, the following options are displayed, which allow you to perform actions on the space or metaspace:

b - begin transaction

c - commit transaction

r - rollback transaction

`s` - Displays the number of entries in the space.

`br` - Displays the entries in the space. If no filter is specified, all entries in the space will be displayed.

`p` - put a tuple into a space

`pm` - put a list of tuples into a space

`cp` - do a compare and put of a tuple into a space

`g` - get a tuple from a space

`t` - take a tuple from a space

`l` - lock a tuple in a space

`u` - unlock a tuple in a space

`q` - exit the example

Some options prompt you for a key. The key is an integer value.

You might also be prompted for a value. A value is a string. When prompted for a value, if you press **Return** instead of entering a string, the value is considered null. For example, the following key/value pairs are valid:

1, ant

2, bat

3,

4, dog

5, eel

Using ASOperations with Shared-Nothing Persistence

See Shared-Nothing Persistence, page 189 for detailed information on how to run `ASOperations` using the built-in shared-nothing persistence feature.

ASBatchOperations

`ASBatchOperations` demonstrates how to perform batch operations on spaces.

Overview

`ASBatchOperations` is similar to `ASOperations`, except that instead of getting or putting one data item at a time, you can get or put a list of data items at one time.

The following are some of the features of ActiveSpaces that ASBatchOperations exercises:

- **Metaspace** connect, close
- **Space** define, getAll, putAll, putAll and lockAll, compareAndPutAll, takeAll, lockAll, unlockAll

Starting ASBatchOperations

The following examples show how to invoke ASBatchOperations for each of the API sets:

Java Invocation

```
java ASBatchOperations -metaspace examplems -member_name batchop
```

C Invocation

```
ASBatchOperations -metaspace examplems -member_name batchop
```

.NET Invocation

AS_HOME\**examples\\dotnet\\ASBatchOperations.exe -metaspace examplems -member_name batchop**

`ASBatchOperations` by default joins a space (`myspace`) as a leech. If you try to run `ASBatchOperations` using the default settings, you will get the following message asking you to start a seeder:

```
waiting for the space to be ready...
please start 1 seeder nodes (ex. as-agent)
```

You have two options:

- Start an as-agent before running `ASBatchOperations`.
- Run `ASBatchOperations` and specify `-role seeder` as a command line option.

Starting ASBatchOperations With Security

The following example shows the command line options that you can use when starting `ASBatchOperations` to have it join the security domain `exdomain` and use a space with an additional encrypted field:

```
-discovery tcp://127.0.0.1:50000 -member_name batchop
-security_token exdomain_token.txt -encrypt_field
```

These command line options start `ASBatchOperations` using the default metaspace named `ms` and allow it to connect to a security domain controller that has been started using the example security policy file `example_policy.txt`.

When you use `ASBatchOperations` to put data into the space, you see the following additional prompt:

```
Put All: Enter the value to be encrypted (string):
```

The value you enter at this prompt is stored into a field named `secure_value`, which is defined as a string. Any string value that you enter in response to this prompt is encrypted when it is stored in the space or when it is persisted.

See the section Data Encryption, page 146 in Chapter 4., "Implementing ActiveSpaces Security" for information on encrypting data fields.

Using ASBatchOperations

After `ASBatchOperations` initializes, the following options are displayed from the main loop of the program and allow you to perform actions on the space or metaspace:

`p` - put all, allows you to enter a sequence of data to be put into the space

`pl` - put all and lock the space entries, same as **`p`** option except the space entries are locked.

Locking means the entries cannot be modified but they can still be read. An entry remains locked until it is specifically unlocked.

`g` - get all, allows the user to enter a list of keys for entries to be retrieved from the space, retrieves the entries, then displays the entries.

`t` - take all, allows the user to enter a list of keys for entries to be removed from the space, removes the entries, then displays the removed entries.

`l` - lock all, allows the user to enter a list of keys for entries to be locked in a space, locks the entries, then displays the locked entries.

`q` - exit the program

After each of the above options is performed, a list of suboptions is displayed to continue to allow you to work with your list of entries. Suboptions are provided for all of the main options, except **`t`** (take all), because the list of entries that were originally taken from the space no longer exists in the space.

The following suboptions are available to be invoked on the list of space entries generated while invoking the main options:

`r` - remove all entries in your list from the space

`l` - lock all entries in your list in the space

`u` - unlock all entries in your list in the space

`p` - put new values for your existing entries by doing a `compareAndPutAll()`

`s` - Exit the suboptions menu

ASChat

ASChat demonstrates how to build a peer-to-peer distributed application using ActiveSpaces: a multi-user chat room.

Overview

ASChat exchanges messages with another running instance of ASChat. You should run two instances of ASChat to see the exchange of messages in the space.

ASChat does not use the same default space as the other examples, but defines its own simple space (ASChat), which has the following two fields:

- A mandatory key field called name of type String
- An optional field called message of type String

Each instance of ASChat that you run uses a single tuple to store the user's name and message for 'putting' into the space. The program loops, waiting for the user to enter a new message. The new message then replaces the old message in the tuple before the tuple is put into the space again.

A listener callback is implemented for puts and takes on the space. When another ASChat instance puts or takes messages to or from the space, the listener callback is invoked and information about the put or take is displayed.

One interesting thing to note about how ASChat is implemented is that it uses the Admin interface to define the ASChat space. Most of the other examples define a space using the SpaceDef interface.

Starting ASChat

The following examples show how to invoke `ASChat` for each of the API sets:

Java Invocation

```
java ASChat -metaspace examplems
```

C Invocation

```
AS_HOME\examples\dotnet\ASChat.exe -metaspace examplems
```

.NET Invocation

```
AS_HOME\examples\dotnet\ASChat.exe -metaspace examplems
```

`ASChat` is hard-coded to join the `ASChat` space as a seeder. So it is not necessary to run other instances of as-agent for the `ASChat` space to have seeders.

Starting ASChat With Security

The following example shows the command line options that you can be use when starting `ASChat` to have it join the security domain `exdomain`:

```
-discovery tcp://127.0.0.1:50000 -security_token exdomain_token.txt
```

These command line options start `ASChat` using the default metaspace named `ms` and allow it to connect to a security domain controller that has been started using the example security policy file `example_policy.txt`.

Using ASChat

After `ASChat` initializes, the following options are displayed from the main loop of the program, which allow you to perform actions on the space or metaspace:

```
Please enter your name:
```

After you enter your name, the following message is displayed:

```
[<your name>] has entered the chat room!
```

Any message you now enter into the command window is displayed to all members of the space in the form:

```
[<your name>] <your message>
```

Type **`quit`** in the command window to exit the program.

ASQuery (Java Only)

The `ASQuery` example is provided only in a Java version.

Overview

`ASQuery` is a good example to look at to learn about how to compose search filter strings for use when browsing a space. `ASQuery` first populates the default space (`myspace`) with 1,000,000 tuples, and then displays the amount of time it took to populate the space.

After the space is populated, `ASQuery` creates browsers using different searche filters, scans the entries of the space using each browser, and displays the browse statistics. `ASQuery` adds indexes to the default space (myspace) in order to speed up the filtering of data when processing queries. Indexes are added for the 'key' and 'value' fields.

Starting ASQuery

The following example shows how to invoke `ASQuery` in Java.

Java Invocation

```
java ASQuery -metaspace examplems -member_name query
```

`ASQuery` by default joins a space (myspace) as a leech. If you try to run `ASQuery` using the default settings, you will get the following message asking you to start a seeder:

```
waiting for the space to be ready...
please start 1 seeder nodes (ex. as-agent)
```

You have two options:

- Start an as-agent before running `ASQuery`.
- Run `ASQuery` and specify `-role seeder` as a command line option.

Starting AS Query With Security

The following example shows the command line options that you can use when starting `ASQuery` to have it join the security domain `exdomain` and to use a space with an additional encrypted field:

```
-discovery tcp://127.0.0.1:50000 -member_name query -security_token
exdomain_token.txt -encrypt_field
```

These command line options start `ASQuery` using the default metaspace named `ms` and allow it to connect to a security domain controller that has been started using the example security policy file `example_policy.txt`. `ASQuery` will then additionally populate an encrypted field named `secure_value`. Since encrypted fields cannot be queried, additional queries on the encrypted field have not been added to the example. See the section Data Encryption, page 146 in Chapter 4., “Implementing ActiveSpaces Security” for information on encrypting data fields.

