## Contents

### How to Use a Space
- Batch versus Blocking Operations ............................................. 29
- Storing Data into a Space .......................................................... 29
- Retrieving Data from a Space ....................................................... 30
- Concurrently Updating Data in a Space ......................................... 31
- Using Space Browsers ................................................................. 32
- Listening to Changes ................................................................. 35
- Filters ....................................................................................... 36
- Remotely Invoking Code over a Space ........................................... 39
- Transactions ............................................................................. 40
- Deployment ............................................................................. 41
- Agents ...................................................................................... 42
- Connecting with a Metaspace: Special Considerations ..................... 42

### Chapter 4 Architecture ............................................................. 45
- Result and Status Codes ............................................................... 46
- The ASCommon Object ............................................................... 50
- Metaspace .................................................................................. 51
  - Connecting to the Metaspace .................................................... 51
  - Metaspace Name ..................................................................... 52
  - ConnectionDef Object ............................................................. 52
  - Disconnecting from the Metaspace ............................................ 54
  - Metaspace Membership ............................................................ 55
- Managing Spaces ......................................................................... 56
  - Defining a Space ..................................................................... 56
  - Getting a Space Definition ...................................................... 57
  - Space Attributes ..................................................................... 58
  - Field Definitions ...................................................................... 64
- Space Membership .................................................................... 66
  - Joining a Space ..................................................................... 66
  - Leaving a Space ..................................................................... 66
  - Persisting on a Space .............................................................. 67
  - Space Life Cycle ..................................................................... 67
- Tuples ........................................................................................ 69
  - Storing a Tuple in a Space ....................................................... 72
  - Updating a Tuple in a Space ..................................................... 72
- Batch operations ........................................................................ 74
- Locking and Working with Locked Entries .................................... 75
- SpaceEntry ............................................................................. 76
Important Information

SOME TIBCO SOFTWARE EMBEDS OR BUNDLES OTHER TIBCO SOFTWARE. USE OF SUCH EMBEDDED OR BUNDLED TIBCO SOFTWARE IS SOLELY TO ENABLE THE FUNCTIONALITY (OR PROVIDE LIMITED ADD-ON FUNCTIONALITY) OF THE LICENSED TIBCO SOFTWARE. THE EMBEDDED OR BUNDLED SOFTWARE IS NOT LICENSED TO BE USED OR ACCESSED BY ANY OTHER TIBCO SOFTWARE OR FOR ANY OTHER PURPOSE.

USE OF TIBCO SOFTWARE AND THIS DOCUMENT IS SUBJECT TO THE TERMS AND CONDITIONS OF A LICENSE AGREEMENT FOUND IN EITHER A SEPARATELY EXECUTED SOFTWARE LICENSE AGREEMENT, OR, IF THERE IS NO SUCH SEPARATE AGREEMENT, THE CLICKWRAP END USER LICENSE AGREEMENT WHICH IS DISPLAYED DURING DOWNLOAD OR INSTALLATION OF THE SOFTWARE (AND WHICH IS DUPLICATED IN LICENSE.PDF) OR IF THERE IS NO SUCH SOFTWARE LICENSE AGREEMENT OR CLICKWRAP END USER LICENSE AGREEMENT, THE LICENSE(S) LOCATED IN THE "LICENSE" FILE(S) OF THE SOFTWARE. USE OF THIS DOCUMENT IS SUBJECT TO THOSE TERMS AND CONDITIONS, AND YOUR USE HEREOF SHALL CONSTITUTE ACCEPTANCE OF AND AN AGREEMENT TO BE BOUND BY THE SAME.

This document contains confidential information that is subject to U.S. and international copyright laws and treaties. No part of this document may be reproduced in any form without the written authorization of TIBCO Software Inc.

TIB, TIBCO, TIBCO Adapter, Predictive Business, Information Bus, The Power of Now, TIBCO ActiveMatrix BusinessEvents, and TIBCO ActiveSpaces are either registered trademarks or trademarks of TIBCO Software Inc. in the United States and/or other countries.

EJB, Java EE, J2EE, and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

All other product and company names and marks mentioned in this document are the property of their respective owners and are mentioned for identification purposes only.

THIS SOFTWARE MAY BE AVAILABLE ON MULTIPLE OPERATING SYSTEMS. HOWEVER, NOT ALL OPERATING SYSTEM PLATFORMS FOR A SPECIFIC SOFTWARE VERSION ARE RELEASED AT THE SAME TIME. SEE THE README FILE FOR THE AVAILABILITY OF THIS SOFTWARE VERSION ON A SPECIFIC OPERATING SYSTEM PLATFORM.

THIS DOCUMENT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS DOCUMENT COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THIS DOCUMENT. TIBCO SOFTWARE INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS DOCUMENT AT ANY TIME.

THE CONTENTS OF THIS DOCUMENT MAY BE MODIFIED AND/OR QUALIFIED, DIRECTLY OR INDIRECTLY, BY OTHER DOCUMENTATION WHICH ACCOMPANIES THIS SOFTWARE, INCLUDING BUT NOT LIMITED TO ANY RELEASE NOTES AND "READ ME" FILES.

Copyright © 2009–2011 TIBCO Software Inc. ALL RIGHTS RESERVED.

TIBCO Software Inc. Confidential Information
Preface

This manual describes TIBCO ActiveSpaces® software. The manual includes concepts and procedures needed to understand ActiveSpaces and use it effectively.

Topics

- Related Documentation, page iv
- Typographical Conventions, page v
- How to Contact TIBCO Support, page vii
This section lists documentation resources you may find useful.

Other TIBCO Product Documentation

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

- TIBCO Rendezvous® software: Behind the scenes, TIBCO Rendezvous software handles the transport of data and messages between member processes over the network for TIBCO ActiveSpaces.

  In particular, 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

<table>
<thead>
<tr>
<th>Convention</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TIBCO_HOME</strong></td>
<td>All TIBCO products are installed under the same directory. This directory is referenced in documentation as <em>TIBCO_HOME</em>. The value of <em>TIBCO_HOME</em> depends on the operating system. For example, on Windows systems, the default value is <code>C:\tibco</code>.</td>
</tr>
<tr>
<td><strong>Code font</strong></td>
<td>Code font identifies commands, code examples, filenames, pathnames, and output displayed in a command window. For example: Use <code>MyCommand</code> to start the foo process.</td>
</tr>
</tbody>
</table>
| **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.
### Typographical Conventions

<table>
<thead>
<tr>
<th>Convention</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="tip_icon.png" alt="tip_icon" /></td>
<td>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.</td>
</tr>
<tr>
<td><img src="warning_icon.png" alt="warning_icon" /></td>
<td>The warning icon indicates the potential for a damaging situation, for example, data loss or corruption if certain steps are taken or not taken.</td>
</tr>
</tbody>
</table>

### Table 1  General Typographical Conventions (Cont’d)

<table>
<thead>
<tr>
<th>Convention</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="tip_icon.png" alt="tip_icon" /></td>
<td>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.</td>
</tr>
<tr>
<td><img src="warning_icon.png" alt="warning_icon" /></td>
<td>The warning icon indicates the potential for a damaging situation, for example, data loss or corruption if certain steps are taken or not taken.</td>
</tr>
</tbody>
</table>

### Table 2  Syntax Typographical Conventions

<table>
<thead>
<tr>
<th>Convention</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[ ]</code></td>
<td>An optional item in a command or code syntax. For example:</td>
</tr>
<tr>
<td></td>
<td>MyCommand [optional_parameter] required_parameter</td>
</tr>
<tr>
<td>`</td>
<td>`</td>
</tr>
<tr>
<td></td>
<td>MyCommand para1</td>
</tr>
<tr>
<td><code>{ }</code></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>MyCommand {param1 param2}</td>
</tr>
<tr>
<td></td>
<td>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:</td>
</tr>
<tr>
<td></td>
<td>MyCommand {param1</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>MyCommand param1 [param2] {param3</td>
</tr>
</tbody>
</table>
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, page 2
- Usage Profiles, page 5
ActiveSpaces software is a peer-to-peer distributed in-memory data grid. With ActiveSpaces, it is easy to create distributed applications that exchange and modify data shared between processes and across a network.

What is ActiveSpaces?

- ActiveSpaces software provides an application programmer with software support for coherent data storage and retrieval, network data transport, and network data representation. Java and C libraries are provided.
- It facilitates and speeds up storage and retrieval of data in a distributed manner so that programmers can concentrate on writing business logic. There is no need to worry about where to store new data, where current data is stored, or if it is out-of-date.
- It combines features of a database with features of a middleware messaging system, in a simple, elegant interface.
- ActiveSpaces software supports many hardware and software platforms, so programs running on many different kinds of computers on a network can communicate seamlessly.
- ActiveSpaces scales linearly as machines 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.
- ActiveSpaces scales transparently: the capacity of the space automatically grows when new peers join. As the number of peers changes, applications continue to function smoothly, requiring no code modifications or restarts.
- Using an adaptive algorithm, data and replicates of data are minimally re-distributed as peers join and leave the peer-to-peer network.

ActiveSpaces ensures that any change in data is reflected on all nodes as it happens and that no node will deliver stale data when invoking get or using a browser.

From the programmer’s perspective, the ActiveSpaces software suite includes three main components:

1. An ActiveSpaces programming language interface (API), available in C and in Java.
2. The ActiveSpaces library files.
3. The underlying reliable multicast network communication infrastructure.
From an administrator’s perspective, the following components are most important:

1. ActiveSpaces applications.
2. The ActiveSpaces agent processes.
3. The Administration Command Line Interface (Admin CLI) tool and language.

ActiveSpaces applications are programs that use ActiveSpaces software to work collaboratively over a shared data set that is represented by one or more tuple spaces. An ActiveSpaces distributed application system is a set of ActiveSpaces programs that cooperate to fulfill a mission.

Tuples are partitioned across "seeders," those members that are configured to contribute memory and processing resources to a space. Tuples are automatically redistributed 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, with ActiveSpaces rebalancing is done transparently using a "minimal redistribution" algorithm.

Replication allows the distribution of replicates of data on different peers for fault tolerance. ActiveSpaces’ data access optimization feature results in use of a replicate if one is locally available for optimal performance. If a seeder suddenly fails, the replicate is immediately promoted to seeder, and the new seeder creates new replicates.

Benefits of Programming with ActiveSpaces

Programming with ActiveSpaces provides the following benefits:

- Distributed for speed, resiliency, and scalability
- Simplifies distributed system development
- Location transparency: no need to worry about where or how to store or find data
- Decoupling of producers and consumers of data
- Ability to be notified automatically as soon as data is modified
- Scalability and fault-tolerance

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 replication
- Implements a form of horizontal partitioning
- Has locks for concurrent access control
- Includes support for transactions
- Supports CRUD operations (Create, Read, Update, Delete)

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 offers these additional features:
- The ability to receive initial values when creating a listener.
- The ability to run continuously updated queries in real-time.
Usage Profiles

The ActiveSpaces data grid can solve a variety of application problems. The nature of the problem determines the best configuration. Several factors should be considered to determine the appropriate configuration: the size of the data, the frequency of data updates versus reads, and the relative importance of update speed versus absolute consistency of the 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 space-enabled.

Distributed Data Cache

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

ActiveSpaces handles the task of keeping the cache synchronized across any number of hosts. Instead of the costly operation of going to a disk, data is fetched quickly from an in-memory data cache across a local network. Coherency and locking is handled by the space.

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 like 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 cope with large data sets, and with data that is transient and volatile.

Space-based Architecture

When a new system is being designed from scratch, you can take maximum advantage of the features of a space-based approach. Space-based architectures use a grid approach for both data storage and processing. Data storage and access is virtualized. Data communication and process coordination are both handled by the space. 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 one large computing problem. A complex problem is decomposed into smaller pieces that can be executed across many machines in parallel.
Chapter 2  

**Installation**

This software may be available on multiple operating systems. However, not all operating system platforms for a specific software version are released at the same time. Please see the readme file for the availability of this software version on a specific operating system platform.

This chapter explains the procedures for installing TIBCO ActiveSpaces on the supported Windows and UNIX platforms.

**Topics**

- Prerequisites, page 8
- Installing ActiveSpaces, page 10
- Setting Environment Variables, page 14
- Testing your Installation, page 16
- Uninstalling ActiveSpaces, page 18
Prerequisites

Supported Platforms

The platforms supported by TIBCO ActiveSpaces 1.1.1 are as follows:

**Windows**
- Windows XP (x86 and x86-64)
- Windows Server 2003 (x86 and x86-64)
- Windows Server 2008 (x86 and x86-64)
- Windows Vista Business Edition (x86 and x86-64)

Disk space required in each case is 1GB.

