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, Two-Second Advantage, TIBCO ActiveMatrix BusinessWorks, 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 © 2010, 2016 TIBCO Software Inc. ALL RIGHTS RESERVED, TIBCO Software Inc. Confidential Information
Contents

About this book ................................................................................................................. ix
   Related documentation ................................................................................................. ix
   Conventions .................................................................................................................... ix
   Community ..................................................................................................................... x

1. Introduction ................................................................................................................... 1
   What is Distributed Transactional Memory? ................................................................. 1
   Managed objects ............................................................................................................ 1
   Transactions .................................................................................................................. 2
   Durable object store .................................................................................................... 2
   Keys and queries .......................................................................................................... 2
   Asynchronous methods ............................................................................................... 2
   High availability ........................................................................................................... 2
   Distributed Computing ............................................................................................... 3
   Online upgrades .......................................................................................................... 4
   Applications .................................................................................................................. 4

2. Applications .................................................................................................................. 5
   Business solution .......................................................................................................... 5
   Design-time .................................................................................................................... 6
   Deploy-time ................................................................................................................... 8
   Platform services ......................................................................................................... 10

3. Administration .............................................................................................................. 15
   Conceptual model ........................................................................................................ 15
   Service names .............................................................................................................. 17
   Discovery service ........................................................................................................ 18
   Clusters ....................................................................................................................... 22
   Nodes ........................................................................................................................... 24
   Engines ....................................................................................................................... 27
   Management tools ...................................................................................................... 29

4. Managed objects .......................................................................................................... 31
   Life cycle ..................................................................................................................... 31
   Extents ......................................................................................................................... 31
   Triggers ....................................................................................................................... 31
   Keys and Queries ........................................................................................................ 31
   Asynchronous methods ............................................................................................... 33
   Named Caches ............................................................................................................ 33

5. Transactions .................................................................................................................. 37
   Local and distributed transactions .............................................................................. 37
   Isolation ....................................................................................................................... 43
   Locking ....................................................................................................................... 43
   Deadlock detection ..................................................................................................... 45
   Transaction logging .................................................................................................. 48

6. Distributed computing ................................................................................................. 49
   Connectivity ............................................................................................................... 50
   Location transparency ............................................................................................... 50
   Locations .................................................................................................................... 51
   Location discovery ..................................................................................................... 52
   Life-cycle .................................................................................................................... 52
   Remote node states ................................................................................................... 53
   Deferred Write Protocol ............................................................................................. 54
   Detecting failed nodes ............................................................................................... 55
   Network error handling ............................................................................................. 56
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.14</td>
<td>Partition failover handling</td>
<td>87</td>
</tr>
<tr>
<td>7.15</td>
<td>Multi-master scenario</td>
<td>88</td>
</tr>
<tr>
<td>7.16</td>
<td>Quorum state machine - minimum number of active remote nodes</td>
<td>90</td>
</tr>
<tr>
<td>7.17</td>
<td>Quorum state machine - voting</td>
<td>91</td>
</tr>
<tr>
<td>7.18</td>
<td>Active cluster</td>
<td>93</td>
</tr>
<tr>
<td>7.19</td>
<td>Failed cluster</td>
<td>93</td>
</tr>
<tr>
<td>7.20</td>
<td>Merge operation - using broadcast partition discovery</td>
<td>94</td>
</tr>
<tr>
<td>7.21</td>
<td>Split cluster</td>
<td>95</td>
</tr>
<tr>
<td>7.22</td>
<td>Geographic redundancy</td>
<td>97</td>
</tr>
<tr>
<td>8.1</td>
<td>Type exchange</td>
<td>100</td>
</tr>
<tr>
<td>9.1</td>
<td>Configuration model</td>
<td>103</td>
</tr>
<tr>
<td>9.2</td>
<td>Configuration life cycle</td>
<td>105</td>
</tr>
<tr>
<td>10.1</td>
<td>Activating Components</td>
<td>110</td>
</tr>
<tr>
<td>10.2</td>
<td>Deactivating Components</td>
<td>111</td>
</tr>
<tr>
<td>10.3</td>
<td>Component Activation Failure</td>
<td>112</td>
</tr>
<tr>
<td>10.4</td>
<td>Component Activation</td>
<td>113</td>
</tr>
<tr>
<td>10.5</td>
<td>Component Deactivation</td>
<td>114</td>
</tr>
</tbody>
</table>
List of Tables

3.1. node service type properties ................................................................. 21
3.2. cluster service type properties ............................................................ 22
3.3. http service type properties ................................................................. 22
3.4. distribution service type properties ..................................................... 22
3.5. application service type properties ..................................................... 22
6.1. Remote node states .............................................................................. 53
7.1. Partition states ..................................................................................... 80
7.2. Partition status .................................................................................... 82
7.3. Node quorum states ............................................................................ 89
9.1. State transition audits .......................................................................... 106
9.2. State transition methods ..................................................................... 107
About this book

This guide describes the architecture of Distributed Transactional Memory. It provides a technical overview of all Distributed Transactional Memory functionality.

It is intended for the following types of readers:

• Anyone looking for a technical overview of Distributed Transactional Memory features.

• Java developers who want to get started developing Java applications using Distributed Transactional Memory.

• System administrators and operators who want to understand the Distributed Transactional Memory application and management architecture.

Related documentation

This book is part of a set of Distributed Transactional Memory documentation, which also includes:

Distributed Transactional Memory Installation — This guide describes how to install the Distributed Transactional Memory software.

Distributed Transactional Memory Quick Start — This guide describes how to quickly get started using Java IDEs to develop Distributed Transactional Memory applications.

Distributed Transactional Memory Java Developer's Guide — This guide describes how to program Distributed Transactional Memory.

Distributed Transactional Memory Administration — This guide describes how to install, configure, and monitor an Distributed Transactional Memory deployment.

Distributed Transactional Memory Performance Tuning Guide — This guide describes the tools and techniques to tune Distributed Transactional Memory applications.

Distributed Transactional Memory System Sizing Guide — This guide describes how to size system resources for Distributed Transactional Memory applications.

Distributed Transactional Memory Javadoc — The reference documentation for all Distributed Transactional Memory APIs.

Conventions

The following conventions are used in this book:

Bold — Used to refer to particular items on a user interface such as the Event Monitor button.

Constant Width — Used for anything that you would type literally such as keywords, data types, parameter names, etc.

Constant Width Italic — Used as a place holder for values that you should replace with an actual value.

Example node names are single character capital letters starting at A.
Example cluster names are single character capital letters starting at X.

**Community**

The Distributed Transactional Memory online community is located at https://devzone.tibco.com. The online community provides direct access to other Distributed Transactional Memory users and the Distributed Transactional Memory development team. Please join us online for current discussions on Distributed Transactional Memory and the latest information on bug fixes and new releases.
Introduction

What is Distributed Transactional Memory?

Distributed Transactional Memory is an in-memory transactional application platform that provides scalable high-performance transaction processing with durable object management and replication. Distributed Transactional Memory allows organizations to develop highly available, distributed, transactional applications using the standard Java POJO programming model.

Distributed Transactional Memory provides these capabilities:

- **Transactions** - high performance, distributed "All-or-None" ACID work.
- **In-Memory Durable Object Store** - ultra low-latency transactional persistence.
- **Transactional High Availability** - transparent memory-to-memory replication with instant fail-over and fail-back.
- **Disaster Recovery** - configurable cross-data center redundancy.
- **Distributed Computing** - location transparent objects and method invocation allowing transparent horizontal scaling.
- **Online Application and Product Upgrades** - no service outage upgrades for both applications and product versions.
- **Application Architecture** - flexible application architecture that provides a single self-contained archive to simplify deployment.

Managed objects

Distributed Transactional Memory features are available using Managed Objects which provide:

- **Transactions**
Transactions

All Distributed Transactional Memory Managed Objects are transactional. Distributed Transactional Memory transactions support transactional locking, deadlock detection, and isolation. Distributed Transactional Memory supports single writer, multi-reader locking, with transparent lock promotion. Deadlock detection and retry is transparently handled by the Distributed Transactional Memory JVM. Transactional isolation ensures that object state modifications are not visible outside of a transaction until the transaction commits.

Distributed Transactional Memory transactions can optionally span multiple JVMs on the same or different machines. Distributed locking and deadlock detection is provided.

All transactional features are native in the Distributed Transactional Memory JVM and do not require any external transaction manager or database.

Durable object store

Managed Objects are always persistent in shared memory. This allows the object to live beyond the lifetime of the JVM. Shared memory Managed Objects also support extents and triggers. There is optional support for transparently integrating managed objects to a secondary store, such as an RBDMS, data grid, archival store, etc.

Keys and queries

Managed Objects can optionally have one or more keys defined. An index is maintained in shared memory for each key defined on a Managed Object. This allows high-performance queries to be performed against Managed Objects using a shared memory index. Queries can be scoped to the local node, a sub-set of the nodes in the cluster, or all nodes in the cluster.

Asynchronous methods

Asynchronous methods allow applications to queue a method for execution in a separate transaction. Transactional guarantees ensure that the method is executed once and only once in a separate transaction.

High availability

Distributed Transactional Memory provides these high availability services:

- Transactional replication across one or more nodes
Complete application transparency

Dynamic partition definition

Dynamic cluster membership

Dynamic object to partition mapping

Geographic redundancy

Multi-master detection with avoidance and reconciliation

A partitioned Managed Object has a single active node and zero or more replica nodes. All object state modifications are transactionally completed on the current active node and all replica nodes. Replica nodes take over processing for an object in priority order when the currently active node becomes unavailable. Support is provided for restoring an object's state from a replica node during application execution without any service interruption.

Applications can read and modify a partitioned object on any node. Distributed Transactional Memory transparently ensures that the updates occur on the current active node for the object. This is transparent to the application.

Partitioned Managed Objects are contained in a Partition. Multiple Partitions can exist on a single node. Partitions are associated with a priority list of nodes - the highest priority available node is the current active node for a partition. Partitions can be migrated to different nodes during application execution without any service interruption. Partitions can be dynamically created by applications or the operator.

Nodes can dynamically join and leave clusters. Active nodes, partition states, and object data is updated as required to reflect the current nodes in the cluster.

A Managed Object is partitioned by associating the object type with a Partition Mapper. The Partition Mapper dynamically assigns Managed Objects to a Partition at runtime. The Managed Object to Partition mapping can be dynamically changed to re-distribute application load across different nodes without any service interruption.

Nodes associated with a Partition can span geographies, providing support for transactionally consistent geographic redundancy across data centers. Transactional integrity is maintained across the geographies and failover and restore can occur across data centers.

Configurable multi-master, aka split-brain, detection is supported which allows a cluster to be either taken offline when a required node quorum is not available, or to continue processing in a non-quorum condition. Operator control is provided to merge object data on nodes that were running in a multi-master condition. Conflicts detected during the merge are reported to the application for conflict resolution.

A highly available timer service is provided to support transparent application timer notifications across failover and restore.

All high availability services are available without any external software or hardware.

Distributed Computing

A Managed Object can be distributed. A distributed Managed Object supports transparent remote method invocation and field access. A distributed Managed Object has a single master node on
which all behavior is executed at any given time. A highly available Managed Object's master node is the current active node for the partition in which it is contained. Distribution is transparent to applications.

**Online upgrades**

Class definitions can be changed on individual nodes without requiring a cluster service outage. These class changes can include both behavior changes and object shape changes (adding, removing, changing fields). Existing objects are dynamically upgraded as nodes communicate to other nodes in the cluster. There is no impact on nodes that are running the previous version of the classes. Class changes can also be backed out without requiring a cluster service outage.

Product versions can also be updated on individual nodes in a cluster without impacting other nodes in the cluster. This allows different product versions to be running in the cluster at the same time to support rolling upgrades across a cluster without requiring a service outage.

**Applications**

An application consists of one or more *application fragments*, or fragment, configuration files, and dependencies, packaged into an *application archive*. An application archive is deployed on a node, along with an optional *node deploy configuration* that can provide override configuration in the application archive and also provide deployment time specific configuration. Different application fragment types are supported, so an application can consist of homogeneous or heterogenous fragments.
This chapter provides an overview of Distributed Transactional Memory applications. Although some details will vary from one application to another, the conceptual framework presented in this chapter (and this book) is common to all, and forms the basis for understanding later chapters. The concepts explained in this chapter include:

- application context within a business solution.
- developing an application.
- deploying an application.
- a brief overview of the platform services that impact application design.

Additional details on designing and deploying applications can be found in the Java Developer's Guide and Administration Guide respectively.

**Business solution**

An application forms part of a business solution as shown in Figure 2.1. A business solution may consist of one more applications deployed on a number of nodes grouped into a cluster. Each node is a container for one or more engines on which an application executes. Applications are described in more detail in the section called “Design-time” on page 6, and clusters, nodes, and engines are described in more detail in the section called “Deploy-time” on page 8 and in Chapter 3.
Figure 2.1. Business Solution

Some important properties of applications:

- distributed: applications are transparently distributed across various machines.
- highly available: if one machine fails, processing of in-flight and new transactions can continue uninterrupted on another machine.
- elastic: machines can be added or removed as needed based on the current load.
- extensible: applications can be upgraded with changed, or entirely new, behavior without a service outage.
- configurable: applications are highly configurable with configuration changes occurring without interrupting in-progress work.
- geographically redundant: applications can be deployed across wide-area networks to support disaster recovery.
- discoverable: applications are discoverable using a discovery service to eliminate any dependency on actual machine network addresses.

Design-time

Applications are generated as part of a software development process. This process is collectively called design-time and consists of standard software development tasks. The output of the design-time tasks is an application archive which contains everything required to deploy an application.
Figure 2.2. Application archive

An application archive contains all of the details known at design-time by a developer or an architect. It does not contain, other than possible default values, any deployment time details, for example network topology information. The contents of an application archive are:

- an application definition.
- one or more fragments.
- optional application dependencies, e.g. 3rd-party JAR files or native libraries.
- optional default application configuration.
- optional upgrade and restore plans.

A fragment is an executable part of an application. A fragment consists of:

- executable software.
- optional fragment dependencies, e.g. 3rd-party JAR files or native libraries.
- optional default fragment configuration.

Each fragment has a fragment type, which identifies the type of engine that will run the fragment at deployment time, for example Java.
Deploy-time

An application is deployed onto a cluster of nodes providing high-availability, elastic scaling and distributed administration. The application nodes are interconnected using distribution, and connected to external systems using adapters. This is shown in Figure 2.3.

Figure 2.3. Deployed application

Applications are deployed to nodes in a cluster using the application archive generated at design-time and an optional node deploy configuration. A node deploy configuration is used to:

- specify deployment time configuration values, for example port numbers, host names, memory sizes, etc.
- override default configuration in an application archive.
- add additional deployment time configuration, for example administration users.
Figure 2.4. Deploying an application

When an application is deployed, a *cluster name* and a *node name* are specified as part of deploying the application to a node. All nodes on which an application is deployed belong to the same cluster. See the section called “Service names” on page 17 for more details on cluster and node names.