ASPaint (Java and .NET Only)

`ASPaint` is an implementation of a shared whiteboard. This example is provided only for Java and .NET.

Overview

`ASPaint` showcases various features of ActiveSpaces, but is not designed to be used as a template for coding best practices.

To get a full understanding of `ASPaint`, launch more than one instance of the program so that you can see the eventing aspect of ActiveSpaces in action.

You can also check the seeded checkbox and see the distribution algorithm working in a visual manner. Also try adding or removing seeders from the space to see redistribution at work. You can also try filter ss and remote space invocation.

Starting ASPaint

Java Invocation

In Java, you can start `ASPaint` in two ways:

1. By entering:

   `java com.tibco.paint.AsPaintApp`

2. By entering

   `java -jar ASPaint.jar`

Starting `ASPaint` using the `-jar` option requires that you start `ASPaint` from the *AS_HOME*`/examples/java` directory, because the manifest in `ASPaint.jar` looks for `lib/as-common.jar` in the current directory.

.NET Invocation

AS_HOME`\`**`examples\dotnet\ASPaint.exe`**

Using ASPaint

When you invoke `ASPaint`, you are prompted to enter the following metaspace connection attributes:

- Member name

- Metaspace name
- Discovery URL
- Listen URL
- Remote discovery URL

Enter these in the dialog window that appears.

`ASPaint` displays a window that allows you to draw into a "whiteboard." The interface is similar to the MS Paint interface.

`ASPaint` creates a space named `paint`. The `ASPaint` example requires that two seeders be connected to the `paint` space before you can use the tools to draw or enter text on the whiteboard. Therefore, you should start a second instance of `ASPaint`.

When both instances of `ASPaint` are running, what you draw on the whiteboard of one `ASPaint` instance is automatically reflected on the whiteboard of the second `ASPaint` instance.

If you check the **Seeded** check box, the command window indicates when a member joins or leaves the metaspace.

ASPersistence

The ASPersistence example demonstrates how to implement the external shared-all persistence interface.

Overview

With shared-all persistence, all nodes in a space persist their data into a single data store. To use shared-all persistence with ActiveSpaces, you must implement the Persister interface. The implemented persister is then used to store and retrieve data from the data store.

The ASPersistence example provides two example implementations of the Persister interface.

- The first implementation, SimplePersister, uses an in-memory HashMap as the data store.
- The second implementation, ASPersister, uses a database as the data store.

The onWrite method of the persister is invoked when data is put into the space.

The onRead method of the persister is invoked when data is read from the data store back into the space.To see data being read back out of the data store, the space needs to be defined with a capacity so that when the capacity is exceeded, the least recently used data in the space is ejected from the space. When data that has been previously ejected from the space is read back into the space, you will see the onRead method of the persister invoked.

The ASPersistence examples does not provide the means to read or write data into the space. Therefore, you should run the ASOperations example program along with the ASPersistence example.

Starting ASPersistence

The following examples show how to invoke ASPersistence for each of the API sets:

Java Invocation

```
java ASPersistence -metaspace examplems -member_name sharedall
-capacity 2 -eviction_policy lru
```

C Invocation

```
ASPersistence -metaspace examplems -member_name sharedall -capacity
2 -eviction_policy lru
```

.NET Invocation

```
AS_HOME\examples\dotnet\ASPersistence.exe -metaspace examplems
-member_name sharedall -capacity 2 -eviction_policy lru
```

ASPersistence does not use the default space but uses a basic space definition and sets the persistence type of the space to 'shared-all'. ASPersistence uses a default space name of shared_all_persisted to identify this slightly different space so that you will not get a space definition mismatch with the other examples which may already be running.

The following examples indicate how to invoke ASOperations to work in conjunction with the ASPersistence example so that you can see the various methods of the persister being invoked.

Java Invocation for ASOperations with ASPersistence

```
java ASOperations -metaspace examplems -role seeder -persistence
shared_all -capacity 2 -eviction_policy lru
```

C Invocation for ASOperations with ASPersistence

```
ASOperations -metaspace examplems -role seeder -persistence
shared_all -capacity 2 -eviction_policy lru
```

.NET Invocation

```
AS_HOME\examples\dotnet\ASOperations.exe -metaspace examplems
-space shared_all_persisted -persistence share_all -capacity 2
-eviction_policy lru
```

Starting ASPersistence With Security

The following example shows the command line options that you can be use when starting ASPersistence to have it join the security domain exdomain:

```
-discovery tcp://127.0.0.1:50000 -member_name sharedall -capacity 2
-eviction_policy lru -security_token exdomain_token.txt
```

To invoke ASOperations to work in conjunction with the ASPersistence example when it has been started with security, use the following command line options:

```
-discovery tcp://127.0.0.1:50000 -role seeder -persistence
shared_all -capacity 2 -eviction_policy lru  -security_token
exdomain_token.txt
```

These command line options start `ASPersistence` and `ASOperations` using the default metaspace name ms and allow them to connect to a security domain controller that has been started using the example security policy file example_policy.txt. Using ASPersistence

Because encrypted fields cannot be persisted with shared all persistence, you should not try to use the `-encrypt_field` command line option when starting `ASPersistence` or `ASOperations` for this scenario.

Use `ASOperations` to put the following data into the space:

```
1, any
2, bat
3, cat
```

Each time a put is done using `ASOperations`, you will see the `onWrite` method of the Persister interface invoked in the command window where the `ASPersistence` example was started.

Since we have defined the space to have a capacity of 2 but we have put three entries into the space, the first entry will be evicted from the space. To verify this, enter **`br`** at the command prompt displayed by the `ASOperations` example and press **Enter** when prompted for a filter. The last two entries put into the space should be displayed.

Now use `ASOperations` to get the entry with an index of 1 back into the space. Enter **`g`** for get in the command window of `ASOperations`. Enter **`1`** for the key and press **Enter**. You will see the `onRead` method of the `Persister` interface invoked in the `ASPersistence` command window.

Enter **`quit`** in the command window of the `ASPersistence` example to shut down the application. The `onClose` method of the `Persister` interface is invoked when `ASPersistence` is stopped.

Using ASPersistence with a Database

You can use the `ASPersistence` example with a database for the data store. The default database supported in the code is PostgreSQL. The class `MySQLConnection` is also provided to show you how easy it is to build on the current `ASPersistence` example to support a different database.

Complete these steps to use the `ASPersistence` example with a database:

1. Ensure your database server is running.

2. Create a database for use with the `ASPersistence` example.
3. Modify your *CLASSPATH* to include the JDBC driver jar file for your database
4. Modify the file `ASPersistence.java` as follows:
   a. Uncomment the following import statement at the beginning of the file:
      ```
      //import persistence.ASPersister;
      ```
   b. Comment out the following import statement at the beginning of the file:
      ```
      import persistence.SimplePersister;
      ```
   c. Comment out the following lines in the `ASPersistence` constructor:
      ```
      persister = space.setPersister(new
      SimplePersister(spaceDef));
      ```
   d. Uncomment the following line in the ASPersistence constructor:
      ```
      //persister = space.setPersister(new ASPersister(metaspace));
      ```
5. Modify the file `ASPersister.java` as follows:
   a. Ensure that the values for *DB_NAME*, *DB_HOST*, *USER_NAME*, and *USER_PASS* are set appropriately for your database.
   b. Ensure that *JDBC_DRIVER* and *JDBC_URL* are set appropriately for your database.
   c. In the `onOpen` method, ensure that the appropriate database connection class is being instantiated.
6. Rebuild `Examples.jar.`
7. Invoke the `ASPersistence` and `ASOperations` examples as described in Starting ASPersistence, page 185, and ensure that your *CLASSPATH* has been set appropriately as described in Step 3.

Shared-Nothing Persistence

This example shows how to make an application be its own persister using the internal shared-nothing persistence feature.

Overview

Shared-nothing persistence can be demonstrated using the `ASOperations` example. With shared-nothing persistence, each seeder in the space stores its space entries locally on disk.

Using `ASOperations` for shared-nothing persistence, you will be able to see how data is recovered from disk. If all seeders in a space go down, data recovery is initiated using as-admin after the seeders are brought back up.

The following is the default location for persisting data:

data_store_path/*metaspace_name*/*space_name*/*member_name*

Where *data_store_path* defaults to your user home directory. Use the `-data_store` command line option to change the default *data_store_path* location.

Shared-nothing persistence is built into ActiveSpaces, so there is no persistence interface that you need to implement. The only thing you have to do is to define a space with a persistence type of shared-nothing.