**UNIX**
- Red Hat Enterprise Linux 4, 5 (x86 and x86-64)
- HP-UX 11i v3 (Itanium-64)
- Solaris 10 (x86, x86-64)
- Solaris Sparc

Optional Supporting Software

**Java JDK or JRE**

Installation of the Java JDK or JRE, version 1.6.0_18 or above, on a machine running ActiveSpaces enhances the functionality of ActiveSpaces as follows:

- The JDK will allow use of the Java API and Administration Command Line Interface (Admin CLI),
- The JRE alone will allow use of the Admin CLI, but not the Java API.

The JDK or JRE is not required on machines where the Java API or Admin CLI will not be used.
TIBCO Rendezvous

TIBCO Rendezvous is required if you wish to use the Rendezvous discovery transport (using a discovery URL that begins with tibrv:). Rendezvous must be installed on each machine where you install ActiveSpaces.

If you will only make use of the embedded PGM discovery transport (using a discovery URL that begins with tibpgm:), TIBCO Rendezvous is not required.

TIBCO ActiveSpaces Enterprise Edition includes limited license of Rendezvous. The license is limited to use for local area communication (RVD). Separate purchase of TIBCO Rendezvous software is required for routing between multiple subnets. The TIBCO Rendezvous component included with ActiveSpaces should only be used with ActiveSpaces.

For more information about discovery transport in ActiveSpaces, see Connecting to the Metaspace on page 51.
Installing ActiveSpaces

This section describes the installation procedure for ActiveSpaces using the TIBCO Universal Installer.

Installing in GUI Mode

To install ActiveSpaces using GUI mode, perform the following steps.
1. Open the physical media or download the ActiveSpaces product package.
2. Extract the ActiveSpaces product archive file to a temporary directory.
3. Navigate to the temporary directory that contains the universal installer.
4. Run TIBCO Universal Installer.
5. The Welcome screen appears. Click Next.
6. The License Agreement screen appears.

After reading through the license text, click I accept the terms of the license agreement and then click Next.
7. The **Important Notice** information appears. Read through this information and click **Next**.

8. The **Installation Environment** screen appears. Specify a location for installing ActiveSpaces.
9. Install all features by clicking **Typical** or choose the features to install by clicking **Custom**. After making your choice, click **Next**.

10. If you selected **Custom**, in **step 9**, a screen similar to the following appears. If you did not select **Custom**, proceed to **step 11**. Unselect the check box next to the features you do not want to install, and click **Next**.
11. The installer prepares the features for installation. A pre-install summary screen appears. Click **Install**.

![Pre Install Summary](image)

12. Click **Yes to All** to close any pop up windows that display during installation.

A post-install report summarizes the installation results. Click **Finish** to close the installer window.
Setting Environment Variables

This section describes the environment variables that must be set after the ActiveSpaces installer is run.

**TIBCO ActiveSpaces bin and lib Directories**

In order for the operating system to find the ActiveSpaces libraries when an ActiveSpaces application is run, the following paths must be added to the PATH (Windows and UNIX) or LD_LIBRARY_PATH (UNIX) environment variables on machines where ActiveSpaces is installed:

- ActiveSpaces bin directory: to the PATH on Windows and UNIX
- ActiveSpaces lib directory: to the PATH on Windows, or the LD_LIBRARY_PATH on UNIX

**Windows example**

If ActiveSpaces is installed in `C:\tibco\as\1.1.1`:

add `C:\tibco\as\1.1.1\lib` and `C:\tibco\as\1.1.1\bin` to the PATH environment variable

**UNIX example**

If ActiveSpaces is installed in `/usr/tibco/as/1.1.1`:

add `/usr/tibco/as/1.1.1/lib` to the LD_LIBRARY_PATH and `/usr/tibco/as/1.1.1/bin` to the PATH environment variable

**TIBCO Rendezvous bin and lib Directories**

In order to use TIBCO Rendezvous as the transport for ActiveSpaces, Rendezvous must be installed on each machine running ActiveSpaces, and these paths must be added to the PATH (Windows) or LD_LIBRARY_PATH (UNIX):

- Rendezvous bin directory: to the PATH on Windows and UNIX
- Rendezvous lib directory: to the PATH on Windows, or the LD_LIBRARY_PATH on UNIX

**Windows example**

If TIBCO Rendezvous is installed in `C:\tibco\tibrv`:
add `C:\tibco\tibrv\lib` and `C:\tibco\tibrv\bin` to the PATH environment variable.

**UNIX example**

If TIBCO Rendezvous is installed in `/usr/tibco/tibrv`:

add `/usr/tibco/tibrv/lib` to the `LD_LIBRARY_PATH` and `/usr/tibco/tibrv/bin` to the PATH environment variable.

**JRE or JDK bin and lib Directories**

In order for the operating system to find the Java class libraries when an ActiveSpaces application is run, the bin and lib directories for the JDK or JRE must be included in the PATH environment variable, for example, `C:\Program Files\Java\jdk1.6.0\bin` and `C:\Program Files\Java\jdk1.6.0\lib`.

**New Environment Variables Required**

**CLASSPATH environment variable**

In order for Java ActiveSpaces applications to run, the CLASSPATH variable should include the following JAR files:

- `as-common.jar`
  
  All applications require this JAR file to be in the CLASSPATH.

- `as-admin.jar` and `antlr-3.2.jar`
  
  All applications making use of the Admin object require these JAR files to be in the CLASSPATH.

**AS_HOME environment variable**

Create a System environment variable called `AS_HOME` that points to the directory where ActiveSpaces is installed, for instance, `C:\tibco\as\1.1.1` on Windows or `/usr/tibco/as/1.1.1` on UNIX.

**JAVA_HOME environment variable**

The JAVA_HOME environment variable must be created if it does not already exist. It should point to the top level directory of the Java JDK or JRE, for example, `C:\Program Files\Java\jdk1.6.0`. 

TIBCO ActiveSpaces User’s Guide
Testing your Installation

When the installation is completed and the required environment variables are set, you can test your ActiveSpaces installation to ensure that it is functioning properly. A simple way of testing the installation is described here.

1. In a command prompt or shell window launch the Admin Command Line Interface (CLI) by navigating to the ActiveSpaces bin directory, then invoking as-admin.cmd.
2. At the Admin CLI prompt type connect.
3. When connected, type show members. The output of this command should be the ID of a single member of the default metaspace, ms (that instance of the Admin CLI).

**Example:**

```
C:\Documents and Settings\username>cd C:\tibco\as\1.1\bin
C:\tibco\as\1.1\lib>as-admin.cmd
as-admin> connect
process_id=5180
Connected to metaspace ms
as-admin> show members
a62c8e3-7750-4ace3c8d-356 metaspace manager
```

4. In a second command prompt (or shell) window, launch another instance of the Admin CLI,
5. Type connect again.
6. Once connected, type show members again. The output should now show the IDs of two members (both instances of the Admin CLI).

**Example:**

```
C:\Documents and Settings\username>cd C:\tibco\as\1.1\bin
C:\tibco\as\1.1.1\lib>as-admin.cmd
as-admin> connect
process_id=6932
Connected to metaspace ms
as-admin> show members
```
The output of the `show members` command will also indicate for each process whether it is the MANAGER or is simply a MEMBER of the metaspace.

The examples in this section use Windows conventions. On UNIX, the Admin CLI executable is called `as-admin.sh`, instead of `as-admin.cmd`. The default location for the file on UNIX is `/usr/tibco/as/1.1.1/bin`, instead of `C:\tibco\as\1.1.1\bin`.

For more information about the Admin CLI and the commands `connect` and `show members`, see Chapter 5, Administrative Commands, on page 87. You can also type ? at any time to see a list of Admin CLI commands and their parameters.
Use the following procedure to uninstall the software:

1. Stop all TIBCO ActiveSpaces processes.

2. Run the uninstaller:
   
   Navigate to `TIBCO_HOME\_uninstall` and run `universal_uninstall` executable.

3. The Welcome screen appears. Click **Next**.

4. To uninstall only ActiveSpaces, select **Custom Uninstall**.

   To uninstall all TIBCO products (installed using the universal installer), select **Typical Uninstall**. After making your choice, click **Next**.

5. If you selected **Custom Uninstall**, then in the next screen select the specific products to uninstall.
The choices include only TIBCO products that you installed using the universal installer.

6. Click the **Uninstall** button to remove the selected products.

7. To complete the procedure, clean up the environment variables that you modified after the ActiveSpaces installation.
Chapter 3  Fundamentals

This chapter explains the fundamental concepts and terminology of TIBCO ActiveSpaces.

Topics

- What is a Metaspace?, page 22
- What is a Space?, page 23
- How is a Space Implemented?, page 25
- What are Tuples, Entries, and Results?, page 27
- How to Use a Space, page 29
What is a Metaspace?

A metaspace represents two things:

1. It is an instance of a cluster of applications processes (typically, but not necessarily, deployed on multiple hosts interconnected by a network) using ActiveSpaces.

2. It is an administrative container for a set of spaces (system spaces and user spaces).

The first thing an application needs to do in order to be able to use ActiveSpaces is connect to a metaspace. Once connected, the application can then define and make use of any number of spaces.

There can be multiple independent metaspaces deployed over a single network of hosts, each with a different set of members and spaces, and each identified by a name and a set of network transport attributes. Applications can connect to more than one metaspace at a time, but can only have a single connection for each metaspace.

A metaspace is a virtual entity: it is created by the first application that connects to it, and it disappears when the last member disconnects from it. The metaspace grows or shrinks automatically as members connect to it, and disconnect from it.

If the Metaspace is configured to make use of multicasting for discovery, there is no need to provide a list of IP addresses of core Metaspace members in order to connect to it.
What is a Space?

A space provides shared virtual storage for data.

- A space is a *shared* entity in the sense that many applications can access a space concurrently and each application has the same coherent view of the data contained in the space. The spaces in ActiveSpaces are called tuple spaces, and the items stored in them are called tuples.

A space is also able to proactively notify those 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 a *virtual* entity in the sense that it is distributed and implemented collaboratively by a group of processes located on multiple hosts and communicating over the network.

ActiveSpaces handles changes in this 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.

What the Space is Not

ActiveSpaces is a data-grid product implementing a distributed in-memory tuple space; it is not just a distributed cache. While the labels are sometimes used interchangeably, there are important distinctions between the two.

*ActiveSpaces differs from a distributed cache*

The first difference is that a cache may *evict* entries at any time if it needs to make room for new entries, while a tuple space data grid will not. The major implication of this distinction is that 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) never as a system of record itself. While 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.

To use a space as a cache, it should be configured with the following attributes:

- It should have a capacity (the maximum number of entries per seeder that can be stored in the space), and
• It should have an eviction policy (such as what happens when the space is full and a request to insert a new entry is made) of 'Least Recently Used'. In that case, and in that case only, the space will evict one of the entries 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 entry will fail due to the capacity being reached.

The second difference is that a distributed cache does not have a notification mechanism to proactively tell applications of changes in the data stored in the cache. Unlike ActiveSpaces, then, a distributed cache cannot be used for distributed process coordination.
How is a Space Implemented?

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. Seeders are applications that join the space indicating their willingness to lend some of the resources of the host on which they are deployed to scale the service provided by ActiveSpaces. In effect, the seeding applications have a portion of the Space embedded in their process.

ActiveSpaces distributes the data stored in the space evenly between all of the processes that have joined the space as seeders. ActiveSpaces is an elastic distributed system, in that seeders can join and leave the space (effectively scaling it up or down) at any time without the need to restart or reconfigure any other participant in the space. When this happens, the distribution of entries is automatically rebalance if necessary to maintain even distribution between the seeders.

An application can also join a space without contributing any of its host’s resources to the scalability of the space, without experiencing any limitation in functionality or in its ability to use the space. Such a space member is called a leech (as opposed to a seeder) and is a peer of the space that is simply not considered by the distribution algorithm and therefore can join or leave the space without incurring any kind of re-distribution of the data.

ActiveSpaces also includes a process called as-agent that has only one function: it automatically joins distributes spaces as a seeder. In order to understand when applications should join a space as seeder or as leeches please consider the following:

- Even though ActiveSpaces is a true peer-to-peer architecture, rather than client-server, you can deploy applications as leeches (effectively clients of the space service) with the as-agents acting as a “server cluster.”
- An application that joins a space as a seeder will experience better performance than it would as a leech for some operations, but this will come at the expense of higher RAM and CPU usage than as a leech.
- The entries in the space are stored randomly (more specifically, using a hash of the value(s) of the key field(s), in practice as good as random) but 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.
• The as-agent process can also be used to "keep the data alive" when all of the instances of an application have disconnected from the metaspace.

• Every time a seeder joins or leaves a space, there can be a temporary impact on space performance while redistribution or replication happens. Leeches on the other hand will not incur any impact when joining or leaving a space.

• This choice of distribution role has to be made on a per space basis: it may be that the best solution is to join some spaces as a seeder and others as a leech.
What are Tuples, Entries, and Results?

**Tuples**

A *tuple* is similar to a row in a database table: it is a container for data. More specifically, it is a sequence of named elements called fields (similar to the columns in a database table) which contain values of a specific type. Each tuple in a space represents a set of related data.