A node is a container that runs one or more *engines* to execute the fragments in an application. An engine is an execution context that is specific to the fragment type. For example, a Java fragment would execute on an engine with its own JVM. Each fragment in an application is executed on one
or more engines. A complete description of node and engine lifecycle can be found in the section called “Nodes” on page 24.

**Platform services**

Applications use the services of the Distributed Transactional Memory platform. These services include:

- transaction management
- transparent distribution of data and processing
- a security model
- high-availability and distribution
- extensible configuration service
- extensible management services
- adapter framework to provide connectivity to external systems

While the Distributed Transactional Memory platform provides a set of powerful capabilities, it is the combination of the application-specific logic and configuration that define the actual behavior of the business solution, as shown in Figure 2.5.

![Figure 2.5. Platform services](image)

Notice that Figure 2.5 shows management not only across the Distributed Transactional Memory platform, but also across the application-specific part of the solution as well. This is because the management service is extensible to the entire application, so you can manage the entire application in a consistent way.
The following sections provide a high-level overview of the platform services that most impact application design.

**Configuration**

The behavior of a business solution can be changed by activating different configurations. Many aspects of a solution are configurable, from minor changes to wholesale redefinition of business logic. Distribution, security policy, adapter connectivity options, and many other features of a solution are configured using the Distributed Transactional Memory *configuration service*. The configuration service provides:

- support for application specific configuration.
- a standard syntax to express configuration data.
- a unified identification for configuration using a configuration type, name, and version.
- a unified set of administrative commands to manage configuration.
- notification when configuration states change.

The configuration service loads configuration data from files. The syntax of the configuration files is *Human-Optimized Configuration Object Notation* (HOCON). HOCON is a superset of *JavaScript Object Notation* (JSON), enhanced for readability. Configuration files contain the configuration type, name, and version identifiers, along with the configuration data. Configuration files can be loaded from any machine that has connectivity to the machine running the target node. Configuration can also be loaded on multiple nodes at the same time.

There are many different configurable elements in an application; each one can have several different versions loaded but only one active version. Figure 2.6 shows an example of how you might load a new version 2.0.0 of an adapter configuration `x.y.z`, while leaving the current version 1.1.3 as the active version.
Figure 2.6. Loading a new configuration

There can be any number of different versions loaded for a particular configuration type; any of these can be activated by a management command. Figure 2.7 shows an example of the activation of version 2.0.0 of an adapter configuration type \texttt{x.y.z}. The previously active version, \texttt{1.1.3}, is now inactive.

Figure 2.7. Changing the active configuration

**Connectivity**

Applications communicate with external systems and other nodes in a cluster using:

- adapters, which provide connectivity between a node and external systems.
- a distributed communication model to transparently communicate between nodes.
Each of these is configurable and controllable using the standard administration tools.

**Endpoints, sessions, and services** A node communicates with external systems using an adapter built using an adapter framework. This framework defines the following concepts:

- **endpoint**: this is an internal representation of a remote system. An endpoint also manages the creation and allocation of a number of sessions
- **session**: a session is a connection with an external system
- **service**: a set of endpoints can be grouped into a service for administrative or operational purposes, for example, so that they can be enabled and disabled as a unit

Figure 2.8 depicts the relationship between service, endpoint, and session.

**Figure 2.8. Connectivity architecture**

Within a node, an endpoint represents a logical destination that elements can communicate with, ignoring the intricate details of sessions and external system details. Importantly, all data format and protocol conversion between the application and the external system is done in adapters. An endpoint can manage either incoming (server) or outgoing (client) sessions.

**High availability**

High availability provides an easy way to ensure application availability using replicated data. Two or more nodes (generally on different machines to reduce risk) are linked together to provide redundancy for each other. Application data are replicated to one or more backup nodes depending on the availability requirements. Nodes can be configured such that all work occurs on one node with another one acting as the backup. This is called active/passive high availability. Nodes can also be configured to be simultaneously processing work. This is called active/active high availability.

Figure 2.9 shows two nodes configured in a primary and backup topology with a client directing traffic for processing to the primary node (Node A). As requests are processed on the primary node, modifications, creations, updates, and deletions of application data are transactionally replicated onto the backup node (Node B).
Figure 2.9. High availability with primary and backup nodes active

If the primary node goes out of service for some reason, the configured backup node takes over processing for the out-of-service node. Figure 2.10 shows this taking place.

Figure 2.10. High availability with processing on backup node

When node A is later brought back up and restored, it will again be the primary node and the processing will move back to it from node B.

It is possible to have all application processing active on one node; an alternative approach provides load balancing by allocating the application work across several nodes. It is also possible, using administration tools, to migrate processing from one node to another. This can be used to scale up a system by adding nodes, or to move processing off of a machine for maintenance reasons.
3

Administration

This chapter provides a high-level overview of the administration architecture and capabilities available to manage Distributed Transactional Memory applications.

Chapter 2 described the general structure of applications. In that chapter applications and fragments were introduced and how they are deployed on nodes in a cluster. This chapter provides more details on the deployment and administration models.

Conceptual model

The following concepts are used to describe the deployment architecture:

- **Machine** - an execution context for a node.
- **Application** - business specific functionality.
- **Fragment** - an executable part of an application.
- **Cluster** - a logical grouping of nodes that communicate to support an application.
- **Node** - a container for engines.
- **Engine** - executable context for a fragment.
Figure 3.1. Conceptual model

An application is executed on one or more nodes.
An application contains one or more fragments.
A fragment is executed on one or more engines.
One or more nodes can run on a single machine.
A node can belong to one cluster.
A cluster can host a single application.
A node can host one or more engines.

Physical model

The concepts described in the section called “Conceptual model” on page 15 are mapped to physical entities as follows:

- **Machine** - a physical or virtual computer.
- **Application** - an application archive containing one or more fragments generated at design-time as described in the section called “Design-time” on page 6.
- **Fragment** - a fragment archive containing executable application code generated at design-time as described in the section called “Design-time” on page 6.
- **Cluster** - a collection of nodes interconnected by a network.
- **Node** - a set of operating system processes running on a machine and monitored and managed by a node coordinator process.
• **Engine** - an operating system process executing a fragment that is managed by the node coordinator.

Figure 2.4 shows a diagram of all of the physical entities.

## Service names

All nodes are uniquely identified by a *service name*. A service name consists of these parts:

- a cluster name
- optional grouping
- a node name

Cluster, group, and node names are *labels*. The valid characters for a label are letters, numbers, and the "-" character. No other punctuation is permitted. Labels must use the UTF8 character set encoding.

A *fully qualified* service name consists of at least two *labels* separated by a ".". A fully-qualified service name includes the cluster name, the node name, and all grouping labels.

A *partially qualified* service name consists of at least one label, the cluster name, with a leading ".". A partially qualified service name does not include the node name, and optionally only a subset of any grouping labels. A single label without a leading "." is a special case and is treated as a cluster name.

Service names are arranged in a hierarchical tree structure, where the cluster name is the root of the tree and a node name is the leaf. The optional grouping part of a service name are the branches between a cluster name and a node name, or more formally:

```
servicename = [<nodename-label>.[[<group-label>.]])*<clusternam label>
```

Here are some example service names:

- a.applicationcluster
- b.eastcoast.applicationcluster
- c.eastcoast.applicationcluster
- d.westcoast.applicationcluster
- e.westcoast.applicationcluster

These service names uniquely identify five different nodes, all in the same cluster.

Service names are used by the discovery service (see the section called “Discovery service” on page 18) to locate network address information. Discovery can be performed using a fully-qualified or a partially-qualified service name. Fully-qualified service names always identify a single node. Partially-qualified service names may resolve to one or more nodes. For example using the service names above:

```
applicationcluster // resolves to all five nodes - notice no leading "."
.applicationcluster // resolves to all five nodes
.eastcoast.applicationcluster // resolves to nodes b and c.
.westcoast.applicationcluster // resolves to nodes d and e.
a.applicationcluster // resolves to node a.applicationcluster.
```

From the examples above, it can be seen that service name grouping allows sets of nodes to be resolved using a partially qualified service name. This is useful for administrating multiple nodes together. For example:
• nodes in different geographical locations might be configured with different connectivity.
• nodes grouped together to provide different high-availability guarantees.
• nodes that host one type of functionality may require different configuration from other nodes.
• nodes hosting different clients might require different operational rules.

**Discovery service**

The discovery service allows details about a node to be discovered using a service name, instead of specific network address.

When a node is installed it is available for discovery by the discovery service. A node registers multiple *service records*. Each service record has a different *service type*. Each service type makes available a different set of *service properties* that can be discovered. The different service types and properties are used by both nodes and client tools to dynamically discover information needed to perform specific tasks. For example, the administration client tool uses service discovery to dynamically find the administration port of a node.

It is recommended that service names be used instead of network addresses when accessing nodes.
Service discovery uses the UDP protocol to provide its services. To enable service discovery to work across machines, the UDP protocol must be enabled in all network routers and switches between nodes using service discovery.

Each node starts a UDP listener on all interfaces on the machine on which the node is running. All nodes in a cluster must use the same UDP listener port to successfully use service discovery.

When a service discovery client, e.g. the administration tool, performs a service lookup, it sends a broadcast using the default broadcast port number, or a user specified port number. If the service discovery client is running on a multi-homed machine, the broadcast is sent on the interface that matches the local host name. Support is provided for explicitly specifying the interface(s) on which the broadcast should be sent. The client must send the broadcast on the same port number on which the nodes are listening for service discovery to resolve the nodes.

When a node receives a service discovery broadcast request, if the fully, or partially qualified, service name matches the node's service name, it sends a response directly back to the address that sent the request. Clients performing a service lookup may receive multiple responses to a single request for partially qualified service names and when multiple nodes are configured to provide proxy discovery services for the same node (see the section called “Proxy discovery” on page 20).
When a discovery client performs a service discovery lookup with a fully qualified service name, the discovery client completes the lookup when the first response is received, or no response is received after a configurable amount of time. When a lookup is performed with a partially qualified service name, a discovery client always waits a configurable amount of time for responses to the lookup.

Figure 3.3. Service discovery network architecture

**Proxy discovery**

As described in the section called “Network architecture” on page 19, service discovery relies on the underlying network supporting UDP broadcast. There are cases where this is not the case; common examples are:

- Nodes communicating over a WAN to provide disaster recovery.
- Nodes deployed to cloud infrastructure.

To support service discovery in these environments *proxy discovery* can be used. Proxy discovery allows nodes that are not the requested service name to respond with the service properties for the requested service name. The network address information to return is configured in the node acting as the proxy. While multiple nodes can act as a proxy for the same service name, it is critical that all of the nodes have the same information configured to ensure that consistent results are returned for discovery requests.
Figure 3.4. Proxy discovery

Figure 3.4 shows node A providing proxy discovery services for node C because of the WAN connection between node A and C.

**Service types**

These service types are registered by nodes:

- **node** - network address information for administration port.
- **cluster** - cluster name of node.
- **http** - network address information for administration Web server.
- **distribution** - network address information for distribution.
- **application** - application running in cluster.

The service properties for each of these services types is summarized in the tables below.

**Table 3.1. node service type properties**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service Name</td>
<td>Fully qualified service name for node.</td>
<td>A.X</td>
</tr>
<tr>
<td>Network Address</td>
<td>Node administration network address.</td>
<td>dtm-adm://dtm.tibco.com:32299</td>
</tr>
<tr>
<td>applicationVersion</td>
<td>Version number of application running on node.</td>
<td>1.0</td>
</tr>
<tr>
<td>applicationName</td>
<td>Name of application running on node.</td>
<td>MyAppApplication</td>
</tr>
</tbody>
</table>
Clusters

Nodes are grouped into clusters. A cluster provides unified administration and distribution services for one or more nodes. Nodes in a cluster can be running on different machine architectures, and have different product or application versions.

Clusters are dynamically created and destroyed as nodes are installed and removed. No explicit action is required to create or destroy a cluster. The cluster to create is determined from the cluster name label in the fully-qualified service name specified when a node is installed. See the section called “Service names” on page 17 for details. A node can only be a single cluster - they cannot belong to multiple clusters. When all nodes in a cluster are removed, the cluster is destroyed.
An application is active as long as a cluster is active, i.e. it has at least one node installed and running. If all nodes in a cluster are stopped and removed, the application is stopped and removed.

**Managing nodes**

Each node has an *administration address*; this is a unique network address where a node listens for incoming administration requests. Administration commands are directed to a node using either the administration address, or indirectly using a discovery service lookup using a service name.

![Diagram of nodes and cluster](image)

It is recommended that service names be used for administration commands instead of a specific network address.

Administrative commands can be sent to multiple nodes in a cluster using a partially qualified service name. When using a partially qualified service name, the administrative command is sent to all resolved nodes. Figure 3.5 shows a load configuration command being sent to all nodes in cluster X with a single command. A partially qualified service name of X, which is the cluster name, is used so it resolves to all nodes in the cluster.

**Figure 3.5. Multi-node administration commands**

Nodes being addressed by either the administration network address or a service name can be in the same data center or in a different data center communicating over a WAN.
Nodes

The most fundamental control that you have over a node is to manage its life cycle. A node can be installed, started, stopped, and removed, as shown in Figure 3.6.

![Node life cycle diagram]

Figure 3.6. Node life cycle

The following sections describe what happens as a node transitions between these states.

Install node

To install a node, you use an administration client to perform an `install node` command. As shown in Figure 3.7, installing a node requires:

- the Distributed Transactional Memory product to be installed on the machine on which the node is being installed.
- the application archive for the application being installed.
- an optional node deploy configuration file.
After a node is successfully installed, the following has occurred:

- application container services have been started.
- the administration services have been started.
- the application archive has been installed into the node directory.
- the default application configuration and the node deploy configuration have been processed.
- all application fragment engines have been installed.
- the node is in the **Stopped** state.

The node can now be started.
**Start node**

Starting a node is done using an administrative client as shown in Figure 3.8.

![Diagram of starting a node](image1)

**Figure 3.8. Starting a node**

When a node is started these actions occur:

- configuration files are loaded and activated.
- all application fragment engines are started.
- the node joins the cluster.
- the node transitions to the **Started** state.

**Stop node**

Stopping a node is done using an administrative client as shown in Figure 3.9.

![Diagram of stopping a node](image2)

**Figure 3.9. Stopping a node**

When a node is stopped these actions occur:

- configuration files are deactivated and unloaded.
• all application fragment engines are stopped.
• the node leaves the cluster.
• the node transitions to the **Stopped** state.

**Remove node**

Removing a node is done using an administrative client as shown in Figure 3.10.

![Figure 3.10. Removing a node](image)

**Figure 3.10. Removing a node**

When a node is removed these actions occur:

• all application container services are stopped.
• administration services are stopped.
• all application fragment engines are removed.
• the node directory is removed.