Starting ASOperations for Shared-Nothing Persistence

The following examples show how to invoke `ASOperations` for shared-nothing persistence.

Java Invocation

```
java ASOperations -metaspace examplems -space
shared_nothing_persisted -persistence shared_nothing -role seeder
[-data_store path]
```

C Invocation

```
ASOperations -metaspace examplems -space shared_nothing_persisted
-persistence shared_nothing -role seeder [-data_store path]
```

.NET Invocation

```
AS_HOME\examples\dotnet\ASOperations.exe -metaspace examplems
-persistence shared_nothing -role seeder [-data_store path]
```

When using `ASOperations` for shared-nothing persistence, you must start `ASOperations` as a seeder so that data will be stored on the node running `ASOperations`. The basic space definition is used, but the persistence type of the space is set to `shared-nothing`. To identify this slightly different space, `ASOperations` uses a default space name of “shared_nothing_persisted” when it detects that shared-nothing persistence has been specified. This is done to prevent conflicts with other examples which may already be running and using the default space definition.

Starting as-agents for Shared-Nothing Persistence

You can also run as-agents to provide additional persistence nodes for the space. The following are examples of how to invoke as-agent to work in conjunction with `ASOperations` when invoked for shared-nothing persistence:

Java Invocation

```
java [-d64] -jar AS_HOME/lib/as-agent.jar -metaspace examplems
[-data_store path]
```

C Invocation

```
AS_HOME/bin/as-agent.exe -metaspace examplems [-data_store path]
```

.NET Invocation

```
AS_HOME\bin\Agent.NET.exe -metaspace examplems [-data_store path]
```

Starting ASOperations for Shared-Nothing Persistence With Security

The following example shows the command line options that you can use when starting `ASOperations` for shared-nothing persistence to have it join the security domain `exdomain`:

```
-persistence shared_nothing -role seeder [-data_store path]
-security_token exdomain_token.txt -encrypt_field
```

These command line options start `ASOperations` using the default metaspace named `ms` and allow it to connect to a security domain controller that has been started using the example security policy file `example_policy.txt`.

When you use `ASOperations` to put data into the space, you will see the following additional prompt:

```
Put: Enter the value to be encrypted (string):
```

The value you enter at this prompt is stored into a field named `secure_value`, which is defined as a string. Any string value that you enter in response to this prompt is encrypted when it is stored in the space and when it is persisted.

See the section Data Encryption, page 146 in Chapter 4., "Implementing ActiveSpaces Security" for information on encrypting data fields.

Using Shared-Nothing Persistence

One of the interesting things to see demonstrated for shared-nothing persistence, is how recovery occurs when all of the seeders for a space go down and then come back up.

Complete these steps to see how a space is recovered when using `ASOperations` for shared-nothing persistence:

1. Start `ASOperations` as described above.

   We will refer to this process as `ASOperations1`.
2. Start another `ASOperations` as described above.

   We will refer to this process as `ASOperations2`.
3. Put several entries into the space using either `ASOperations1` or `ASOperations2`.
4. Use the **`br`** command with `ASOperations2` to verify the entries in the space.
5. Exit the `ASOperations1` process (enter **`quit`** at the command line).
6. Use the **`br`** command with **`ASOperations2`** to verify that the entries have been redistributed to `ASOperations2`.
7. Exit the `ASOperations2` process (enter **`quit`** at the command line).
8. Restart ASOperations2.

   You will see a message indicating that the space is not ready and instructing you to issue a `recover space` admin command.
9. Start as-admin and connect to the metaspace:

   `connect name "examplems"`
10. Issue the following recover command in as-admin:

    `recover space shared_nothing_persisted with data`
11. When you see the message indicating the persisted data has been loaded in the window used to start `ASOperations2`, use the **`br`** command with `ASOperations2` to verify that the entries in the space have been restored.

ASRequestReplyServer and ASRequestReplyClient

These examples show how two spaces can work together as a reply server and reply client.

Overview

The `ASRequestReplyClient` and `ASRequestReplyServer` examples work together to demonstrate how two spaces can be used by applications to communicate in a request and reply manner. The two spaces are named `request` and `reply`.

In this example, `ASRequestReplyServer` defines the spaces and `ASRequestReplyClient` joins the already defined spaces. This is different from most of the other examples where the space is always defined.

Another interesting thing to note about how `ASRequestReplyServer` is implemented is that it uses the Admin interface to define the 'request' and 'reply' spaces. Most of the other examples define a space using the SpaceDef interface.

`ASRequestReplyClient` loops, putting messages into the 'request' space. When it is notified of messages in the 'reply' space, `ASRequestReplyClient` takes the replies from the 'reply' space.

For every 100 replies received, `ASRequestReplyClient` prints a message indicating the number of requests that have been sent and the number of replies received.

`ASRequestReplyServer` periodically polls for messages on the 'request' space. When a message is put into the 'request' space, `ASRequestReplyServer` takes it from the space. The name of the ASRequestReplyServer is added to the original message as the responder to the request. This slightly modified message is then put into the 'reply' space. For every 100 requests received, `ASRequestReplyServer` prints a message indicating the number of requests it has processed.

The `request` space has a definition consisting of two fields:

- A mandatory key field called `id` of type 'integer'
- A mandatory key field called `requester` of type 'string'

The `reply` space has a definition consisting of three fields:

- A mandatory key field called `id` of type 'integer'
- A mandatory key field called `requester` of type 'string'
- A mandatory field called `responder` of type 'string'

`ASRequestReplyServer` is a seeder for the `request` space and a leech for the `reply space`.

`ASRequestReplyClient` is a leech for the `request` space and a seeder for the `reply` space.

Starting ASRequestReplyServer

You should start `ASRequestReplyServer` first. The following examples indicate how to invoke `ASRequestReplyServer` for each of the APIs.

Java Invocation

```
java ASRequestReplyServer -metaspace examplems -member_name
rrserver
```

C Invocation

```
ASRequestReplyServer -metaspace examplems -member_name rrserver
```

.NET Invocation

```
ASRequestReplyServer -metaspace examplems -member_name rrserver
```

You should start `ASRequestReplyClient` second. The following examples indicate how to invoke `ASRequestReplyClient` for each of the APIs.

Starting ASRequestReplyClient

The following examples indicate how to invoke `ASRequestReplyClient` for each of the APIs.

Java Invocation

```
java ASRequestReplyClient -metaspace examplems -member_name
rrclient
```

C Invocation

```
ASRequestReplyClient -metaspace examplems -member_name rrclient
```

.NET Invocation

```
ASRequestReplyClient -metaspace examplems -member_name rrclient
```

Starting ASRequestReplyServer and ASRequestReplyClient with Security

The following example shows the command line options that you can use when starting `ASRequestReplyServer` to have it join the security domain `exdomain`:

```
-discovery tcp://127.0.0.1:50000 -member_name rrserver
-security_token exdomain_token.txt
```

To invoke `ASRequestReplyClient` to work in conjunction with the `ASRequestReplyServer` example when it has been started with security, use the following command line options:

```
-discovery tcp://127.0.0.1:50000 -member_name rrclient
-security_token exdomain_token.txt
```

These command line options start `ASRequestReplyServer` and `ASRequestReplyClient` using the default metaspace name `ms` and allow them to connect to a security domain controller that has been starting using the example security policy file `example_policy.txt`.

Remote Space Invocation: InvokeClient

The example directory for each API set also contains an example showing how to implement and use remote space invocation.

Overview

The `InvokeClient` example demonstrates how to invoke a user defined method on:

- An entry associated with a given key in a space
- Only on the invoking member of a space
- On all seeders in a space
- On all members of a space

Implementations of the `Invocable` and `MemberInvocable` interfaces are provided.

Starting InvokeClient

The following examples show how to invoke `InvokeClient` for each API set.

Java Invocation

```
java InvokeClient -metaspace examplems -member_name invoke
```

C Invocation

```
InvokeClient -metaspace examplems -member_name invoke
```

.NET Invocation

```
AS_HOME/examples/dotnet/InvokeClient.exe -metaspace examplems
-member_name invoke
```

Starting InvokeClient with Security

The following example shows the command line options that you can use when starting InvokeClient to have it join the security domain `exdomain` and to use a space with an additional encrypted field:

```
-discovery tcp://127.0.0.1:50000 -member_name invoke
-security_token exdomain_token.txt -encrypt_field
```

These command line options start `InvokeClient` using the default metaspace named `ms` and allow it to connect to a security domain controller that has been started using the example security policy file `example_policy.txt`. Using the `-encrypt_field` command line option allows `InvokeClient` to connect to the same space defined when `ASOperations` is started with security and the `-encrypt_field` command line option.