Fields have a name and a type. A tuple can be seen as a kind of map on 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.

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 the software may perform an automated field conversion if possible, or the operation will fail.

*Table 3, Field Type Conversions* shows which type conversions are supported. The symbols x and l have the following meanings:

- **x**: Conversion is supported, but with loss of precision.
- **l**: Conversion is supported with no loss of precision.

An empty cell in the table indicates that conversion between the two types is not supported.

<table>
<thead>
<tr>
<th></th>
<th>Boolean</th>
<th>Short</th>
<th>Integer</th>
<th>Long</th>
<th>Float</th>
<th>Double</th>
<th>Blob</th>
<th>String</th>
<th>DateTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Short</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Integer</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>Long</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Float</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>Double</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>Blob</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TIBCO ActiveSpaces User’s Guide
There is a `get` and `put` method for each type of field. In Java, an overloaded `put` method is also provided for convenience.

Some of the fields in a tuple are considered as `key` fields, as specified in the space’s definition.

**Entries**

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 TTL value (time-to-live), that is, how much time is left before the entry expires, at which time it will be deleted from the space.

The following features distinguish an entry from a tuple.

- An entry represents a tuple that has been put into a space.
- An entry includes metadata such as time-to-live, lock type, and status.

**Results**

Most 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.
How to Use a Space

A space provides the user with the ability to store data, retrieve data, and be notified when the data contained in a space is modified in any way as soon as the modification happens.

Besides reading and writing entries by key field values, it is also possible to query and iterate, consume, or lock through multiple entries stored (or updated) in a space using a SpaceBrowser. It is also possible to subscribe to a space and have a callback method invoked when any change happens on the space using a Listener.

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. It is a fact of life for any fast distributed system such as ActiveSpaces that if seeders are distributed over a network, then some operations will require at least one network round-trip time 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.

Storing Data into a Space

A user 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 will be overwritten by the new tuple.

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

- Field names and types should match the fields of the space's definition.
  
  If a tuple's field does not match the space's definition, ActiveSpaces will attempt 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.
Retrieving Data from a Space

ActiveSpaces offers three ways to retrieve (or consume) data from a space:

- A tuple space is an implementation of the associative memory paradigm and allows the user to get a complete copy of the tuple of the tuple associated with specific values of its key fields. This is done by using the space’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 equal to `NOT_FOUND`.

- It is possible to create listeners on a space that invoke a user callback function as filtered initial data and new data are pushed from the space to the listeners. For more information on listeners, see Listening to Changes on page 35.

- It is also possible to create space browsers on the space that let the user retrieve filtered data initially stored in the space and new data tuple by tuple and on demand. For more information on space browsers, see Using Space Browsers on page 32.

The choice of which method to use to retrieve data from a space should be obvious to the user according to its application logic: for retrieving a single tuple using an exact key match the `get` function should be used. For retrieving either all or a filtered subset of the data contained in the space both listeners and space browsers offer the same functionality and the choice of which one to use depends on whether the user needs a multi-threaded event-driven callback-oriented approach, or would rather iterate through the tuples at its own pace (i.e. on demand, using the space browser’s `next` method).

Consuming or Removing Data from a Space

Tuples can be removed 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 as 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 to the `take` operation will be equal to `OK`, the complete tuple will be 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 `NOT_FOUND`. 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 `NOT_FOUND`. 
This means that, unlike a simple delete operation that would succeed even if there is nothing to delete, the take operation can be used to effectively "consume" data from a space (for example using a space browser), and therefore to easily distribute workload using ActiveSpaces.

It is also possible to perform a take operation on all or a filtered subset of the tuples contained in a space using a space browser. For more information on space browsers, see Using Space Browsers on page 32.

Concurrently Updating Data in a Space

When multiple processes concurrently get and update tuples in a space, there is a possibility that two processes may try to update the same tuple at the same time. In that case, it is often necessary to serialize those 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,” there is a possibility that both transactions starting at the same time get the same current account balance, both apply their individual change in the account value, and which ever of the two is the last to set the new account balance will overwrite the other’s modification.

There are two ways to solve this problem using ActiveSpaces:

1. An optimistic approach is best suited when the likelihood of having such a collision is low. In this case, application programmers 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 touched in any way, the operation will fail, which is an indication for the programmer that he should refresh its view of the data and re-apply the change.

2. A pessimistic approach to the concurrent update problem is best suited 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 tuples in the space. The space’s lock method is an atomic get and lock, and takes the same argument as the get method.
A locked tuple is read-only for all members of the space except the one that has locked it. Only one member can lock a specific tuple at any given time. If a member other than the lock owner tries to overwrite, take, update, or lock a locked tuple, that operation may block until the lock is cleared (if that behavior is desired for the space, in which case please refer to the space’s LockWait attribute) or will fail, indicating that the tuple is locked.

Once a tuple is locked, the owner of the lock can either unlock it, remove it from the space, or update it (which automatically clears the lock).

It is also possible to iteratively lock all or a filtered subset of the entries in a space using a space browser.

Finally, it is possible to 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.

Using Space Browsers

Another method of interacting with the space can be used when working with groups of tuples, rather than with the single tuple key lookup of the space’s get method. Space browsers allow the user 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, it is continuously updated according to the changes in the data contained in the space being browsed.

This means that 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.
Entry browsers and the event browser

There are two main types of space browser: entry browsers (of which there are three kinds) and event browsers. The three kinds of entry browser are `GetBrowser`, `TakeBrowser`, and `LockBrowser`. Entry browsers allow the user to not only retrieve the next entry in a series of entries (a GET browser), but also to directly operate on it, that is, to take or consume it (a TAKE browser) or lock it (a LOCK browser). Event Browsers allow the user to iterate through the stream of events (changes) happening on the Space.

The following points address further differences between entry browsers and event browsers:

- Entry browsers and event browsers both have two methods, `next()` and `stop()`. But an entry browser’s `next` method returns a `SpaceEntry`, while the event browser’s `next` method returns a `SpaceEvent`.
- An entry browser also has a `getType()` method, which the event browser does not have.
- An entry browser’s `next` method will do a `get`, `take`, or `lock`, according to the browser’s type: `GetBrowser`, `TakeBrowser`, or `LockBrowser`.
  - The `GET Browser next()` method does a `get` on the next entry to browse (very much like a regular iterator).
  - The `TAKE Browser next()` method, however, does a `take` on the next entry (that is, it atomically retrieves and removes the next entry currently available to take from the space).
  - The `LOCK Browser next()` method does a `lock` on the next entry to browse (that is, it atomically retrieves and locks the next entry currently available to lock in the space).
- The `EventBrowser next` method returns a `SpaceEvent` rather than an entry.
- 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 the user to see deletions and expirations of entries they have already iterated through.

Space browsers deliver the entries (and initial PUT events) for the initial values in no particular order and in an order that may change from one instance of a space browser to another.
Since a space browser is continuously updated, it does not have a `hasNext` method, but instead it has a timeout: the amount of time the user is willing for his 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). Being continuously updated means that if multiple browsers of type `TAKE` created on the same space are used to take entries from the space using `next`, a particular entry will only be 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 of two scopes, *time scope* and *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 down the period of time of interest.

- `snapshot` means that the browser starts with all the entries in the space at the time the browser is created (or initial values), but is not updated with new entries that are put into the space after that moment.
  - Note that the browser’s timeout value is ignored when the Time scope is `snapshot` since in this case the browser will only iterate through a finite set of entries (only those that are present in the Space at the time of the browser’s creation).

- `new` means that the browser starts empty, and is updated only with entries (or associated events) put into the space after the moment of the browser’s creation.

- `all` means that the browser starts with all the entries in the space, and is continuously updated with new entries.

- `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 entries or events being browsed.

- `all` is used to browse over all the entries (or associated events) in the space

- `seeded` is used to browse only over the entries (or associated events) actually distributed to the member creating the browser
Listening to Changes

ActiveSpaces has the ability to 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. 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 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 single callback function, which will be invoked for all event types, to the `tibasListener_Create` function. The new `tibasListener` object created by `tibasListenerCreate` is then activated using the `tibasMetaspace_Listen` or `tibasSpace_Listen` functions. The callback function will be passed a `tibasSpaceEvent` object, whose type can be determined by invoking the `tibasSpaceEvent_GetType` function.

- 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 joins or leaves, and now the local node is seeding or unseeding. 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 or leaves, and now the local node is seeding or unseeding. Only applicable if the listener distribution scope is SEEDED.

It is also possible to ask for a current snapshot of the entries stored in the space (sometimes referred to as initial values) to be 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 the user to specify a filter string 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

The way filters work in ActiveSpaces is similar to the way message selectors work in JMS (Java Message Service).

Filters can make reference to any of the fields contained in the Tuples. Filters do not provide any ordering or sorting of the entries stored in the space.
### Operators supported in filters

Table 4 shows the operators that are supported in the ActiveSpaces filters:

**Table 4  Operators for ActiveSpaces filters**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;, &gt;=</td>
<td>greater than</td>
</tr>
<tr>
<td>NOT or ! or &lt;&gt;</td>
<td>not</td>
</tr>
<tr>
<td>*</td>
<td>multiply</td>
</tr>
<tr>
<td>=</td>
<td>equal</td>
</tr>
<tr>
<td>!=</td>
<td>not equal</td>
</tr>
<tr>
<td>ABS</td>
<td>absolute value</td>
</tr>
<tr>
<td>MOD</td>
<td>modulo</td>
</tr>
<tr>
<td>NOTBETWEEN</td>
<td>not between</td>
</tr>
<tr>
<td>BETWEEN</td>
<td>between</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>NOTLIKE</td>
<td>not like (regex)</td>
</tr>
<tr>
<td>LIKE</td>
<td>like (regex)</td>
</tr>
<tr>
<td>&lt;, &lt;=</td>
<td>less than</td>
</tr>
<tr>
<td>+</td>
<td>addition</td>
</tr>
<tr>
<td>OR</td>
<td>or</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
</tr>
<tr>
<td>AND</td>
<td>and</td>
</tr>
<tr>
<td>IN</td>
<td>range, as in “age in (1,3,5,7,11)”</td>
</tr>
<tr>
<td>NULL</td>
<td>does not exist</td>
</tr>
<tr>
<td>NOT NULL</td>
<td>exists</td>
</tr>
<tr>
<td>IS</td>
<td>only used with NULL or NOT NULL, as in “x IS NULL” or ”x IS NOT NULL”</td>
</tr>
</tbody>
</table>
Table 4  Operators for ActiveSpaces filters

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOR</td>
<td>nor, as in &quot;age NOT 30 NOR 40&quot;</td>
</tr>
<tr>
<td>/* ....</td>
<td>comments</td>
</tr>
<tr>
<td>// ....</td>
<td>comments</td>
</tr>
</tbody>
</table>

Table 5  Formats for filter values

Table 5  Formats for filter values

<table>
<thead>
<tr>
<th>Octal Value</th>
<th>\oXXX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hexadecimal Value</td>
<td>\xXXX</td>
</tr>
<tr>
<td>Exponents (as in 1E10 or 1E-10)</td>
<td>XXXEYY</td>
</tr>
<tr>
<td>Date Time</td>
<td>YYYY-MM-DDTHH:MM:SS</td>
</tr>
<tr>
<td>Date</td>
<td>YYYY-MM-DD</td>
</tr>
<tr>
<td>Time</td>
<td>HH:MM:SS:uuuu+-XXXXGMT</td>
</tr>
<tr>
<td>True</td>
<td>TRUE</td>
</tr>
<tr>
<td>False</td>
<td>FALSE</td>
</tr>
</tbody>
</table>
Remotely Invoking Code over a Space

ActiveSpaces offers the ability for Space members to remotely invoke code on other members of the Space. This feature allows the code to be collocated with the data for optimal performance.

To illustrate, 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 will incur a fair amount of network traffic, since retrieving an entry may require a round-trip over the network, and updating that entry may require another round-trip over the network. The latency induced by those network round-trips will have a negative impact on the overall throughput of the processing.

- 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 will update 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 currently only available in Java, and therefore requires that all seeders on the Space be Java processes (including the as-agents if there are any). It only takes care of method invocation and not of distributing the code to the Space members; for example, the method being invoked must be available in the CLASSPATH of all Space members.

Invocation patterns

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 will be 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 InvocableMember) 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 InvocableMember 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**

The Enterprise Edition of ActiveSpaces offers the ability 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 are 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 1.1, 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. This also means that there is no central server used for the coordination of operations, but rather any number of peers working together to offer a common service.

In the case of storing a tuple, ActiveSpaces uses a distributed hashing algorithm applied on the values of the key fields of a tuple to distribute as evenly as possible the seeding of the tuples (that is, the 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. It also means that the more seeders to a space, the larger the amount of data that can be stored in that space.