**Engines**

One or more engines can be running on a node. Each engine hosts a single fragment and has a unique name. In the default case, there is an engine executing for each fragment in an application. However, this can be changed in the node deploy configuration so that multiple engines execute the same fragment.

When a node is installed, all engines are also installed. See the section called “Install node” on page 24 for details.

When a node is removed, all engines are also removed. See the section called “Remove node” on page 27 for details.

Figure 3.11 shows the engine life-cycle, which is affected by both engine and node commands.
Figure 3.11. Engine life-cycle

Start engine

In addition to engines being started when a node is started (see the section called “Start node” on page 26), engines can also be individually started using an administration command as shown in Figure 3.12.

```plaintext
start engine name=myengine
```

Figure 3.12. Starting an engine

Starting an individual engine has no impact on any other engines running on a node. A started engine can be stopped as described in the section called “Stop engine” on page 29 without having to stop the node.
Stop engine

In addition to engines being stopped when a node is stopped (see the section called “Stop node” on page 26), engines can also be individually stopped using an administration command as shown in Figure 3.13.

![Stop Engine Diagram]

**Figure 3.13. Stopping an engine**

Stopping an individual engine has no impact on any other engines running on a node. A stopped engine can be restarted as described in the section called “Start engine” on page 28.

Management tools

This section introduces the Distributed Transactional Memory administration tools. These tools are discussed in more detail in the Administration Guide.

Administrator is a web-based GUI that supports managing all nodes in a cluster. Administrator allows any Web Browser to be used to manage clusters, nodes, and applications.

epadmin provides a command line tool to support all administrative commands. epadmin provides a simple mechanism to script operational functions.

In general, the Web client is easier to use interactively and epadmin provides advantages for scripting and easy repeatability.

All administrative commands are also supported using JMX. Platform log messages are also exposed as JMX notifications. This allows any off-the-shelf JMX console to be used to manage nodes.
4

**Managed objects**

As described above Managed Objects are backed by shared memory. They can also be distributed and replicated.

**Life cycle**

Managed Objects are not garbage collected. They must be explicitly deleted by the application. Managed Objects exist following a normal JVM or machine shutdown. They also survive node and machine failures if they are replicated to another machine.

**Extents**

An extent is a collection of all Managed Objects that have been accessed on the local node. All Managed Objects have extents automatically maintained. Extents contain references to objects created on the local node and remote references for objects that were pushed (replicated) or pulled to the local node.

**Triggers**

Managed Objects optionally support triggers. A trigger provides a mechanism to be notified when a Managed Object is updated, deleted, or a conflict is detected while restoring a node following a multi-master scenario.

**Keys and Queries**

Managed Objects can optionally have one or more keys defined using annotations. When a key is defined on a Managed Object, an index is maintained in shared memory as Managed Objects are created and deleted. An index associated with a replicated or distributed Managed Object is maintained on all nodes to which the object is exists.
By default key values are immutable - they cannot be changed after an object is created. Mutable keys are also allowed if explicitly specified in the key annotation.

Explicit transaction locking can be specified when doing a query. These lock types can be specified:

- None - no transaction lock is taken on the objects returned by the query.
- Read - a transaction read lock is taken on all objects returned by the query.
- Write - a transaction write lock is taken on all objects returned by the query.

The lock type specified when performing a query only has impact on the query result. It does not affect the standard transaction locking as described in the section called “Locking” on page 43 when operating on the objects returned from the query.

A query can be scoped to the local node only, a user defined sub-set of the nodes in a cluster, or all nodes in a cluster. This allows object instances to be located on any node in a cluster. When a query executes on remote nodes it is called a **distributed query**.

When a query is executed on multiple remote nodes, the query executes in parallel and the result set is combined into a single result set returned to the caller. The returned result set is guaranteed to contain only a single instance of an object if an object exists on multiple nodes (replicated or distributed).

A query scoped to a local node only returns objects that are already on the local node. These objects may have been created on the local node, returned from a remote node using a distributed query, or replicated to the local node.

If an object is returned from a remote node that doesn't already exist on the local node it is implicitly created on the local node. This causes a write lock to be taken for this object. The lock type specified when performing the query is ignored in this case. The caching of objects returned from remote nodes is controlled using **Named Caches** as described in the section called “Named Caches” on page 33.

When a user-defined query scope is used, the nodes in the query scope can be audited when the query is executed. The possible audit modes are:

- Verify that the query scope contains at least one node. No other auditing is performed.
- Verify that the query scope contains at least one node and that distribution is enabled. Any inactive nodes are skipped when a query is performed.
- Verify that the query scope contains at least one node and that distribution is enabled. Any inactive nodes cause a query to fail with an exception.

Query support is provided for:

- Unique and non-unique queries
- Ordered and unordered queries
- Range queries
- Cardinality
- Atomic selection of an object that is created if it does not exist
Asynchronous methods

Methods on managed objects can be defined as asynchronous. Asynchronous methods are not queued for execution until the current transaction commits. When the current transaction commits, a new transaction is started and the method is executed in the new transaction. If a deadlock is detected while executing an asynchronous method, the transaction is aborted, a new transaction is started, and the method is re-executed.

The default transaction isolation of the transaction started to execute an asynchronous method is Serializable. The default isolation level can be changed to Read Committed - Snapshot using an annotation.

Asynchronous methods are queued to the target object and are executed one at a time, in the same order in which they were queued. Only one asynchronous method can be executed by a particular object at a time. The following ordering guarantees are made:

• An object executes asynchronous methods from a single sender object in the same order that they are sent.

• An object executes asynchronous methods from multiple senders in an indeterminate order. This order may or may not be the same order in which they were sent.

• An asynchronous method sent from an object to itself is processed before any other queued asynchronous methods to that object.

Asynchronous methods can be called on a distributed object. The method will be executed on the master node for the object. However, the method is always queued on the local node - it is not sent to the remote target node until after the current transaction commits.

If the target object of an asynchronous method is deleted before the method executes, the method execution is discarded.

When a JVM is shutdown, any queued asynchronous methods that have not executed are executed when the JVM is restarted.

Named Caches

Named caches provide a mechanism to control the amount of memory used to cache managed objects. Named caches can be dynamically defined, and managed objects added, at runtime without impacting a running application. Named caches support configurable cache policies and support for automatic, and explicit managed object flushing.

The default caching policies for managed objects when they are not associated with a named cache are:

• Local managed objects are always cached.

• Distributed objects (see Chapter 6) are never cached.

• Replica objects (see Chapter 7) are always cached, and cannot be flushed.

Named caches are defined using an API or administrative commands.
Cache policies

Named caches support these cache policies:

- **Always** - object data is always accessed from shared memory on the local node. These objects are never flushed from shared memory.

- **Never** - object data is never accessed from shared memory on the local node. These objects are always flushed from shared memory. This cache policy is defined by setting the cache size to zero.

- **Sized** - object data is always accessed from shared memory on the local node. These objects are automatically flushed from shared memory when they exceed a configurable maximum memory consumption size.

Cache policies are specified per named cache and they can be dynamically changed at runtime.

The implications of caching a distributed object are described in the section called “Reading and writing object fields” on page 50.

Cache association

Managed objects are associated with a named cache by class name at runtime. When a class is associated with a named cache all objects of that type are moved into the cache, along with any objects that extend the parent class, that are not already associated with a cache.

Named caches support inheritance. If a class is associated with a named cache all objects with that class as their parent are moved into the named cache. If another named cache is defined and a child class of the parent is associated with it, only the child objects (and any of its children) are moved into the named cache. All other objects are left in the parent's named cache.

Object flushing

All managed objects, except for replica objects, can be flushed from shared memory.

Cached objects are flushed from shared memory:

- explicitly using an API.

- automatically at the end of the current transaction (only distributed objects not in a named cache).

- using a background flusher when associated with a named cache.

Regardless of how an object is flushed, it has this behavior:

- flushing a local managed object, including partitioned objects on the active node, is equivalent to deleting the object, any installed delete triggers will be executed.

- flushing a distributed object removes the object data, including any key data, from local shared memory.

- flushing a replica object is a no-op. Replica objects cannot be flushed since that would break the redundancy guarantee made in the partition definition.
Figure 4.1 shows how a distributed object is refreshed after it was flushed from memory. O1` is a distributed reference to O1 that was stored in an object field on Node One. Accessing the field containing the O1` distributed reference on Node One will cause the object data to be refreshed from Node Two.

**Figure 4.1. Object refresh**

Distributed objects not in a named cache are automatically flushed from shared memory at the end of the transaction in which they were accessed. These objects are never in shared memory longer than a single transaction.

A background flusher evicts objects from shared memory in named caches. Objects are flushed from shared memory when the total bytes in shared memory exceeds the configured maximum size. Objects are flushed from shared memory using a Least Recently Used algorithm. The background flusher operates asynchronously, so the maximum memory utilization may be temporarily exceeded.

Objects are also automatically flushed from shared memory when memory throttling is in affect, for example when a distributed query fetches a large number of remote objects that cause local cache limits to be exceeded.

When calculating the size of shared memory required for a node, cached objects must be included in the sizing. See the Distributed Transactional Memory Sizing Guide.

**Flush notifier** Optionally a flush notifier can be installed by applications to control whether an object is flushed or not. When a flush notifier is installed it is called in the same transaction in which the flush occurs. The notifier is passed the object that is being flushed, and the notifier can either accept the flush, or reject it. If the notifier rejects the flush the object is not flushed from shared memory. The flush notifier is called no matter how an object flush was initiated.
This section describes Distributed Transactional Memory transactional functionality in more detail.

**Local and distributed transactions**

Transactions may be either local or distributed.

Local transactions are used on a single node even if they span multiple JVMs on the node.

Distributed transactions are used between Distributed Transactional Memory nodes. When a transaction spans Distributed Transactional Memory nodes a global transaction is started on the node that initiates the distributed work. The initiating node acts as the transaction coordinator. There is no independent transaction coordinator in Distributed Transactional Memory. All Distributed Transactional Memory nodes act as a transaction coordinator for distributed work that they initiate.
Figure 5.1. Distributed transaction

Nodes may be added to a distributed transaction not only by the node that initiated the distributed transaction, but by any node that participates in the distributed transaction.
Figure 5.2. Distributed transaction node participants

Figure 5.2 shows how nodes are added to a distributed transaction. In this diagram Node 1 starts a local transaction, LT1, and then initiates a global transaction, GT, to Node 2. Node 2 starts a local transaction, LT2, on behalf of the global transaction GT and then initiates work on Node 3 in
the same global transaction GT. Node 3 initiates another local transaction LT3, and then initiates work on Node 4, which starts another local transaction, LT4.

The response back from each of the nodes contains information on the local transaction that was started on the node, and any other nodes that started local transactions. This allows the initiating node, to determine which nodes need to be committed when the global transaction is committed. This is shown in the diagram in the commit processing on Node 1 - a commit is sent to all four nodes, even though Node 1 only initiated a global transaction to Node 2.

There is no programmatic difference between local and distributed transactions. Distributed Transactional Memory initiates the appropriate transaction type transparently depending on whether local or remote objects are in the transaction. There is a difference in how deadlocks are detected. See the section called “Deadlock detection” on page 45.

**Prepare**

Distributed transactions optionally have a prepare phase. A prepare phase provides a mechanism to integrate with external transactional resources. A failure in any of the prepare notifiers causes the transaction to rollback.

A prepare phase is used if there are any updates in a transaction, or transaction notifiers (see the section called “Transaction notifiers” on page 41) are installed for a transaction. The transaction notifiers can be installed on any node that is participating in a distributed transaction. If no updates are done in a transaction, or no transaction notifiers are installed, the prepare phase is skipped to optimize the performance of distributed transactions by eliminating the additional network I/O required with prepares.
Figure 5.3. Distributed transaction with prepare

See also the section called “Deferred Write Protocol” on page 54.

Transaction notifiers

Applications can optionally install transaction notifiers that are called during the prepare phase and when the transaction commits or rolls back. Transaction notifiers can be used to integrate with external systems - both transactional and non-transactional. Transaction notifiers are executed on the node on which they were installed. A distributed transaction may have transaction notifiers installed on multiple nodes by the application. In this case, the notifiers are executed on each node on which they were installed.
Figure 5.4. Distributed transaction notifiers
Isolation

Distributed Transactional Memory transactions support the following transaction isolation levels for objects:

- **Serializable** - modifications are only visible outside of the current transaction when it commits. Transaction read locks are taken for the duration of the transaction to ensure read consistency. All writes are blocked while a transaction read lock is held. This is the default transaction isolation level.

- **Read Committed - Snapshot** - modifications are only visible outside of the current transaction when it commits. Snapshots are taken from the last committed transaction (i.e. It is not a dirty read) to ensure read consistency during a transaction. No transaction read locks are taken during the transaction allowing object modifications to occur while reading an object. Read consistency is provided by the snapshot data across all fields in an object.

Both object isolation levels, serializable and read committed - snapshot, provide consistent, or repeatable reads during a transaction on the same node. This means that the same object field read multiple times in a transaction returns the same value. Read consistency is not guaranteed across nodes. See the section called “State conflicts” on page 44 for details on how data inconsistencies are handled.

Extents always use this transaction isolation level:

- **Read Committed** - extent iterations and cardinality will return inconsistent results in the same transaction if other transactions create or delete objects in an extent.

Locking

Transaction locks are used to maintain data consistency for the duration of a transaction. Transaction locks are only taken on objects. The transaction isolation level impacts the locking that occurs during a transaction. A serializable transaction isolation takes both transaction read and transaction write locks. A read committed - snapshot transaction isolation level only takes transaction write locks, no transaction read locks are taken.

A transaction lock is taken on an object when a field is accessed (serializable transaction isolation only) or modified. The transaction lock is released when the transaction commits or rolls back. Executing a method on an object does not take a transaction lock unless an object field is accessed (serializable transaction isolation only) or modified in the method. This implies that multiple threads can be executing the same method on the same object at the same time.

No transaction locks are taken on extents when objects are created or deleted. This allows better parallelism for object creation and deletion, but it does have implications for transactional isolation. See the *Distributed Transactional Memory Java Developer’s Guide* for details.

Distributed Transactional Memory supports multiple reader, single writer transaction locks. For example, multiple concurrent transactions can read the same object fields, but only a single transaction can modify an object field.

When a transaction is using a serializable transaction isolation, transaction read locks can be promoted to a transaction write lock if an object field is read, and then the field is modified in the same transaction. A transaction read lock would be taken on the initial field read and then promoted to a transaction write lock when the field is written. If multiple transactions attempt to promote a trans-
action read lock on the same object, all transactions, but one, will generate a promotion deadlock. A promotion deadlock causes the transaction to rollback, dropping its transaction locks. The transaction is then replayed causing the transaction to reacquire the transaction locks.

Distributed objects support the same transaction locking as objects on the local node.