Using InvokeClient

After `InvokeClient` initializes, the following options are displayed that allow you to perform actions on the space:

- **key** invokes the user defined method on the space member that holds the tuple with a key of 1.
- **self** invokes the user-defined method on the invoking member in the space (e.g. InvokeClient).
- **seeders** invokes the user-defined method on all seeders in the space.
- **members** invokes the user defined method on all members in the space.

The Invocable interface is used when the `key` option is selected. The `MemberInvocable` interface is used when the `self`, `seeders`, or `members` option is selected.

The purpose of the `key` option is to see that the user-defined method is invoked on the seeder that contains the entry with the key of 1. To see anything with the `key` option, you should start up two instances of `ASOperations`, using the `-role seeder` command line option, and put some entries into the space, making sure that you have an entry with a key of 1.

When the example is run and the `key` option is selected, you will see the-user defined method of the Invocable interface run by the seeder that contains the entry with a key of 1.

The purpose of the other options is to let you see how the user-defined implementation of `MemberInvocable` is called on different members of the space. The `InvokeClient` example joins the space as a leech. To see what happens when the user-defined method is invoked with these different options, start up two instances of `ASOperations`, using the `-role seeder` command line option, along with `InvokeClient`. In this case:

- The `self` option should cause the user-defined method to be invoked on `InvokeClient`.
- The `seeders` option should cause the user-defined method to be invoked on the other `ASOperations` instances.

- The `members` option should cause the user-defined method to be invoked on `InvokeClient` and the other `ASOperations` instances.

Overview of ASBrowser, ASEventBrowser, and ASListener

The `ASBrowser`, `ASEventBrowser`, and `ASListener` examples show how simple it is to use the space browser and listener features to iterate over the contents of a space, or to listen for events on a space. `ASBrowser` and `ASListener` do exactly the same thing, but each uses a different set of ActiveSpaces API calls to achieve its goal. `ASEventBrowser` implements an event listener. Examine all three programs to evaluate when to use a browser and when to use a listener in your own code.

These examples are also very useful when designing and debugging your own applications. Since they do not care about the definition of the space being browsed or listened to, they can be used, for example, to get a dump of the data contained in a space, or to monitor the actions taken on a space by another application.

For example, try to launch `ASOperations` and, at the same time, use `ASListener` on the space called `myspace` to see the effects of the various space operations.

Do not forget to experiment with the `timescope` attribute as well.

ASBrowser

Overview

The `ASBrowser` example creates a space browser which, by default, retrieves all of the entries in the default space 'myspace,' displays them in the command window, and then exits.

The `ASBrowser` example does not create any entries in the space. Therefore you should run `ASOperations` to create the space and put data into it so you will have something in the space to "browse." See ASOperations, page 173 for more specific information on running `ASOperations`.

The `ASBrowser` example accepts the following additional command line arguments:

`-timeout 0 | -1 |` *xxx* where: 0=no wait (default), -1=wait forever, xxx=timeout in milliseconds

`-filter "" | filter_string` where: ""=no filter (default)

`-distribution_scope all | seeded` (default: `all`)

`-timescope snapshot | all | new` (default: `snapshot`)

`-browser get | take` (default: `get`)

`-prefetch` *entries_to_prefetch* (default: 1000)

Starting ASBrowser

The following examples indicate how to invoke `ASBrowser` for each of the API sets:

Java Invocation

```
java tools.ASBrowser -metaspace examplems -member_name browser
```

C Invocation

```
ASBrowser -metaspace examplems -member_name browser
```

.NET Invocation

```
ASBrowser -metaspace examplems -member_name browser
```

Starting ASBrowser with Security

The following example shows the command line options that you can use when starting `ASBrowser` to have it join the security domain `exdomain` :

```
-member_name browser -discovery tcp://127.0.0.1:50000
-security_token exdomain_token.txt
```

These command line options start ASBrowser using the default metaspace named `ms` and allow it to connect to a security domain controller that has been started using the example security policy file `example_policy.txt`.

Using ASBrowser

By varying the additional command line arguments of `ASBrowser`, you can affect the behavior of the ASBrowser example. For example, to cause `ASBrowser` to block and wait endlessly for any new entries added to the space, start `ASBrowser` with the following additional command line options:

```
-timescope new -timeout -1
```

ASEventBrowser

Overview

The `ASEventBrowser` example creates an event browser for a space. When started, the event browser displays all of the current entries in the space and then blocks, waiting for new operations to occur on the space. When a new operation occurs, the event browser displays the information about that event.

`ASEventBrowser` does not create any entries in the space. Therefore you should run `ASOperations` to create the space and put data into it so you will have something in the space to "browse." See ASOperations, page 173 for more information on running `ASOperations`.

The `ASEventBrowser` example recognizes the following additional command line arguments:

`-timeout 0 | -1 | *xxx* where:`
0 =n o wait (default)
-1 = wait forever
`*xxx*=timeout in milliseconds`

`-filter “” | filter_string` where: “”=no filter (default)

`-distribution_scope all | seeded` (default: `all`)

`-timescope snapshot | all | new` (default: `snapshot`)

Starting ASEventBrowser

The following examples indicate how to invoke `ASEventBrowser` for each API set.

Java Invocation

```
java tools.ASEventBrowser -metaspace examplems -member_name
evtbrowser
```

C Invocation

```
ASEventBrowser -metaspace examplems -member_name evtbrowser
```

.NET Invocation

```
ASEventBrowser -metaspace examplems -member_name evtbrowser
```

Starting ASEventBrowser with Security

The following example shows the command line options that you can use when starting `ASEventBrowser` to have it join the security domain `exdomain`:

```
-member_name evtbrowser -discovery tcp://127.0.0.1:50000
-security_token exdomain_token.txt
```

These command line options start ASEventBrowser using the default metaspace named `ms` and allow it to connect to a security domain controller that has been started using the example security policy file `example_policy.txt`.

Using ASEventBrowser

By varying the additional command line arguments of the `ASEventBrowser` example, you can affect the behavior of the `ASEventBrowser` example. For example, to cause `ASEventBrowser` to only display new entries added to the space, start `ASEventBrowser` with the following additional command line options:

```
-timescope new
```

ASListener

Overview

The `ASListener` example creates a listener for a space. The listener callback is invoked whenever a put, take, expire, seed, or unseed event occurs for an entry in the space. The listener callback displays information about the event that caused it to be invoked.

The `ASListener` example does not create any entries in the space. Therefore you should run `ASOperations` to create the space and put data into it so you will have something which triggers the ASListener example. he space and put data into it so you will have something in the space to "browse." See ASOperations, page 173 for more specific information on running `ASOperations`.

The ASListener example recognizes the following additional command line arguments:

`-filter "" |` _*string* where: ""=no filter (default)

`-distribution_scope all | seeded` (default: `all`)

`-timescope snapshot | all | new | new_events` (default: `new`)

Starting ASListener

The following examples describe how to invoke `ASListener` for each of the API sets.

Java Invocation

```
java tools.ASListener -metaspace examplems -member_name listener
```

C Invocation

```
ASListener -metaspace examplems -member_name listener
```

.NET Invocation

```
ASListener -metaspace examplems -member_name listener
```

Starting ASListener with Security

The following example shows the command line options that you can use when starting `ASListener` to have it join the security domain `exdomain`:

```
-member_name listener -discovery tcp://127.0.0.1:50000
-security_token exdomain_token.txt
```

These command line options start `ASListener` using the default metaspace named ms and allow it to connect to a security domain controller that has been started using the example security policy file `example_policy.txt`.

Using ASListener

By varying the additional command line arguments of the `ASListener` example, you can affect the behavior of `ASListener`. For example, to cause `ASListener` to display all entries added to the space, start `ASListener` with the following additional command line option:

```
-timescope all
```

MetaspaceMemberMonitor

Overview

The `MetaspaceMemberMonitor` example creates a MetaspaceMemberListener for a metaspace. The listener callback is invoked whenever a member joins, leaves or changes their role in the metaspace (e.g. member vs manager). The listener callback displays information about the event which caused it to be invoked.

`MetaspaceMemberMonitor` is only interested in events that affect the members of a metaspace. If `MetaspaceMemberMonitor` connects to a metaspace with existing members, it displays information about those members when it first starts up, then display information about members as they connect to or leave the metaspace.