This means that 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

Hosts participating in the same metaspace communicate with each other primarily using the TCP listen protocol, but also use a discovery protocol (either Rendezvous or the embedded PGM reliable discovery protocol) to discover each other and in some cases for the internal dissemination of events. While it is possible to encapsulate the discovery communications over TCP (for example using Rendezvous’ remote daemon connection feature or the TIBCO Messaging
Appliance), it is likely that the most common deployment of ActiveSpaces will be on machines homogeneously interconnected by a multicast-enabled LAN in order to benefit from the simplified metaspace connection arguments that multicast allows.

In this default deployment scenario, metaspace members need to be able both to receive each other’s multicast transmissions and to 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 Connecting with a Metaspace: Special Considerations on page 42.

Agents

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. Agents can also be used to ensure that the desired degree of replication specified for a space can be achieved.

For more information, see Using as-agent on page 107.

Connecting with a 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. It is however possible to connect simultaneously to several different Metaspaces

- When a process joins a space through its metaspace connection, it will only join a space once. If you call `getSession` 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.
Chapter 4  Architecture

This chapter explains the architecture of TIBCO ActiveSpaces.

Topics

• Result and Status Codes, page 46
• The ASCommon Object, page 50
• Metaspace, page 51
• Managing Spaces, page 56
• Space Membership, page 66
• Getting a Space Definition, page 57
• Tuples, page 69
• Batch operations, page 74
• Locking and Working with Locked Entries, page 75
• SpaceEntry, page 76
• Listeners, page 77
• SpaceEvent, page 79
• Entry Browsers, page 80
• Event Browsers, page 82
• Transactions, page 83
• Persistence, page 84
• Remote Space Invocation, page 85
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 entry in the space or because the entry 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 entry being returned, this entry 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 entry 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>
<thead>
<tr>
<th>Constant</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>no error</td>
<td>The operation was successful and an entry was found.</td>
</tr>
<tr>
<td>NOT_FOUND</td>
<td>error</td>
<td>The operation was successful from a system standpoint, but a matching entry was not found in the space.</td>
</tr>
</tbody>
</table>
### Table 6: Status Codes

<table>
<thead>
<tr>
<th>Constant</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALREADY_EXISTS</td>
<td>error</td>
<td>The operation was successful from a system standpoint, but updating from null failed because there is currently an entry in the space for the requested key field(s) value(s).</td>
</tr>
<tr>
<td>LOCKED</td>
<td>error</td>
<td>The operation was successful from a system standpoint, but the entry was still locked by the end of the LockWait timeout.</td>
</tr>
<tr>
<td>NOT_LOCKED</td>
<td>error</td>
<td>The attempt to unlock the entry was successful from a system standpoint, but the entry was already unlocked.</td>
</tr>
<tr>
<td>INCOMPATIBLE_TYPE</td>
<td>error</td>
<td>The operation was successful from a system standpoint, but the type of a field contained in the tuple is incompatible with the space definition.</td>
</tr>
<tr>
<td>INCOMPATIBLE_TUPLE</td>
<td>error</td>
<td>The operation was successful from a system standpoint, but the name of a field contained in the tuple is incompatible with the space definition.</td>
</tr>
<tr>
<td>REMOTE_ERROR</td>
<td>error</td>
<td>The remote operation, such as persister operation, experienced an error.</td>
</tr>
<tr>
<td>MISMATCHED_LOCK</td>
<td>error</td>
<td>The lock expired in the space and another member already locked the entry.</td>
</tr>
<tr>
<td>SYS_ERROR</td>
<td>severe error</td>
<td>The operation failed because of a system error.</td>
</tr>
<tr>
<td>Constant</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>INVALID_LOCK</td>
<td>runtime exception</td>
<td>The lock object passed to the function is invalid.</td>
</tr>
<tr>
<td>INVALID_COLLECTION</td>
<td>runtime exception</td>
<td>The list object passed to the function is invalid.</td>
</tr>
<tr>
<td>INVALID_DATETIME</td>
<td>runtime exception</td>
<td>The datetime object passed to the function is invalid.</td>
</tr>
<tr>
<td>INVALID_ARG</td>
<td>runtime exception</td>
<td>One of the arguments passed to the function was invalid (such as an invalid enumerated variable).</td>
</tr>
<tr>
<td>INVALID_NAME</td>
<td>runtime exception</td>
<td>The name of the metaspace/space/field is invalid.</td>
</tr>
<tr>
<td>INVALID_URL</td>
<td>runtime exception</td>
<td>The URL used to connect to the metaspace is invalid.</td>
</tr>
<tr>
<td>INVALID_ADMIN</td>
<td>runtime exception</td>
<td>The Admin object passed to the function is invalid.</td>
</tr>
<tr>
<td>INVALID_MEMBER</td>
<td>runtime exception</td>
<td>The Member object passed to the function is invalid.</td>
</tr>
<tr>
<td>INVALID_METASPACE</td>
<td>runtime exception</td>
<td>The Metaspace object passed to the function is invalid.</td>
</tr>
<tr>
<td>INVALID_SPACE</td>
<td>runtime exception</td>
<td>The Space object passed to the function is invalid.</td>
</tr>
<tr>
<td>INVALID_TUPLE</td>
<td>runtime exception</td>
<td>The Tuple object passed to the function is invalid.</td>
</tr>
<tr>
<td>INVALID_ENTRY</td>
<td>runtime exception</td>
<td>The Entry object passed to the function is invalid.</td>
</tr>
<tr>
<td>INVALID_BROWSER</td>
<td>runtime exception</td>
<td>The Browser object passed to the function is invalid.</td>
</tr>
</tbody>
</table>
### Result and Status Codes

<table>
<thead>
<tr>
<th>Constant</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INVALID_EVENT_BROWSER</td>
<td>runtime exception</td>
<td>The EventBrowser object passed to the function is invalid.</td>
</tr>
<tr>
<td>INVALID_LISTENER</td>
<td>runtime exception</td>
<td>The Listener object passed to the function is invalid.</td>
</tr>
<tr>
<td>INVALID_FILTER</td>
<td>runtime exception</td>
<td>The Filter object passed to the function is invalid.</td>
</tr>
<tr>
<td>INVALID_EVENT</td>
<td>runtime exception</td>
<td>The Event object passed to the function is invalid.</td>
</tr>
<tr>
<td>INVALID_ITER</td>
<td>runtime exception</td>
<td>The iterator object passed to the function is invalid.</td>
</tr>
<tr>
<td>INVALID_DEF</td>
<td>runtime exception</td>
<td>The definition object passed to the function is invalid.</td>
</tr>
<tr>
<td>INVALID_ACTION</td>
<td>runtime exception</td>
<td>The action object passed to the function is invalid.</td>
</tr>
<tr>
<td>INVALID_ACTION_RESULT</td>
<td>runtime exception</td>
<td>The action result object passed to the function is invalid.</td>
</tr>
<tr>
<td>INVALID_OP</td>
<td>runtime exception</td>
<td>The op object passed to the function is invalid.</td>
</tr>
<tr>
<td>INVALID_PERSISTER</td>
<td>runtime exception</td>
<td>The persister object passed to the function is invalid.</td>
</tr>
<tr>
<td>INVALID_RESULT</td>
<td>runtime exception</td>
<td>The result object passed to the function is invalid.</td>
</tr>
</tbody>
</table>

Table 6  Status Codes
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.
Metaspace

A metaspace is a logical concept representing two things.

1. From a deployment perspective, it represents the cluster of hosts and processes sharing the same metaspace name and set of discovery transport attributes, making a particular instance of an ActiveSpaces deployment. The hosts and processes in a metaspace can work together by joining the same tuple spaces.

2. From an administrative point of view, a metaspace is a container for a set of spaces. There are two kinds of spaces contained in a metaspace: system spaces, which are defined by ActiveSpaces itself, and user spaces, meaning spaces that are defined by a user.

The metaspace is created when the first process connects to it, and it disappears when the last process disconnects from it. Once a metaspace is created, it initially contains only the system spaces. As users create spaces in the metaspace (either using the administrative CLI tool, or the AS API calls), the definition of those spaces (along with other administrative data) is stored in some of the system spaces. This means that space and field definitions are not persisted beyond the last member of the metaspace disconnecting from it.

Multiple metaspaces may exist at the same time, each one containing a completely independent set of spaces. This means for example that changes to a space called clients in a metaspace named Dev have no impact on a space named clients in a metaspace named Prod. Since any single application can not connect to two different metaspaces using the same metaspace name, Metaspaces should always use different names.

In the API, the metaspace is represented by a metaspace object that is created using the Metaspace's static connect method. One of the members of the metaspace takes on the role of manager for the metaspace membership. In the API, the role of MEMBER or MANAGER for a metaspace member is represented by the enum Member.ManagementRole (tibas_managementRole in C).

Connecting to the Metaspace

Typically, one of the first things an ActiveSpaces process does is connect to a metaspace. The Metaspace.connect() method—or, in C, the tibasMetaspace_Connect() call—takes two parameters:

- name
- ConnectionDef object

The parameters are explained in more detail in the sections that follow.
Metaspace Name

The metaspace name is a string containing the name of that 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.

ConnectionDef Object

The ConnectionDef object contains the attributes of the connection to the Metaspace:

- discovery attribute
- listen attribute

Discovery Attribute

The discovery attribute describes the method used to discover the other members of the Metaspace on the network. Discovery of the current Metaspace members is facilitated through the use of multicasting, and the attribute of this discovery mechanism are expressed in the form of an URL. All intended members of a metaspace must specify compatible discovery URLs in order for them to become members of the same metaspace. 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, will be created instead of a single one.

TIBCO Rendezvous Discovery URL format

The discovery URL to use TIBCO Rendezvous has the following format:

\texttt{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:

\texttt{interface;discovery\_group\_address}

If not specified, ActiveSpaces will use the default interface and discovery group address 239.8.8.9 (so the URL will be equivalent to
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*.

**PGM (Pragmatic General Multicast) URL format**

The following discovery URL format means that the PGM discovery transport is used:

```
tibpgm://destination port/
interface;discovery group address/optional transport arguments
```

- **destination port** specifies the destination port used by the PGM transport. If not specified, it will use the default value of 7888.
- **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.
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 indicates 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.

Disconnecting from the Metaspace

When applications terminate or no longer need to use the spaces in a particular metaspace they should disconnect from it using the metaspace’s disconnect method. The disconnect 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(), browse, browseEvents, listen function call increments the use count on a Metaspace object. This use count is used to ensure that 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 metaspace disconnect happens as expected. This is the case both for JAVA and for C APIs.
**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 a unique member name within that metaspace. It is possible for the user to get access to this name by invoking the metaspace’s `getSelfMember` method which returns a `Member` object.

**Getting the List of User-defined Space Names**

The list of names of the user spaces that are currently defined in the metaspace can be retrieved using the metaspace’s `getUserSpaceNames` method, which in Java returns a `String[]` and in C returns a `StringList` object (`tibasStringList *`).

**Setting or Getting the Log Level**

The amount of logging produced by the ActiveSpaces core library can be adjusted and retrieved using the metaspace’s `setLogLevel` and `getLogLevel` methods respectively. `ERROR` is the lowest level, followed by `WARN`, and `INFO` (the highest level).
Managing Spaces

Spaces are the main feature that is offered by ActiveSpaces. They can be seen as a form of virtual shared memory that can be used by many applications distributed over a network to store, retrieve, and consume data (as well as a data distribution and synchronization mechanism) in a platform independent manner.

A successful connection to a particular metaspace enables the user to define, drop, join, and leave spaces, and also get an existing space’s definition and list of members.

Defining a Space

In ActiveSpaces, a space must first be defined in the metaspace before it can be joined by applications and agents. Once a space is defined, it is actually created when a member of the metaspace joins it (and becomes the first member of that space). Conversely, it is destroyed (although it remains defined) when the last member leaves it (and there are no more members of the space).

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 some of the system spaces).

Space Definition Through the Admin CLI

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

Example creating a space using the Admin CLI:

```plaintext
define space 'myspace' ('Key' int, 'Value' string) key ('Key')
```

Space Definition Through the API

Using the SpaceDef object Defining a space through the API is done using the defineSpace method of the Metaspace object, which takes a SpaceDefinition object as its sole parameter.
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')```

Note that space names are case-sensitive.

**Getting a Space Definition**

It is possible to get the space definition for a space that has been previously defined in the metaspace 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**

It is possible to 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.

**Space Definition**

A space definition is composed of 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 represented in the API by a `SpaceDef` object that is either created from scratch by invoking the `SpaceDef`’s `create()` method, or returned by the metaspace or space’s `getSpaceDef` methods.
Space Attributes

The attributes of spaces include distribution, replication, expiration, and TTLs, each of which is described in this section.

Distribution

A space may be either distributed or non-distributed. For distributed spaces, management of the entries in the space is shared among the seeders that have joined the space. If the space is non-distributed, a single seeder is responsible for all entries in the space. (For non-distributed spaces, other seeders may still store the entries in the space, according to the replication degree.)