**State conflicts**

A state conflict is reported by Distributed Transactional Memory when an object modification (create, write, delete) operation from a remote node detects that the data on the local node has changed underneath it. This is possible in a distributed system because the object may be modified from multiple nodes in the system. State conflicts can occur with both the standard distributed transaction protocol and the deferred write protocol (see the section called “Deferred Write Protocol” on page 54).

If a state conflict is detected an error is returned to the remote node where the object state is discarded, the transaction rolled back, and then replayed. The effect of this is that the object state will be resynchronized on the remote node. The application is never aware that a state conflict occurred. The only impact is on application performance.

Figure 5.5 shows an example of a state conflict. The sequence diagram shows these steps:

1. Transaction T1 on node 1 reads an object from node 2 and commits.
2. Transaction T2 on node 3 reads the same object from node 2 and commits.
3. Transaction T3 on node 3 modifies the object on node 2 and commits.
4. Transaction T4 on node 1 attempts to modify the same object on node 2, but the object has changed since the last time it was read onto node 1. A state conflict is detected and node 1 is instructed to rollback transaction T4 and to discard all object state.
5. Transaction T4 is replayed on node 1 as T5. The object state is first refreshed from node 2, and then the object is successfully modified.
Deadlock detection

Since transactions are running simultaneously, it is possible to have deadlocks in applications. Distributed Transactional Memory automatically detects deadlocks and handles them in the following manner:

- the transaction that detected the deadlock is chosen as the *victim*, this transaction is rolled back and replayed.
- another transaction waiting on a transaction lock that was released is chosen as the *winner* and allowed to complete.

Figure 5.5. State conflict
Figure 5.6. Deadlock detection

Figure 5.6 shows a deadlock caused by these actions:

1. Transaction 1 requests, and is granted, a read lock on Object 1.

2. Transaction 2 requests, and is granted, a read lock on Object 2.

3. Transaction 1 requests, but is not granted, a write lock on Object 2. The write lock is not granted because of the read lock held on Object 2 by Transaction 2. Objects cannot be modified while other transactions are reading the object.

4. Transaction 2 requests, but is not granted, a write lock on Object 1. This is a deadlock because both transactions would block indefinitely waiting for the other to complete. Transaction 2 is chosen as the victim and rolled back.

5. Transaction 1 is granted the requested write lock on Object 2 because Transaction 2's read lock on Object 2 was released when Transaction 2 was rolled back.

Notice that both transactions are attempting to promote a read lock to a write lock. This deadlock can be avoided by taking the write lock initially, instead of promoting from a read lock. See the Distributed Transactional Memory Java Developer's Guide for details on how to use explicit locking to avoid lock promotion deadlocks.

Deadlock detection and resolution is transparent to the application programmer, but deadlocks are expensive in both responsiveness and machine resources so they should be avoided.
Local transactions detect deadlocks immediately in the execution path. There is no timeout value associated with local transactions.

Distributed transactions use a configurable time-out value to detect deadlocks. If a lock cannot be obtained on a remote node within the configured time-out period, the distributed transaction is rolled back, releasing all locks. The transaction is then restarted.

![Graphical representation of distributed deadlock detection](image)

**Figure 5.7. Distributed deadlock detection**

Because distributed deadlock detection is based on a time-out, applications with distributed deadlocks will perform poorly because the configured time-out has to be large enough to ensure that there are never any false deadlocks reported during normal application processing.
Transaction logging

To support rollback of a transaction, all object modifications must be logged. The Distributed Transactional Memory logging mechanism is done in memory by keeping a copy of the before image of any changes. Any object references that are no longer referenced in a transaction are protected from garbage collection so they are still available if the current transaction rolls back.

If the current transaction commits, all logged data is discarded and any reference locks to deleted objects are released.

If the current transaction rolls back, the original state of all objects is restored. Any objects created in the transaction are released to allow them to be garbage collected.
Distributed computing

Any Distributed Transactional Memory Managed Object can be a distributed object. A distributed object transparently provides remote method invocation and access to object fields across nodes. The full transactional guarantees made by Distributed Transactional Memory for non-distributed objects are also true for distributed objects.

Access to a distributed object is through a normal Java object reference. All Managed Object references contain data to identify the node where the object was created.

The same instance of an object cannot exist on multiple nodes. Copies of an object's state may be located on multiple nodes to improve performance or robustness, but the master copy is located on a single node - by default the node where the object was created.

All object methods transparently execute on the master node for an object. Any methods invoked on an object reference are sent to the master node and executed there.

Objects of the same type can be created on multiple nodes. This is done by installing the application class files, or implementation, on multiple nodes. This is a common application architecture to support object partitioning and caching or service availability mechanisms.
Figure 6.1. Distributed method execution

Figure 6.1 shows an Order class that has its implementation installed on two nodes - Node One and Node Two. Two instances of the Order class have been created, one on Node One and one on Node Two. When the Order.cancel() method is executed on Node One, using the order(Node Two) instance, the method is executed on Node Two. The opposite is true for the order(Node One) instance.

Connectivity

The distribution protocol uses either TCP/IP, SSL, or Infiniband connectivity between nodes with a platform independent encoding. The platform independent encoding allows heterogeneous hardware platforms to communicate with each in a distributed transactional system. The optional automatic node discovery protocol uses UDP.

Location transparency

Distributed Transactional Memory provides location transparency for objects. This means that when an application accesses an object, its location is transparent — it may be on the local or on a remote node.

Location transparency is accomplished through the use of distributed references. All Managed Objects created in Distributed Transactional Memory have a distributed reference that contains the master node for the object. An object's identity, as defined by its distributed reference, does not change throughout the lifetime of the object.

Methods invoked on an object are always executed on the master node for an object.

Reading and writing object fields

Object field data is transparently read from and written to the master node when fields are accessed on a local node.
Read operations are dispatched to the master node to read field data depending on whether the local node has the data cached locally or not. If the field data is not available on the local node a distributed read will be done when a field is accessed. The read will complete before the get of the field returns to the caller. All reads are done on the master node in the same transaction in which the field access occurs.

When a field associated with a remote object is modified on a local node, by default, the update is deferred until the local transaction enters the prepare state. This is called deferred writes. See the section called “Deferred Write Protocol” on page 54 for details.

**Extents**

When an extent is accessed using a local query, only object references on the local node are returned - no read is dispatched to any remote nodes. References are in a local extent either because the object was created on the local node, it was returned in a method call, or it was pushed to the local node as part of object replication. Distributed queries can be used to access the global extent of all objects.

**Locations**

Every node is uniquely identified by:

- a cluster unique name
- a cluster unique location code
- a cluster unique shared memory timestamp

The default node name is set to the local host name. The default node name can be changed during node installation. This allows multiple Distributed Transactional Memory nodes to run on the same machine.

The location code is automatically derived from the node name using a hashing algorithm. The location code is a numeric identifier that is used to determine the actual network location of the master node for an object. The location code is stored with each Managed Object. The initial value of the location code for an object is the location code of the node on which the object was created.

Highly available objects can migrate to other nodes as part of failover, or to support load balancing. When object migration occurs the location code associated with all of the migrated objects is updated to use the location code of the node to which they were migrated. This update occurs on all nodes on which the objects exist. After the completion of an object migration, the new master node for the object is the new node, which may be different than the node on which the object was created.

The shared memory timestamp is assigned when the shared memory is first created for a node. This occurs the first time a node is started following an installation. The shared memory timestamp is a component of the opaque distributed reference. It ensures that the distributed reference is globally unique.
**Location discovery**

Location discovery provides runtime mapping between location codes, or node names, and network addresses. This is called *location discovery*.

Location discovery is done two ways:

- static discovery using configuration information.
- dynamic discovery using service discovery.

Configuration can be used to define the mapping between a node name and a network address. Configuring this mapping is allowed at any time, but it is only required if service discovery cannot be used for location discovery. An example of when this would be necessary is if a remote node is across a wide area network where service discovery is not allowed. This is called *static discovery*.

If configuration information is not provided for a location name, service discovery is used to perform location discovery. This has the advantage that no configuration for remote nodes has to be done on the local node - it is all discovered at runtime. This is called *dynamic discovery*.

![Warning](image.png)

When a network address is discovered with both static and dynamic discovery, the configured static discovery information is used.

Location discovery is performed in the following cases:

- A create of an object in a partition with a remote active node.
- A method or field is set on a remote object.

When an object is associated with a partition whose active node is remote, a location discovery request is done by node name, to locate the network information associated with the node name.

When an operation is dispatched on a remote object, a location discovery request is done by location code, to locate the network information associated with a location code.

Location code information is cached on the local node once it has been discovered.

**Life-cycle**

Initialization and termination of the distribution services are tied to activation and deactivation of distribution configuration data. A node without active distribution configuration cannot provide distributed services to a cluster. When distribution configuration is activated the following steps are taken to initialize distribution:

1. Mark the local node state as starting
2. Start dynamic discovery service if enabled
3. Start network listeners
4. Start keep-alive server
5. Mark the local node state as active
After initialization completes, the node is automatically part of the cluster. It can now provide access to distributed objects or provide high-availability services to other nodes in the cluster.

## Remote node states

Remote nodes can have one of the states in Table 6.1 on page 53.

### Table 6.1. Remote node states

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undiscovered</td>
<td>Node cannot be discovered. Network address information is not available from this remote node. Remote node is unavailable.</td>
</tr>
<tr>
<td>Discovered</td>
<td>The network address information for this node is discovered, either using dynamic or static discovery, but no connection could be established to the node. Remote node is unavailable.</td>
</tr>
<tr>
<td>In Up Notifier</td>
<td>Node is transitioning to an Up state. This is a transitory state. Any installed node available notifiers are being executed.</td>
</tr>
<tr>
<td>Up</td>
<td>Active connections are available to this node. Remote node is active.</td>
</tr>
<tr>
<td>In Down Notifier</td>
<td>Node is transitioning to the Down state. This is a transitory state. Any installed node unavailable notifiers are being executed.</td>
</tr>
<tr>
<td>Down</td>
<td>Node is inactive. No connections are active to this node, and new connection attempts fail with an error. Remote node is unavailable.</td>
</tr>
<tr>
<td>Duplicate Location</td>
<td>A duplicate location code was detected during connection establishment. No communication can occur with this node until this error is corrected. Remote node is unavailable.</td>
</tr>
<tr>
<td>Duplicate Timestamp</td>
<td>A duplicate installation time-stamp was detected during connection establishment. No communication can occur with this node until this error is corrected. Remote node is unavailable.</td>
</tr>
<tr>
<td>Unsupported Protocol</td>
<td>An unsupported protocol version was detected during connection establishment. No communication can occur with this node until this error is corrected. Remote node is unavailable.</td>
</tr>
</tbody>
</table>

### Remote node state change notifiers

Application installed node state change notifiers are called when a remote node transitions from active to unavailable and from unavailable to active. The In Up Notifier and In Down Notifier states defined in Table 6.1 on page 53 are seen when a node notifier is being called.

When a node state change notifier is installed, it is guaranteed to be called for all active remote nodes already discovered by the local node. Node notifier execution is serialized for a specific remote node. A call to a notifier must complete before another notifier is called. For example, if a remote node becomes unavailable while an active notifier is being executed, the unavailable notifier is not called until the active notifier completes.

Node state change notifiers are called in a transaction.
Deferred Write Protocol

By default, all distributed object updates use a *deferred write protocol*. The deferred write protocol defers all network I/O until the commit phase of a transaction. This allows the batching of all of the object updates, and the prepare request, into a single network I/O for each node, improving network performance. The size of the network buffer used for the network I/O is controlled in the distribution configuration. See the Distributed Transactional Memory Administration for details on distribution configuration.

The deferred write protocol is shown in Figure 6.2 for two nodes.

![Diagram showing the deferred write protocol for two nodes](image)

**Figure 6.2. Deferred write protocol**

Notice that no transaction locks are taken on node B as distributed objects are modified on node A until the prepare step.

---

Chapter 6. Distributed computing
Distributed object creates and deletes perform network I/O immediately, they are not deferred until commit time. There is no prepare phase enabled for these transactions. See Figure 5.1.

The deferred write protocol is disabled if a method call is done on a distributed object. Any modifications to the distributed object on the local node are flushed to the remote node before the method is executed on the remote node. This ensures that any updates made on the local node are available on the remote node when the method executes.

After the method executes on the remote node any modifications on the remote node are copied back to the initiating node. This ensures that the data is again consistent on the local node on which the method was originally executed.

The deferred write protocol can be disabled in the high availability configuration. In general, it should be enabled. However, if an application only accesses object fields using accessors, instead of directly accessing fields, it will be more performant to disable the deferred write protocol since no modifications are ever done on the local node. See the Distributed Transactional Memory Administration for details on high availability configuration.

Detecting failed nodes

Distributed Transactional Memory supports keep-alive messages between all nodes in a cluster. Keep-alive requests are used to actively determine whether a remote node is still reachable. Keep alive messages are sent to remote nodes using the configurable keepAliveSendIntervalSeconds time interval.

Figure 6.3 shows how a node is detected as being down. Every time a keep-alive request is sent to a remote node, a timer is started with a duration of nonResponseTimeoutSeconds. This timer is reset when a keep-alive response is received from the remote node. If a keep-alive response is not received within the nonResponseTimeoutSeconds interval, a keep-alive request is sent on the next network interface configured for the node (if any). If there are no other network interfaces configured for the node, or the nonResponseTimeoutSeconds has expired on all configured interfaces, all connections to the remote node are dropped, and the remote node is marked Down.

Connection failures to remote nodes are also detected by the keep-alive protocol. When a connection failure is detected, as opposed to a keep-alive response not being received, the connection is reattempted to the remote node before trying the next configured network interface for the remote node (if any). This connection reattempt is done to transparently handle transient network connectivity failures without reporting a false node down event.

It is important to understand that the total time before a remote node is marked Down is the number of configured interfaces times the nonResponseTimeoutSeconds configuration value in the case of keep-alive responses not being received. In the case of connection failures, the total time could be twice the nonResponseTimeoutSeconds times the number of configured interfaces, if both connection attempts to the remote node (the initial one and the retry) hang attempting to connect with the remote node.

For example, in the case of keep live responses not being received, if there are two network interfaces configured, and the nonResponseTimeoutSeconds value is four seconds, it will be eight seconds before the node is marked Down. In the case of connection establishment failures, where each connection attempt hangs, the total time would be sixteen seconds before the node is marked Down.
Network error handling

Distribution uses TCP as the underlying network protocol. In general, TCP provides reliable connectivity between machines on a network. However, it is possible that network errors can occur that cause a TCP connection to drop. When a TCP connection is dropped, requests and responses between nodes participating in a distributed transaction are not received. Network errors are detected by the keep-alive protocol described in the section called “Detecting failed nodes” on page 55 and handled by the distributed transaction protocol.

Network connectivity failures are caused by:
• A non-response keep alive timeout occurring.

• TCP retry timers expiring.

• Lost routes to remote machines.

These errors are usually caused by network cables being disconnected, router crashes, or machine interfaces being disabled.