Starting MetaspaceMemberMonitor

The following examples show how to invoke `MetaspaceMemberMonitor` for each of the API sets.

Java Invocation

```
java tools.MetaspaceMemberMonitor -metaspace examplems -member_name
mmmonitor
```

C Invocation

```
MetaspaceMemberMonitor -metaspace examplems -member_name mmmonitor
```

.NET Invocation

```
AS_HOME\examples\dotnet\MetaspaceMemberMonitor.exe -metaspace
examplems -member_name mmmonitor
```

Starting MetaspaceMemberMonitor with Security

The following example shows the command line options that you can use when starting `MetaspaceMemberMonitor` to have it join the security domain `exdomain`.

```
-member_name mmmonitor -discovery tcp://127.0.0.1:50000
-security_token exdomain_token.txt
```

These command line options start MetaspaceMemberMonitor using the default metaspace named `ms` and allow it to connect to a security domain controller that has been started using the example security policy file `example_policy.txt`.

Using MetaspaceMemberMonitor

To see output from `MetaspaceMemberMonitor`, start `MetaspaceMemberMonitor` then start as-agent, and then start other example programs such as `ASChat` or `ASOperations` to see what is displayed when each program connects to or leaves the metaspace.

Enter **`quit`** in the command window to exit the program.

SpaceDefMonitor

Overview

The `SpaceDefMonitor` example creates a SpaceDefListener for a metaspace. The listener callback is invoked whenever a space is defined or dropped from the metaspace. The listener callback displays information about the event that caused it to be invoked.

`SpaceDefMonitor` is only interested in events that affect the definition of spaces in a metaspace. If `SpaceDefMonitor` connects to a metaspace with existing spaces, it displays information about those spaces when it first starts up, and then displays information about spaces as they are defined in or dropped from the metaspace.

Starting SpaceDefMonitor

The following examples show how to invoke `SpaceDefMonitor` for each of the API sets.

Java Invocation

```
java tools.SpaceDefMonitor -metaspace examplems -member_name
sdmonitor
```

C Invocation

```
SpaceDefMonitor -metaspace examplems -member_name sdmonitor
```

.NET Invocation

```
AS_HOME/examples/dotnet/SpaceDefMonitor.exe -metaspace examplems
-member_name sdmonitor
```

Starting SpaceDefMonitor with Security

The following example shows the command line options that you can use when starting `SpaceDefMonitor` to have it join the security domain `exdomain`.

```
-member_name sdmonitor -discovery tcp://127.0.0.1:50000
-security_token exdomain_token.txt
```

These command line options start `SpaceDefMonitor` using the default metaspace named `ms` and allow it to connect to a security domain controller that has been started using the example security policy file `example_policy.txt`.

Using SpaceDefMonitor

To see output from `SpaceDefMonitor`, start `SpaceDefMonitor`, and then start other example programs that define new spaces, such as `ASChat` or `ASOperations` to see what is displayed when each program starts up. Exit all example programs, then start up the Admin CLI and enter the following:

```
as-admin> connect name “examplems” membername “admin”
as-admin> show spaces
```

For each space displayed by `show spaces`, do the following and watch what is displayed in the SpaceDefMonitor command window:

```
as-admin> drop “ASChat”
as-admin> drop “myspace”
as-admin> quit
```

Enter **`quit`** in the SpaceDefMonitor command window to exit the program.

SpaceStateMonitor

Overview

The `SpaceStateMonitor` example creates a SpaceStateListener for a metaspace. The listener callback is invoked whenever the state of a space changes (e.g., `READY`, `FAILED`). The listener callback displays information about the event which caused it to be invoked.

`SpaceStateMonitor` is only interested in events which affect the state of spaces in a metaspace. If SpaceStateMonitor connects to a metaspace with existing spaces, it will display information about those spaces when it first starts up, then display information about spaces as their state changes.

Starting SpaceStateMonitor

The following examples show how to invoke `SpaceStateMonitor` for each of the API sets.

Java Invocation

```
java tools.SpaceStateMonitor -metaspace examplems -member_name
ssmonitor
```

C Invocation

```
SpaceStateMonitor -metaspace examplems -member_name ssmonitor
```

.NET Invocation

AS_HOME/**examples/dotnet/SpaceStateMonitor.exe -metaspace examplems -member_name ssmonitor**

You can listen to space state changes for a specific space by using the -space command line option.

Starting SpaceStateMonitor with Security

The following example shows the command line options that you can use when starting `SpaceStateMonitor` to have it join the security domain `exdomain`:

```
-member_name ssmonitor -discovery tcp://127.0.0.1:50000
-security_token exdomain_token.txt
```

These command line options start `SpaceStateMonitor` using the default metaspace named `ms` and allow it to connect to a security domain controller that has been started using the example security policy file `example_policy.txt`.

Using SpaceStateMonitor

To see output from `SpaceStateMonitor`, start `SpaceStateMonitor`, and then start other example programs that define new spaces, such as `ASChat` or `ASOperations`, to see what is displayed when each program starts up. Exit all example programs and watch how the space state changes in the `SpaceStateMonitor` command window.

Enter **`quit`** in the `SpaceStateMonitor` command window to exit the program.

SpaceMemberMonitor

Overview

The `SpaceMemberMonitor` example creates a SpaceMemberListener for a space. The listener callback is invoked whenever a member joins, leaves, or changes their role in the space (e.g. leech vs seeder). The listener callback displays information about the event which caused it to be invoked.

`SpaceMemberMonitor` is only interested in events that affect the members of an existing space. It does not create the space itself.

If `SpaceMemberMonitor` connects to a metaspace with existing members, it displays information about those members when it first starts up, and then displays information about members as they connect to or leave the space.

Starting SpaceMemberMonitor

The following examples show how to invoke `SpaceMemberMonitor` for each of the API sets.

Java Invocation

```
java tools.SpaceMemberMonitor -metaspace examplems -space myspace
-member_name smmonitor
```

C Invocation

```
SpaceMemberMonitor -metaspace examplems -space myspace -member_name
smmonitor
```

.NET Invocation

AS_HOME`/examples/dotnet/SpaceMemberMonitor.exe -metaspace examplems -space myspace -member_name smmonitor`

If the space name is not specified on the command line, an exception will be thrown.

Starting SpaceMemberMonitor with Security

The following example shows the command line options that you can use when starting `SpaceStateMonitor` to have it join the security domain `exdomain`.

```
-member_name smmonitor -discovery tcp://127.0.0.1:50000
-security_token exdomain_token.txt
```

These command line options start `SpaceMemberMonitor` using the default metaspace named `ms` and allow it to connect to a security domain controller that has been started using the example security policy file `example_policy.txt`.

Using SpaceMemberMonitor

To see output from `SpaceMemberMonitor`, start `SpaceMemberMonitor` and then start as-agent or other example programs, such as `ASOperations`, to see what is displayed when each program connects to or leaves the space.

Enter **`quit`** in the command window to exit the program.

ASDomainController

Overview

You can use the `ASDomainController` example as a security domain controller when running the rest of the ActiveSpaces examples with security. `ASDomainController` uses a security policy file to connect to a metaspace and become a security domain controller for the metaspace.

The `ASDomainController` example does not connect to any spaces or provide any functionality besides what is needed to act as a security domain controller. Once `ASDomainController` successfully connects to a metaspace, it loops, waiting for user input to tell it to shut down.

Starting ASDomainController

The following examples show how to invoke `ASDomainController` for each of the API sets. If you do not start the example from the security subdirectory of the examples, be sure to enter the full path of the `example_policy.txt file`.

Java Invocation

```
java security.ASDomainController -discovery tcp://127.0.0.1:50000
-listen
tcp://127.0.0.1:50000 -security_policy example_policy.txt
```

C Invocation

```
ASDomainController -discovery tcp://127.0.0.1:50000 -listen
tcp://127.0.0.1:50000 -security_policy example_policy.txt
```

.NET Invocation

```
AS_HOME/examples/dotnet/ASDomainController.exe -discovery
tcp://127.0.0.1:50000
-listen tcp://127.0.0.1:50000 -security_policy example_policy.txt
```

Using ASDomainController

Once the `ASDomainController` example starts, you can then start other examples using security. `ASDomainController` acts as the security domain controller for the metaspace it connects to. In the above example invocation, the default metaspace named `ms` is connected to by `ASDomainController`. To have `ASDomainController` act as the security domain controller for a different metaspace, use the `-metaspace` command line argument to specify the name of the desired metaspace.