When a space is distributed, the responsibility of the storing of the tuples is evenly distributed amongst all the seeders joined to the space. When a space is non-distributed, the responsibility for storing all the tuples in the space is assigned to one of the seeders joined to the space (other seeders joined to the space may also replicate those tuples if a degree of replication is specified).

In order to offer the best possible (most even) distribution of entries in a space regardless of the number of entries in the space, 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. There is no need to define a number of “partitions” for the space that would only provide optimal distribution when the number of entries stored in the space (or the number of seeders for the space) is within a certain range: in essence, every single entry in the space is a partition.

The space’s distribution policy can be set or gotten 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.

Distributed Space

By default, spaces are distributed. This means that 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 amongst 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 to a single key in a distributed space, and ensures that those changes are propagated as they happen. This means that ActiveSpaces guarantees that every single member of the spaces will see changes to the values associated with a particular key in the exact same order, regardless of physical location or level of participation in the space.

Non-distributed Space

A non-distributed space will be entirely managed by a single member. The primary 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 moment in time, one member of the space is in charge of managing the entries for the space. It is called the seeder. 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, it means that the space will not be usable until that 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

In order to provide fault-tolerance and to prevent loss of tuples if one of the seeders of a space suddenly disappears from the metaspace, it is possible to specify a degree of replication for a space.

Replication in ActiveSpaces is performed in a distributed active-active manner: seeders both seed some entries and replicate entries assigned to other seeders. This means that the replication itself is distributed: rather than having a designated backup for each seeder that replicates all of the entries that this seeder seeds, the entries that it seeds are replicated by all of the other seeders. ActiveSpaces seeders, then, work in an active-active mode, in the sense that there are no "backup seeders" just waiting for a "primary seeder" to fail to start becoming active. Instead, all seeders are always active both seeding and replicating entries. This is a more efficient mode of replication because it means that if a seeder fails, there is no need to redistribute the entries amongst the remaining seeders to ensure that the distribution remains balanced. All that is needed is for ActiveSpaces to rebuild the replication degree, which is less work than having to redistribute, as well, and therefore ensures that there is less impact on performance when a seeder fails.
A degree of replication of 0 (the default value) means 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 will disappear from the space. If, instead, that member leaves in an orderly manner by invoking the call to leave the space or the call to disconnect from the metaspace, there is no loss of data.

A degree of replication of 1 means that each tuple seeded by one member of the space will also be replicated by one other seeder of the space. If a seeder suddenly disappears from the metaspace, the entries that it was seeding will automatically be seeded by the nodes that were replicating them, and no data is lost. Seeders do not have designated replicating members; rather, all of the entries 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 entries are still evenly balanced over the remaining set of seeder members of that space. It also means that ActiveSpaces’s fault-tolerance is achieved in an active-active manner.

A degree of replication of 2 (or 3 or more) means that each tuple seeded by a member of the space is replicated by 2 (or 3 or more) other seeder members of the space. This means that up to 2 (or 3 or more) seeder members of a space can suddenly disappear of the metaspace (before the degree of replication can be re-built by the remaining members of the space) without any data being lost.

A special replication degree of REPLICATE_ALL is also available. When it is specified for a space, all of the entries in that space will 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 entry in the space, and can therefore perform a read operation locally, using either its seeded or replicated view of the entry.

The degree of replication can be set or gotten using the SpaceDef object’s setReplicationCount or getReplicationCount methods. The default value is 0, that is, no replication.

**Synchronous and Asynchronous Replication**

Along with the degree of replication, it is also possible to define whether replication will happen in synchronous or asynchronous mode for the space.
Asynchronous replication

Asynchronous replication is the default behavior. It offers the best performance, since the amount of time it takes to complete an operation that modifies one of the entries in the space does not depend on the degree of replication. While the asynchronous replication protocol of ActiveSpaces has very low latency and is resilient in most failure scenarios, it does not offer an absolute guarantee that the data modification has been replicated to the desired degree by the time the operation completes.

Using asynchronous replication does not imply a degradation in the coherency or consistency of the view of the data between members of a space. Under normal operating conditions all members of the space will be notified of the change in the data almost at the same time as the operation returns, even when asynchronous replication is used.

Synchronous replication

Synchronous replication offers the highest level of safety, at the expense of performance. When synchronous replication is in use for a space, it means that an operation that modifies one of the entries in the space will only return an indication of success when that modification has been positively replicated up to the degree of replication required for the space.

Comparison of asynchronous and synchronous replication

One implication of the type of replication of a space on operations on that space is that asynchronous replication is more permissive in the sense that it allows operations that modify the data stored in a space even when the degree of replication can not be achieved at the time for lack of enough seeders. (Replication will be automatically achieved, however, as soon as enough seeders have joined the space.) Synchronous replication does not allow such operations. Asynchronous replication, then, is a best effort quality of replication, while synchronous replication is a strict enforcement of the replication degree.

Example

Assume that a space is defined with a degree of replication of 1, meaning the data should be seeded by one member and replicated by another seeder member of the space. Further assume that there is only one seeder member of the space at a given time. In this case, if the space uses synchronous replication, operations other than get will fail (until a second seeder joins the space). If the space uses asynchronous replication, all operations will be allowed, but the replication degree will not be achieved (until a second seeder joins the space).

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.
Chapter 4  Architecture

Capacity

It is possible to define a capacity for the space in order to control the amount of memory used by the seeders for storing and replicating entries in the space. The capacity is expressed in number of entries per seeder and defaults to -1, which means an infinite number of entries per seeder.

If a capacity is specified, the eviction policy is used to indicate the outcome of an operation that would result in an additional entry being seeded by a seeder that is already at capacity. The two choices for the eviction policy are exception, which means that the operation will fail with the appropriate exception being stored in the Result object, or eviction, which means that the seeder will evict another entry using the eviction algorithm Least Recently Used (LRU), meaning least recently read or modified entry) eviction algorithm.

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.

Persistence

Since ActiveSpaces stores the entries of a space in the memory of the space's seeders, the data contained in a space disappears when the last seeder of the space leaves. In order to be able to maintain the data in the space through times where no seeders are available, it is necessary to persist this data to another (slower but more permanent) data storage system such as a database, a key-value store, or even a file-system. ActiveSpaces allows a space to be marked as persistent, meaning that the space can be "re-hydrated" from the persistence layer at startup, and changes to the space can be reflected to that persistence layer.

If the space is defined as both persistent and with a capacity and an eviction policy of LRU, then it is possible to 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"). It should be noted that in this mode of operation (unlike key value operations), browsers and listeners operate only on the entries stored in the space at the time they are used, which could represent a subset of all of the data contained in the persistent data-store.

A Space can be marked as persisted (in which case it will need at least one persister as well as the minimum number of seeders to be usable), or not (in this case it does not need a persister but still needs seeders).
Expiration and TTLs

This section explains the concepts of expiration, time-to-live (TTL), and LockTTL as they are used in ActiveSpaces.

**Entry TTL**  ActiveSpaces is a data-grid, not a cache. This means that entries stored into a space will not be evicted from the space in order to make room for a new entries unless the space is specifically configured to evict. It is for this reason that a space can be used as a *system of record*. However, in some cases it is convenient to have an automated *garbage collection* mechanism built into the space in order to expire old obsolete entries from it. In order to do this, it is possible to define a time-to-live for the entries stored in a space. The TTL is the number of milliseconds that must have elapsed since the entry was created or last modified before it will be considered for expiration.

The entry TTL can be set or gotten using the `SpaceDef` objects's `setTTL` and `getTTL` methods, respectively. The default value (`DEFAULT_ENTRY_TTL`) is `TTL_FOREVER`, which means that entries in the space never expire. When an entry 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.

**LockTTL**  Applications that have joined a space have the ability to lock entries in that space. By default, locked entries will remain locked until either the application that created the lock clears it, or until that application disconnects from the metaspace, whether in an orderly manner or not. However, in order to avoid potential dead-lock situations, it is also possible to specify a maximum lock-time-to-live for a space, which means that if an application does not clear a lock on its own within a certain number of milliseconds, then the lock will automatically be cleared.

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

**LockWait**  While an entry is locked, no space member besides the creator of the lock can perform an operation on it other than a get. For this reason, 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 entries for the shortest possible amount of time in order to maximize concurrency of the overall process.

If there is an expectation that on a space the locking of entries will be a very short-lived event, ActiveSpaces can greatly simplify the application programmer’s job by providing the ability to specify a LockWait value for the space. The LockWait value is the number of milliseconds an operation attempting to modify a locked entry can block while waiting for the lock to clear. If at the end of the LockWait period the entry is still locked then the operation to modify that
locked entry will throw an exception indicating that the operation could not be performed because the entry 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, entries about to be replaced by uncommitted operations are locked by that transaction).

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 entry will immediately return with a failure indicating that the entry is currently locked.

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) or the underscore (_) character and then contain any combination of letters (upper or lower case) and numbers.

Note that field names are case-sensitive.

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB</td>
<td>A BLOB (binary large object) type is an array of 8 bit values.</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>The BOOLEAN type has one of two values, true or false, represented by the integers 1 or 0, respectively.</td>
</tr>
<tr>
<td>CHAR</td>
<td>Char type represents a char.</td>
</tr>
</tbody>
</table>
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.

### Key Fields

At least one of the defined fields needs to be used as a *key field* for the space definition to be valid. The set of fields to be used as key fields can be specified using the `SpaceDef`’s `setKey` method and passing it a number of strings containing the space names in Java, or a single string containing a space-separated list of field names in C. It is also possible to get a list of the name of the fields marked as key fields in a `SpaceDef` by using the `SpaceDef`’s `getKey` method.

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.
Space Membership

Joining a Space

Once a space has been defined in the metaspace, it can then be joined by applications and will be automatically joined by agents if the space is distributed. A space can be joined, and a valid space object obtained, by invoking the getSpace method of the Metaspace object, which takes a space name and a distribution role as arguments. The space name must be the name of a space defined in the metaspace, and the role specifies if the application wants to join the space as a seeder or as a 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.

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 will return a reference to a Space object that can then be used to interact with the space. Users should also remember that 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.

The space object is reference-counted, meaning that successive calls to getSpace will 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 that second getSpace call). An application's role for a space can also be adjusted using the Space's setDistributionRole method.

Leaving a Space

An application can leave a space by invoking the space object's leave() method (or, in C, by invoking tibasSpace_Free), or by stopping the listeners or browsers that may have joined the space automatically when created.

As the space object is reference-counted, 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. Just like 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 will be adjusted when leave() is invoked on the instances of the Space object returned by each call to getSpace.
Persisting on a Space

Because ActiveSpaces is a true peer-to-peer distributed system, it is the responsibility of some of the space members — the *persisters* — to provide the service of interacting with a persistence layer, just like it is the responsibility of some of the space members — the *seeders* — to 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.

Space Life Cycle

Spaces have a life cycle that starts with the space being defined for the first time 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` method 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 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 being that the space’s load method can be invoked when the space is in the LOADING state (typically by the registered persister `onLoad` method).

Applications can check that the space is in the READY state before attempting to use it by using the space’s `isReady()` method.
Applications can also synchronize themselves with the space’s state by using the space’s `waitForReady` method. This method takes a timeout which is the number of milliseconds for which it will block while waiting for the space to reach the READY state and returns a boolean indicating whether the timeout was reached or not (Java also has a convenience version of the method that does not take a timeout and just blocks until the space is ready).
Tuples

One of the simplest ways 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 entry 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. It is possible, for example, to extract a tuple out of the entry 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 it is possible to 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 lower case) or the underscore character (_), and then must contain 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 64 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 a convenience 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 `TIBAS_NOT_FOUND` 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>
<thead>
<tr>
<th>Upcasting Supported to...</th>
<th>Type of Field in Tuple</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short</td>
<td>Short, Int, Long, Float, Double</td>
</tr>
<tr>
<td>Int</td>
<td>Short, Int, Long, Float, Double</td>
</tr>
<tr>
<td>Long</td>
<td>Short, Int, Long, Float, Double</td>
</tr>
<tr>
<td>Float</td>
<td>Short, Int, Long, Float, Double</td>
</tr>
<tr>
<td>Double</td>
<td>Short, Int, Long, Float, Double</td>
</tr>
<tr>
<td>Blob</td>
<td>Blob, String</td>
</tr>
</tbody>
</table>

It is possible to 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.

It is also possible to 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 introspect 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.

It is possible to 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**

It is possible to get the name and the space definition of the space represented by the `Space` object by invoking the `getName` and `getSpaceDef` methods respectively.

**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 an entry 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 entry inside a Result object. (In the case of the `take` method, it also atomically removes that entry from the space.)

If there is currently no entry in the space containing a tuple whose key field values match exactly the values in the tuple passed as argument, the value of the Status field in the returned Result (or the return value of the function in C) will be equal to NOT_FOUND. Users should be aware that this situation—where there is no `Entry` object stored in the space for those key values—is not considered an error. Rather, it is an indication that we positively know there is no entry. Therefore, users should always check the value of the Status field in the returned Result object (or the return value of the function in C).