As discussed in the section called “Local and distributed transactions” on page 37, all distributed transactions have a transaction initiator that acts as the transaction coordinator. The transaction initiator can detect network failures when sending a request, or reading a response from a remote node. When the transaction initiator detects a network failure, the transaction is rolled back. Other nodes in a distributed transaction can also detect network failures. When this happens, rollback is returned to the transaction initiator, and again the transaction initiator rolls back the transaction. This is shown in Figure 6.4.
Figure 6.4. Connection failure handling

When the transaction initiator performs a rollback because of a connection failure - either detected by the initiator or another node in the distributed transaction, the rollback is sent to all known nodes. Known nodes are those that were located using location discovery (see the section called “Location discovery” on page 52). This must be done because the initiator does not know which nodes are participating in the distributed transaction. Notice that a rollback is sent to all known nodes in Figure 6.4. The rollback is retried until network connectivity is restored to all nodes.
Transaction rollback is synchronized to ensure that the transaction is safely aborted on all participating nodes, no matter the current node state.

**Distributed transaction failure handling**

Any communication failures to remote nodes detected during a global transaction before a commit sequence is started cause an exception that an application can handle (see the *Distributed Transactional Memory Java Developer’s Guide*). This allows the application to explicitly decide whether to commit or rollback the current transaction. If the exception is not caught, the transaction will be automatically rolled back.

Undetected communication failures to remote nodes do not impact the commit of the transaction. This failure scenario is shown in Figure 6.5. In this case, Node 2 failed and was restarted after all locks were taken on Node 2, but before the commit sequence was started by the transaction initiator - Node 1. Once the commit sequence starts it continues to completion. The request to commit is ignored on Node 2 because the transaction state was lost when Node 2 restarted.
Figure 6.5. Undetected communication failure

Transaction initiator node failures are handled transparently using a transaction outcome voting algorithm. There are two cases that must be handled:
• Transaction initiator fails before commit sequence starts.

• Transaction initiator fails during the commit sequence.

When a node that is participating in a distributed transaction detects the failure of a transaction initiator, it queries all other nodes for the outcome of the transaction. If the transaction was committed on any other participating nodes, the transaction is committed on the node that detected the node failure. If the transaction was aborted on any other participating nodes, the transaction is aborted on the node that detected the failure. If the transaction is still in progress on the other participating nodes, the transaction is aborted on the node that detected the failure.

Transaction outcome voting before the commit sequence is shown in Figure 6.6. In Figure 6.6 the initiating node, Node 1, fails before initiating the commit sequence. When Node 2 detects the failure it performs the transaction outcome voting algorithm by querying other nodes in the cluster to see if they are participating in this transaction. Since there are no other nodes in this cluster, the Transaction Status request is a noop and the transaction is immediately aborted on Node 2, releasing all locks held by the distributed transaction.

![Figure 6.6. Transaction initiator fails prior to initiating commit sequence](image)

Transaction outcome voting during a commit sequence is shown in Figure 6.7. In Figure 6.7 the initiating node, Node 1, fails during the commit sequence after committing the transaction on Node

---

61
2, but before it is committed on Node 3. When Node 3 detects the failure it performs the transaction outcome voting algorithm by querying Node 2 for the resolution of the global transaction. Since the transaction was committed on Node 2 it is committed on Node 3.

Figure 6.7. Transaction initiator fails during commit sequence
To support transaction outcome voting each node maintains a history of all committed and aborted transactions for each remote node participating in a global transaction. The number of historical transactions to maintain is configurable and should be based on the time for the longest running distributed transaction. For example, if 1000 transactions per second are being processed from a remote node, and the longest transaction on average is ten times longer than the mean, the transaction history buffer should be configured for 10,000 transactions.

For each transaction from each remote node, the following is captured:

- global transaction identifier
- node login time-stamp
- transaction resolution

The size of each transaction history record is 24 bytes.
Distributed Transactional Memory provides these high availability services:

- Synchronous and asynchronous object replication
- Dynamic object partitioning
- Application transparent partition failover, restoration and migration
- Node quorum support with multi-master detection and avoidance
- Recovery from multi-master situations with conflict resolution
- Geographic redundancy

The high-availability features are supported using configuration and are transparent to applications. There are also public APIs available for all of the high-availability services to allow more complex high-availability requirements to be met by building applications that are aware of the the native high-availability services. Configuring high-availability services is described in the Administration Guide, while programming to the API is described in the Java Developer's Guide.

The remaining sections in the chapter provide an architectural overview of the high-availability services.

## Conceptual model

Figure 7.1 shows the high availability concepts added to the standard deployment conceptual model described in the section called “Conceptual model” on page 15.

- **Availability Zone** - a collection of nodes that provide redundancy and quorum for each other.
- **Data Distribution Policy** - policy for distributing data across an availability zone.
• **Data Distribution Policy Binding** - binds an application data type to a data distribution policy and a data mapper.

• **Data Mapper** - implements a specific data distribution policy, for example round-robin, consistent hashing, etc.

• **Dynamic Data Distribution Policy** - a policy that automatically distributes data evenly across all nodes in an availability zone. Data is automatically redistributed as nodes are added and removed.

• **Dynamic Partition** - a partition created as required to evenly distribute data when using a dynamic data distribution policy.

• **Quorum Policy** - a policy, and actions, to prevent a partition from being active on multiple nodes simultaneously.

• **Partition** - the unit of data distribution within a data distribution policy.

• **Static Data Distribution Policy** - a policy that explicitly distributes data across all nodes in an availability zone to ensure optimal data locality. Data is not automatically redistributed as nodes are added and removed.

• **Static Partition** - a partition explicitly defined to distributed data when using a static data distribution policy.

Each of the concepts in Figure 7.1 is identified as either design-time or deploy-time. Design-time concepts are defined in the application definition configuration (see the section called “Design-time” on page 6) and the deploy-time concepts are defined in the node deploy configuration (see the section called “Deploy-time” on page 8).
Figure 7.1. High availability concepts

An availability zone belongs to a single cluster, has one or more nodes, and one or more optional quorum policies.

A cluster can have zero or more availability zones.

A data distribution policy is associated with one or more availability zones and a data distribution policy binding.

A data distribution policy binding associates a data distribution policy, an application fragment, and a data mapper.

A data mapper is associated with one or more data distribution policy bindings.

An application fragment is associated with one or more data distribution policy bindings.

A dynamic data distribution policy is associated with one or more dynamic partitions.

A dynamic partition is associated with one dynamic data distribution policy.

A node can belong to zero or more availability zones.

A quorum policy is associated with one availability zone.

A partition is associated with one data distribution policy.

A static data distribution policy is associated with one or more static partitions.
A static partition is associated with one static data distribution policy.

**Data distribution policies**

A data distribution policy defines how application data is distributed, or *partitioned*, across multiple nodes. Partitioning of application data allows large application data sets to be distributed across multiple nodes, typically each running on a separate machine. This provides a simple mechanism for applications to grow horizontally by scaling the amount of machine resources available for storing and processing application data. Data distribution policies are associated with one or more availability zones.

To support partitioning application data, an application fragment must be associated with one or more data distribution policies. A data distribution policy defines these characteristics:

- How partitions (see the section called “Partitions” on page 72) are distributed across available machine resources in an availability zone.
- The replication requirements for the application data, this includes the replication style, asynchronous vs. synchronous, and the number of replica copies of the data.
- The partitioning algorithm to use to distribute the application data across the available partitions.

Partitions can be distributed across the available machine resources, either dynamically, a *dynamic data distribution policy*, or statically, a *static data distribution policy*.

A dynamic data distribution policy automatically creates *dynamic partitions* based on the available nodes. There is no requirement to explicitly configure dynamic partitions or to map them to specific nodes. Data in a dynamic data distribution policy is redistributed across dynamic partitions when a node joins or leaves a cluster. The repartitioning is triggered by reloading the configuration on all nodes in the cluster.

Figure 7.2 shows a dynamic data distribution policy that provides round-robin data distribution across three nodes. As data is created it is assigned to the nodes using a round-robin algorithm; **Data 1** is created on Node A, **Data 2** is created on Node B, **Data 3** is created on Node C, and the next data created will be created back on Node A. There are dynamic partitions created to support this data distribution policy, but they are omitted from the diagram since they are implicitly created by the runtime. If additional nodes are added, the round-robin algorithm will include them when distributed data.
Figure 7.2. Dynamic data distribution policy

A static data distribution policy requires that static partitions be explicitly configured and mapped to nodes. Data is never repartitioned when using a static data distribution policy. Data can only be migrated to other nodes by loading updated configuration, but the data is never re-assigned to a new static partition once it has been created.

Figure 7.3 shows a static data distribution policy that uses a data mapper that maps odd data to a partition named **Odd** and even data to a partition named **Even**. As data is created it is created on the active node for the partition, so all Odd data is created on **Node A** and all Even data is created on **Node B**. Adding a new new node to a static distribution policy will not impact where the data is created, it will continue to be created on the nodes that are hosting the partition to which the data is mapped. Contrast this to the behavior of the dynamic data distribution policy - where new nodes affect the distribution of the data.
Figure 7.3. Static data distribution policy

The number of copies of data, or replicas, is defined by a data distribution policy. The policy also defines whether replication should be done synchronously or asynchronously, and other runtime tuning and recovery properties.

Data is distributed across partitions at runtime using a data mapper. A data mapper maps application data to a specific partition at runtime, usually based on the content of the application data. Built-in data mappers are provided, and applications can provide their own data mappers to perform application-specific data mapping.

Availability zones

Figure 7.4 shows a simple example of a single cluster with three nodes deployed in a single data center. There is one availability zone defined which contains all of the nodes in the cluster. The
availability zone would define a data distribution policy appropriate for the deployed application, and possibly define a quorum policy that ensures that quorum is always maintained.

![Diagram](image.png)

**Figure 7.4. Single data center availability zone**

Figure 7.5 shows a more complex example of a cluster that spans two data centers to provide disaster recovery for an application. This examples defines these availability zones:

- **West Coast** - availability zone within the West Coast data center.
- **East Coast** - availability zone within the East Coast data center.
- **Disaster Recovery** - availability zone that spans the west and east coast data centers to provide data redundancy across data centers.

Notice that nodes C and E are in multiple availability zones, with each availability zone having possibly different quorum policies.

The application data types that represent the critical application data must be in the same data distribution policy. This distribution policy is then associated with the Disaster Recovery availability zone to ensure that the data is redundant across data centers.
Partitions provide the unit of data distribution within a data distribution policy.

A partition is identified by a name. Partition names must be globally unique across all nodes in a cluster. Each partition has a node list consisting of one or more nodes. The node list is specified in priority order, with the highest priority available node in the node list the active node for the partition. All other nodes in the node list are replica nodes for the partition. Replica nodes can use either synchronous or asynchronous replication (see the section called “Replication” on page 82).

If the active node becomes unavailable, the next highest available replica node in the node list automatically becomes the active node for the partition.

All data in a partition with replica nodes has a copy of the data transparently maintained on all replica nodes. This data is called replica data.
Figure 7.6. Partition definitions

Figure 7.6 defines three partitions named One, Two, and Three. Partitions One and Two support replication of all contained data, with node B replication done synchronously and node C replication done asynchronously. Partition Three has only a single node B defined so there is no replication in this partition. All data assigned to partition Three during creation are transparently created on node B. A node A failure will cause the active node for partition One and Two to change to node B. A node B failure has no impact on the active node for partition One and Two, but it causes all data in partition Three to be lost since there is no other node hosting this partition.

**Partitioned object**

Data is partitioned by installing a data mapper on a managed object using configuration. A managed object with a data mapper installed is called a partitioned object. A data mapper is responsible for assigning a managed object to a partition. Partition assignment occurs:

- when an object is created
- during partition mapping updates (see the section called “Updating object partition mapping” on page 86).

Data mappers are inherited by all subtypes of a parent type. A child type can install a new data mapper to override a parent's data mapper.

A partitioned object is always associated with a single partition, but the partition it is associated with can change during the lifetime of the object.

The algorithm used by a data mapper to assign an object to a partition is application specific. It can use any of the following criteria to make a partition assignment:

- object instance information
system resources (e.g. CPU, shared memory utilization, etc.) utilization

load balancing, e.g. consistent hashing, round-robin, priorities, etc.

any other application specific criteria

Built-in data mappers for consistent hashing and round-robin are provided.

**Distributed consistent hashing** The distributed consistent hashing data mapper distributes partitioned objects evenly across all nodes in a cluster; maintaining an even object distribution even as nodes are added or removed from a cluster. When a new node joins a cluster, it takes its share of the objects from the other nodes in the cluster. When a node leaves a cluster, the remaining nodes share the objects that were on the removed node.

Assigning an object to a partition using distributed consistent hashing consists of these steps:

1. Generate a hash key from data in the object.
2. Access the hash ring buffer location associated with the generated hash key value.
3. Map the object to the partition name stored at the accessed hash ring buffer location.

The important thing about the consistent hashing algorithm is that the same data values consistently map to the same partition.

The size of the hash ring buffer controls the resolution of the object mapping into partitions - a smaller hash ring may cause a lumpy distribution, while a larger hash ring will more evenly spread the objects across all available partitions.

The number of partitions determines the granularity of the data distribution across the nodes in a cluster. The number of partitions also constrains the total number of nodes over which the data can be distributed. For example, if there are only four partitions available, the data can only be distributed across four nodes, no matter how many nodes are in the cluster.

The size of the hash ring buffer should be significantly larger than the number of partitions for optimal data distribution.

Figure 7.7 shows an example of mapping data to partitions using distributed consistent hashing. The colored circles on the consistent hash ring buffer in the diagram represent the hash ring buffer location for a specific hash key value. These locations contain a partition name. In this example, both Data 1 and Data 2 map to partition Two on node B, while Data 3 maps to partition Three on node C, and Data 4 maps to partition Four on node D.

While the example shows each node only hosting a single partition, this is not realistic, since adding more nodes would not provide better data distribution since there are no additional partitions to migrate to a new node. Real configurations have many partitions assigned to each node.
Round-robin  The round-robin data mapper distributes data evenly across all static partitions, *not nodes*, in a static data distribution policy. Data is sent in order to each of the partitions defined in the static partition policy. The distribution of the static partitions across nodes is defined by the partition to node mapping defined by the static data distribution policy. For example in Figure 7.8, two thirds of the data ends up on Node A, since there are two partitions defined on Node A, and only one third on Node B, since Node B has only a single partition defined.
Figure 7.9 shows three nodes, A, B, and C, and a partition P. Partition P is defined with an active node of A and a replica node of B. Partition P is also defined on node C but node C is not in the node list for the partition. On node C, partition P is considered a sparse partition.
The partition state and node list of sparse partitions is maintained as the partition definition changes in the cluster. However, no objects are replicated to these nodes, and these nodes cannot become the active node for the partition. When an object in a sparse partition is created or updated, the create and update is pushed to the active and any replica nodes in the partition.