Once `ASDomainController` has started, it will displayed a short menu of options. The following options are allowed:

`h` - display command line help information

`q` - exit `ASDomainController`

User Authentication Example

Using the example security policy file, `example_policy.txt`, you can easily enable user authentication using your local operating system for authentication. When user authentication is enabled, any of the examples, which are started with security using the security token file `exdomain_token.txt` will prompt the user for their domain name, user name and password when trying to connect to a metaspace in the 'exdomain' security domain.

To enable user authentication for use with the examples:

1. Change directory to the security subdirectory of the examples.
2. Open the file `example_policy.txt` with a text editor.
3. Locate the following line:
   ```
   authentication=none
   ```
4. Change it to the following to enable system level user authentication:
   ```
   authentication=userpwd;source=system;service=login;
   hint=SystemLoginInformation
   ```
5. Save the file.

If you have only changed the user authentication setting in the security policy file, it should not be necessary to regenerate the security token file. What is entered for the hint is displayed to the user to give them an indication of which login username and password to enter.

To run an example with user authentication:

1. Stop any examples that were started with security for the `exdomain` security domain.

2. Restart any security domain controllers, for the `exdomain` security domain, using the modified security policy file.
3. Start your example as you normally would start it with security.

When your example tries to connect to the metaspace, you will first be asked to enter your system login domain, username, and password. If your user information cannot be authenticated against your operating system, your connection to the metaspace will be denied.

ASUserAuthenticator

Overview

When using user authentication, the default ActiveSpaces behavior of prompting users to enter their authentication information may not be adequate. Therefore, TIBCO ActiveSpaces provides a callback mechanism that allows you to customize how user authentication information is retrieved for your users.

The `ASUserAuthenticator` example demonstrates how to use the user authentication callback mechanism to implement your own functionality for retrieving user authentication information. However, you must first have configured the example security policy file to enable user authentication as described in User Authentication Example, page 214. After `ASUserAuthenticator` successfully connects to a metaspace, it loops, waiting for user input to tell it to shut down.

Starting ASUserAuthenticator

The following examples show how to invoke `ASUserAuthenticator` for each of the API sets. If you do not start the example from the security subdirectory of the examples, be sure to enter the full path of the `exdomain_token.txt` file.

Java Invocation

```
java security.ASUserAuthenticator -discovery tcp://127.0.0.1:50000
-security_token exdomain_token.txt
```

C Invocation

```
ASUserAuthenticator -discovery tcp://127.0.0.1:50000
-security_token exdomain_token.txt
```

.NET Invocation

```
AS_HOME/examples/dotnet/ASUserAuthenticator.exe -discovery
tcp://127.0.0.1:50000 -security_token exdomain_token.txt
```

Using ASUserAuthenticator

Once the `ASUserAuthenticator` example starts, you will see the following prompts:

```
Enter user name and password for SystemLoginInfo
```

```
Login Domain:
Login User Name:
Login Password:
```

You will notice in this example that the password you enter is echoed back to you. `ASUserAuthenticator` is only an example and uses the input mechanisms provided by the implementation language. Various software packages are available from third-party vendors which you can use to not echo what is being input by the user.

`ASUserAuthenticator` connects to the default metaspace named `ms`. Once `ASUserAuthenticator` has started, it will displayed a short menu of options. The following options are allowed:

`h` - display command line help information

`q` - exit `ASUserAuthenticator`

User Access Control Example

User access control works in conjunction with user authentication. To get user access control to work, you first have to enable user authentication as described in the User Authentication Example, page 214.

The example security policy file `example_policy.txt` has a predefined user named `user1` who has been configured to have the following permissions for all spaces in the default metaspace named `ms`:

- seeder
- read
- write
- encrypt

To enable user access control for you as the user, do the following:

1. Change directory to the security subdirectory of the examples.
2. Open the file `example_policy.txt` with a text editor.
3. Locate the following line:
   ```
   access_control=false;default=deny
   ```
4. Change it to the following to enable user access control:
   ```
   access_control=true;default=deny
   ```
5. Locate the line:
   ```
   group1 = user1
   ```
6. Change `user1` to your <domain\username>.

7. Save the file.

To see how user access control works when running `ASOperations` with security, do the following:

1. Stop any examples that were started with security for the `exdomain` security domain.
2. Restart any security domain controllers, for the `exdomain` security domain, using the modified security policy file.
3. Start `ASOperations` with security.

When `ASOperations` tries to connect to the metaspace, you will be prompted for your user login information. After that `ASOperations` will join the space. You should be able to do some puts and gets on the space and browse the space.

ASPerf

Overview

ASPerf is an example that demonstrates how to perform timing measurements to calculate how long it takes ActiveSpaces to perform an action. ASPerf consists of a master and any number of slaves. The master is used to remotely invoke methods on the slaves. Each remote invocation method on a slave times the action being performed and the timings are returned in the results of the remote invocation. The master then displays the results returned from each of the slaves.

The ASPerf example uses two different spaces:

- **ASPerfCtrl** Used by the master to remotely invoke methods on the slaves.
- **ASPerfShared** Used by all of the remote invocation methods of the slaves for performing the actions of the remote invocation methods, such as putting data into the space.

Additionally, each slave creates its own individual space, which the slave can use instead of using the ASPerfShared space that it shares with all of the other slaves. Remote invocations from the master indicate which space a slave should use. In this way, you can see the timing differences between when a slave uses its own local space or uses the shared space in its remote invocation methods.

The ASPerf master creates the ASPerfCtrl space and then joins it as a leech. Each slave joins the ASPerfCtrl space as a seeder. In this way, the ASPerf master can use remote invocation on all seeders to invoke the same remote invocation method on all of the slaves of the ASPerfCtrl space.

ASPerf slaves can join the ASPerfShared space or their own local space as seeders or leeches.

Whether a slave joins these spaces as a leech or a seeder is controlled by the command line options of the ASPerf slave. The minimum number of seeders required for the ASPerfShared space and the ASPerf slave's own space is one. So if an ASPerf slave is started as a leech, an as-agent must also be run so that the slave's own space will have a seeder.

Since an as-agent joins all spaces in a metaspace as a seeder, when the ASPerf master tries to remotely invoke a method on all seeders of the ASPerfCtrl space, the remote invocation is also be tried on the as-agent. Since the as-agent does not know about the ASPerf remote invocation methods, as-agent reports an error back to the ASPerf master. For this reason, a special version of as-agent, ASPerfAgent, is provided with the ASPerf example.

Similar to `as-agent`, `ASPerfAgent` joins spaces in a metaspace as a seeder and thereby lends its computing resources to the space. However, `ASPerfAgent` specifically *does not* join the `ASPerfCtrl` space. So when the `ASPerf` master does remote invocations on all seeders of the `ASPerfCtrl` space, the remote invocation is not be tried on the `ASPerfAgent` and unnecessary errors are not reported back to the master.

Starting the ASPerf Master

The following are examples of how to invoke the **`ASPerf`** master for each of the APIs:

Java Invocation

```
java ASPerf -metaspace examplems -member_name master
```

.NET Invocation

```
ASPerf.exe -metaspace examplems -member_name master
```

C Invocation

```
ASPerf.exe -metaspace examplems -member_name master
```

The `ASPerf` master joins the `ASPerfCtrl` space as a leech. When you start the `ASPerf` master, you see the following message until an `ASPerf` slave is started:

```
Waiting for the control space to be ready (no slave connected
yet)...
```

After an `ASPerf` slave is started, the master continues, and you see the following prompt, which allows you to enter commands to cause the remote invocation of methods on the slaves:

```
Invoke on slaves (enter 'h' for help):
```

The following are some of the remote invocation methods that can be invoked using the `ASPerf` master:

- **LatencyGet, LatencyPut, LatencyTake** Returns the execution times when space operations are individually performed. For example, by default, `LatencyPut` returns the time it takes to store 1000 tuples of data into a space by doing 1000 puts.
- **ThroughputGet, ThroughputPut, ThroughputTake** Returns the execution times when space operations are performed as batches. For example, by default, `ThroughputPut` returns the time it takes to store 1000 tuples of data into a space by putting the tuples into the space in batches of 100.

To see the full list of commands, enter **h** at the prompt after the `ASPerf` master starts up.