It is also possible to explicitly remove an entry that was previously obtained from the space by using the space’s `remove` method, which takes an entry (rather than a tuple) as its argument.
Storing a Tuple in a Space

Storing a tuple in a space is done using the space's put method, which takes the tuple to put as an argument and returns a copy of the entry this put now created in the space (in C it is also possible to pass a NULL instead of a pointer to a tibas_entry variable argument in order to optimize resource usage if the user does not need to get a copy of the created entry back). If the tuple contains fields that do not match the names of the fields in the space's definition, those fields are stripped from the tuple stored in the space.

If there is already an entry containing a tuple with the same key field values stored the space at the time the put is invoked, this old tuple is quietly replaced by the new one. If this behavior is not desired, and the users wants to avoid overwriting an existing entry by mistake, a special form of the space's update method should be used.

At the time the put is invoked, if there is already a matching entry in the space and that entry is locked, the method may block. It may block for up to the amount of time specified in the space definition's LockWait attribute. After this amount of time, if the stored entry is still locked, the put will fail, and that 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.

It is also possible to atomically put and lock a tuple in a space using the putAndLock method.

Updating a Tuple in a Space

It is possible to perform an atomic compare and set operation on a tuple stored in a space using the space's update method. This method compares the provided oldEntry with the entry currently stored in the space, and if they are strictly equal, then atomically replaces the oldEntry with a new one containing the newtuple. The method takes two arguments, oldEntry and newtuple. The value of the Result's Status field (or the return value of the function in C) can be:

- **on success** OK, and—if the entry stored in the space at the time the method is invoked was indeed the same entry as the one provided as the oldEntry argument—a copy of the newly created entry stored in the space is contained in the Result object (or pointed to by the entry pointer argument in C); or,

- **on failure** NOT_FOUND, (if the provided oldEntry is not stored in the space any more at the time the method is invoked, either because it was replaced by a prior put or update, or because it was taken or removed).

The update method will only succeed if the key field values of the newtuple argument match those of the tuple contained in the oldEntry argument.
The only exception to this rule is that it is allowed to pass null as the `oldEntry` parameter, indicating that the method should succeed only if there is no tuple in the space with the same key field values as the `newTuple`. In the case where there is already an entry in the space containing a tuple with the same key values, the status returned by the update (from null) operation is `ALREADY_EXISTS`.

It is also possible to atomically update and lock a tuple in a space using the `updateAndLock` method.

Note that, when the `oldEntry` argument passed is a locked entry, the update operation also has the effect of clearing the lock. This is because most of the time once an application has locked an entry it will then usually either unlock it or update its value and unlock it. Having the update operation perform the updating of the value of the locked entry and the unlocking of that updated entry in a single operation allows for optimization of the performance of the entire "lock-update-unlock" operational pattern. It is therefore pointless to call unlock on an entry after it's been updated. If the application needs to update the value of the locked entry and to keep it locked, then it should use the `updateAndLock` operation instead.

Having the update operation perform the updating of the value of the locked entry and the unlocking of that updated entry in a single operation allows for optimization of the performance of the whole 'lock-update-unlock' operational pattern. It is therefore useless and wrong to call unlock on an entry after it's been updated. If the application needs to update the value of the locked entry and to keep it locked, then it should use the `updateAndLock` operation instead.
Batch operations

Due to the fact that ActiveSpaces will most of the time be deployed with seeders on many machines on the same network, it is a fact of life that some of the operations may 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`. 
Locking and Working with Locked Entries

It is possible for a user to make an entry read-only by locking it. There are two methods that create a new locked entry, `putAndLock` and `updateAndLock`. There are also two methods that can be used to lock an existing entry: using the space’s `lock` method and passing it a tuple as an argument, in which case it behaves as an atomic `get and lock`, or using that same lock method (`tibasSpace_LockEntry` in C), but passing it an entry as an argument, in which case it will attempt to lock that particular entry if it is still in the space.

Regardless of the way the method is invoked, there are two possible outcomes:

- The `Result` object’s Status field (or the return value of the function in C) is equal to `OK` and the operation was successful in which case a "new" entry is returned in the `Result` object (or pointed to by the entry pointer argument in C), or,
- The `Result` object’s Status field is equal to `NOT_FOUND` (there is no tuple with those key field values in the space, or the entry is no longer in the space), or it is equal to `LOCKED` (there is a tuple with those key field values in the space but it is locked by someone else, or someone else locked the entry).

Once an entry is successfully locked, the object returned by the successful lock method (or `putAndLock` or `updateAndLock`) is used as the key to the lock, and must be included as a parameter to any subsequent operations on that locked entry. The four possible methods that can be invoked on a locked entry are:

- **remove** Deletes the entry from the space.
- **unlock** Unlocks the entry, and makes it read-write again.
- **update** Updates the values of the non-key fields and automatically unlocks the entry. Locks may have a TTL (time-to-live), and in case the application is too slow to invoke this `update` method, the lock may have expired before the invocation, and therefore someone else may have had a chance to replace or remove it. The `update` method will succeed, even if the lock was automatically cleared because it exceeded its TTL, as long as the entry was not modified after the lock was cleared.
- **updateAndLock** Updates the values of the non-key fields and keeps the entry locked. Subsequent operations should use the updated entry returned by this operation, not the one returned by the previous operation that locked it in the first place.
SpaceEntry

A SpaceEntry represents a copy of an entry that was stored in the space at some point in time (at the time the method that returned this copy was invoked). The following methods return SpaceEntry objects: the space’s get, put, update, lock, putAndLock, updateAndLock methods, the entry browser’s next methods and, finally, the SpaceEvent object’s getEntry method.

Since an entry in a space contains the tuple being stored as well as the metadata associated with that tuple, the SpaceEntry object offers the following methods:

- **getTuple** Extracts and returns the tuple from the entry.
- **getSpaceName** Returns the name of the space to which this entry belongs.
- **getTTL** Returns the time in milliseconds remaining before the entry can be considered for expiration.
- **getLockType** Returns either NO_LOCK, meaning that the entry was not locked at the time this SpaceEntry object was generated, or EXCLUSIVE_LOCK, meaning that it was locked.
- **refresh** Attempts to refresh the metadata of the entry in this SpaceEntry object (the TTL and the LockType). It returns true if the entry is still in the space, or false otherwise (meaning that the entry has been overwritten or removed from the space since the SpaceEntry object was first generated).
Listeners

Listeners are used to monitor events that represent changes to the entries 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 entries 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 that will be invoked for all event types. This user callback function will be passed a single tibas_spaceEvent object which 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 entries. 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 entries. 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 the section on filters: see Filters on page 36. Note that some of the filtering may be executed in a distributed manner by ActiveSpaces in order to optimize performance.

A listener 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 listener contains only PUT events corresponding to the entries 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 entries 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 entries already contained in the space at creation time.
- `all` means that the listener starts with all the entries 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 entries or events being browsed.

- `all` is used to listen to events related to all entries in the space.
- `seeded` is used to listen only to events associated with the entries 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 entry.
- **UNSEED** Generated when the member that created the listener no longer seeds an entry.
SpaceEvent

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:

- **getSpaceEntry** Returns the entry associated with the space event (that is, the entry that was put or updated, the entry that was taken or removed, the entry that expired, or the entry 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`. 
Entry Browsers

An entry browser is used to iterate through the contents of a space. It is created 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.

An entry browser has two 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 entry browser:

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

You can optionally specify a filter string when creating an entry browser. A filtered entry browser will only return entries that match the specified filter. This is done by adding a filter argument when you invoke the `browse()` method on the space object. That argument contains a filter expressed as a string in the language described in the section on filters: see Filters on page 36. Note that some of the filtering may be executed in a distributed manner by ActiveSpaces in order to optimize performance.

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

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

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

The **BrowserDef** object for an entry browser can include a **timeout** value. The timeout is the amount of time for which an entry 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 entries in the snapshot have been iterated through will return `null` right away.
Event Browsers

An event browser’s `next()` method returns an event (the same kind of object you would get in a listener). The browser iterates through the events happening on a space. The list of events remaining to be iterated through is continuously updated according to changes in the space.

You can optionally specify a filter string when creating an event browser. A filtered event browser will only return events that match the specified filter. This is done by adding a filter argument when you invoke the `browseEvents()` method on the `Space` object. That argument contains a filter expressed as a string in the language described in the section on filters: see Filters on page 36. Note that some of the filtering may be executed in a distributed manner by ActiveSpaces in order to optimize performance.

Just as with an entry browser, an event browser can have either of two scopes, `time scope` and `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 down the period of time of interest.

- `snapshot` means that the browser will only deliver PUT events representing the entries stored in the space at creation time.
- `new` means that only events pertaining to entries stored or overridden in the space after creation time will be browsed on.
- `new events` means that all events happening on the space after creation time will be browsed on. Unlike the time scope `new` described above, this time scope includes events (such as TAKE or EXPIRE events) related to entries already contained in the space at creation time.
- `all` means that the browser will start by delivering PUT events for all the entries stored in the space at creation time, followed by all subsequent events on the space.

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

- `all` means that events for all of the entries stored in the space will be browsed on.
- `seeded` means that only events related to the entries seeded by the member will be browsed on.

The `BrowserDef` object for an event browser can include a `timeout` value. The timeout is the amount of time for which an event browser’s `next()` method can block while waiting for a new event. If there is still no new event to browse at the end of the timeout, the `next()` method will return `null`.
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.
Persistence

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 an entry 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 an entry by key value did not result in a matching entry being found in the space.
- **onLoad** 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 entries stored in the persistence layer.
Remote Space Invocation

TIBCO ActiveSpaces now allows applications to remotely invoke code execution over the space to other members (this feature is only available in Java in TIBCO ActiveSpaces version 1.1.1).

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 `InvocableMember`.
- The class needs to be present in all of the member's CLASSPATHs.

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).
Chapter 5  Administrative Commands

This chapter describes the commands for administering TIBCO ActiveSpaces through the Administration Command Line Interface (Admin CLI).

Topics

- The Admin CLI, page 88
- clear, page 91
- connect, page 92
- define | create space, page 94
- disconnect, page 97
- drop space, page 98
- export metaspace, page 99
- help, page 100
- show | describe member, page 101
- show | describe members, page 102
- show | describe space, page 103
- show | describe spaces, page 105
- quit | exit | bye, page 106
- Using as-agent, page 107
The Admin CLI

Administrative tasks for TIBCO ActiveSpaces are performed through a utility called the Administration Command Line Interface (Admin CLI). These tasks include connecting to a metaspace, creating a space, and displaying information about existing spaces and members. This chapter lists and describes the commands available in the Admin CLI.

Launching the Admin CLI

Launch the Admin CLI by invoking the executable file `as-admin.cmd` in a command prompt window. The default location of this file on Windows is `C:\tibco\as\1.1.1\bin`.

The examples in this section use Windows conventions. On UNIX, the Admin CLI executable is called `as-admin.sh`, instead of `as-admin.cmd`. The default location for the file on UNIX is `/usr/tibco/as/1.1.1/bin`, instead of `C:\tibco\as\1.1.1\bin`.

Using a script file to pass arguments when launching the Admin CLI

Using a script file, you can pass command line arguments when you launch the Admin CLI. Usage for the `as-admin` command is as follows:

```
as-admin -i admin-cmds-filename
```

The above usage information for `as-admin` can be obtained by invoking `ad-admin -h` from the ActiveSpaces bin directory in a command line window.

Comments can be inserted into the script file by starting a line with `#`, as follows:

```
# this is a comment
```

This mechanism allows execution of commands in a batch mode through the Admin CLI. There are several limitations on the structure of the file:

1. No white space or special characters are allowed before the command itself.
2. Each line is executed sequentially by the Admin CLI.
3. The admin tool will not automatically exit and return to the shell after executing the commands of the file unless the script file contains a quit command.

Using the Admin CLI command `export metaspace [to <filename>]` (see `export metaspace on page 99`) will generate a file containing the schema for the existing metaspace. The `admin-cmds-filename` value can be the path to this generated file. In this way, a new metaspace can be created using the exported metaspace schema.

**Example**

1. Create a text file that invokes the Admin CLI. Here is sample of the contents of such a file. In this example, the file is named `example01.ser`, and is located in the directory `C:\temp2`.

   ```
   connect name 'ms' discovery 'tibpgm' listen 'tcp';
   define space name 'tests pace' (field name 'key' type 'integer' field name 'value' type 'string') key ('key');
   ```

2. Launch the Admin CLI using the syntax given in the usage statement above, `as-admin -i admin-cmds-filename`.

   ```
   as-admin -i c:\temp2\example01.ser
   ```

**Additional characteristics of the Admin CLI tool:**

- When inside the tool, hitting the question mark key (?) at any time will bring up context-sensitive help.
- Arrow keys and the tab key function in the command window as in typical advanced shells.
- It is possible to invoke a shell command by using the escape character '!', for example, `!dir` or `!ls` to list the files in the current directory.
- Field names and literals must be enclosed in single or double quotes.
- In most cases, the Admin CLI will start by using the `connect` command to connect to a metaspace.
- The default discovery mechanism is PGM. If RV discovery or TCP unicast discovery is needed, the appropriate discovery URL needs to be specified.