Figure 7.9. Sparse partition

Sparse partition definitions are useful for application specific mechanisms that require a node to have a distributed view of partition state, without being the active node or participating in replication.

**Defining partitions**

Partitions are defined directly by the application or an administrator on a running system. Partitions should be defined and enabled (see the section called “Enabling and disabling partitions” on page 78) on all nodes on which the partition should be known. This allows an application to:

- immediately use a partition. Partitions can be safely used after they are enabled. There is no requirement that the active node has already enabled a partition to use it safely on a replica node.
- restore a node following a failure. See the section called “Restore” on page 79 for details.
As an example, here are the steps to define a partition \( P \) in a cluster with an active node of \( A \) and a replica node of \( B \).

1. Nodes \( A \) and \( B \) are started and have discovered each other.
2. Node \( A \) defines partition \( P \) with a node list of \( A, B \).
3. Node \( A \) enables partition \( P \).
4. Node \( B \) defines partition \( P \) with a node list of \( A, B \).
5. Node \( B \) enables partition \( P \).

Partition definitions can be redefined to allow partitions to be migrated to different nodes. See the section called “Migrating a partition” on page 86 for details.

The only time that node list inconsistencies are detected is when object re-partitioning is done (see the section called “Updating object partition mapping” on page 86), or a sparse partition is being defined.

**Enabling and disabling partitions**

Once, a partition has been defined, it must be enabled. Enabling a partition causes the local node to transition the partition from the Initial state to the Active state. Partition activation may include migration of object data from other nodes to the local node. It may also include updating the active node for the partition in the cluster. Enabling an already Active partition has no affect.

Disabling a partition causes the local node to stop hosting the partition. The local node is removed from the node list in the partition definition on all nodes in the cluster. If the local node is the active node for a partition, the partition will migrate to the next node in the node list and become active on that node. As part of migrating the partition all objects in the partition on the local node are removed from shared memory.

When a partition is disabled with only the local node in the node list there is no impact to the objects contained in the partition on the local node since a partition migration does not occur. These objects can continue to be read by the application. However, unless the partition mapper is removed, no new objects can be created in the disabled partition because there is no active node for the partition.

**Remotely defined and enabled partitions** When a partition is defined, the partition definition is broadcast to all discovered nodes in the cluster. The RemoteDefined status (see the section called “Partition status” on page 82) is used to indicate a partition that was remotely defined. When the partition is enabled, the partition status change is again broadcast to all discovered nodes in the cluster. The RemoteEnabled status (see the section called “Partition status” on page 82) is used to indicate a partition that was remotely enabled.

While the broadcast of partition definitions and status changes can eliminate the requirement to define and enable partitions on all nodes in a cluster that must be aware of a partition, it is recommended that this behavior not be relied on in production system deployments.

The example below demonstrates why relying on partition broadcast can cause problems.

1. Nodes \( A, B, \) and \( C \) are all started and discover each other.
2. Node \( A \) defines partition \( P \) with a node list of \( A, B, C \). Replica nodes \( B \) and \( C \) rely on the partition broadcast to remotely enable the partition.
3. Node B is taken out of service. Failover (see the section called “Failover” on page 79) changes the partition node list to A, C.

4. Node B is restarted and all nodes discover each other, but since node B does not define and enable partition P during application initialization the node list remains A, C.

At this point, manual intervention is required to redefine partition P to add B back as a replica. This manual intervention is eliminated if all nodes always define and enable all partitions during application initialization.

**Failover**

A partition with one or more replica nodes defined in its node list will failover if its current active node fails. The next highest priority available node in the node list will take over processing for this partition.

When a node fails, it is removed from the node list for the partition definition in the cluster. All undiscovered nodes in the node list for the partition are also removed from the partition definition. For example, if node A fails with the partition definitions in Figure 7.6 active, the node list is updated to remove node A leaving these partition definitions active in the cluster.

<table>
<thead>
<tr>
<th>Name: One</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node List: B, C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name: Two</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node List: B</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name: Three</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node List: B</td>
</tr>
</tbody>
</table>

**Figure 7.10. Updated partition node list**

Once a node has been removed from the node list for a partition, no communication occurs to that node for the partition.

**Restore**

A node is restored to service by defining and enabling all partitions that will be hosted on the node. This includes partitions for which the node being restored is the active or replica node. When a partition is enabled on the node being restored partition migration occurs, which copies all objects in the hosted partitions to the node.
To restore node A to service after the failure in the section called “Failover” on page 79, requires the following steps:

- define and enable partition One with active node A and replicas B and C.
- define and enable partition Two with active node A and replica B.

After these steps are executed, and partition migration completes, node A is back online and the partition definitions are back to the original definitions in Figure 7.6.

**Partition states**

Partitions can have one of the following states:

**Table 7.1. Partition states**

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>Partition was defined, but not enabled. Objects cannot be mapped to this partition in this state.</td>
</tr>
<tr>
<td>Active</td>
<td>Partition is running on the active node for the partition.</td>
</tr>
<tr>
<td>Migrating</td>
<td>The active node for a partition is being updated. This state occurs during failover, restore, and during operator migration of a partition.</td>
</tr>
<tr>
<td>Replicating</td>
<td>Partition replicas are being updated, but the active node is not changing. Objects are being pushed to the replica nodes that were added, then removed from replica nodes that were deleted from the partition's node list. This state occurs when an existing partition's node list is redefined.</td>
</tr>
<tr>
<td>Updating</td>
<td>Partition object membership is being updated. This state is entered when a re-partition is occurring.</td>
</tr>
<tr>
<td>Unavailable</td>
<td>Partition is not active on any node. Objects cannot be mapped to this partition in this state.</td>
</tr>
</tbody>
</table>

Figure 7.11 shows the state machine that controls the transitions between all of these states.
Figure 7.11. Partition state machine

The external events in the state machine map to an API call or an administrator command. The internal events are generated as part of node processing.

Partition state change notifiers  Partition state change notifiers are called at partition state transitions if an application installs them. Partition state change notifiers are called in these cases:

- the transition into and out of the transient states defined in Figure 7.11. These notifiers are called on every node in the cluster that has the notifiers installed and the partition defined and enabled.
- the transition directly from the Active state to the Unavailable state in Figure 7.11. These notifiers are only called on the local node on which this state transition occurred.
Partition status

Partitions also have a status, which defines how the local definition of the partition was done, and whether it has been enabled. The valid states are defined in Table 7.2 on page 82.

Table 7.2. Partition status

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LocalDefined</td>
<td>The partition was defined on the local node.</td>
</tr>
<tr>
<td>RemoteDefined</td>
<td>The partition was never defined on the local node. It was only remotely defined.</td>
</tr>
<tr>
<td>RemoteEnabled</td>
<td>The partition was never enabled on the local node. It was only remotely enabled.</td>
</tr>
<tr>
<td>LocalEnabled</td>
<td>The partition was enabled on the local node.</td>
</tr>
<tr>
<td>LocalDisabled</td>
<td>The partition was disabled on the local node.</td>
</tr>
</tbody>
</table>

All of the partition status values are controlled by an administrative operation, or API, on the local node except for the RemoteEnabled and RemoteDefined statuses. The RemoteEnabled and RemoteDefined statuses occurs when local partition state was not defined and enabled on the local node, it was only updated on a remote node.

If the local node leaves the cluster and is restarted, it must redefine and enable a partition locally before rejoining the cluster to rejoin as a member of the partition. For this reason it is recommended that all nodes perform define and enable for all partitions in which they participate, even if they are a replica node in the partition.

Replication

Partitioned objects are replicated to multiple nodes based on the node list in their partition definition. Objects that have been replicated to one or more nodes are highly available and are available to the application following a node failure.

Replication can be synchronous or asynchronous on a per-node basis in a partition. A mix of synchronous and asynchronous replication within the same partition is supported. For example in Figure 7.6, partition One is defined to use synchronous replication to node B and asynchronous replication to node C.

Synchronous replication guarantees that all replica nodes are updated in the same transaction in which the replicated object was modified. There can be no loss of data. However, the latency to update all of the replica nodes is part of the initiating transaction. By default, synchronous replication uses the deferred write protocol described in the section called “Deferred Write Protocol” on page 54.

Asynchronous replication guarantees that any modified objects are queued in a separate transaction. The object queue is per node and is maintained on the same node on which the modification occurred. Modified objects are updated on the replica nodes in the same order in which the modification occurred in the original transaction. The advantage of asynchronous replication is that it removes the update latency from the initiating transaction. However, there is potential for data loss if a failure occurs on the initiating node before the queued modifications have been replicated.

Figure 7.12 shows asynchronous replication behavior when a modification is made on the active node for a partition. The following steps are taken in this diagram:

1. A transaction is started.
2. Replicated objects are modified on the active node.

3. The modified objects are transactionally queued on the active node.

4. The transaction commits.

5. A separate transaction is started on the active node to replicate the objects to the target replica node.

6. The transaction is committed after all queued object modifications are replicated to the target node.

**Figure 7.12. Asynchronous replication**

Because asynchronous replication is done in a separate transaction consistency errors can occur. When consistency errors are detected they are ignored, the replicated object is discarded, and a warning message is generated. These errors include:

- Duplicate keys.
- Duplicate object references caused by creates on an asynchronous replica.
- Invalid object references caused by deletes on an asynchronous replica.

All other object modifications in the transaction are performed when consistency errors are detected.

Figure 7.13 provides more details on the differences between synchronous and asynchronous replication. The key things to notice are:

- Synchronous modifications (creates, deletes, and updates) are always used when updating the active node and any synchronous replica nodes.
- Modifications to asynchronous replica nodes are always done from the active node, this is true even for modifications done on asynchronous replica nodes.

⚠️ It is strongly recommend that no modifications be done on asynchronous replica nodes since there are no transactional integrity guarantees between when the modification occurs on the asynchronous replica and when it is reapplied from the active node.
Synchronous updates are always done from the node on which the modification occurred - this can be the active or a replica node.

These cases are shown in Figure 7.13. These steps are shown in the diagram for a partition P with the specified node list:

1. A transaction is started on node C - a replica node.
2. A replicated object in partition P is modified on node C.
3. When the transaction is committed on node C, the update is synchronously done on node A (the active node) and node B (a synchronous replica).
4. Node A (the active node) queues the update for node D - an asynchronous replica node.
5. A new transaction is started on node A and the update is applied to node D.
Figure 7.13. Replication protocol

Error handling

If an I/O error is detected attempting to send creates, updates, or deletes to a replica node, an error is logged on the node initiating the replication and the object modifications for the replica node are discarded and the replica node is removed from the node list for the partition. These errors include:

- the replica node has not been discovered yet
- the replica node is down
- an error occurred while sending the modifications

The replica node will be re-synchronized with the active node when the replica node is restored (see the section called “Restore” on page 79).
Updating object partition mapping

Partitioned objects can be re-partitioned on an active system. This provides a mechanism for mapping objects to new partitions.

The partition mapping for objects is updated using an administrative command or an API. Partition mapping updates can only be initiated on the active node for a partition. When the partition update is requested an audit is performed to ensure that the node list is consistent for all discovered nodes in the cluster. This audit is done to ensure that no object data is migrated to other nodes as part of remapping the partitions.

When a partition update is requested, all installed partition mappers on the active node are called for all partitioned objects. The objects will be moved to the partition returned by the partition mapper.

Object partition mapping updates only occur if the partition mapper installed by the application supports a dynamic mapping of objects to partitions. If the partition mapper only supports a static mapping of objects to partitions no remapping will occur.

New partition mappers can be installed on a node to perform partition updates as shown in these steps:

1. Define and enable a new partition on the local node.
2. Install a new partition mapper that maps objects to the new partition.
3. Perform the partition update.
4. Optionally migrate the partition as needed.

This technique has the advantage that objects created while the partition update is executing will be mapped to the new partition.

Migrating a partition

Partitions support migration to different nodes without requiring system downtime. Partition migration is initiated using an administrator or an API on the current active node for the partition. The following changes can be made to a partition definition:

- Change the priority of the node list, including the active node.
- Add new nodes to the node list
- Remove nodes from the node list
- Update partition properties.

When the partition migration is initiated all object data is copied as required to support the updated partition definition, this may include changing the active node for the partition.

For example, these steps will the migrate the active node from A to C for partition P:

1. Node C defines partition P with a node list of C, B.
2. Node C enables the partition and partition P migrates to node C.
When the partition migration is complete, partition P is now active on node C with node B still the replica. Node A is no longer hosting this partition.

It is also possible to force replication to all replica nodes during a partition migration by setting the force replication property when initiating partition migration. Setting the force replication property will cause all replica nodes to be resynchronized with the active node during partition migration. In general forcing replication is not required since replica nodes resynchronize with the active node when partitions are defined and enabled on the replica node.

**Active node transparency**

As discussed in the section called “Location transparency” on page 50, partitioned objects are also distributed objects. This provides application transparent access to the current active node for a partition. Applications simply create objects, read and modify object fields, and invoke methods. The Distributed Transactional Memory runtime ensures that the action occurs on the current active node for the partition associated with the object.

When an active node fails, and the partition is migrated to a new active node, the failover to the new active node is transparent to the application. No creates, updates, or method invocations are lost during partition failover as long as the node that initiated the transaction was not the failing node. Failover processing is done in a single transaction to ensure that it is atomic. See Figure 7.14.

**Object locking during migration**

When a partition is migrated to a new active node all objects in the partition must be write locked on both the new and old active nodes, and all replica nodes. This ensures that the objects are not modified as they are migrated to the new node.

When an object is copied to a new node, either because the active node is changing, or a replica node changed, a write lock is taken on the current active node and a write lock is taken on the replica node. This ensures that the object is not modified during the copy operation.

To minimize the amount of locking during an object migration, separate transactions are used to perform the remote copy operations. The number of objects copied in a single transaction is controlled
by the objects locked per transaction partition property. Minimizing the number of objects locked in a single transaction during object migration minimizes application lock contention with the object locking required by object migration.

**Node quorum**

Distributed Transactional Memory uses a quorum mechanism to detect, and optionally, prevent partitions from becoming active on multiple nodes. When a partition is active on multiple nodes a *multiple master, or split-brain*, scenario has occurred. A partition can become active on multiple nodes when connectivity between one or more nodes in a cluster is lost, but the nodes themselves remain active. Connectivity between nodes can be lost for a variety of reasons, including network router, network interface card, or cable failures.

![Figure 7.15. Multi-master scenario](image)

Figure 7.15 shows a situation where a partition may be active on multiple nodes if partitions exist that have all of these nodes in their node list. In this case, Node Two assumes that Node One and Node Three are down, and makes itself the active node for these partitions. A similar thing happens on Node One and Node Three - they assume Node Two is down and take over any partitions that were active on Node Two. At this point these partitions have multiple active nodes that are unaware of each other.

The node quorum mechanism provides these mutually exclusive methods to determine whether a node quorum exists:

- minimum number of active remote nodes in a cluster.
- percentage of votes from currently active nodes in a cluster.