Starting the ASPerf Slave

The following are examples of how to invoke an `ASPerf` slave for each of the APIs:

Java Invocation

```
java ASPerfSlave -metaspace examplems -member_name slave1 -role
seeder
```

.NET Invocation

```
ASPerfSlave.exe -metaspace examplems -member_name slave1 -role
seeder
```

C Invocation

```
ASPerfSlave.exe -metaspace examplems -member_name slave1 -role
seeder
```

Starting the ASPerf Agent

The following are examples of how to invoke an `ASPerf` agent for each of the APIs:

Java Invocation

```
java ASPerfAgent -metaspace examplems -member_name agent1
```

.NET Invocation

```
ASPerfAgent.exe -metaspace examplems -member_name agent1
```

C Invocation

```
ASPerfAgent.exe -metaspace examplems -member_name agent1
```

Appendix A Result and Status Codes

Many space operations return a *result object*—or in C, a *status value*— or, in the case of batch operations, a list of result objects. The operations that return a Result always return a result even if the operation failed (likewise, a status is always returned in C). Result objects always contain a status code (which is the same as the status codes returned directly by the functions in C).

Each status code has one of three types: *no error*, *error*, or *severe error*. *No error* indicates that the operation was successful, and data was returned. *Errors* indicate that the operation was successful from a system standpoint, but no data could be returned (because there was no tuple in the space or because the tuple in the space was locked). *Severe errors* indicate that the operation failed because of a system problem. If the status indicates an error or severe error, it is possible to get an exception (or an error object in C) using the `getError()` or `getSevereError()` methods of the Result (or in C, using `tibasError_GetError()` or `tibasError_GetSevereError())`.

`Result` (and `ResultList`) also has convenience `hasError()` and `hasSevereError()` methods that return **`true`** if the `Result` object (or any of the `Results` contained in the list) has an `Error` or `SevereError`.

If the operation was successful and resulted in an tuple being returned, this tuple can be retrieved from the `Result` object using the `getEntry` method, in this case it is also possible to directly retrieve the tuple contained in that tuple using the `Result` object's `getTuple` method.

If one of the objects passed as an argument to the space method is invalid, this method will throw a runtime exception.

The following table lists status codes returned by TIBCO ActiveSpaces functions:

Table 8 Status Codes

Constant	Type	Description
`TIBAS_OK`	no error	The operation was successful and a tuple was found.
`TIBAS_ALREADY_EXISTS`	error	The operation failed because there is currently a tuple in the space for the requested key field(s) value(s).

Table 8 Status Codes

Constant	Type	Description
`TIBAS_LOCKED`	error	The operation failed because the tuple is locked by another thread or process (depending on the LockScope().
`TIBAS_MISMATCHED_LOCK`	error	The lock expired in the space and another member already locked the tuple.
`TIBAS_INCOMPATIBLE_TUPLE`	error	The operation failed because the name of a field contained in the Tuple is incompatible with the space definition.
`TIBAS_MISMATCHED_TUPLE`	error	The operation failed because the Tuple is incompatible with the space definition.
`TIBAS_INCOMPATIBLE_TYPE`	severe error	The operation failed because two data types involved in an operation are incompatible
`TIBAS_LIMIT_EXCEEDED`	error	The operation failed because a predefined limit on the space, such as capacity, has been exceeded.
`TIBAS_INVALID_ARG`	severe error	The operation failed because an invalid argument was passed to an operation.
`TIBAS_SYS_ERROR`	severe error	The operation failed because of a system error.

Glossary

A

ActiveSpaces distributed application system

A set of ActiveSpaces programs that cooperate to fulfill a mission.

ActiveSpaces program

A program that uses ActiveSpaces software to work collaboratively over a shared data set that is represented by one or more tuple spaces.

ACL

Access Control List (ACL). A list of subjects and groups and the permissions granted to the subjects or to members of groups, which controls access to ActiveSpaces resources. You specify ACLs in the policy file for a security domain with the `access_control` setting, which can be set to *true* or *false*.

agent

A optional standalone process or *daemon* that is part of ActiveSpaces and provides services or features to the space. Using the Admin CLI, the administrator launches ActiveSpaces agents on the host or hosts where these services will run. They currently provide the following services:

1. Persist the *system spaces* configuration information to disk.
2. Provide additional scalability and stability to spaces.

associative array

A collection of values with unique keys, where each key is associated with one value. The keys can be any object, not necessarily an integer. A space can be used as an associative array.

authentication source

An ActiveSpaces component that presents credentials to an external authenticator and decides whether a supplied *Credential*—retrieved by the *Authenticator*—is valid. The authentication source can be an LDAP client, an OS or Pluggable Authentication Modules (PAM) login, a smart-card, an API invoker and so on. The authenticator performs the verification by connecting to an external resource (such as an LDAP v3 directory server) or by invoking local system calls, such as win32 `LogonUser()` or with UNIX/Linux, `pam_authenticate()`.

authenticator

An ActiveSpaces component that establishes a unique association between a subject and an identity through its credentials. The authenticator securely obtains *Credential*s from authenticating components to generate *Subject—Credential* associations. The authenticator converts and forwards the credential to the component, which verifies its validity.

C

cache

A generic term commonly used to refer to a repository of data that duplicates original values stored elsewhere, making that data more readily available to be fetched quickly where it is needed. In ActiveSpaces, a *cache* is distinguished from a *tuple space* in that data may be evicted from a cache without notification, for instance, to make space for other data. In this case, the evicted data, if needed later, will be fetched from the original data store, of which the cache is merely a copy. Data is never evicted from a space without notification; it is only removed if it expires or is deliberately taken out. It is possible to configure a space to act as a cache by setting a capacity and an eviction policy other than **none** in the space's definition.

cluster

A group of linked computers working closely together to increase scalability, and to maximize performance and availability beyond what can be achieved by a single computer of comparable power.

coherency

When multiple identical copies of data are maintained, *coherency* is a quality that indicates the copies are kept in synch when the original data changes.

credential

An object that can be used to establish the identity of a requestor node. This can be data:

- Known to the requestor, such as a username/password.
- Possessed by the requestor, such as an X509v3 certificate, security token, shared secret, IP address, and so on.

D

distributed cache

A cache that uses multiple locations for storage of data.

distributed in-memory tuple space

A generic term for the category of software product that includes ActiveSpaces. The data in a tuple space is *distributed* over multiple machines for scalability and failover, and it is stored *in memory* for optimal performance.

data grid

A data store that is distributed over a cluster comprised of multiple machines or members. With ActiveSpaces, the capacity of the data grid scales linearly as you add members to the cluster.

data partitioning

Distributing a set of data over a cluster of members. ActiveSpaces performs data partitioning transparently, based on the members that have been provided to the tuple space. Developers do not need to concern themselves with which parts of the data are stored on which members.

data

A means of providing fault tolerance where a copy of data from one member is stored on another member, so that no member can be a single point of failure. When is enabled for a space, the replicates are updated whenever tuple data changes through a `put` or `take` command. (A `get` command will not cause replicates to be updated, since it does not change the data.)

There are two kinds of , *synchronous* and *asynchronous*. Synchronous will have an impact on performance, since it involves putting (or taking) data and replicating it in a single transaction. With asynchronous , there is little perceptible impact on performance, but there is a small amount of time where the data is not fully replicated.

Whether or not a space is replicated and, if so, whether the is synchronous or asynchronous, is specified when the space is created. The administrator can also specify the *degree* of , that is, how many replicas of the data will be created.

With synchronous , the administrator or application has immediate verification of whether or not the was successful, because if it was *not*, then the `put` or `take` command that triggered the attempted will itself fail, returning an error message. In asynchronous mode, the command will succeed, regardless of successful . An application or administrator can listen to advisory spaces to determine whether there was a problem with for an instance of asynchronous .

If a space is being used as a cache-aside, the space will normally be created *without* , since the system of record for that data will be a database. In this case, if the single member containing the space goes down or is offline, the data can be obtained from the database.

domain data key

Used for memory- and local-persistence encryption. It can be generated or regenerated by using the `as-admin` tool;.

E

entry

An entry represents a tuple that is stored in a space. While a tuple is made up of a key and value, an entry is made up of the tuple plus the metadata associated with its being stored in a space. In addition to details used internally by ActiveSpaces, the metadata includes the entry's time-to-live value (TTL), that is, how much time is left before the entry expires, at which time it will be deleted from the space.

event

In ActiveSpaces, an event reflects a change to some of the data in a space or a change in state of a space or member.

event listener

See *space listener.*

event notification

An asynchronous message sent to event listeners when data changes. The message takes the form of an invocation of a callback method on the space listener.