For more information on transport, see Connecting to the Metaspace on page 51.
The Execute method

Admin CLI administrative commands (for example defining a space) can be executed directly from within an application. This is done by using the Metaspace object's execute method and passing it a string representing the Admin CLI command. A string is returned containing the output resulting from executing the command.
## clear

*Admin CLI Command*

<table>
<thead>
<tr>
<th>Syntax</th>
<th>clear</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>clear is used to remove all lines of text from the command window except for a single Admin CLI prompt.</td>
</tr>
<tr>
<td>Parameters</td>
<td>None.</td>
</tr>
</tbody>
</table>
connect

Admin CLI Command

Syntax  connect [name <name>] [discovery <string>] [listen <string>]

Purpose  connect is used to connect to a metaspace. Connecting to a metaspace is a necessary initializing step for the Admin CLI in order to begin working with ActiveSpaces, just as it is for an ActiveSpaces application. The Admin CLI can only be connected to one metaspace at a time.

Parameters  The table lists the parameters for this command with a description of each parameter.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Optional. If a metaspace with this name does not exist, it will be created as a result of this command. If no name is specified, the Admin CLI will connect to the default metaspace, called ms.</td>
</tr>
</tbody>
</table>
| discovery | Optional. The discovery URL can take one of 3 forms:
  - If Unicast discovery is used, then a list of ‘well known’ IP addresses and ports must be passed in a URL with the following syntax:
    tcp://ip1:port1;ip2:port2;...
  - If multicast discovery is to be used then the URL must be one of the following depending on which reliable multicast transport is to be used:
    The tibrv (multicast) URL takes three parameters: service, network, and daemon. In many cases, the default values are sufficient.
    Syntax: tibrv://service=service/network=network/
    daemon=daemon
    or
    tibpgm://destination port/interface;discovery
    group address/optional transport arguments
    See PGM (Pragmatic General Multicast) URL format on page 53 for more information of PGM discovery URLs.
    See TIBCO Rendezvous Discovery URL format on page 52 for more information on RV discovery URLs. |
The following examples illustrate the syntax of the `connect` command:

- `connect`
- `connect name <metaspace_name>
- `connect discovery <discovery_url>
- `connect listen <listen_url>
- `connect discovery <discovery_url> listen <listen_url>
- `connect name <metaspace_name> discovery <discovery_url>
- `connect name <metaspace_name> discovery <discovery_url> listen <listen_url>`

Table 8  `connect` Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>listen</td>
<td>Optional. Specifies which interface and port the administrative process should create its listening TCP socket on. Syntax: tcp://interface:port. See Listen URL format on page 54 for more information on listen URLs.</td>
</tr>
</tbody>
</table>

Example

```
name 'ms' discovery 'tcp://192.168.1.10'
```

Note that parameter values must be enclosed in either single or double quotes.
**Define | Create Space**

**Command**

```
 Syntax (define | create) space name <space-name> ( field name <field-name> type <field-type> [nullable true|false] *) key ( <field-name> [, <field-name> ... ] ) [distribution_policy <distributed|non_distributed>] [replication_count <n>] [sync_replicated true|false] [min_seeders <n>] [capacity <n>] [eviction_policy <none|lru>] [ttl <n>] [lock_ttl <n>] [lock_wait <n>] [persisted true|false]

 Purpose Used to create a space.

 Remarks Supported datatypes for fields are: boolean, char, short, integer, long, float, double, string, datetime, blob

 Parameters The parameters for this command are listed and described in Table 9, define space Parameters.

 Table 9 define space Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Required.</td>
</tr>
<tr>
<td>field</td>
<td>Required.</td>
</tr>
<tr>
<td></td>
<td>The datatype for a field must be one of the following: boolean, char, short, integer, long, float, double, string, datetime, blob.</td>
</tr>
<tr>
<td>nullable</td>
<td>Optional. Can be either true or false (no quotes). By default is equal to false. If a field has nullable set to true, it means that tuples put into the space do not need to contain a field of that name</td>
</tr>
<tr>
<td>key</td>
<td>Required. Identifies one or more fields (already specified with the field parameter) that will serve as a unique key for the space.</td>
</tr>
<tr>
<td>distribution_policy</td>
<td>Optional. Determines whether management of entries in the space is shared among the seeders that have joined the space (distributed) or a single seeder is responsible for all entries in the space (non_distributed). The default value is distributed.</td>
</tr>
</tbody>
</table>
**Table 9  define space Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sync_replicated</td>
<td>Optional. A value of true means that replication will happen in synchronous mode for the space, so that when an operation modifies one of the entries in the space, the operation will only return an indication of success when that modification has been positively replicated up to the degree of replication required for the space. The default value is false.</td>
</tr>
<tr>
<td>replication_count</td>
<td>Optional. Specifies the degree of replication for the space, i.e., the number of replicates required.</td>
</tr>
<tr>
<td>min_seeders</td>
<td>Optional. Specifies the minimum number of seeders that should be joined to the space before the space becomes ready to accept operations. The default value is 1.</td>
</tr>
<tr>
<td>capacity</td>
<td>Optional. Specifies a maximum number of entries per seeder for the space. When the capacity is reached the result of any additional request to put (insert) a new entry in the space will depend on the value of the eviction_policy attribute. The default value is -1 (no capacity).</td>
</tr>
<tr>
<td>eviction_policy</td>
<td>Optional. If a put operation on a space would cause a seeder to exceed the space’s capacity attribute, then the value of this attribute will dictate the result of this operation: if the value is 'none' (in quotes) then there will be no eviction and the operation will fail because the seeder is already at capacity. If the value is 'lru' (in quotes) then the seeder will evict another entry from the space using the 'least recently used' eviction algorithm. The default value is 'none' (no eviction).</td>
</tr>
<tr>
<td>Persisted</td>
<td>Optional. A setting of true means that the space needs to be persisted. Default value is false.</td>
</tr>
<tr>
<td>TTL</td>
<td>Time to live in milliseconds. The default is -1 (forever).</td>
</tr>
<tr>
<td>LOCK_TTL</td>
<td>Specifies in milliseconds the duration of a lock placed on the space. The default is -1 (forever).</td>
</tr>
</tbody>
</table>
Chapter 5  Administrative Commands

Examples

Simple example

```bash
define space name 'mypace' (field name 'key' type 'integer' field name 'value' type 'string') key ('key')
```

With additional parameters

```bash
define space name 'testspace' (field name 'key' type 'double' field name 'value' type 'blob') key ('key') distribution_policy 'non_distributed' replication_count 1 sync_replicated true capacity 10000 eviction_policy 'none'
```

Table 9  define space Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCK_WAIT</td>
<td>For a space that is locked, specifies how long a member process will wait for it to become unlocked. The default is -1 (forever). Other valid values are 0 or any positive value. The unit of measure is milliseconds.</td>
</tr>
<tr>
<td>update_transport</td>
<td>The transport protocol used to distribute notifications of updates to the data stored in the space. Can be either listen or discovery, signifying whether the reliable listen protocol or the reliable discovery protocol (as they are defined by the connection attributes to the metaspace) is used.</td>
</tr>
</tbody>
</table>
disconnect

Command

Syntax: disconnect

Purpose: Disconnects the Admin CLI from the metaspace to which it is currently connected.

Parameters: None.
drop space

Syntax  drop space <name>

Purpose  Drops the named space.

Remarks  It is required that no other members be connected to the space when the drop
space command is executed.

Parameters  The table shows parameters for this command.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the space to be dropped.</td>
</tr>
</tbody>
</table>
### export metaspace

**Command**

**Syntax**

```bash
export metaspace [to <filepath>]
```

**Purpose**

Exports the definitions for the metaspace and its spaces (if any are currently defined) to a designated text file or to the screen.

**Remarks**

The file that is created can be used as a parameter when invoking as-admin with the `-i` parameter, as shown below.

#### Example using as-admin.cmd -i

1. Export a metaspace to a file by invoking `export metaspace`, including `to` and a filepath parameter. Use single- or double-quotes around the filepath:

   ```bash
   as-admin> export metaspace to 'C:\temp3\saved_metaspase'
   ```

2. Quit as-admin:

   ```bash
   as-admin> quit
   ```

3. Launch as-admin with the `-i` flag and the filepath (no quotation marks):

   ```bash
   C:\tibco\as\1.1.1\bin>as-admin.cmd -i C:\temp3\saved_metaspase
   ```

4. The output will indicate that metaspace has been recreated and as-admin is now connected to this newly-created metaspace:

   ```bash
   C:\tibco\as\1.1.1\bin>"C:\Program Files\Java\jdk1.6.0\bin\java -jar ..\lib\as-common.jar -i C:\temp3\saved_metaspase
   process_id=8148
   Connected to metaspase ms
   ```

**Parameters**

The table shows parameters for this command.

#### Table 11  export metaspase Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filepath</td>
<td>Optional. The path to a location where the text file will be saved. If no filepath is given, the metaspace information is displayed on the screen.</td>
</tr>
</tbody>
</table>
# help

**Command**

<table>
<thead>
<tr>
<th>Syntax</th>
<th>help</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Provides a complete list of Admin CLI commands and their syntax.</td>
</tr>
<tr>
<td>Remarks</td>
<td>The <code>help</code> command also displays the keyboard shortcuts available within the Admin CLI. Context-sensitive help (such as the syntax for the current command) is displayed when the question mark (‘?’) character is typed. The syntax for the as-admin command, with all possible parameters, is displayed by invoking <code>as-admin -h</code>:</td>
</tr>
<tr>
<td>Parameters</td>
<td>None.</td>
</tr>
</tbody>
</table>

```plaintext
Usage::
    as-admin -h -d debug -i admin-cmds-filename
```
**show | describe member**

*Command*

**Syntax**  
show | describe member <name>

**Purpose**  
Outputs information to the screen about the member specified with the *name* parameter.

**Parameters**  
The table shows parameters for this command.

*Table 12 show member Parameters*

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name (member ID) of the member for which information is desired.</td>
</tr>
</tbody>
</table>
show | describe members

Command

<table>
<thead>
<tr>
<th>Syntax</th>
<th>show</th>
<th>describe members</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Outputs to the screen a list of all members currently running in the metaspace.</td>
<td></td>
</tr>
<tr>
<td>Remarks</td>
<td>In addition to the name (member ID) of each space in the metaspace, the output includes the role of the member in the space, that is, whether the member is a metaspace manager (for the metaspace group membership protocol) or just a metaspace member</td>
<td></td>
</tr>
<tr>
<td>Parameters</td>
<td>None.</td>
<td></td>
</tr>
</tbody>
</table>
show | describe space

Command

Syntax:  show | describe space <name>

Purpose: Outputs information to the screen about the space specified with the name parameter.

Remarks: See Output below for information on the output of the show space command.

Parameters: The table shows parameters for this command.

Table 13  show space Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The name of the space for which information is desired. The name value must be in single or double quotes.</td>
</tr>
</tbody>
</table>

Output

The output of the show space command is displayed in the form of three tables as described in this section.

Space definition attributes

The first section of the output provides information about the space's attributes, arranged in the following columns:

Space Name  The name of the space, as given when the space was defined.

Space State  The current state of the space. The value can be initial (meaning that the Space needs more seeders or more persisters), loading (meaning the space is being loaded from the persistence layer), or ready (meaning the space has enough seeders, persisters, and has been loaded)

Distribution Policy  Can be either distributed (meaning the space is distributed) or non_distributed.

Replication count  The replication count is displayed. Default is 0, meaning there is no replication.

Sync Replicated  Can be true or false.

Capacity  The capacity of the space in number of entries per seeder. -1 indicates no capacity.
Eviction policy  The policy of the eviction to be applied when a space operation would cause the capacity to be exceeded. Can be none (no eviction) or LRU (least recently used eviction).

Min seeders  The minimum number of seeders that need to be joined to the space before the space becomes ready.

Persisted  A value of true is displayed if the space needs to be persisted or a value of false otherwise.

Update transport  The transport protocol used to distribute notifications of updates to the data stored in the space. Can be either the reliable listen protocol or the reliable discovery protocol as they are defined by the connection attributes to the metaspace.

Entry TTL  The TTL (time-to-live) of the entries stored in the spaces in milliseconds. Default is -1 (forever).

Lock wait  The Lock wait defined for the space is displayed. Default is -1 (forever).

Lock TTL  The Lock TTL (time-to-live) defined for the space is displayed. Default is -1 (forever).

Fields
For each field, the name and type of the field, and whether or not the field is nullable (optional), is displayed.

Below this list of field definitions are names of all the fields that make up the space’s key.

Statistics
Count of put, get, take, seeded, and replicated entries is displayed. These counts are displayed per member and the cumulative count for all members connected to this space.
# show | describe spaces

**Command**

<table>
<thead>
<tr>
<th>Syntax</th>
<th>show</th>
<th>describe spaces</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose</strong></td>
<td>Lists all spaces defined for the metaspace.</td>
<td></td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td>None.</td>
<td></td>
</tr>
</tbody>
</table>
quit | exit | bye

Command

Syntax    quit | exit | bye

Purpose   Exits the Admin CLI.

Parameters None.
In addition to `as-admin.cmd`, the Admin CLI tool, ActiveSpaces provides a pre-built Agent process called `as-agent`. Users can run `as-agent` on any host. The sole purpose of `as-agent` is to join all distributed spaces in the specified metaspace as a seeder. Agents can also be used to ensure that the desired degree of replication specified for a space can be achieved.

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. Any process using the ActiveSpaces API can become a seeder on a space, however there are situations in which applications may not always be running and joined to the space, or not in sufficient numbers. Also, the fact that the data contained in the space disappears along with the last seeder in the space may become a problem. The `as-agent` process can be used to ensure that there are always one or more seeders for each of the spaces in the metaspace.

Agents remain connected to the metaspace, and always seed the system spaces, keeping the metaspace’s space definitions alive, even when all other applications have quit the metaspace. Start up the `as-agent` process by invoking `as-agent.exe` in the ActiveSpaces bin directory. By default, this is located at `C:\tibco\as\1.1.1\bin` (Windows) or `/usr/tibco/as/1.1.1/bin` (UNIX). The `as-agent` process can be launched with a number of optional parameters. All parameters are optional, but parameters that are specified must be given as pairs. The syntax for launching `as-agent` with parameters is shown here:

```
as-agent.exe -metaspace <metaspace_name> -discovery <discovery_url> -listen <listen_url> -log <log_file> -debug <log_level>
```

Usage help for `as-agent` is displayed when the following command is invoked from the bin directory: `C:\tibco\as\1.1.1\bin> as-agent -help`

The output of this help request is shown here. It includes the default values used if no parameter is provided by the user:

```
Usage
-metaspace <metaspace_name>  default ms
-discovery <url> / <url_list> default tibpgm://
-listen <url>  default tcp://
-log <log_file>  
-debug <log_level>  default 3 (INFO)
```

Discovery url format:
- `tcp://ip1:port,ip2:port2,ip3:port3`
- `tibpgm://dport/interface;multicast/key1=value1;key2=value2;key3=value3`
- `tibrv://service/network/daemon`

Listen url format:
- `tcp://interface:port`
Explanations of as-agent Parameters

There is also a Java version of as-agent available in the lib directory that can be launched using java -jar as-agent, and which takes the same argument as the C version of the agent described above. The Java version of the as-agent behaves exactly as the C version, but will be able to service requests to remotely invoke methods as triggered by other applications using the space (obviously as long as it has the classes being invoked present in it’s CLASSPATH).

For more details about discovery and listen URLs, see:

- Connecting to the Metaspace, page 51
- Discovery Attribute, page 52
- Listen Attribute, page 54

log file example

If the parameter -log as is used, then the output will be as-<processid>.log.

-debug <log_level>

The log levels are as follows:

1 = ERROR_LEVEL
2 = WARNING_LEVEL
3 = INFO_LEVEL

The default is 3 (INFO). The log information displayed on the console is minimal and can not be controlled through this parameter. This parameter is only for log files. If a log file is not specified, then the debug (log level) value is ignored.
Chapter 6  Examples

The example programs are provided with the product to use when learning to program with ActiveSpaces.

Topics

- Overview, page 110
- Building the Examples, page 111
- Running the Examples, page 112
Overview

TIBCO ActiveSpaces is a middleware used by application programmers. In addition to this User’s Guide, the example programs provided with the product are one of the most important resource to use when learning to program with ActiveSpaces. It is highly recommended that any person new to ActiveSpaces reserves some time to study these example programs.

Some of those examples (such as ASOperations) are created in order to provide boilerplate code templates that can be copied and pasted directly into the applications the user is writing; others are just examples of types of applications that can easily be created using ActiveSpaces.
As long as the environment variables (especially the AS_HOME environment variable) are set up correctly, it should be straightforward for a developer to build the example applications.

In C, a makefile is provided for convenience and typing make is all that is required to build them. For the Java examples, build.xml is also provided to facilitate examples building: if you have the ant tool installed on your machine, typing ant in the appropriate directory is all that is needed to do to build them. Note that the Java examples also contain a separate ASPaint example application that does not exist in C.
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 in C, as almost all of them are self-contained and can be run just by launching the executable file. In Java, the ant build script will create an Examples.jar file containing all of the example classes and they can be launched using

```
java -cp Examples.jar <ClassName>
```

The ASOperation is the default class for the jar file and can be launched directly using

```
java -jar Examples.jar.
```

Since many examples are distributed themselves, remember to try and run multiple instances of the example applications at the same time to experience how easy it is to create coherent and consistent distributed system using ActiveSpaces.

Because ActiveSpaces is platform independent, the C and Java versions of the examples are completely interoperable: you can, for example, run one instance of the example application in C and one in Java and they will interact just as well as if all instances where C (or Java).

ASOperations and ASBatchOperations

The first example program a user should examine is the ASOperations example. It defines and joins (as a seeder) a space called myspace, with a very basic definition consisting of three fields:

- A mandatory key field called key of type integer
- Two optional fields called value (of type String) and time (of type DateTime)

Once connected to the metaspace and after defining and joining the space, the application prompts the user to one of the following actions:

- g does a get on the space
- p does a put on the space
- t does a take on the space
- l does a lock on the space
- b begins a transaction
- c commits the transaction
- r rollbacks the transaction
• s does a size (such as counts the number of entries matching the optional filter in the space)

When asked for a key, just enter an integer number; when asked for a value, just enter any string or hit Return to signify that you do not want to set a value since the value field is nullable.

It is more interesting to launch more than one instance of ASOperations in order to see two processes interacting over the same space.

ASOperations is important because it shows the 'best practice' way to code those simple basic space interactions, programmers are encouraged to copy and paste parts of the ASOperation example into their program and to use those code sniplets as the templates of how to use the features offered by ActiveSpaces in their own programs.

As it's name indicates, ASBatchOperations does the same thing as ASOperation only using the batch versions of the space calls.

ASPersistence

This example is an implementation of the persistence interface, effectively a reduced version of ASOperation that creates a persisted space and register its own implementation of the ActiveSpaces persistence Interface. The Java example uses JDBC to persist to any database offering a JDBC interface, while the C version simply prints to the console.

ASBrowser, ASEventBrowser and ASListener

These two examples show how simple it is to use the space browser and listener features to iterate over the content 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. Examine both to evaluate when you to use a browser and when to use a listener in your own code.

These examples are also very handy 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 play with the timescope attribute as well.
ASChat

This example shows how simple it is to build true peer-to-peer distributed applications using ActiveSpaces: in this case, a multi-user chat-room.

ASRequestReplyServer and ASRequestReplyClient

These examples show how to 'use a space as a queue' in order to distribute request processing. Start at least one instance of ASRequestReplyServer first, then start and stop as many instances of the server and the client application as you want. Notice how requests sent by the clients are distributed amongst the servers.

Remote Space Invocation

The Java example directory also contains examples showing how to implement and use remote space invocation.

ASPaint

ASPaint is a Java only implementation of a shared whiteboard. It was written primarily to showcase the various features of AS, rather than to be used as a template for coding best practices. You can build it by using ant in the ASPaint directory, and you can launch it by using

```
java -jar ASPaint
```

Obviously, it is only interesting to play with when you launch more than one instance of ASPaint simultaneously 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 very visual manner. Try adding or removing seeders from the space and see re-distribution at work. You can also try filters and even remote space invocation.
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 tuplespaces.

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.

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.

class
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.

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 replication
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 replication 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 replication, synchronous and asynchronous. Synchronous replication will have an impact on performance, since it involves putting (or taking) data and replicating it in a single transaction. With asynchronous replication, 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 replication is synchronous or asynchronous, is specified when the space is created. The administrator can also specify the degree of replication, that is, how many replicas of the data will be created.

With synchronous replication, the administrator or application has immediate verification of whether or not the replication was successful, because if it was not, then the put or take command that triggered the attempted replication will itself fail, returning an error message. In asynchronous mode, the command will succeed, regardless of successful replication. An application or administrator can listen to advisory spaces to determine whether there was a problem with replication for an instance of asynchronous replication.

If a space is being used as a cache-aside, the space will normally be created without replication, 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.

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.

**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.

**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.

**key**
A unique value based on the value of one or more fields.

**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.

**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.

**peer**
A process that has connected to a metaspace and joined a space as either a *seeder* or a *leech*.

**R**

**relaxed coherency**
If there are multiple copies of a tuple (due to replication 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*.)

**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.

**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.

**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 seedership 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 replication 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**.)

**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**

**tupel**
A typed data object that is stored in a space. Similar to a row in a database table.

**tupel 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

A
ActiveSpaces distributed application system 115
ActiveSpaces features 3
ActiveSpaces program 115
Admin CLI 88
agent 115
agents 42
AS_HOME environment variable 15
ASBrowser, ASEventBrowser and ASListener 113
ASChat 114
ASOperations and ASBatchOperations 112
ASPaint 114
ASPersistence 113
ASRequestReplyServer and
   ASRequestReplyClient 114
associative array 115

B
batch operations 74
batch versus blocking operations 29
benefits of programming with ActiveSpaces 3
building the examples 111

C
cache 115
CLASSPATH environment variable 15
clear command 91
cluster 115
doerentency 115
concurrently updating data in a Space 31
connect command 92
connecting to the Metaspace 51
connecting with a Metaspace
   special considerations 42
consuming or removing data from a Space 30
customer support vii

d
data grid 116
data partitioning 116
data replication 116
define | create space command 94
defining a Space 56
deployment 41
disconnect command 97
disconnecting from the Metaspace 54
distributed cache 115
distributed Data Cache 5
distributed in-memory tuple space 116
distributed Space 58
drop space command 98

E
Entries 28
tentry 116
tentry browsers 80
tentry browsers and the event browser 33
tevent 117
tevent browsers 82
tevent listener 117
tevent notification 117
Example using as-admin.cmd -i 99
Execute method 90
explanations of as-agent parameters 108
export metaspace command 99

F
field 117
field definitions 64
filters 36
formats for filter values 38

G
getting or taking a single Tuple from the Space 71
getting the list of user defined space names 55
getting the name and definition of a Space 71
grid computing 6

H
hash map 117
help command 100
how is a Space implemented? 25
how to use a Space 29

I
in-memory operational Data Store 5
installing ActiveSpaces 10
installing in GUI mode 10

J
Java JDK or JRE 8
JAVA_HOME environment variable 15
joining a Space 66
JRE or JDK bin and lib directories 15

K
key 117
Key fields 65

L
launching the Admin CLI 88
leaving a Space 66
leech 117
listener 117
Listeners 77
lock 117
locking and working with locked entries 75
locking data in a Space 31

M
managing Spaces 56
member 117
Metaspace 51
metaspace 117
Metaspace membership 55
Metaspace name 52

N
networking considerations 41
new environment variables required 15
node 118
non-distributed Space 59

O
operators supported in filters 37
optional supporting software 8
P

peer 118
persistence 84
persisting on a Space 67
PGM (Pragmatic General Multicast) URL Format 53
product overview 2

Q

quit | exit | bye command 106

R

relaxed coherency 118
remote Space invocation 114
result and status codes 46
Results 28
retrieving data from a Space 30
running the examples 112

S

scalability 118
scopes of a Space browser 34
seeder 118
setting environment variables 14
show | describe member command 101
show | describe members command 102
show | describe space command 103
show | describe spaces command 105
Space attributes 58
space browser 118
Space definition through the Admin CLI 56
Space definition through the API 56
Space life cycle 67
space listener 119
Space membership 66
space-based architecture 5
SpaceEntry 76
SpaceEvent 79
storing a Tuple in a Space 72
storing data into a Space 29
strict coherency 119
support, contacting vii
supported platforms 8
system spaces 119

T

technical support vii
testing your installation 16
TIBCO ActiveSpaces bin and lib directories 14
TIBCO Rendezvous 9
TIBCO Rendezvous bin and lib directories 14
TIBCO Rendezvous Discovery URL Format 52
TIBCO_HOME v
transactions 40, 83
tuple 119
tuple fields 69
tuple space 119
Tuples 27, 69

U

uninstalling ActiveSpaces 18
updating a Tuple in a Space 72
usage profiles 5
user spaces 119
using as-agent 107
using Space browsers 32

W

what are Tuples, Entries, and Results? 27
what is a Metaspace? 22
what is a Space? 23
what is ActiveSpaces? 2