When using the minimum number of active remote nodes to determine a node quorum, the node quorum is not met when the number of active remote nodes drops below the configured minimum number of active nodes.

When using voting percentages, the node quorum is not met when the percentage of votes in a cluster drops below the configured node quorum percentage. By default each node is assigned one
vote. However, this can be changed using configuration. This allows certain nodes to be given more weight in the node quorum calculation by assigning them a larger number of votes.

When node quorum monitoring is enabled, high-availability services are Disabled if a node quorum is not met. This ensures that partitions can never be active on multiple nodes. When a node quorum is restored, by remote nodes being rediscovered, the node state is set to Partial or Active depending on the number of active remote nodes and the node quorum mechanism being used. See the section called “Node quorum states” on page 89 for complete details on node quorum states.

See the Distributed Transactional Memory Administration Guide for details on designing and configuring node quorum support.

## Node quorum states

The valid node quorum states are defined in Table 7.3 on page 89.

### Table 7.3. Node quorum states

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active</td>
<td>All discovered nodes are Up. A node quorum exists.</td>
</tr>
<tr>
<td>Partial</td>
<td>One or more discovered nodes are Down. A node quorum still exists.</td>
</tr>
<tr>
<td>Disabled</td>
<td>A node quorum does not exist. High availability services are disabled on this node. The state of all hosted partitions has been set to Unavailable. Keep-alive processing from remote nodes is disabled. This ensures that remote nodes detect this node as unavailable.</td>
</tr>
</tbody>
</table>

Figure 7.16 shows the state machine that controls the transitions between all of these states when node quorum is using the minimum number of active nodes method to determine whether a quorum exists.
Figure 7.16. Quorum state machine - minimum number of active remote nodes

Figure 7.17 shows the state machine that controls the transitions between all of these states when node quorum is using the voting method to determine whether a quorum exists.
Disabling node quorum

There are cases where disabling node quorum is desired. Examples are:
Network connectivity and external routing ensures that requests are always targeted at the same node if it is available.

Geographic redundancy, where the loss of a WAN should not bring down the local nodes.

To support these cases, the node quorum mechanism can be disabled using configuration (see the Distributed Transactional Memory Administration Guide). When node quorum is disabled, high availability services will never be disabled on a node because of a lack of quorum. With the node quorum mechanism disabled, a node can only be in the Active or Partial node quorum states defined in Table 7.3 on page 89 - it never transitions to the Disabled state. Because of this, it is possible that partitions may have multiple active nodes simultaneously.

**Restoring a cluster**

This section describes how to restore a cluster following a multi-master scenario. These terms are used to describe the roles played by nodes in restoring after a multi-master scenario:

- **source** - the source of the object data. The object data from the initiating node is merged on this node. Installed compensation triggers are executed on this node.

- **initiating** - the node that initiated the restore operation. The object data on this node will be replaced with the data from the source node.

To recover partitions that were active on multiple nodes, support is provided for merging objects using an application implemented compensation trigger. If a conflict is detected, the compensation trigger is executed on the source node to allow the conflict to be resolved.

The types of conflicts that are detected are:

- **Instance Added** - an instance exists on the initiating node, but not on the source node.

- **Key Conflict** - the same key value exists on both the initiating and source nodes, but they are different instances.

- **State Conflict** - the same instance exists on both the initiating and source nodes, but the data is different.

The application implemented compensation trigger is always executed on the source node. The compensation trigger has access to data from the initiating and source nodes.

Figure 7.18 shows an example cluster with a single partition, P, that has node A as the active node and node B as the replica node.
Figure 7.18. Active cluster

Figure 7.19 shows the same cluster after connectivity is lost between node A and node B with node quorum disabled. The partition P definition on node A has been updated to remove node B as a replica because it is no longer possible to communicate with node B. Node B has removed node A from the partition definition because it believes that node A has failed so it has taken over responsibility for partition P.

Figure 7.19. Failed cluster

Once connectivity has been restored between all nodes in the cluster, and the nodes have discovered each other, the operator can initiate the restore of the cluster. The restore (see the section called "Restore" on page 79) is initiated on the initiating node which is node A in this example. All partitions on the initiating node are merged with the same partitions on the source nodes on which the partitions are also active. In the case where a partition was active on multiple remote nodes, the node to merge from can be specified per partition, when the restore is initiated. If no remote node is specified for a partition, the last remote node to respond to the Is partition(n) active? request (see Figure 7.20) will be the source node.
Figure 7.20. Merge operation - using broadcast partition discovery

Figure 7.20 shows the steps taken to restore the nodes in Figure 7.19. The restore command is executed on node A which is acting as the initiating node. Node B is acting as the source node in this example.

The steps in Figure 7.20 are:

1. Operator requests restore on A.
2. A sends a broadcast to the cluster to determine which other nodes have partition P active.
3. B responds that partition P is active on it.
4. A sends all objects in partition P to B.
5. B compares all of the objects received from A with its local objects in partition P. If there is a conflict, any application reconciliation triggers are executed. See the section called “Default conflict resolution” on page 95 for default conflict resolution behavior if no application reconciliation triggers are installed.
6. A notifies B that it is taking over partition P. This is done since node A should be the active node after the restore is complete.
7. B pushes all objects in partition P to A and sets the new active node for partition P to A.
8. The restore command completes with A as the new active node for partition P (Figure 7.18).
The steps to restore a node, when the restore from node was specified in the restore operation are very similar to the ones above, except that instead of a broadcast to find the source node, a request is sent directly to the specified source node.

The example in this section has the A node as the final active node for the partition. However, there is no requirement that this is the case. The active node for a partition could be any other node in the cluster after the restore completes, including the source node.

Figure 7.21 shows another possible multi-master scenario where the network outage causes a cluster to be split into multiple sub-clusters. In this diagram there are two sub-clusters:

- Sub-cluster one contains nodes A and B
- Sub-cluster two contains nodes C and C

![Figure 7.21. Split cluster](image)

To restore this cluster, the operator must decide which sub-cluster nodes should be treated as the initiating nodes and restore from the source nodes in the other sub-cluster. The steps to restore the individual nodes are identical to the ones described above.

There is no requirement that the initiating and source nodes have to span sub-cluster boundaries. The source and initiating nodes can be in the same sub-clusters.

**Default conflict resolution** The default conflict resolution behavior if no compensation triggers are installed is:

- **Instance Added** - the instance from the initiating node is added to the partition.
- **Key Conflict** - the instance on the initiating node is discarded. The instance on the source node is kept.
- **State Conflict** - the instance on the initiating node is discarded. The instance on the source node is kept.

**Node quorum**
Geographic redundancy

All of the Distributed Transactional Memory high availability features can be used across a WAN to support application deployment topologies that require geographic redundancy without any additional hardware or software. The same transactional guarantees are provided to nodes communicating over a WAN, as are provided over a LAN.

Figure 7.22 shows an example system configuration that replicates partitions across the WAN so that separate data centers can take over should one completely fail. This example system configuration defines:

- Partition A with node list One, Two, Four
- Partition B with node list Three, Four, Two

Under normal operation partition A's active node is One, and highly available objects are replicated to node Two, and across the WAN to node Four, and partition B's active node is Three, and highly available objects are replicated to node Four, and across the WAN to node Two. In the case of a Data Center North outage, partition A will transition to being active on node Four in Data Center South. In the case of a Data Center South outage, partition B will transition to being active on node Two in Data Center North.
The following should be considered when deploying geographically redundant application nodes:

- network latency between locations. This network latency will impact the transaction latency for every partitioned object modification in partitions that span the WAN.

- total network bandwidth between locations. The network bandwidth must be able to sustain the total throughput of all of the simultaneous transactions at each location that require replication across the WAN.

Geographically distributed nodes should be configured to use the static discovery protocol described in the section called “Location discovery” on page 52.
Nodes in a cluster can be upgraded independently of other nodes in the cluster. These upgrades include:

- Product versions
- Application versions
- Operating system versions

The upgrade functionality ensures that a cluster never has to be completely brought down for any upgrades.

All nodes in a cluster can be at different product versions. Different product versions are detected when a node joins a cluster and any required protocol negotiation is done automatically at that time. This allows product versions to be upgraded on each node independently.

Different application versions can also be running on each node in a cluster. Application differences between two nodes are detected, and the objects are made compatible at runtime, either transparently, or by application specific code to resolve the inconsistencies. This allows application versions to be upgraded on each node independently.

All nodes in a cluster can use different operating system versions. This allows operating system version upgrades to be done on each node independently.

**Application versions**

Classes in an application are versioned using a `serialVersionUID`. The rules used to determine which class is the latest version are:

- The class with a larger `serialVersionUID` value is considered as a newer version than the one with a smaller value.
A class that does not have a `serialVersionUID` defined is considered older than a class with a `serialVersionUID` defined.

If classes have the same `serialVersionUID` value the node with the newest shared memory time stamp (see the section called “Locations” on page 51) is considered newest.

### Detecting version changes

Version changes are detected automatically during initialization and as classes are loaded into JVMs running on a node. As nodes connect to each other, and as new types are loaded into a JVM, a *type exchange* occurs between the two nodes. A type exchange is performed for both application classes and product runtime structures. The type exchange protocol is shown in Figure 8.1.

**Figure 8.1. Type exchange**

The steps in Figure 8.1 are:

1. Node one sends CRC values for all types defined on node one.
2. Node two compares the CRC values for all types sent from node one found on node two.
3. If the CRC values are different for a type, node two sends node one its definition of the type.
4. Node one saves the definition of the types received from node two in a *type mismatch* table for node two.
5. Node one sends node two its definition of the mismatched types received from node two.
6. Node two saves the type definitions received from node one in a type mismatch table for node one.

The CRC defined above, is a computed numeric value that is used to determine whether a type definition has changed. The CRC value is identical on nodes that have the same type definition. The type information sent if the CRC values differ is a complete type definition that includes:

- field definitions
- inheritance hierarchy
- version information

The use of a CRC to determine type changes mimizes network bandwidth in the case where type information is identical.

Type mismatch tables exist for each node for which mismatched type information was detected. Type mismatch tables contain this information:

- Complete type definition, including the type name.
- Version number

Whenever objects are marshaled for a type (reading and writing), the type mismatch table is checked to see if the type matches for the two nodes communicating. If a type is found in the type mismatch table - the object is upgraded as described in the section called “Object upgrades” on page 101.

## Object upgrades

Objects are always upgraded on the node that contains the newest version of the class (see the section called “Application versions” on page 99). This technique is called *most current version makes right*. This is true for both sending and receiving objects between nodes. This ensures that no application changes are required on nodes running an earlier version of a class.

Object upgrades can be transparent, or non-transparent. Transparent changes are handled automatically without any required application support. Non-transparent changes require an application to implement an *object mismatch trigger*. See the *Distributed Transactional Memory Java Developer's Guide* for details on supported upgrades and transparent vs. non-transparent changes.

## Error handling

The overriding error handling policy for upgraded classes is to *do no harm* on nodes running older versions.

If an error is detected when reading an object from a remote node with an earlier version of a class definition, the error is logged, but not propagated back to the transaction initiator on the remote node. The error is not propagated to the initiator because the previous version of the class file has no knowledge of the new class version and it would not have any mechanism to handle the error. This is consistent with the *do no harm policy*.

Possible causes of errors are:

- application defect in upgrade code
- non-unique key errors because of inconsistent key values
The node administrator can make a decision on whether these errors are acceptable. If they are not acceptable, the node is taken offline and the upgraded classes restored to a previous working version. Another upgrade can be attempted after resolving the errors.

When an object is sent to a remote node with an earlier version of a class definition, any errors detected on the node with the earlier class version are propagated back to the transaction initiator. In this case, the new class version can either handle the errors, or it indicates a bug in the version mapping code provided by the application. Again, this is consistent with the do no harm policy.
Distributed Transactional Memory supports online versioning of configuration data. This allows a configuration to change without having to restart a running application. Configuration data is stored as managed objects in shared memory. Applications can define their own configuration data by defining a Java class. Application defined configuration data is operationally managed the same way as predefined Distributed Transactional Memory configuration data.

Figure 9.1 shows the configuration concepts.
These concepts are defined as:

- **Type** - a specific category of configuration data that is loaded in a single configuration file. A configuration type consists of one or more configuration classes.

- **Class** - a Java configuration class. This Java class defines a new configuration object. All configuration classes are associated with a configuration type.

- **Name** - a unique name per configuration type. Multiple unique names can be associated with a configuration type. The configuration name is the unit of versioning.

- **Version** - a unique configuration version per configuration name. Multiple versions can be associated with a configuration name, but only one can be active.

- **Objects** - zero or more configuration objects associated with a configuration version. All of the configuration objects are associated with one of the configuration classes associated with the related configuration type.

- **Notifier** - a configuration notifier that handles configuration state changes (see the section called "Configuration notifiers" on page 106).

Configuration data is loaded into Distributed Transactional Memory using configuration files. The detailed syntax of these configuration files is described in the *Distributed Transactional Memory Administration*. In addition to the configuration data for the configuration objects, the configuration files also contain:

- **Type** – type of configuration data

- **Name** – configuration name

- **Version** – version number of configuration file

The `type`, `name`, and `version` information in the configuration files maps directly to the configuration concepts described above.

The `type` information in a configuration file is used to locate any configuration notifiers associated with the configuration data. The `name` and `version` are used to create or replace a configuration when the configuration is activated. See the section called “Configuration life cycle” on page 104 for more details.

For example, this configuration file is associated with a configuration `type` of `distribution`, it has a `name` of `myconfiguration`, and it is `version 1.0`.

```java
//
// This file defines version 1.0 of a distribution configuration named myconfiguration
//
configuration "myconfiguration" version "1.0" type "distribution"
{
    ...
};
```

**Configuration life cycle**

All configuration can go through the life cycle shown in Figure 9.2.
Figure 9.2. Configuration life cycle

The possible configuration states are:

- **Loaded** - configuration data has been loaded into a Distributed Transactional Memory node. This is a transient state. The configuration data automatically transitions to the Inactive state once it has been successfully loaded.

- **Inactive** - configuration data is loaded into a node, but it is not the active version.

- **Active** - the configuration version is active.

- **Removed** - configuration data has been removed from the node. This is a transient state.

Only one active version is allowed for each configuration `name` within a `type`. For example if there are two versions, version 1.0 and version 2.0, of a configuration file with a `name` value of `myconfiguration` and a `type` of `distribution`, only one can be active at a time in a node.

An audit step occurs before any configuration state changes to ensure that the configuration change does not cause runtime application failures. If an audit fails, the configuration state change does not occur and the application is left in the previous known good state.
Replacing a version

When one version of a configuration type and name is active, and a new version is activated, the old version is replaced. That is, the old version is deactivated and the new version is activated as a single Distributed Transactional Memory transaction. For example, loading and activating version 2.0 to replace version 1.0 takes place as follows:

1. Configuration type distribution and name myconfiguration version 1.0 is active.
2. Configuration type distribution and name myconfiguration version 2.0 is loaded, passes audit, and is activated.
3. Configuration type distribution and name myconfiguration version 1.0 is now inactive, and configuration type distribution and name myconfiguration version 2.0 is active.