F

field

A field is a portion of a tuple, similar to a single value (or row) in a column of a database table. A field is associated with a name, a type, and a value.

G

group

A group that can be used to organize users and domain objects, thus simplifying administration. Security groups allow you to assign the same security permissions to a large numbers of users or requestors, such as employees in a single department or in a single location or nodes in a security domain, ensuring that security permissions are consistent across all members of a group.

The security group can include:

- Individual subjects
- Other security groups

For example, a security group named *group3* can include users (`user6` and `user7`) and also other security groups, such as a group named `group2`.

- Objects or entities

For example, `My Ldap X509Cert CN`, which represents ian X509 v3certificate

H

hash map

An *associative array* that uses a hash function to optimize search and insertion operations. The hash function transforms the key into a hash, a number that is used as an index in an array to locate the values during a lookup.

I

Identity

A set of properties of an entity that can be used to uniquely distinguish it from other entities. A logical association with:

- An individual computer/node that is running the ActiveSpaces software.
- A program or a program component that is invoking the ActiveSpaces API.

K

key

A unique value based on the value of one or more fields.

The key is used to implement the insertion policy for a tuple. The key is also used by the ActiveSpaces distribution algorithm to determine how data is distributed.

L

leech

A member that joins a space but does not lend any resources to the space, such as memory or processing power. Distinct from a *seeder.*

listener

See *space listener.*

lock

An application can lock an entry so that the entry cannot be modified (but can still be read) until the lock is explicitly removed.

M

member

A process, either an application or an agent, that is linked to the ActiveSpaces libraries and is joined to a space as one of a cluster of members. A single machine may contain more than one member. A member can be a *seeder* or a *leech*, depending on whether or not it lends resources to the space.

A member can be a seeder in one space and a leech in another.

metaspace

An administrative collection of system spaces and user spaces sharing the same transport argument, which includes a multicast address that can be used for messages (event notifications).

A metaspace is a container for managing a number of user spaces, and a group of members that are working together in a cluster. The metaspace is the initial handle to ActiveSpaces. An application or member first joins a metaspace, and through it, gets access to other objects and functionality.

N

node

A term sometimes used in place of the term *member*. This usage can be confusing, because the term *node* is most often used outside of ActiveSpaces to refer to a machine, whereas within ActiveSpaces, a single machine may contain more than one member.

P

peer

A process that has connected to a metaspace and joined a space as either a *seeder* or a *leech*.

permission

A specific set of access control permissions possessed by either a user or a group, which defines how to access *Resource*s in the *Security Domain*. Permissions are granted to specific scopes—to metaspaces or spaces.

The privilege that is granted depends on the scope level; for example, at the metaspace level, specified metaspaces or all metaspaces can be granted access to transactions, or to connections, and at the space level, specified spaces or all spaces can be granted access to read, write, delete, browse, lock, or seed operations.

Rights or privileges are specified in a permissions table that you code in the policy file for a domain

R

relaxed coherency

If there are multiple copies of a tuple (due to or local caching), any change to the tuple is reflected in those copies as quickly as possible. ActiveSpaces uses relaxed coherency in most modes of operation. (See *strict coherency*.)

requestor

An ActiveSpaces node that requests access to resources (by attempting to join one or more metaspace) controlled by one or more *Security Domain Controllers.*

resource

ActiveSpaces objects such as metaspaces, spaces or tuples.

S

scalability

For data stores, the ability to contain ever-increasing amounts of data. ActiveSpaces offers *linear scalability*, meaning that storage capacity and performance increase at a constant rate as members are added to a space.

security domain

A context in which uniform and consistent security settings can be enforced on a defined set of metaspaces. A metaspace can only be contained (managed) by one domain at a time. Within a domain, secured transports are negotiated between pairs of nodes, independent of other nodes. Memory- and local-persistence encryption use a shared secret, stored protected by the managing node's identity (see the definition for *Security Domain Controller*).

security domain controller

An ActiveSpaces node that is dedicated to enforcing the policy definitions for one or more *Security Domain*s. You should set up multiple security domain controllers for each security domain to provide fault tolerance for security.

security policy

The definition of security preferences for one or more *Security Domain*s. The policy configuration is specified in a policy file that you create by using the `as-admin` **define | create security_policy** command. The policy file contains:

- A domain identity that you specify in the command to create the security policy. command.

  For information on creating the domain identity, see Creating a Security Policy File, page 136.
- Optional RSA private keys and an X509v3 certificate that ActiveSpaces creates when you create a policy file.

  For information on creating the domain identity, see Security Policy Files, page 135.
- Configuration keywords that specify specific attributes of the security policy, such as access lists, and transport quality (QoS).

security token

A file that is deployed on nodes that need to connect to access- controlled and/or secured ActiveSpaces *Resource*s. The token comprises:

- An X509v3 node identity (optional). You can specify a node identity when you run the `as-admin` **secpolicy new token** command.
- An X509v3 trust anchor for a *Security Domain* to establish initial trust. The trust anchor consists of the security key and security certificate for the domain which are contained in the policy file for the domain.
- A discovery URL. This is provided in the policy file.
- A transport service quality (QoS) identifier to define the strength of the secured transport between the client (see definition for *Requestor*) and managing nodes. This is provided in the policy file.

seeder

A member that joins a space and lends resources, such as memory and processing power, to the scalability of the space. Distinct from a *leech*.

In a distributed space, all peers are responsible for seeding certain tuples.

In a non-distributed space, one of the peers is assigned to be the seeder, determined by the ActiveSpaces distribution algorithm.

Ideally, peers are relatively stable, since there is overhead to reorganize the distribution of the tuples among the remaining peers when a peer leaves the space. For this reason, a transient application—one that will leave and join the space frequently—should generally be configured to join the space as a leech, rather than as a peer.

Note that *agents* are always seeders, not leeches. Agents provide an efficient, stable means of increasing the scalability of a space. Also, note that multiple seeders cannot be created from a single client program.

For each entry in a space, the ActiveSpaces distribution algorithm designates one seeder as the seeder of that tuple, whether or not the tuple is replicated on other members. The seeder holds and owns the authoritative copy of the complete tuple.

If the space has multiple seeders, a tuple may be held by different seeders at different times. If the current seeder of the entry leaves the space, another seeder is chosen as the entry's new seeder, and the entry is then copied over to the new seeder.

Shared-All Persistence

With shared-all persistence, certain space members are designated as *persisters* — to provide the service of interacting with a persistence layer, just as some of the space members — the *seeders* — provide the basic space service.

Shared-Nothing Persistence

Each node that joins a space as a seeder maintains a copy of the space data on disk. Each node that joins as a seeder writes its data to disk and reads the data when needed for recovery and for cache misses

space browser

A space browser, created with the ActiveSpaces API, allows an application to iterate through the entries in a space. There are four kinds of space browsers: EventBrowser, GetBrowser, TakeBrowser, and LockBrowser. All space browsers have a single method, `next`, which returns an entry to the calling process. Space browsers are described in more detail in this document and in the API documentation.

space listener

The portion of your code that comprises a callback function to be invoked by ActiveSpaces when certain data changes or certain events occur. A listener is similar to a *subscriber* in a publish-subscribe messaging system.

The events for which callback functions will be invoked are **lock**, **put**, **take**, and **unlock**.

Depending on the distribution scope of the listener two additional callback functions, `onSeed` and `onUnseed`, can be invoked to monitor seeder changes due to re-distribution of entries when a seeder joins or leaves a space.

strict coherency

When there are multiple copies of a set of data (due to or local caching), strict coherency means that any change to the data must be applied to all copies at the same time. Because this adversely impacts performance, and because *relaxed coherency* offers nearly the same degree of coherence, ActiveSpaces provides strict coherency only for spaces that are non-distributed, that is, where only a single copy of the data exists. (See *relaxed coherency.*)

subject

An entity that is associated with an *Identity* through a *Credential*. A subject represents a single aspect of a *Credential*; for example, the name of a user or the common name value of the distinguished name component of an X509v3 certificate.

system spaces

A set of administrative spaces that are created and maintained by ActiveSpaces and are used to describe the attributes of the spaces. Distinct from *user spaces*.

T

tuple

A typed data object that is stored in a space. Similar to a row in a database table.

tuple space

A collection of tuples. Similar to a table in a database.

U

user spaces

Spaces that are defined by the user. Distinct from *system spaces*.

Index