Because the configuration replacement is done in a single Distributed Transactional Memory transaction, there is no disruption to a running application.

Deactivating a version

Deactivating a configuration version does not restore any previously active version. Another version must be activated, or loaded and activated, as a separate step. (Until this is done, there is no active version.) Nor does deactivating a version unload it; it must be explicitly removed to achieve this. Until removed, a deactivated version remains available to be reactivated again without having to reload the configuration data.

Configuration notifiers

Applications may install configuration notifiers to respond to configuration events that are raised as the configuration transitions through its life cycle. See the Distributed Transactional Memory Java Developer's Guide for details on how configuration notifiers are installed. Configuration notifiers are associated with a configuration type. Multiple notifiers can be installed for a configuration type. If multiple configuration notifiers are installed, the order in which they are called is undefined.

Configuration notifiers support:

- auditing of configuration data and application state before a state change occurs
- modifying application behavior based on a configuration state change

Audit notifier methods should ensure that the configuration state transition being audited can occur successfully. If the state transition cannot occur successfully, either because of invalid configuration data values or the current state of the application, the audit method reports a failure. If an audit fails, the configuration state change does not occur.

Table 9.1. State transition audits

<table>
<thead>
<tr>
<th>State Transition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>Configuration load audit. This audit occurs after the configuration data is loaded into memory.</td>
</tr>
<tr>
<td>activate</td>
<td>Configuration activate audit. This audit method is called when there is no previous version of the configuration data with the specified type and name active.</td>
</tr>
</tbody>
</table>
Configuration replace audit. This audit method is called when there is a previous version of the specified type and name active.

inactive Configuration deactivation audit.

remove Configuration remove audit.

Following a successful audit (except for load), a notifier method is called to perform application specific behavior associated with the configuration state transition. The application state change methods cannot fail - all validation should have been done by the associated audit method.

**Table 9.2. State transition methods**

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>Configuration data successfully loaded.</td>
</tr>
<tr>
<td>active</td>
<td>Configuration activation succeeded. This method is called when there is no previous version of the configuration data with the specified type and name active.</td>
</tr>
<tr>
<td>replace</td>
<td>Replace existing configuration data. This method is called when there is a previous version of the specified type and name active.</td>
</tr>
<tr>
<td>inactive</td>
<td>Configuration data successfully deactivated.</td>
</tr>
</tbody>
</table>

Notice that there is no method associated with removing configuration data. Configuration data removal is handled without any application involvement, other than auditing that the configuration data can be removed.
Components

A component is a JAR file that contains a property file named `ast.properties`. Components may optionally contain configuration files and notifiers. The configuration files and notifiers are specified in the `ast.properties` file. The order in which the configuration files are loaded and activated, and the notifiers executed, is also specified in the `ast.properties` file.

When an Distributed Transactional Memory JVM starts, all components are automatically activated in the order they are found in the class path. All component activation completes before `main` is called. The activation of all components occurs in a single transaction.
Figure 10.1. Activating Components

When a JVM exits, all components are deactivated in the reverse order in which they were activated. All component deactivation occurs in a single transaction.
Figure 10.2. Deactivating Components

The failure of any component activation during JVM startup causes the transaction to be rolled back and the JVM startup to fail. The rollback of the transaction causes the deactivation and unloading of any configuration files loaded and activated by previously successful component activations.
Figure 10.3. Component Activation Failure

During JVM shutdown, if a component attempts to contact a remote JVM on the same, or a different node, and the JVM is not available, the component deactivation transaction is rolled back and component deactivation is terminated. The JVM then shuts down. The result of this failure is that any components loaded by the JVM are not deactivated. The most common reason that a remote JVM is not available is that it is also being shutdown. To avoid this condition, it is recommended that component notifiers minimize the use of objects requiring access to an external JVM.

Activation

These steps are taken to activate a component:

1. Read the ast.properties file for the component.
2. Create an instance of each specified notifier and store the reference to prevent it from being garbage collected.
3. Call the pre-configuration initialization method for each notifier.
4. Load and activate each specified configuration file.
5. Call the post-configuration initialization method for each notifier.
Deactivation

Figure 10.4. Component Activation

These steps are taken to deactivate a component:

1. Call the pre-configuration termination method for each notifier.

2. Deactivate and unload each configuration file in the reverse order in which they were loaded and activated.

3. Call the post-configuration termination method for each notifier.

4. Release each notifier instance in the reverse order in which they were created.

The execution order of notifier deactivation methods and JVM shutdown hooks is undefined.
Figure 10.5. Component Deactivation
Distributed Transactional Memory system management is done using any of the following:

- Administrator via a web browser
- a command line tool named `epadmin`
- a Java Management Extensions (JMX) console

Applications can extend the standard Distributed Transactional Memory management features. Application management features are automatically visible using the standard Distributed Transactional Memory system management tools.

An application adds system management features by implementing a target. A target is a grouping of common management functions. A target has one or more commands. Each command provides a specific management function.

A command can optionally return one or more rows of data. Each row of data must have the same number of columns. The first row returned contains the column names.

Commands can execute synchronously or asynchronously. A synchronous command completes its function before it returns. An asynchronous command continues to execute after returning.

When a command is executed it is in a transaction. The transaction is committed after the command returns. This is true for both synchronous and asynchronous commands. An exception thrown by a command causes the transaction to be rolled back. A new transaction must be started when an asynchronous command calls a method on a target after returning from the initial invocation by the administration framework.

**Node logging**

Log messages generated by nodes are available in:
• node log files
• Administrator web UI
• JMX notifications

Security

All system management commands require authentication before they can be executed. The authentication information is used to both identify the user executing the command and to also check role based security policies to ensure that the user has access to the requested command. Access control is enforced before a command is executed. If the user executing the command does not have access to the requested command, an error is returned without executing the command.

Access control rules are configured for each system management target independently.

See the Distributed Transactional Memory Administration for complete details.
Index

A
accessing data
  transaction locks and, 43
Active, 80
active node, 72
  migrating, 86
adapters, 12
  endpoint, 13
  service, 13
  session, 13
administration, 15-29
  geographic redundancy, 23
  service discovery, 18
Administrator, 29
application, 4, 16
  archive, 4
  configurable, 6
  discoverable, 6
  distributed, 6
  elastic, 6
  extensible, 6
  fragments, 4
  geographic redundancy, 6
  highly available, 6
  node deploy configuration, 4
  versions, 99
application architecture, 5-14
applications
  application archive, 7
  deploy-time, 8
  design-time, 6
  fragment, 7
  fragment type, 7
ast.properties, 109
asynchronous methods, 33
  distributed reference, 33
  execution ordering, 33
  overview, 2
  shutdown behavior, 33
  target object deleted, 33
  transaction isolation, 33
availability zone, 67, 70

B
business solution
  applications, 10
  context, 5

C
cache policy, 34
  always, 34
  never, 34
  sized, 34
caching
  default caching distributed object, 33
  default caching local managed object, 33
  default caching replica object, 33
class upgrades
  overview, 4
classes
  versions, 99
cluster name, 9
cluster upgrades, 99
  application versions, 99
  operating system versions, 99
  product versions, 99
clusters, 16, 22
commit, 43
components, 109-114
  activation, 112
  component failure during activation, 111
  component failure during deactivation, 112
  deactivation, 113
  jvm shutdown, 110
  jvm startup, 109
configuration, 11, 103-107
  active, 105
  active version, 11
  audit, 105
  class, 104
  inactive, 105
  life cycle, 104
  loaded, 105
  notifier, 104
  notifiers, 106
  objects, 104
  removed, 105
  replacing, 106
  states, 105
  syntax, 11
  type, 104
  version, 104-105
conflict resolution
  instance added, 92, 95
  key conflict, 92, 95
  state conflict, 92, 95
  trigger, 92
connectivity
  adapters, 12
creating and deleting objects
  extent not locked, 43
high availability, 52
creating, updating, and deleting objects
  state conflict during, 44

D
data distribution policy, 67-68
data distribution policy binding, 67
data mapper, 67, 73
  distributed consistent hashing, 74
  inheritance, 73
  partition assignment algorithms, 73
  round-robin, 75
deadlocks
detection, 45
defered write protocol
disabled on remote method invocation, 55
discovery (see location discovery)
distributed
defered write protocol, 54
distributed consistent hashing, 74
distributed objects, 34
  (see also cache policy)
distributed reference
  shared memory timestamp, 51
distributed references, 50
Distributed Transactional Memory
  JVM, 1
distributed transactions
  detected communication failures, 59
  network errors, 56
  transaction initiator failure, 60
  undetected communication failures, 59
distribution, 49-63
  and transactions, 37
    (see also accessing data)
extents, 51
heterogeneous platform support, 50
life-cycle, 52
managed objects, 49
master node, 49
overview, 3
read field, 51
remote node states, 53
SSL, 50
TCP/IP, 50
  transaction deadlock timeout, 47
UDP, 50
write field, 51
durable object store (see persistence)
  overview, 2
dynamic data distribution policy, 67-68
dynamic discovery, 52
dynamic partition, 67-68

E
endpoint, 13
gine, 9, 16
  relationship to node, 16
  start, 28
  stop, 29
engines
  life cycle, 27
  epadmin, 29
extent, 31
  and locking, 43

F
failover, 87
  active node migration, 87
flush notifier, 35
fragment, 16, 67

G
garbage collection, 31
geographic redundancy, 96
  administration, 23
  network bandwidth, 97
  network latency, 97

H
HA (see high availability)
high availability, 13, 65-97
  geographic redundancy, 96
multi-master, 88
node quorum, 88
overview, 2
  partition active on multiple nodes, 88
high availability objects
  remote create, 52
highly available data
  data mapper, 73

I
Initial, 80
installation, 24
isolation level (see transactions, isolation)

J
JMX, 29
JVM
  multiple JVMs in a transaction, 37

K
keep-alive, 55
keepAliveSendIntervalSeconds, 55
keys
  immutable and mutable, 32
keys and queries
  overview, 2

L
LocalDefined, 82
LocalDisabled, 82
LocalEnabled, 82
location code, 51
location codes
  mapping to network addresses, 52
migration, 51
location discovery, 52
locking, 43
logging, 115

M
managed object
  cache policy, 34
managed objects, 1, 31-35
  (see also replicated objects)
    asynchronous methods, 33
    caching, 33
    explicit deletion required, 31
    keys, 31
    life cycle of, 31
    overview, 1
    queries, 32
management, 10
  nodes, 23
management tools, 29
Migrating, 80
migrating
  force replication, 87
migration
  master node, 51

N
named cache, 33
  adding a class, 34
  inheritance, 34
network addresses
  mapping to location codes, 52
node, 9, 23
  installation, 24
  life cycle, 24
  logging, 115
  remove, 27
  start, 26
  state change notifiers, 53
  stop, 26
node deploy configuration, 8
node names
  with highly available create, 52
node quorum
  disabled, 92
  node visibility, 88
  recovering partitions with multiple active nodes, 91
  states, 89
  votes, 88
node quorum state
  Active, 89
  Disabled, 89
  Partial, 89
node state
  Discovered, 53
  Down, 53
  Duplicate Location, 53
  Duplicate Timestamp, 53
  In Down Notifier, 53
  In Up Notifier, 53
  Undiscovered, 53
  Unsupported Protocol, 53
  Up, 53
nodes, 16
  detecting failure of, 55
  failure detection interval, 55
  identification of, 51
  location code, 51
  migrating partition, 86
  naming, 51
time stamp, 51
  transaction spanning of, 37
  transparent access to active node, 87
nonResponseTimeoutSeconds, 55
notifiers
  transaction, 41

O
object flushing
  distributed object, 34
  local object, 34
  notifier, 35
  replica object, 34
  throttling, 35
object identity, 50
object locking (see migration)
object references, 49
  distributed, 50
objects
  location transparency of, 50
  network location of, 51
  representation on multiple nodes, 49
partition, 67
  active node, 72
  disable, 78
  enable, 78
  objects in disabled partition, 78
  remote define, 78
  remote enable, 78
  replica node, 72
  sparse, 76
partition failover
  node list updates, 79
partition state, 80
  active, 80
  initial, 80
  migrating, 80
  notifiers, 81
  replicating, 80
  unavailable, 80
  updating, 80
partition status, 82
  LocalDefined, 82
  LocalDisabled, 82
  LocalEnabled, 82
  RemoteDefined, 82
  RemoteEnabled, 82
partitioned object, 73
partitioned objects
  updating partitions, 86
partitions, 72
  active node transparency, 87
  definition of, 77
  failover, 79
  inconsistent node lists, 78
  migrating, 86
  redefining, 78
  restore, 79
performance
  and distributed deadlock detection, 47
prepare
  transaction, 40
proxy service discovery, 20
queries
  distributed, 32
  explicit transaction locking, 32
  locking when remote object returned, 32
  scope audit, 32
  scoping, 32
quorum policy, 67

RemoteDefined, 82
RemoteEnabled, 82
remove, 27
replica data, 72
replica node
  migrating, 86
replica nodes, 72
Replicating, 80
replication, 82
  asynchronous, 82
  asynchronous consistency errors, 83
  asynchronous modifications always done on active node, 83
  synchronous, 82
  synchronous modifications, 83
replication node
  error handling, 85
rollback, 43
  due to deadlock, 45
  for automatic state conflict resolution, 44
  logging, 48
secondary store, 2
security, 116
serialVersionUID, 99
service, 13
service discovery, 18
  network protocol, 19
  properties, 18
  proxy service discovery, 20
  waiting for responses, 20
service name, 17
  fully qualified, 17
  partially qualified, 17
service type
  application, 21
  cluster, 21
  distribution, 21
  http, 21
  node, 21
session, 13
  fully qualified, 17
  partially qualified, 17
state conflicts, 44
static data distribution policy, 67, 69
static discovery, 52
static partition, 68
stop, 26, 29
system management, 115-116
  security, 116
target, 115

T
transaction isolation
  extents, 43
  objects, 43
  read committed snapshot, 43
  read consistency, 43
  serializable, 43
transactions, 37-48
  distributed, 37
  failover processing, 87
  global transaction spanning nodes, 37
  history record size, 63
  isolation, 43
  local vs distributed, 37
  locking behavior, 43
  logging, 48
  outcome voting, 63
  overview, 2
  transaction outcome voting history buffer configuration, 63
triggers, 31
  type
detecting changes, 100

U
UDP
  service discovery, 19
Unavailable, 80
Updating, 80
upgrade
  do no harm policy, 101
  error handling, 101
  most current version makes right, 101
  non-transparent changes, 101
  read error handling, 101
  transparent changes, 101
  write error handling, 102

X
XA integration
  transaction, 40